CS322 Fall 2003: Programming Language Design
- Lecture Notes –

Grigore Rosu
University of Illinois at Urbana-Champaign

Abstract

This report contains the complete lecture notes for CS322, Programming Language Design, taught by Grigore Rosu in the Fall 2003 semester at the University of Illinois at Urbana Champaign. This large PDF document has been generated automatically from the CS322's website at: http://fsl.cs.uiuc.edu/~grosu/classes/2003/fall/cs322/.

Of particular importance may be the novel technique for defining concurrent languages that starts at page 673, based on a first-order representation of computations (called "continuations" for simplicity, though only their tail is an actual "continuation structure").

CS322 - Programming Language Design (Fall 2003)

Students enrolled in this class are expected to check this web page regularly. Complete lecture notes will be posted here.

Course Description

CS322 is an advanced course on principles of programming language design. Major language design paradigms will be investigated and mathematically defined (or specified), including: static versus dynamic binding, call-by-value, by reference, by name, and by need, type checking and type inference, objects and classes, concurrency. Since the rigorous definitional framework will be *executable*, interpreters for the designed languages will be obtained for free. Software analysis tools reasoning about programs in these languages will arise naturally. Major theoretical models will be discussed.

Meetings: Tu/Th 3:30 - 4:45, 243 Mechanical Engineering Building
Credit: 1 or .75 unit
Professor: Grigore Rosu (Office: DCL 2213, WWW: http://cs.uiuc.edu/grosu, Email: grosu@cs.uiuc.edu)
Office hours: 2:00 - 4:00 on Wednesdays
Web site: http://gureni.cs.uiuc.edu/~grosu/classes/2003/fall/cs322

Newsgroup: class.cs322

Lecture notes

- 28 Aug, Lecture 01. General Information and Introduction.
  1 HW1 exercise.
- 02 Sep, Lecture 02. Maude.
  2 HW1 exercises.
- 04 Sep, Lecture 03. Initial Models, Induction and Recursive Definitions.
  2 HW1 exercises.
- 09 Sep, Lecture 04. Defining a Simple Programming Language.
  3 HW1 exercises.
  prog-lang1.maude.
- 11 Sep, Lecture 05. Designing a Functional Programming Language - Basic Notions and Features.

HW1 due, in class.
HW1 Possible Solutions

- 16 Sep, Lecture 06. Designing a Functional Programming Language - Syntax.
  1 HW2 exercise.
  prog-lang2.maude.
  prog-lang2.pdf.
- 18 Sep, Lecture 07. Designing a Functional Programming Language - Semantics; part 1.
  1 HW2 exercise.
- 23 Sep, Lecture 08. Designing a Functional Programming Language - Semantics; part 2.
  2 HW2 exercises.
- 25 Sep, Lecture 09. Designing a Functional Programming Language - Semantics; part 3.
  1 HW2 exercise.
  HW2 in one file.
- 30 Sep, Lecture 10. Typed Languages - Static Type Checking (part 1).
  type-checking.maude.
  type-checking.pdf.
- 02 Oct, Lecture 11. Typed Languages - Static Type Checking (part 2) and Dynamic Type Checking.
  HW2 due, by email before the class.
  2 HW3 exercises.
  dynamic-type-checking.maude.
  dynamic-type-checking.pdf.
- 07 Oct, Lecture 12. Typed Languages - Type Inference (part 1).
  1 HW3 exercise.
  type-infenrece.maude.
  type-inference.pdf.
- 09 Oct, Lecture 13. Typed Languages - Type Inference (part 2).
  1 HW3 exercise.
- 14 Oct, Lecture 14. Defining an Object Oriented Programming Language (part 1).
- 16 Oct, Lecture 15. Defining an Object Oriented Programming Language (part 2).
  new-funct-lang.maude.
  new-funct-lang.pdf.
  oo-lang.maude.
  oo-lang.pdf.

oo-lang-fixed.maude.
oo-lang-fixed.pdf.
2 HW4 exercises.

- 21 Oct, Lecture 16. Defining an Typed Object Oriented Language (part 1).
  1 HW4 exercise.
  HW3 due, uploaded on the server by midnight.
- 23 Oct, Lecture 17. Defining an Typed Object Oriented Language (part 2).
  1 HW4 exercise.
  oo-type-checking.maude.
  oo-.type-checking.pdf.
- 28 Oct, Lecture 18. Continuation-based Definition of a Functional Language (part 1).
  new-funct-lang2.maude.
  new-funct-lang2.pdf.
- 30 Oct, Lecture 19. Continuation-based Definition of a Functional Language (part 2).
  1 HW5 exercise.
  continuations-semantics.maude.
  continuations-semantics.pdf.
- 04 Nov, Lecture 20. Exceptions. Continuation-Passing Style (CPS) Transformation (Part 1).
  1 HW5 exercise.
  exceptions.maude.
  exceptions.pdf.
- 06 Nov, Lecture 21. Continuation-Passing Style (CPS) Transformation (Part 2).
  1 HW5 exercise.
  cps.maude.
  cps.pdf.
  HW4 due on Friday, 07 Nov, uploaded on the server by midnight.
- 11 Nov, Lecture 22. Operational Semantics.
  1 HW6 exercise.
  all-in-one.maude.
  all-in-one.pdf.
- 13 Nov, Lecture 23. Denotational Semantics (Part 1).
- 18 Nov, Lecture 24. Denotational Semantics (Part 2).
  2 HW6 exercises.
- 20 Nov, Lecture 25. Axiomatic Semantics (Hoare Logic).
  1 HW6 exercise.
  HW5 due on Monday, 24 Nov, uploaded on the server by noon.

- 02 Dec, Lecture 26. Defining a Multithreaded Language (Part 1).
  k-eq.maude.
  k-eq.pdf.
- 04 Dec, Lecture 27. Defining a Multithreaded Language (Part 2).
  k-rl.maude.
  k-rl.pdf.
  1 (the only one) HW7 exercise.
  IMPORTANT! Use the following syntax for your project: bc-syntax.maude.
- 09 Dec, Lecture 28
  HW6 due in class. If you need more time let me know, but do not forget that there is a HW7 as well as a project, which both need to be handled by Dec 12 (according to UIUC's policy)..
- 11 Dec, Lecture 29, Recapitulation.
  sample-final.pdf.
  HW7 (1 exercise) and Project due on Dec 12, 11:59PM, uploaded on the server.
- 15 Dec, Final Exam. Room 243 ME Bldg, 1:30 - 4:30 PM.

CS322 - Programming Language Design

Lecture 1: General Information and Introduction

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

General Information

- Class Webpage and Newsgroup:
  `http://fsl.cs.uiuc.edu/~grosu/classes/2003/fall/cs322`
  `news://class.cs322`
- Lectures: Tuesdays and Thursdays 3:30 - 4:45, 243 M E Bldg
- Office hours: Wednesdays 2:00 - 4:00, DCL 2213
- Instructor: Grigore Roşu
  - Office: DCL 2213
  - Email: grosu@cs.uiuc.edu
  - WWW: `http://cs.uiuc.edu/grosu`
  - Secretary: Molly Flesner
    (DCL 2120, `mflesner@uiuc.edu`, 244-6813)
- Prerequisites: CS321 or equivalent, or instruction's approval

- Textbooks
  1) Friedman, Wand and Haynes, *Essentials of Programming Languages*, MIT Press, Second Edition, 2001
  2) Winskel. *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993

  (Self contained lecture notes will be posted on class' webpage!)
- Other sources
  - The Maude Language: `http://maude.cs.uiuc.edu`
  - Proceedings of Conferences on Programming Languages
    * **POPL** (ACM Symp. on Principles of Prog. Lang.)
    * **PLDI** (ACM Symposium on Programming Language Design and Implementation)
    * **OOPSLA** (ACM Conference on Object Oriented Programming Systems, Languages and Applications)
    * **JFP** (Journal of Functional Programming)

Grading

- Students registered for 1 unit
  - Homework assignments: **45%**
  - Final exam: **35%**
  - Individual project: **25%**
- Students registered for 0.75 units
  - Homework assignments: **60%**
  - Final exam: **40%**

The Homework Assignments

- This is a labor intensive class. The notions presented in class will be often backed by machine supported formalizations which you are supposed to modify or redo entirely as part of your assignments and as part of your project
- Assignments will be complete approximately every 4 lectures, on Tuesdays, and are due the next Tuesday.
- The exercises in the lecture notes are of two types: regular exercises and homework exercises:
  - *Exercise.* The regular exercises are intended to warm you up and to help you better understand other deeper notions
  - *Homework Exercises.* The homework exercises are those which you have to solve as part of your homework
- Therefore, you have between 1 and 3 weeks for each exercises.

The Project

- The project will consist of designing a new programming language with specified features. This language will be an improved version of an existing language, namely GNU's BC (type `man bc`, or simply `bc` on any UNIX platform). The design will be formalized and an interpreter will be provided, which I will test against 25 carefully selected programs.

The Final Exam

- The final exam will test your overall understanding of the concepts discussed in class, and it is expected to be consistent with your assignments' scores

Dates and Deadlines

Th, Aug 28	Lecture 1 - Introduction	
Tu, Sep 2	Lecture 2 - Maude	
Th, Sep 4	Lecture 3 - Induction	
Tu, Sep 9	Lecture 4 - A Simple PL	HW1
Th, Sep 11	Lecture 5 - Designing a PL 1	
Tu, Sep 16	Lecture 6 - Designing a PL 2	HW1 due
Th, Sep 18	Lecture 7 - Designing a PL 3	
Tu, Sep 23	Lecture 8 - Designing a PL 4	HW2
Th, Sep 25	Lecture 9 - Type Checking	
Tu, Sep 30	Lecture 10 - Type Inference	HW2 due

Th, Oct 2	Lecture 11 - Objects and Classes 1	
Tu, Oct 7	Lecture 12 - Objects and Classes 2	HW3
Th, Oct 9	Lecture 13	
Tu, Oct 14	Lecture 14	HW3 due
Th, Oct 16	Lecture 15	
Tu, Oct 21	Lecture 16	HW4
Th, Oct 23	Lecture 17	
Tu, Oct 28	Lecture 18	HW4 due
Th, Oct 30	Lecture 19	
Tu, Nov 4	Lecture 20	HW5

Th, Nov 6	Lecture 21	
Tu, Nov 11	Lecture 22	HW5 due
Th, Nov 13	Lecture 23	
Tu, Nov 18	Lecture 24	HW6
Th, Nov 20	Lecture 25	
Tu, Dec 2	Lecture 26	HW6 due
Th, Dec 4	Lecture 27	
Tu, Dec 9	Lecture 28	
Th, Dec 11	Lecture 29 - Recapitulation	Project due
Dec 15-20	Final Exam	

Collaboration and Other Policies

- You are free to discuss the homework assignments with other students (and are encouraged to do so!). The focus of any such discussion should be limited to figuring the problem specification, not coming up with a solution. You may *not* jointly write or code any assignment. To do so will be considered cheating! All cheating will be penalized by automatically assigning a failing grade for the course and instigating further disciplinary action with the appropriate university disciplinary body.
- You should retain copies of your assignments until you receive your final grade. In the event of a discrepancy between your scores on assignments and those on the exams, you may be asked to explain any work you performed. Your grade may be adversely affected by an inability to explain your work or by

failure to retain copies of it. All students are required to take the exam in order to receive a grade in the course.

- The course has a newsgroup. You are encouraged to use this group to ask questions, answer mundane system questions for other students, discuss homeworks, etc. In consideration for your peers, please don't use the it to post flames, irrelevant messages, ads, etc.

Course Description

- Advanced course on principles of programming language design
- Major language design paradigms will be investigated and mathematically defined (or specified), including:
  - Static versus dynamic binding
  - Call-by-value, reference, name, need
  - Type checking and type inference
  - Objects and classes, concurrency
- Since the rigorous definitional framework will be *executable*, *interpreters* for the designed languages will be obtained *for free*
- Software analysis tools reasoning about programs in these languages will arise naturally
- Major theoretical models will be discussed

Course Objectives

- Present and define rigorously the major features and design concepts in programming languages
- Show how elegantly and easily one can design a programming language if one uses the right tools and framework
- Understand the significant theory of programming languages
- The practical objective of this course is *not* to *implement* programming languages, but rather to *specify or define* them formally and modularly, on a feature by feature basis
  - Interpreters will, however, be obtained for free, because in the discussed framework one can *execute* specifications
  - For that reason, we take the freedom to interchangeably use the words *define* and *implement* in this course

Specification versus Implementation

What is the difference between *specification* and *implementation*?

- An intuitive way to think of them is to associate the first to the question WHAT and the second to the question HOW
- A specification says *what* properties a system should have, while an implementation says *how* those properties are achieved
- A specification declares the interface of a system as well as the operations that it can perform, together with properties that these must fulfill; e.g., "addition is commutative". An implementation gives concrete implementations of these operations; it may require quite tricky algorithms.
- A specification can have many correct implementations. Such an implementation is said to *satisfy* the specification, and this is formally written $Impl \models Spec$

An Example: Integer Numbers

Integer numbers are part of any programming language that is worth its salt. What is the difference between *specifying* and *implementing* integer numbers?

- A *specification* of integer numbers may say
  - There are some entities called *integers*
  - There is some special integer called *zero* and written 0
  - There are two unary operations *succ* and *pred*
  - There is some binary operation $+$
  - These operators have lots of properties, including:
    - *succ* and *pred* are inverse to each other
    - $+$ is associative, commutative and has 0 as identity
    - $(\forall X, Y : \mathit{integer})\ \mathit{succ}(X) + Y = \mathit{succ}(X + Y)$
    - etc.

- A typical *implementation* of integer numbers may be the standard computer binary representation over k bits, where integers range from -2^k to $2^k - 1$.

Homework Exercise 1 *The integer standard computer binary representation over k bits is not a correct implementation of integer numbers as specified above. Which properties are not satisfied?*

- In this course we will consider an idealistic implementation of integers, in which they can have any finite number of digits. The Maude 2.0 language that we will use in this class provides such an idealistic built-in representation of integer numbers
- It is worth noting that there is often a very subtle difference between specification and implementation, especially in the context of executable specification environments

Specifying Programming Languages

- In this course the emphasis will be on *specifications of programming languages*. For a desired programming language, say $\mathcal{PL}$, the focus will be on devising a specification $Spec(\mathcal{PL})$, defining all the important aspects of $\mathcal{PL}$.
- Ideally, one would want to specify $\mathcal{PL}$ on a feature by feature basis, in a modular way. Supposing that $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$ are desired features of $\mathcal{PL}$, such as static binding, call-by-reference, etc., with specifications $Spec(\mathcal{F}_1)$, $Spec(\mathcal{F}_2)$, ..., $Spec(\mathcal{F}_n)$, then a major goal in this class is to define, or rather "design",

  $$Spec(\mathcal{PL}) \text{ as } Compose(Spec(\mathcal{F}_1), Spec(\mathcal{F}_2), ..., Spec(\mathcal{F}_n)),$$

  where *Compose* is some appropriate module, or specification, composition operator.

Maude

- Maude 2.0 will be the specification language used in this course
- It has a website: `http://maude.cs.uiuc.edu`
- It has a manual and many examples: go to its website
- It can be downloaded and easily installed

Exercise 1 *Install Maude 2.0, print its manual, and run the examples coming with the manual.*

- This course is *not* about Maude! We only use Maude as a definitional framework for programming language concepts
- However, indirectly you will become quite good at using Maude by the end of the course

Important Notes and Advice

- The lecture notes for this class will be all posted on the web and will be as detailed as needed. The textbooks will be useful as auxiliary material in order to have a better understanding of the discussed concepts
- We will *not* use Scheme in this class at all! Scheme is a quite advanced programming language and using it to *implement* interpreters hides some of the real important and interesting issues in specifying a programming language
- However, we will use all the other programming language concepts in the Friedman book, as well as a close variant of the language proposed there as a running test case

CS322 - Programming Language Design

Lecture 2: The Maude Language

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

General Information about Maude

Maude is an executable specification language that will be used in CS322. Its roots go back to Clear (Edinburgh) in 1970s and to OBJ (Stanford, Oxford) in 1980s. Other languages in the same family are CafeOBJ (Japan) and BOBJ (San Diego).

Maude is currently being developed at Stanford Research Institute and at the University of Illinois at Urbana-Champaign. Webpage: `http://maude.cs.uiuc.edu`

We will use Maude to specify programming language features, which, when put together via simple Maude module operations, lead to specifications of programming languages. Due to the fact that Maude is executable, interpreters for programming languages

designed this way will be obtained for free.

Maude version 2.0 should run on the EWS machines. Type `maude2` and you should see the welcome screen:

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.0.1 built: Aug  1 2003 17:25:59
             Copyright 1997-2003 SRI International
                   Mon Sep  1 19:08:16 2003
Maude>
```

It can also be easilly installed on any platform. Download it from its web page. Also print its user manual. You will need it.

How to Use Maude

Maude is interpreted, so you can just type your specifications and commands. However, a better way is to just type everything in one file, say `p.maude`, and then just include that file with the command "`Maude> in p`". Use "`quit`" or simply "`q`", to quit.

You can correct/edit your Maude file and then load it with "`Maude> in p`" as many times as needed. However, keep in mind that Maude maintains only one working session, in particular one module database, until you quit it. This can sometimes lead to "unexpected" errors for beginers, so if you are not sure about an error just quit and then restart Maude.

Modules

Specifications are introduced as *modules*, or *theories*. There are several kinds of modules, but we will only use *functional modules* in this class, which have the syntax

```
fmod <NAME> is
  <BODY>
endfm
```

where <NAME> can be any identifier, typically using capital leters.

The <BODY> of a module can include importation of other modules, sort and operation declarations, and a set of sentences. The sorts together with the operations form the *signature* of that module, and represent the interface of that module to the outside world.

An Example: Peano Natural Numbers

The following defines natural numbers with successor and addition, using Peano's axiomatization:

```
fmod PEANO-NAT is
  sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endfm
```

Declarations are separated by periods, which should typically have white spaces before and after.

Signatures

One sort, `Nat`, and three operations, `zero`, `succ`, and `plus`, form the signature of `PEANO-NAT`. Sorts are declared with the keywords `sort` or `sorts`, and operations with `op` or `ops`.

The three operations have zero, one and two arguments, resp., whose sorts are between `:` and `->`. Operations of zero arguments are also called *constants*, those of one argument are called *unary* and those of two *binary*. The result sort appears after `->`.

Use `ops` when two or more operations of same arguments are declared together, and use white spaces to separate them:

```
ops plus mult : Nat Nat -> Nat .
```

There are few special characters in Maude. If you use `op` in the above then only one operation, called "`plus mult`", is declared.

Equations

The two equations in `PEANO-NAT` are "properties", or constraints, that these operations should satisfy. More precisely, any *correct* implementation of Peano natural numbers should satisfy them.

All equations are quantified universally, so we should read "for all `M` and `N` of sort `Nat`, `plus(s(N), M) = s(plus(N,M))`". All equations in a module can be used in reasoning about the defined structures.

Exercise 1 *Prove that any correct implementation of* `PEANO-NAT` *should satisfy the property*

```
plus(succ(succ(zero)), succ(succ(succ(zero)))) =
 plus(succ(succ(succ(succ(zero)))), succ(zero)).
```

Equations can be applied from *left-to-right* or from *right-to-left* in reasoning, which means that equational proofs may require exponential search, thus making them theoretically intractable.

Rewriting

Maude regards all equations as *rewriting rules*, which means that they are applied only from left to right. Thus, any well-formed term can either be derived infinitely often, or be reduced to a *normal form*, which cannot be reduced anymore by applying equations as rewriting rules.

Maude's command to reduce a term to its normal form is `reduce` or simply `red`. Reduction will be made in the last defined module:

```
Maude> reduce plus(succ(succ(zero)), succ(succ(succ(zero)))) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: succ(succ(succ(succ(succ(zero)))))
Maude>
```

Millions of rewrites per second can be performed, for which reason `Maude` can be (almost) used as a programming language.

Importation

Modules can be imported, using the keywords `protecting` or `extending`. The difference between the two importation modes is subtle and semantical rather than operational. Fell free to use `extending` all the time and read the manual for more information.

The following module extends `PEANO-NAT` with multiplication:

```
fmod PEANO-NAT* is
  extending PEANO-NAT .
  op mult : Nat Nat -> Nat .
  vars M N : Nat .
  eq mult(zero, M) = zero .
  eq mult(succ(N), M) = plus(mult(N, M), M) .
endfm
```

Variable declarations are *not* imported.

The Mix-fix Notation

Some operations are prefix, but others are infix, postfix, or *mix-fix*, i.e., arguments can occur anywhere, like in the case of `if_then_else_`.

Maude supports mix-fix notation by allowing the user to place the arguments of operations wherever one wants by using underscores. Here are some examples:

```
op _+_ : Int Int -> Int .
op _! : Nat -> Nat .
op in_then_else_ : BoolExp Stmt Stmt -> Stmt .
op _?_:_ : BoolExp Exp Exp -> Exp .
```

Exercise 2 *Rewrite* `PEANO-NAT` *and* `PEANO-NAT*` *using mix-fix notation. What happens if you try to reduce an expression containing both* `_+_` *and* `_*_` *without parentheses?*

Many researchers use BNF (Bachus-Naur Form) or CFG (Context Free Grammar) notation to describe the syntax of a particular language. In this course we will use the mix-fix notation, so it is important to understand the relationship between the mix-fix notation and BNF and CFG:

Homework Exercise 1 *Argue that the mix-fix notation is equivalent to BNF and CFG. Find an interesting simple example language and express its syntax using the mix-fix notation, BNF, and production-based CFG (e.g.,* $A \rightarrow AB \mid a$ *and* $B \rightarrow BA \mid b$*).*

Parsing

Mix-fix notation leads to an interesting problem, that of *parsing*. For example, in the modified `PEANO-NAT*` of Exercise 2, how would one parse `x + y * z`?

Maude has a command called `parse`, which parses a term using the syntax of the most recently defined module. To see *all* the parentheses, first type the command "`set print with parentheses on .`" (see Subsection 17.5 in Maude's manual).

"`parse x + y * z .`" reports ambiguous parsing. Use paratheses to disambiguate parsing, such as "`parse x + (y * z) .`". To give priorities to certain operators in order to reduce the number of parenthese, use the *precedence attribute* when you declare operators:

```
op _+_ : Int Int -> Int [prec 33] .
op _*_ : Int Int -> Int [prec 31] .
```


The lower the precedence the stronger the binding! With these precedences for `_+_` and `_*_`, one can parse as follows:

```
Maude> set print with parentheses on .
Maude> parse -10 + 2 * 3 .
Int: (-10 + (2 * 3))
```

Sometimes, especially when debugging, the mixfix notation may be confusing. If you wish to turn it off, use the command:

```
Maude> set print mixfix off .
Maude> parse -10 + 2 * 3 .
Int: _+_(-10, _*_(2, 3))
```

Associativity, Commutativity and Identity Attributes

Many of the binary operations that will be used in this class will be associative (A), commutative (C) or have an identity (I), or combinations of these. E.g., `_+_` is associative, commutative and has `0` as identity. All these can be added as attributes to operations when declared:

```
op _+_ : Int Int -> Int [assoc comm id: 0 prec 33] .
op _*_ : Int Int -> Int [assoc comm id: 1 prec 31] .
```

Notice that each of the A, C, and I attributes are logically equivalent to approptiate equations, such as

```
eq A + (B + C) = (A + B) + C .
eq A + B = B + A .  ---> attention: rewriting does not terminate!
eq A + 0 = A .
```

Then why are the ACI attributes necessary? For several reasons:

1. Associativity removes the need for useless parantheses:

   ```
   Maude> parse -10 + 2 + 3 .
   Int: -10 + 2 + 3
   ```

2. Commutativity will allow rewriting to terminate. The normal form will, however, be *modulo commutativity*.

3. ACI matching is perhaps the most interesting reason. It will be desctibed next.

Matching Modulo Attributes

The following module is a an elegant but tricky specification for lists of integers with a membership operation `_in_`:

```
fmod INT-LIST is protecting INT .
  sort IntList .
  subsort Int < IntList .
  op nil : -> IntList .
  op __ : IntList IntList -> IntList [assoc id: nil] .
  op _in_ : Int IntList -> Bool .
  var I : Int .  vars L L' : IntList .
  eq I in L I L' = true .
  eq I in L = false [owise] .
endfm
```

Note the declaration "`subsort Int < IntList .`", which says that

integers are also lists of integers. Then the constant `nil` and the concatenation operation `__` can generate any finite list of integers:

```
Maude> parse 1 2 3 4 5 .
IntList: 1 2 3 4 5
Maude> red 1 nil 2 nil 3 nil 4 nil 5 6 7 nil .
result IntList: 1 2 3 4 5 6 7
```

Further, by AI matching, the membership operation can be defined without having to traverse the list element by element!

```
Maude> red 3 in 2 3 4 .
result Bool: true
Maude> red 3 in 3 4 5 .
result Bool: true
Maude> red 3 in 1 2 4 .
result Bool: false
```

By AI matching, `I`, `L`, and `L'` in the left-hand-side term of "`eq I`

`in L I L' = true .`" can match any integer or lists of integers, respectively, including `nil`. Since `I` can only be matched by `3`, the other matches follow automatically.

The attribute `owise` in "`eq I in L = false [owise] .`" tells Maude to apply that equation only "otherwise", that is, only if no other equation, in this case the previous one only, can be applied.

If one wants to define sets of integers, then, besides replacing the sort `IntList` probably by `IntSet`, one has to declare the concatenation also commutative, and one has to replace the first equation by "`eq I in I L = true .`".

Homework Exercise 2 *Define a Maude module called* `INT-SET` *specifying sets of integers with membership, union, intersection and difference (elements in one set and not in the other).*

The matching modulo attributes are implemented very efficiently in Maude, so you (typically) don't have to worry about efficiency.

Built-in Modules

There are several built-in modules. We will use the following:

- `BOOL`. Provides a sort `Bool` with two constants `true`, `false` and basic boolean calculus, together with `if_then_else_fi` and `_==_` operations. The later takes two terms, reduces them to their normal forms, and then returns `true` iff they are equal (modulo ACI, as appropriate); otherwise it returns `false`.
- `INT`. Provides a sort `Int`, arbitrary large integers as constants of sort `Int`, together with the usual operations on these.
- `QID`. Provides a sort `Qid` together with arbitrary large quoted identifiers, as constants of sort `Qid`, of the form: `'a`, `'b`, `'a-larger-identifier`, etc.

To see an existing module, built-in or not, type the command

```
Maude> show module <NAME> .
```

For example, "`show module BOOL .`" will output

```
fmod BOOL is protecting TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [prec 61 gather (e E)] .
  vars A B C : Bool .
  eq true and A = A .  eq false and A = false .
  eq A and A = A .     eq false xor A = A .
  eq A xor A = false . eq not A = A xor true .
  eq A and (B xor C) = A and B xor A and C .
  eq A or B = A and B xor A xor B .
  eq A implies B = not (A xor A and B) .
endfm
```

Exercise 3 *Use the command "`show module <NAME> .`" on the three modules above and try to understand them.*

CS322 - Programming Language Design

Lecture 3: Initial Models, Induction and Recursively Defined Operations

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

From now on we may use the word *model* to refer to implementations. This terminology comes from mathematical logics and "model theory", so it has a more mathematical flavor. Anyhow, we should think of both implementations and models as "realizations" of specifications.

A specification can have therefore several models: all those satisfying its properties. However, not all models are always intended. In this lecture we discuss *initial models*, for which *induction* is a valid proof technique, and then we discuss the relationship between these and *recursively defined operations*.

Bad Models

Among the models of a specification, there are some which have quite unexpected properties. For example, a simple specification of lists of bits without any equations, can be defined as follows:

```
fmod BIT-LIST is
  sorts Bit BitList .  subsort Bit < BitList .
  ops 0 1 : -> Bit .
  op nil : -> BitList .
  op _,_ : Bit BitList -> BitList .
endfm
```

One possible model, $\mathcal{M}$, can do the following:

- Implement elements of sort `BitList` as *real numbers* (yes, it looks strange but there is nothing to forbid this so far),
- Implement `nil` and `0` as *0*, and `1` as *1*,
- Implement `_,_` as *addition*.

The model $\mathcal{M}$ has very strange properties, such as

Junk

There are lots of "lists" in $\mathcal{M}$ which are not intended to be lists of bits, such as, for example, the number π.

Confusion

There are distinct lists which in fact collapse when interpreted in $\mathcal{M}$, such as, for example, the lists `0,1` and `1,0`. Concatenation becomes commutative in $\mathcal{M}$, which is highly undesirable.

Initial Models

Initial models are those with *no junk* and *no confusion*. How can we define such a model?

For the specification of lists of bits above, we can build an initial model $\mathcal{T} = (\mathcal{T}_{\texttt{Bit}}, \mathcal{T}_{\texttt{BitList}})$ as the pair of *smallest* sets with the following properties:

1. `0` and `1` belong to $\mathcal{T}_{\texttt{Bit}}$;
2. $\mathcal{T}_{\texttt{Bit}}$ is included in $\mathcal{T}_{\texttt{BitList}}$;
3. `nil` is an element of $\mathcal{T}_{\texttt{BitList}}$, and `0,L` and `1,L` are elements of $\mathcal{T}_{\texttt{BitList}}$ whenever `L` is an element of $\mathcal{T}_{\texttt{BitList}}$.

The model $\mathcal{T}$ is exactly the desired model for lists of bits: contains nothing but lists and no two distinct lists are collapsed!

One can similarly define an initial model for any signature. Intuitively, initial models of signatures consist of *exactly* all the well-formed terms over the syntax specified by the signature.

In the case of specifications, which contain not only signatures but also sentences, initial models can be defined as well. Essentially, they are defined by

> *taking the initial models of the corresponding signatures and collapsing all terms which can be shown equal using specification's sentences*.

As an example, let us reconsider the specification of natural numbers `PEANO-NAT` discussed in Lecture 2:

```
fmod PEANO-NAT is
  sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endfm
```

The initial model of its signature contains all the well-formed terms built over `zero`, `succ` and `plus` (so no variables!). When we also consider the two equations, many distinct such terms will collapse, because they become equal modulo those equations.

E.g., `plus(succ(succ(zero)), succ(succ(succ(zero))))` is equal to `plus(succ(succ(succ(succ(zero)))), succ(zero))`, so they represent only one element in the initial model of `PEANO-NAT`.

The set of all terms which are equivalent modulo the equations of a specification are typically called *equivalence classes*. So initial models of specification contain equivalence classes as elements.

From now on in this class, when we talk about models or implementations of specifications, we will actually mean initial models. Also, when we prove properties of specifications, we prove them as if for their initial models. In fact, Maude's modules introduced with the keywords `fmod ... endfm` are modules whose meaning is given by their initial models.

Induction

Why are we interested in initial models? Because in these models, *induction*, a very powerful proof technique very related to recursive definitions that will be intensively used in defining programming language features later in the course, is a valid proof technique.

Where is the Mistake?

To understand the important concept of initial model as well as its relationship to induction better, let us play a "where is the mistake" game. We prove by induction a property of a specification, and then we show that there are implementations which satisfy the specification but which do *not* satisfy the "proved" property.

We first prove that commutativity of `_+_` is a consequence of the specification of Peano natural numbers defined in Lecture 2, this time using mix-fix notation:


```
fmod PEANO-NAT is sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm
```

To show that `M + N = N + M` for all natural numbers `M` and `N`, we can do a proof by *induction* on either `M` or `N`, say `M`. Since any natural number is either `0` or the successor of a smaller natural number, we need to analyze two cases:

Case 1: `M = 0`. We need to prove that `0 + N = N + 0` for any natural number `N`. By the first equation, we only need to show that for any natural number `N`, it is the case that `N + 0 = N`. The only way to show it is by induction again, this time by `N`. There are two cases again:

Case 1.1: `N = 0`. We have to show that `0 + 0 = 0`, which follows by the first equation.

Case 1.2: Assume `n + 0 = n` for some `n` and prove that `s(n) + 0 = s(n)`. By the second equation, `s(n) + 0 = s(n + 0)`, and by the induction's hypothesis, `s(n + 0) = s(n)`. Done.

Case 2: Assume that `m + N = N + m` for some `m` and for any `N`, and prove that `s(m) + N = N + s(m)` for any `N`. By the second equation and the induction hypothesis, it follows that all what is left to prove is `N + s(m) = s(N + m)`, which we prove again by induction:

Case 2.1: `N = 0`. The equality `0 + s(m) = s(0 + m)` is immediate because of the first equation.

Case 2.2: Assume that `n + s(m) = s(n + m)` for some `n` and show `s(n) + s(m) = s(s(n) + m)`. Indeed, `s(n) + s(m) = s(n + s(m)) = s(s(n + m))` by the second equation and the induction hypothesis, and `s(s(n) + m) = s(s(n + m))` by the second equation.

Therefore, by several applications of induction we proved that addition on Peano natural numbers is commutative. Hence, one would naturally expect that addition should be commutative for *any* implementation of natural numbers satisfying the Peano axioms in `PEANO-NAT`. Well, let us consider the following implementation $\mathcal{S}$:

- Natural numbers are interpreted as strings;
- `0` is interpreted as the empty string;
- `s(N)` is the string `aN`, that is, the character `a` concatenated with the string `N`;
- `_+_` is implemented as string concatenation.

$\mathcal{S}$ obviously satisfies the two axioms of `PEANO-NAT`. However, addition is *not* commutative in $\mathcal{S}$! Where is the problem? What's going on here? There is nothing wrong, just that

> *Proofs by induction are NOT valid for all possible models/implementations, but only for special ones!*

Initial models are among those. We will consider only specifications whose intended models are initial, so proofs by induction are valid.

Homework Exercise 1 *Show that addition is associative in* `PEANO-NAT` *and that multiplication is commutative and associative in* `PEANO-NAT*`*, where we also replace* `mult` *by its mix-fix variant* `_*_`*. These proofs need to be done by induction. Describe also a model/implementation of* `PEANO-NAT*`*, where multiplication is implemented in such a way that it is neither commutative nor associative. You can extend the one on strings if you wish.*

Constructor versus Defined Operations

So far, we have done all the proofs by induction considering that any Peano natural number is generated by `0` and `s`. Why did we do that, considering that `PEANO-NAT` had declared three operations:

```
op 0 : -> Nat .
op s : Nat -> Nat .
op _+_ : Nat Nat -> Nat .
```

Intuitively, the reason is that `_+_` is completely *defined* using `0` and `s`. This implies that any element of the initial model on `PEANO-NAT`, that is, an equivalence class, contains a term which is built with only `0` and `s`, so `0` and `s` can *construct* any Peano number.

It is an undecidable problem to say whether an operator is defined or not in term of others. The complexity comes from the fact that one or more operators can be defined mutually recursively.

However, fortunately there are criteria which work in most of the practical cases, and also methodologies on how to define operators in terms of (intended) constructors.

The former is quite a hard problem and certainly beyond the scope of this class, but the second is quite important because we will define many operations in term of constructors in this class.

Defining Operations using Constructors

There is no silver-bullet recipe on how to define "defined" operators, but essentially the main (safe) idea is to

Define the operator's "behavior" on each constructor.

That is what we did when we defined `plus` in `PEANO-NAT` and `mult` in `PEANO-NAT*`: we first defined them on `zero` and then on `succ`.

In general, if `c1`, ..., `cn` are the intended constructors of a data-type, in order to define a new operation `d`, make sure that all equations

```
eq d(c1(X1,...)) = ...
...
eq d(cn(Xn,...)) = ...
```

are in the specification. This gives no guarantee (e.g., one can "define" `plus` as `plus(succ(M),N) = plus(succ(M),N)`), but it is a good enough principle to follow.

Defining Operations on Lists

Let us consider the following specification of lists:

```
fmod INT-LIST is protecting INT .
  sort IntList .  subsort Int < IntList .
  op __ : Int IntList -> IntList [id: nil] .
  op nil : -> IntList .
endfm
```

Therefore, there are two constructors for lists: the empty list and the concatenation of an integer to a list. Let us next define several other important and useful operations on lists. Notice that the definition of each operator treats each of the constructors separately.

The following defines the usual length operator:

```
fmod LENGTH is protecting INT-LIST .
  op length : IntList -> Nat .
  var I : Int .  var L : IntList .
  eq length(I L) = 1 + length(L) .
  eq length(nil) = 0 .
endfm

red length(1 2 3 4 5) .   ***> should be 5
```


The following defines membership, without speculating matching as we did in Lecture 2 (in fact, this would not be possible anyway because concatenation is not defined associative as before):

```
fmod IN is protecting INT-LIST .
  op _in_ : Int IntList -> Bool .
  vars I J : Int .  var L : IntList .
  eq I in J L = if I == J then true else I in L fi .
  eq I in nil = false .
endfm

red 3 in 2 3 4 .  ***> should be true
red 3 in 3 4 5 .  ***> should be true
red 3 in 1 2 3 .  ***> should be true
red 3 in 1 2 4 .  ***> should be flase
```

The next operator appends two lists of integers:

```
fmod APPEND is protecting INT-LIST .
  op append : IntList IntList -> IntList .
  var I : Int .  vars L1 L2 : IntList .
  eq append(I L1, L2) = I append(L1, L2) .
  eq append(nil, L2) = L2 .
endfm

red append(1 2 3 4, 5 6 7 8) .
***> should be 1 2 3 4 5 6 7 8
```


The following imports APPEND and defines an operation which reverses a list:

```
fmod REV is protecting APPEND .
  op rev : IntList -> IntList .
  var I : Int .  var L : IntList .
  eq rev(I L) = append(rev(L), I) .
  eq rev(nil) = nil .
endfm

red rev(1 2 3 4 5) .   ***> should be 5 4 3 2 1
```

The next module defines an operation which sorts a list of integers by insertion sort:

```
fmod ISORT is protecting INT-LIST .
  op isort : IntList -> IntList .
  vars I J : Int .  var L : IntList .
  eq isort(I L) = insert(I, isort(L)) .
  eq isort(nil) = nil .
  op insert : Int IntList -> IntList .
  eq insert(I, J L) = if I > J then J insert(I,L) else I J L fi .
  eq insert(I, nil) = I .
endfm

red isort(4 7 8 1 4 6 9 4 2 8 3 2 7 9) .
***> should be 1 2 2 3 4 4 4 6 7 7 8 8 9 9
```


Defining Operations on Binary Trees

Let us now consider a specification of binary trees, where a tree can be either empty or an integer with a left and a right subtree:

```
fmod TREE is protecting INT .
  sort Tree .
  op ___ : Tree Int Tree -> Tree .
  op empty : -> Tree .
endfm
```

We next define some operations on such trees, also following the structure of the trees given by the two constructors above.

The next simple operation simply mirrors a tree, that is, it recursively replaces each left subtree by the mirrored right subtree and vice-versa:

```
fmod MIRROR is protecting TREE .
  op mirror : Tree -> Tree .
  vars L R : Tree .  var I : Int .
  eq mirror(L I R) = mirror(R) I mirror(L) .
  eq mirror(empty) = empty .
endfm

red mirror((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be (empty 2 (empty 6 empty)) 5 ((empty 1 empty) 3 empty)
```


Searching in binary trees can be defined as follows:

```
fmod SEARCH is protecting TREE .
  op search : Int Tree -> Bool .
  vars I J : Int .  vars L R : Tree .
  eq search(I, L I R) = true .
  eq search(I, L J R) = search(I, L) or search(I, R) [owise] .
  eq search(I, empty) = false .
endfm

red search(6, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be true
red search(7, (empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be false
```

Notice the use of the attribute `[owise]` for the second equation.

Exercise 1 *Define* `search` *with only two equations, by using the built-in* `if_then_else_fi`.

Putting Together Trees and Lists

We next define a module which imports both modules of trees and of lists on integers, and defines an operation which takes a tree and returns the list of all integers in that tree, in an infix traversal:

```
fmod FLATTEN is
  protecting APPEND .
  protecting TREE .
  op flatten : Tree -> IntList .
  vars L R : Tree .  var I : Int .
  eq flatten(L I R) = append(flatten(L), I flatten(R)) .
  eq flatten(empty) = nil .
endfm
red flatten((empty 3 (empty 1 empty)) 5 ((empty 6 empty) 2 empty)) .
***> should be 3 1 5 6 2
```

Exercise 2 *Do the same for prefix and for postfix traversals.*

Exercise 3 *Read Chapter 1 of the Friedman et al. book. Focus on the general ideas not on the Scheme particularities.*

Homework Exercise 2 *Write an executable specification in Maude 2.0, use binary trees to sort lists of integers. You should define an operation* `btsort : IntList -> IntList`*, which sorts the argument list of integers, as* `isort` *did above. In order to define it, define another operation* `bt-insert : IntList Tree -> Tree`*, which inserts each integer in the list at its place in the tree, and also use the already defined* `flatten` *operation.*

CS322 - Programming Language Design

Lecture 4: Defining a Simple Programming Language

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

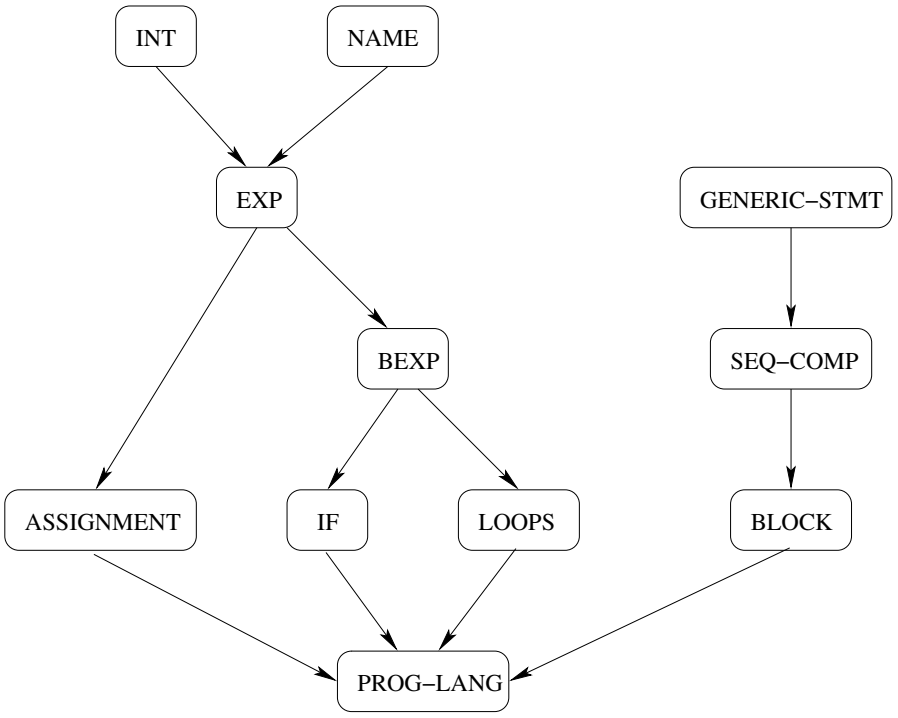
In this lecture we will design and define a very simple programming language, which has only assignments and loops. The methodology that we will follow when designing or defining programming languages is the following:

1. Define the *syntax* of the programming language;
2. Write and *parse* several programs that we would like to be able to execute in our language;
3. Define the *semantics* of our programming language;
4. *Execute* the programs that we parsed at step 2.

The simple language that will be defined in this lecture will contain the following basic features:

- Integer numbers
- Variables
- Expressions formed with integers, variables and arithmetic operators
- Assignment statement
- Boolean expressions
- Conditional statement
- Loops

These features will be introduced modularly, as following:



Syntax

We start by defining the syntax of our programming language. One advantage of using Maude to design programming languages is that one gets a parser for the desired syntax with almost no effort. This is because of its mix-fix notation enriched with operator precedences, and because of the built-in parser for this notation.

Let us start by defining the (variable) names that can be used in our language. One can essentially use any quoted identifier provided by the built-in module `QID`. In addition to those, we also define all one letter names as constants of sort `Name`:

```
fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm
```


Syntax of Expressions

The next module introduces the expressions allowed in our language. They are built from integers and names using the usual arithmetic operators:

```
fmod EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sort Exp .
  subsorts Int Name < Exp .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm
```

The attribute `[ditto]` attached to an operator declaration says that that operator inherits all the attributes it had when it was

previously defined.

In our case, the operators of addition, subtraction and multiplication were already defined in the module INT that we imported; type "show module INT ." to see the attributes of these operators.

The quotient operator was not defined in INT, so we give it a precedence (31); this is the same precedence as multiplication's, which is stronger than that of addition and subtraction.

Syntax of Statements

One may not know in advance how many kinds of statements one includes in ones language. For that reason, we prefer to keep an open design. We define a module called GENERIC-STMT-SYNTAX, which will be imported by other modules defining particular statements:

```
fmod GENERIC-STMT-SYNTAX is
  sort Stmt .
  op skip : -> Stmt .
endfm
```

E.g., the following module defines the assignment statement:

```
fmod ASSIGNMENT-SYNTAX is extending GENERIC-STMT-SYNTAX .
  protecting EXP-SYNTAX .
  op _=_ : Name Exp -> Stmt [prec 40] .
endfm
```

Syntax of Sequential Composition

Sequential composition of statements is basic to any programming language. This is typically attained by using semicolons ";", but we prefer to design our language such that the use of semicolon is optional. This is because many programmers write one statement per line, in which case the semicolons may look artificial:

```
fmod SEQ-COMP-SYNTAX is protecting GENERIC-STMT-SYNTAX .
  sort StmtList .
  subsort Stmt < StmtList .
  op __ : StmtList StmtList -> StmtList [assoc] .
  op _;_ : StmtList StmtList -> StmtList [assoc] .
endfm
```

We next introduce blocks, which are obtained by enclosing a sequence of statements, like in C, C++, Java, etc., using curly brackets. In our language, blocks are parsed as ordinary statements:


```
fmod BLOCK-SYNTAX is extending SEQ-COMP-SYNTAX .
  op {_} : StmtList -> Stmt .
endfm
```

Syntax of Boolean Expressions

Boolean expressions are needed in order to define several important statements, such as conditionals and loops. They are defined on top of expressions:

```
fmod BEXP-SYNTAX is protecting EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm
```

Syntax of Conditionals

Any programming language worth its salt has conditional statements. We also include them in our simple language:

```
fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
endfm
```

Notice that we had to import both `BEXP-SYNTAX` and `GENERIC-STMT-SYNTAX`. Our general methodology is to import as few features as possible when we define a new feature. This way we keep our design as flexible and modular as possible.

Syntax of Loops

Repetition is a crucial and indispensable programming concept. This is typically obtained via `for` loops, `while` loops, `do ... until`, etc. We only consider the first two types of loops here:

```
fmod LOOPS-SYNTAX is extending BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
endfm
```

We have chosen a syntax which is close in spirit to that of C and Java. A `for` loop takes 4 arguments: a statement which is executed first, a boolean expression which is tested at each iteration, a statement which is executed at each iteration, and finally the body of the `for` statement, which is just another statement; in particular, it can be a block enclosed by curly brackets.

Putting All the Syntax Together

We can now put together all the features whose syntax we defined so far, and thus design the syntax of our first programming language:

```
fmod PROG-LANG-SYNTAX is
  extending ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending IF-SYNTAX .
  extending LOOPS-SYNTAX .
  sort Pgm .
  op __ : StmtList Exp -> Pgm .
  op _;_ : StmtList Exp -> Pgm .
endfm
```

So programs in our language consist of a series of statements followed by an expression. The intuitive meaning of these programs

is: statements are executed, the final expression is evaluated, and then its result is returned as the result of the execution.

Parsing Programs

The semantics, or the "meaning", of this simple programming language will be defined rigorously in the sequel. But before that, it is instructive to parse several programs. These programs will be later "executed" within the designed executable specification of the semantics of our language.

```
parse x .             ***> should be  Name: x
parse 'x + 1 .        ***> should be  Exp: 'x + 1
parse x - 10 .        ***> should be  Exp: x - 10
parse 'x = 10 .       ***> should be  Stmt: 'x = 10
parse 'x = 10 + 'x .  ***> should be  Stmt: 'x = 10 + 'x
```

So Maude's parser associates a sort to each correctly parsed term. Due to subsorts, a well-formed term can have multiple sorts. Maude's parser returns *the least* sort that a term can have.

If one parses terms which are not well formed, such as

```
parse x + 2 = 10 .    ***> should be  [StmtList]: x + 2 = 10
```

then Maude cannot find proper sorts for them. The term above is *not* well formed because the assignment operation is declared to have an element of sort `Name` as its left argument, while `x + 2` is parsed as a term of sort `Exp`, which is a supersort of `Name`.

The result of parsing the term above was a sort surrounded by square brackets, as opposed to just a sort as it was the case for the previous terms. Such bracketed sorts typically suggest that the parsed term was not well-formed. One can think of the sort(s) between the brackets, as well as all their subsorts, as the sorts that Maude tried to associate to that term and failed.

The following are several other small terms, using sequential composition of statements. The first is a list of statements, while the others are or are intended to be programs. Notice that all semicolons but the last one are sequential compositions, while the last one separates the list of statements from the result expression. Also, notice that the forth term is not well-formed, so the parser returns `[Pgm]` instead of `Pgm`:

```
parse x = 1 ; y = x .              ***> should be  StmtList
parse skip ; 3 + y .               ***> should be  Pgm
parse x = 1 ; y = x ; y .          ***> should be  Pgm
parse x = 1 ; y + 1 = x ; y .      ***> should be  [Pgm]
parse
      x = 1 ;
      y = (1 + x) * 2 ;
      z = x * 2 + x * y + y * 2 ;
      x + y + z
 .                                 ***> should be  Pgm
```

The following program uses a `for` loop to calculate the power x^y:

```
parse
      x = 17 ;
      y = 100 ;
      p = 1 ;
      for(i = y ; not zero?(i) ; i = i - 1) {
        p = p * x
      }
      p
 .                                  ***> should be  Pgm
```

Notice that the four statements are separated by semicolons, while the result expression is appended using the operator `__ : StmtList Exp -> Pgm` instead of semicolon. If we had not allowed concatenation without semicolon as an operator then we would have had to use semicolons even after `}`.

The following is a tricky but classical implementation of Fibonacci numbers using only two variables. Fibonacci's numbers have the property that $f_0 = 0$, $f_1 = 1$, and $f_{n+2} = f_{n+1} + f_n$ for any $n \geq 2$:

```
parse
      x = 0 ;
      y = 1 ;
      n = 1000 ;
      for(i = 0 ; not(i equals n); i = i + 1) {
        y = y + x ;
        x = y - x
      }
      y
 .                                  ***> should be  Pgm
```

The last example that we consider implements Collatz's conjecture, which says that for any natural number n, the following `while` loop *terminates*:

```
parse
      n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
      c = 0 ;
      while not (n equals 1) {
        c = c + 1 ;
        if even?(n)
        then n = n / 2
        else n = 3 * n + 1
      }
      c
 .                                ***> should be  Pgm
```

You get A+ in this class if you prove it :-). The program above returns the number of iterations of the loop.

Semantics

We will discuss *semantics* of programming languages in more depth in this class, but for now we can just think of semantics of a program as the meaning of that program, or better what it is supposed to do. For example, the program `x = 5 ; 3 + x` is supposed to return 8.

In what follows, we define the semantics of our programming language as a Maude series of specifications.

Due to the fact that Maude is executable, we will then obtain an *interpreter* for our programming language for free.

State

In order to talk about the meaning of programs, we first have to introduce the notion of *state*. A state can be intuitively seen as a data structure storing all the information needed in order to define the meaning of each programming language construct.

For our overly simplified programming language, the only thing needed is the value associated to each name. This way one can evaluate any expression by looking up into the state for the values corresponding to specific name, and so on. Since we have assignment statements in our language, besides looking up for values associated to names, we will also need to add new or change existing associations of names to values in the state.

It is clear that the notion of state is needed to define the semantics of any programming language. In order to be generic, we prefer not to commit to a particular programming language syntax when we

define the module STATE. Therefore, we will consider that a state associates integer values to generic *indexes*. All these suggest that we should define a state as a set of pairs (index,integer), together with lookup and update operations:

```
fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
  vars X : Index .  vars I I' : Int .  var S : State .
  eq ([X,I] S)[X] = I .
  eq ([X,I'] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm
```

The next step is to define the meaning of all the programming language constructs. To make sure we do not forget anything, we do it inductively, over the structure of programs on a module by module basis.

Semantics of Names

Given a state `S`, the meaning of a name `X` in `S` is the value of `X` in `S`, that is `S[X]`. We define a new operation `eval` which gives the meaning of any name in any state:

```
fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  subsort Name < Index .
  op eval : Name State -> Int .
  var X : Name .  var S : State .
  eq eval(X, S) = S[X] .
endfm
```


Semantics of Expressions

The operator `eval` extends to expressions naturally:

```
fmod EXP-SEMANTICS is protecting EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op eval : Exp State -> Int .
  vars E E' : Exp .  var I : Int .  var S : State .
  eq eval(I, S) = I .
  eq eval(E + E', S) = eval(E, S) + eval(E', S) .
  eq eval(E - E', S) = eval(E, S) - eval(E', S) .
  eq eval(E * E', S) = eval(E, S) * eval(E', S) .
  eq eval(E / E', S) = eval(E, S) quo eval(E', S) .
endfm
```

So in order to evaluate `E + E'` in a state `S`, we first evaluate `E`, then `E'`, and then we add the two obtained integers (the operations in the right-hand-side terms are all defined in the built-in module

INT). Our definition speculates the fact that in our programming language evaluating an expression is a side-effect free operation, which is typically *not* the case for most programming languages.

Semantics of Statements

The meaning of statements is different from that of expressions in what their role is to actually change the state of the program. We therefore define an operation `state` which takes a statements and a state, and gives the state after the statement is executed in the given state:

```
fmod GENERIC-STMT-SEMANTICS is protecting GENERIC-STMT-SYNTAX .
  protecting STATE .
  op state : Stmt State -> State .
  eq state(skip, S:State) = S:State .
endfm
```


The empty statement `skip` does not have effect, so it does not change the state. Notice that we have declared the Maude variable `S` of sort `State` on-the-fly, without using the keyword `var`. This may be useful when an equation does not contain many variables, or when a variable is used in only one equation.

An assignment of an expression to a name first needs to evaluate the expression in the current state, and then to update the state correspondingly:

```
fmod ASSIGNMENT-SEMANTICS is protecting ASSIGNMENT-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  extending EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq state(X = E, S) = S[X <- eval(E,S)] .
endfm
```

Semantics of Sequential Composition

A sequence of statements modifies the state incrementally:

```
fmod SEQ-COMP-SEMANTICS is protecting SEQ-COMP-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  op state : StmtList State -> State .
  var St : Stmt .  var Stl : StmtList .  var S : State .
  eq St Stl = St ; Stl .
  eq state(St ; Stl, S) = state(Stl, state(St, S)) .
endfm
```

A block is equivalent to executing its sequence of statements:

```
fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending SEQ-COMP-SEMANTICS .
  var Stl : StmtList .  var S : State .
  eq state({Stl}, S) = state(Stl, S) .
endfm
```


Semantics of Boolean Expressions

The meaning of boolean expressions in a given state is their boolean value after evaluation. Therefore, we need to define another `eval` operation, this time on boolean expressions, which uses the `eval` operator on expressions:

```
fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting EXP-SEMANTICS .
  protecting STATE .
  op eval : BExp State -> Bool .
  vars E E' : Exp .  vars BE BE' : BExp .  var S : State .
  eq eval(E equals E', S) = eval(E, S) == eval(E', S) .
  eq eval(zero?(E), S) = eval(E, S) == 0 .
  eq eval(even?(E), S) = eval(E, S) rem 2 == 0 .
  eq eval(not BE, S) = not eval(BE, S) .
  eq eval(BE and BE', S) = eval(BE, S) and eval(BE', S) .
endfm
```

Semantics of Conditionals

A conditional changes the state depending on the value of the boolean expression. If true then it is equivalent to evaluating the "then" part; otherwise is equivalent to evaluating the "else" part:

```
fmod IF-SEMANTICS is protecting IF-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending GENERIC-STMT-SEMANTICS .
  var BE : BExp .  vars St St' : Stmt .  var S : State .
  eq state(if BE then St else St', S) =
     if eval(BE, S) then state(St, S) else state(St', S) fi .
endfm
```


Semantics of Loops

To simplify the definitions of loops, we prefer to first translate `for` loops into corresponding `while` loops, and then to define only the meaning of `while`:

```
fmod LOOPS-SEMANTICS is protecting LOOPS-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending BLOCK-SEMANTICS .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  vars St St1 St2 St3 : Stmt .  var BE : BExp .  var S : State .
  eq for(St1 ; BE ; St2) St3 = St1 ; while BE {St3 ; St2} .
  eq state(while BE St, S) =
     if eval(BE, S) then state(while BE St, state(St, S)) else S fi .
endfm
```

The equation above captures the essence of `while` loops: its body

is executed as far as its condition holds.

Exercise 1 *In the module above, define* `while` *in terms of* `for` *and then define only the meaning of* `for`*.*

Homework Exercise 1 *Define another looping statement, namely* `do_until_ : Stmt BExp -> Stmt`*, which executes the statement until the condition holds. Also, define both* `while` *and* `for` *in terms of* `do_until_`*, and then define only the meaning of* `do_until_`*. Do you know any programming language which has such a statement?*

Putting All the Semantics Together

We can now put all the pieces together and define the semantics of our simple programming language. The meaning of a program is to evaluate the result expression in the state generated by the sequence of statements preceding it:

```
fmod PROG-LANG-SEMANTICS is protecting PROG-LANG-SYNTAX .
  extending ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending IF-SEMANTICS .
  extending LOOPS-SEMANTICS .
  op eval : Pgm -> Int .
  var Stl : StmtList .  var E : Exp .
  eq Stl E = Stl ; E .
  eq eval(Stl ; E) = eval(E, state(Stl, empty)) .
endfm
```

Getting an Interpreter for Free

Our major goal so far was to *define* or *specify* a simple programming language, not to *implement* it. However, since Maude is executable, we essentially have a *model* of the programming language specification. We can use this model as an *interpreter* for our programming language, that is a special "program" which executes a program by analyzing and interpreting its statements one by one:

```
red eval( skip ; 3 + y) .              ***> should be  NzNat: 3 + empty[y]
red eval( x = 1 ; y = x ; y) .         ***> should be  NzNat: 1
red eval( x = 1 ; y + 1 = x ; y) .
  ***> should be  [Index,Exp,FindResult]: eval((x = 1 ; y + 1 = x) ; y)
red eval( x = 1 ;
          y = (1 + x) * 2 ;
          z = x * 2 + x * y + y * 2 ;
          x + y + z) .                 ***> should be  NzNat: 19
```


The following calculates 17^{1000} :

```
red eval(
      x = 17 ;
      y = 1000 ;
      p = 1 ;
      for(i = y ; not zero?(i) ; i = i - 1) {
        p = p * x
      }
      p
) .
```

It works by calculating 1000 multiplications by 17, and returns the following result in about 20ms:

```
rewrites: 27031 in 0ms cpu (19ms real) (~ rewrites/second)
result NzNat: 28113918290274009317325519346051643357099390088961343927790
3
79468719678351004695108419762713987296026707288752712272251433237189294528
92187911383661665158632699262017614946963739146029379874086783044756457349
49211600455378450828626928364749579848017404426129224666236271347228322848
```

```
612077465646247238058321998185722903431211709147742087557872623125389563831834345666087271820616519211027014390522189875416357626927657201081609526326081722094834062359530937658537207086791785237668381234290524352276187346545536709827653297413110655181551618986660349926871117512433225433439774997182119618676087249461464806141541798469801463479504232051703407859479601086824304244248529910788667292805927087686717126783153039959103167224419184028754127812873201534012059492374934239307439142997143227515408965838618688656218694963470876816001852365506456535783159043991826183041597388066768415612945193482668409022538402380601877993151216468442036695577630483483667661048166053117674083668349788212517812902901004829134044967630345031806995357825175417473124185224121012909730713668471271742192042595053736215712971416309440591384199971466441365610220459369610099230839946904600075846123692088599562213501099246059564684239923141174386159885608000
```


The Fibonacci number program is also interpreted relatively efficiently. It takes also about 20ms to calculate the 1000th Fibonacci number:

```
red eval(
      x = 0 ;
      y = 1 ;
      n = 1000 ;
      for(i = 0 ; not(i equals n); i = i + 1) {
        y = y + x ;
        x = y - x
      }
      y
) .
```

```
rewrites: 38032 in 0ms cpu (23ms real) (~ rewrites/second)
result NzNat: 703303677114228158218352548771835497701812698363587327426049050871545371181969335797422494945626117334877504492417659910881863632654502236471060120533741212738673391111981393731255987676900919022452453234035 01
```

The Collatz conjecture program terminates indeed [:-)] on the huge n below:

```
red eval(
      n = 1783783426478237597439857348095823098297983475834906983749867349 ;
      c = 0 ;
      while not (n equals 1) {
        c = c + 1 ;
        if even?(n)
        then n = n / 2
        else n = 3 * n + 1
      }
      c
) .
```

after 1035 iterations in also about 20ms:

```
rewrites: 35134 in 0ms cpu (17ms real) (~ rewrites/second)
result NzNat: 1035
```


Homework Exercise 2 *Within the current definition of the simplistic programming language, what happens if a name is used in some expression before it was "initialized", that is, before it was assigned a concrete value? Modify the design of the language such that any name is by default automatically initialized with* `0`.

The next homework exercise is a bit more complex. It asks you to design a simple program analysis tool. Each homework assignment will include one harder exercise.

Homework Exercise 3 *Using uninitialized names is bad programming practice, because most programming languages, including C, do not guarantee that names are automatically initialized with a specific value. In this exercise, you are required to define an operation which takes a program and returns the set of all names which are used before initialization. More precisely, you should specify a new module, called* `UNINITIALIZED`*, which imports* `PROG-LANG-SYNTAX` *and defines sets of names (*`NameSet`*) together*

with an operation `uninitialized : Pgm -> NameSet` *which gives the set of names that are used without being initialized. Modularize your definitions as much as possible.*
Hint: *Modify the semantics of the programming language to carry not only the state but also the set of uninitialized names.*

The program analysis tool described in the exercise above falls under the category of *dynamic analysis* tools.

Exercise 2 *Add input/output to the programming language defined in this lecture. In order to do this, you should define one more statement,* `print_ : Exp -> Stmt`*, which outputs the result of evaluating the expression, and one more expression,* `read() : -> Exp .`*, which reads an integer from the input. The input and output buffers should be defined as lists of integers, and passed as auxiliary arguments.*

Homework Exercise 4 (Extra credit). *Same as Homework Exercise 3, but in the context described in Exercise 2. Note that one cannot execute the program anymore because the input is not available, so one should rely on* static analysis.

```
*** ************************** ***
*** Simple calculator language ***
*** ************************** ***

--- ------ ---
--- SYNTAX ---
--- ------ ---

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sort Exp .
  subsorts Int Name < Exp .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod GENERIC-STMT-SYNTAX is
  sort Stmt .
  op skip : -> Stmt .
endfm

fmod ASSIGNMENT-SYNTAX is extending GENERIC-STMT-SYNTAX .
  protecting EXP-SYNTAX .
  op _=_ : Name Exp -> Stmt [prec 40] .
endfm

fmod SEQ-COMP-SYNTAX is protecting GENERIC-STMT-SYNTAX .
  sort StmtList .
  subsort Stmt < StmtList .
  op __ : StmtList StmtList -> StmtList [assoc] .
  op _;_ : StmtList StmtList -> StmtList [assoc] .
endfm
```

```
fmod BLOCK-SYNTAX is extending SEQ-COMP-SYNTAX .
  op {_} : StmtList -> Stmt .
endfm

fmod BEXP-SYNTAX is protecting EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
endfm

fmod LOOPS-SYNTAX is extending BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
endfm

fmod PROG-LANG-SYNTAX is
  extending ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending IF-SYNTAX .
  extending LOOPS-SYNTAX .
  sort Pgm .
  op __ : StmtList Exp -> Pgm .
  op _;_ : StmtList Exp -> Pgm .
endfm

parse x .
parse 'x + 1 .
parse x - 10 .
parse 'x = 10 .
parse 'x = 10 + 'x .
parse x + 2 = 10 .
parse x = 1 ; y = x .
parse skip ; 3 + y .
parse x = 1 ; y = x ; y .
```

```
parse x = 1 ; y + 1 = x ; y .
parse
     x = 1 ;
     y = (1 + x) * 2 ;
     z = x * 2 + x * y + y * 2 ;
     x + y + z
 .
parse
     x = 17 ;
     y = 100 ;
     p = 1 ;
     for(i = y ; not zero?(i) ; i = i - 1) {
       p = p * x
     }
     p
 .
parse
     x = 0 ;
     y = 1 ;
     n = 1000 ;
     for(i = 0 ; not(i equals n); i = i + 1) {
       y = y + x ;
       x = y - x
     }
     y
 .
parse
     n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
     c = 0 ;
     while not (n equals 1) {
       c = c + 1 ;
       if even?(n)
       then n = n / 2
       else n = 3 * n + 1
     }
     c
 .



--- --------- ---
--- SEMANTICS ---
--- --------- ---
```

```
fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
  vars X : Index .  vars I I' : Int .  var S : State .
  eq ([X,I] S)[X] = I .
  eq ([X,I'] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  subsort Name < Index .
  op eval : Name State -> Int .
  var X : Name .  var S : State .
  eq eval(X, S) = S[X] .
endfm

fmod EXP-SEMANTICS is protecting EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op eval : Exp State -> Int .
  vars E E' : Exp .  var I : Int .  var S : State .
  eq eval(I, S) = I .
  eq eval(E + E', S) = eval(E, S) + eval(E', S) .
  eq eval(E - E', S) = eval(E, S) - eval(E', S) .
  eq eval(E * E', S) = eval(E, S) * eval(E', S) .
  eq eval(E / E', S) = eval(E, S) quo eval(E', S) .
endfm

fmod GENERIC-STMT-SEMANTICS is protecting GENERIC-STMT-SYNTAX .
  protecting STATE .
  op state : Stmt State -> State .
  eq state(skip, S:State) = S:State .
endfm

fmod ASSIGNMENT-SEMANTICS is protecting ASSIGNMENT-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  extending EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq state(X = E, S) = S[X <- eval(E,S)] .
```

```
endfm

fmod SEQ-COMP-SEMANTICS is protecting SEQ-COMP-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  op state : StmtList State -> State .
  var St : Stmt .  var Stl : StmtList .  var S : State .
  eq St Stl = St ; Stl .
  eq state(St ; Stl, S) = state(Stl, state(St, S)) .
endfm

fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending SEQ-COMP-SEMANTICS .
  var Stl : StmtList .  var S : State .
  eq state({Stl}, S) = state(Stl, S) .
endfm

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting EXP-SEMANTICS .
  protecting STATE .
  op eval : BExp State -> Bool .
  vars E E' : Exp .  vars BE BE' : BExp .  var S : State .
  eq eval(E equals E', S) = eval(E, S) == eval(E', S) .
  eq eval(zero?(E), S) = eval(E, S) == 0 .
  eq eval(even?(E), S) = eval(E, S) rem 2 == 0 .
  eq eval(not BE, S) = not eval(BE, S) .
  eq eval(BE and BE', S) = eval(BE, S) and eval(BE', S) .
endfm

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending GENERIC-STMT-SEMANTICS .
  var BE : BExp .  vars St St' : Stmt .  var S : State .
  eq state(if BE then St else St', S) =
     if eval(BE, S) then state(St, S) else state(St', S) fi .
endfm

fmod LOOPS-SEMANTICS is protecting LOOPS-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending BLOCK-SEMANTICS .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  vars St St1 St2 St3 : Stmt .  var BE : BExp .  var S : State .
  eq for(St1 ; BE ; St2) St3 = St1 ; while BE {St3 ; St2} .
```

```
  eq state(while BE St, S) =
     if eval(BE, S) then state(while BE St, state(St, S)) else S fi .
endfm

fmod PROG-LANG-SEMANTICS is protecting PROG-LANG-SYNTAX .
  extending ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending IF-SEMANTICS .
  extending LOOPS-SEMANTICS .
  op eval : Pgm -> Int .
  var Stl : StmtList .  var E : Exp .
  eq Stl E = Stl ; E .
  eq eval(Stl ; E) = eval(E, state(Stl, empty)) .
endfm



red eval( skip ; 3 + y) .
red eval( x = 1 ; y = x ; y) .
red eval( x = 1 ; y + 1 = x ; y) .
red eval(
      x = 1 ;
      y = (1 + x) * 2 ;
      z = x * 2 + x * y + y * 2 ;
      x + y + z
) .
red eval(
      x = 17 ;
      y = 1000 ;
      p = 1 ;
      for(i = y ; not zero?(i) ; i = i - 1) {
        p = p * x
      }
      p
) .
red eval(
      x = 0 ;
      y = 1 ;
      n = 1000 ;
      for(i = 0 ; not(i equals n); i = i + 1) {
        y = y + x ;
        x = y - x
      }
      y
```

```
) .
red eval(
    n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
    c = 0 ;
    while not (n equals 1) {
      c = c + 1 ;
      if even?(n)
      then n = n / 2
      else n = 3 * n + 1
    }
    c
) .
```

CS322 - Programming Language Design

Lecture 5: Designing a Functional Language - Basic Notions and Features

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

In the next several lectures we will design and define modularly a functional language. In this lecture we discuss the basic features that we want to include in our language. These features are standard in many functional languages, such as *ML*, *Scheme*, *Haskell*, and so on.

Our purpose in this lecture is *not* to define any of these languages, though you will be able to do it easily at the end of the class, but rather to define their major features in a modular way, so that one can create a new language by just combining these feature modules.

It is very important to first *understand the entire language* you want to design, and only after that to define it formally. Without the big picture in mind, your design can be poor and non-modular, so difficult to change. Today's lecture is dedicated to understanding the language we want to define.

Functional Programming Languages

Functional programming languages are characterized by allowing *functions as first class citizens*. This means that functions are manipulated like any other values in the language, so in particular they can be passed as arguments to functions, can be returned by functions, and so on.

The syntax of functional languages is typically very simple, but there are various, usually non-trivial, semantic choices when one designs a functional programming language. Syntactically, almost everything is an *expression*. Expressions are *evaluated* to *values*. As we did with the previous simple language in the previous lecture, we start with expressions that can be built with integers, names and arithmetic operators, but a conditional on expressions.

Let

The `let <Bindings> in <Exp>` construct is encountered in most of the functional programming languages. Its meaning is essentially to bind some names to values and then to evaluate an expression which may refer to those names. For example,

```
let x = 5
in x
```

is a new expression, which is evaluated to 5. One can have multiple bindings:

```
let x = 5, y = 7
in x + y
```

Nested `let` expressions are naturally allowed:

```
let x = 5
in let y = x
   in y

let x = 1
in let z = let y = x + 4
           in y
   in z
```

Both expressions above are evaluated to 5. The meaning of the `let` language construct in a given state is the following:

> Evaluate the expression in the `in` part in the state obtained after evaluating all the right-hand-side expressions in the bindings and then assigning their values to the corresponding names *in parallel*.

Notice that nothing is specified about the *order* in which the right-hand-side expressions are evaluated! Because of *side effects*, which we will allow in our language, different orders can lead to different behaviors. Different implementations (or models) of our language can take different decisions; one can even evaluate all the expressions concurrently on a multiprocessor platform.

Also, it is important to note that the right-hand-side expressions are evaluated *before* the bindings are applied. The following expression, for example, is evaluated to whatever value `x` has in the current state, which may be different from 10:

```
let x = 10, y = 0, z = x
in let a = 5, b = 7
   in z
```

Functions

Functions, called also *procedures* in our language, use the syntax `proc(<Parameters>) <Exp>`, and syntactically they are nothing but ordinary expressions. The following is therefore an expression:

```
proc(x,y,z) x * (y - z)
```

In order to *apply* or *invoke* a function on a list of arguments, we need another operation, `<Exp>(<ExpList>)`, whose first argument is expected to be a function having as many parameters as expressions in the list.

Static *type checkers*, which we will define later in the course, ensure that functions are applied correctly. For the time being, we allow even "incorrect" expressions syntactically; however, to keep the range of possible implementations of our language broad, the meaning of badly formed expressions will remain undefined.

The following are all well-formed expressions:

```
proc(x,y) 0
(proc(x,y) 0) (2,3)
(proc(y,z) y + 5 * z) (1,2)
```

The first is a function with two arguments which returns `0`, the second applies that function, and the third applies a more complicated function. The expected values after evaluating the last two expressions are, of course, `0` and `11`, respectively.

One may want to bind a name to a function in order to reuse it without typing it again. This can be easily done with the existing language constructs:

```
let f = proc(y,z) y + 5 * z
in  f(1,2) + f(3,4)
```

Evaluating the above expression should yield `34`.

Passing Functions as Arguments

In functional programming languages, functions can be passed as arguments to functions just like any other expressions. E.g.,

```
(proc(x,y) x(y)) (proc(z) 2 * z, 3)
```

evaluates to 6, since the outermost function applies its first argument, which is a function, to its second argument.

Similarly, the following evaluates to 1:

```
let f = proc(x,y) x + y,
    g = proc(x,y) x * y,
    h = proc(x,y,a,b) (x(a,b) - y(a,b))
in h(f,g,1,2)
```


Free vs. Bound Names

Like we had uninitialized variables in programs written in the simple language described in the previous lecture, we can also have free names in expressions.

Intuitively, a name is *free* in an expression if and only if that name is *referred to* in some subexpression without being apriori *declared* or *bound* by a `let` construct or by a function parameter. E.g., `x` is free in the following expressions:

```
x
let y = 10 in x
let y = x in 10
x + let x = 10 in x
let x = 10, y = x in x + y
```

as well as in the expressions

```
proc(y) x
```

```
proc(y) y + x
(proc(y) y + 1)(x)
(proc(y) y + 1)(2) + x
x(1)
(proc(x) x)(x)
```

A name can be therefore free in an expression even though it has several bound occurrences. However, `x` is *not free* in any of the following expressions:

```
let x = 1 in x + let x = 10 in x
proc(x) x
let x = proc(x) x in x(x)
```


Scope of a Name

The same name can be declared and referred to multiple times in an expression. E.g., the following are both correct and evaluate to 5:

```
let x = 4
in let x = x + 1
   in x

let x = 1
in let x = let x = x + 4 in x
   in x
```

A name declaration can be thus *shadowed* by other declarations of the same name. Then for an occurrence of a name in an expression, how can we say to which declaration it refers to? Informally, the *scope* of a declaration is "the part of the expression" in which any occurrence of the declared name refers to *that* declaration.

Static vs. Dynamic Scoping (or Binding)

Scoping of a name is trickier than it seems, because the informal "part of the expression" above cannot always be easily defined. What are the values obtained after evaluating the following?

```
let y = 1
in let f = proc(x) y
   in let y = 2
      in f(0)

let y = 1
in (proc(x,y) (x y)) (proc(x) y, 2)
```

To answer this question, we should first answer the question to which declarations the `y` in `proc(x) y` refers to. There is no definite answer, however, to this question.

Under *static (or lexical) scoping*, it refers to `y = 1`, because this is the most nested declaration of `y` containing the occurrence of `y` in `proc(x) y`. Thus, the scope of `y` in `proc(x) y` can be determined statically, by just analyzing the text of the expression. Therefore, under static scoping, the expressions above evaluate to 1.

Under *dynamic scoping*, the declaration of `y` to which its occurrence in `proc(x) y` refers cannot be detected statically anymore. It is a dynamic property, which is determined during the evaluation of the expression. More precisely, it refers to the latest declaration of `y` that takes place during the evaluation of the expression. Under dynamic scoping, both expressions above evaluate to 2.

Most of the programming languages in current use prefer static scoping of variables or names. Software developers and analysis tools can understand and reason about programs more easily under static scoping. However, there are also languages, like GNU's BC, which are dynamically scoped. Dynamic scoping, however, seems to

be easier to implement. The very first versions of LISP were also dynamically scoped.

Since both types of scoping make sense, in order to attain a maximum of flexibility in the design of our programming language, we will define them as separate Maude modules and import whichever one we want when we put together all the features in a fully functional language.

It is important to be aware of this design decision all the time during the process of defining our language, because it will influence several other design decisions that we will make.

Exercise 1 *Re-read Subsection 1.3 in Friedman on scoping and binding of variables.*

Functions Under Static Scoping

Under static scoping, all the names which occur free in a function declaration refer to statically known previous declarations. It may be quite possible that the names which occurred free in that function's declaration are redeclared before the function is invoked.

Therefore, when a function is invoked under static scoping, it is *wrong* to just evaluate the body of the function in the current state (that's what one should do under dynamic scoping)! What one should do is to *freeze* the state in which the function was declared, and then to evaluate the body of the function in *that state* rather than in the current state.

In the context of side effects the situation will actually be more complicated, since one actually wants to propagate the side effects across invocations of functions. In order to properly accommodate

side effects, the *environments* when the functions are declared rather than the states will be frozen; environments map names to *locations*, which further contain values, instead of directly to values.

This special value keeping both the function and its declaration state or environment is called a *closure* in the literature. We will discuss this concept in depth in subsequent lectures, and define it rigorously when we define our language.

But for the time being, think of a closure as containing all the information needed in order to invoke a function. It is a closure that one gets after evaluating an expression which is a function, so closures are seen as special values in our language design.

Static Scoping and Recursion

Is there anything wrong with the following expression calculating the factorial of a number recursively?

```
let f = proc(n)
          if zero?(n)
          then 1
          else n * f(n - 1)
in f(5)
```

There is nothing wrong with it under dynamic scoping, because once `f(5)` starts being evaluated, the value denoted by `f` is already known and so will stay when its body will be evaluated.

However, under static scoping, the `f` in `f(n - 1)` is not part of the closure associated to `f` by the `let` construct, so `f(n - 1)` *cannot* be evaluated when the function is invoked.

Letrec

Therefore, in order to define recursive functions under static scoping we need a new language construct. This is called `letrec` in many functional programming languages:

```
letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(5)
```

It can be used to also define mutually recursive functions:

```
letrec 'even = proc(x) if zero?(x) then 1 else 'odd(x - 1),
        'odd = proc(x) if zero?(x) then 0 else 'even(x - 1)
in 'odd(17)
```

Unlike `let`, which first evaluates the expressions in its bindings,

then creates the bindings of names to the corresponding values, and then evaluates its body expression in the new state, `letrec` works as follows:

1. Creates bindings of its names to currently unspecified values, which will become concrete values later at step 3;
2. Evaluates the binding expressions in the newly obtained state;
3. Replaces the undefined values at step 1 by the corresponding values obtained at step 2, thus obtaining a new state;
4. Evaluates its body in the new state obtained at step 3.

If one does not use the names bound by `letrec` in any of the binding expressions then it is easy to see that it is behaviorally equivalent to `let`.

However, it is crucial to note that those names bound using `letrec` are accessible in the expressions they are bound to! Those of these

names which are bound to function expressions will be therefore bound to closures including their binding in the state.

More precisely, if `S'` is the new state obtained at step 3 above when `letrec` is evaluated in a state `S`, then the value associated to a name `X` in `S'` is

- The value of `X` in `S` if `X` is not a name bound by `letrec`;
- A closure whose state (or environment, in the context of side effects) is `S'` (or the environment of `S'`, respectively) if `X` is bound to a function expression by `letrec` or to an expression which evaluates to a function;
- An integer or an undefined value otherwise.

While this is exactly what we want in the context of recursive functions, one should be very careful when one declares non-functional bindings with `letrec`. For example, the behavior of the expression


```
let x = 1
in letrec x = 7,
          y = x
   in y
```

is undefined. However, notice that the variable `x` is *not free* in

```
letrec x = 7,
       y = x
in y
```

Instead, it is bound to a location which contains a value which is not defined!

Variable Assignment

So far our functional programming language is *pure*, in the sense that it has *no side effects*. More precisely, this means that the value associated to any name in a state does not change after evaluating an expression.

Indeed, if a name is redeclared by a `let` or `letrec` construct, then a new binding is created for that name, the previous one remaining unchanged. For example, the expression below evaluates to `1`:

```
let x = 1
in let y = let x = x + 4 in x
   in x
```

The evaluation of the expression `let x = x + 4 in x` to `5` has no effect therefore on the value of `x` in the outer `let`.

There are situations, however, when one wants to modify an

existing binding. For example, suppose that one wants to define a function `f` which returns the number of times it has been called. Thus, `f() + f()` would be evaluated to `3`. In typical programming languages, this would be realized by defining some global variable which is incremented in the body of the function. In our current language, the best one can do would be

```
let c = 0
in let f =  proc() let c = c + 1
                   in c
   in f() + f()
```

or

```
let f = let c = 0
        in proc() let c = c + 1
                  in c
in f() + f()
```

Unfortunately, neither of these solves the problem correctly, they

both evaluating to 2. The reason is that the `let` construct in the body of the function *creates a new binding* of `c` each time the function is called, so the outer `c` will never be modified.

By contrast, a *variable assignment* modifies an existing binding, more precisely the one in whose scope the assignment statement takes place. Following many functional programming languages, we let `set <Name> = <Exp>` denote a variable assignment expression.

With this, the two expressions above can be correctly modified to the following, where `d` is just a dummy name used to enforce the evaluation of the variable assignment expression:

```
let c = 0
in let f =  proc() let d = set c = c + 1
                   in c
   in f() + f()
```

and


```
let f = let c = 0
        in proc() let d = set c = c + 1
                  in c
in f() + f()
```

which evaluate to 3.

In order to properly define variable assignments in a programming language, one has to refine the state into *environment* and *store*. The environment maps names to *locations*, while the store maps locations to values. Thus, in order to extract the value associated to a name in a state, one first has to find that name's location in the environment and then extract the value stored at that location.

All language constructs can be defined smoothly and elegantly now. `let` creates new locations for the bound names; assignments modify the values already existing in the store; closures freeze the environments in which functions are declared rather than the entire states. Side effects can be now correctly handled.

Parameter Passing Variations

Once one decides to allow side effects in a programing language, one also needs to decide how argument expressions are passed to functions. So far, whenever a function was invoked, bindings of its parameter names to new values obtained *after evaluating* the expressions passed as arguments were created. This kind of argument passing is called *call-by-value*. For example, the following expression evaluates to 2:

```
let x = 0
in let f = proc(x) let d = set x = x + 1
                   in x
   in f(x) + f(x)
```

Other kinds of argument passing can be encountered in other programming languages and can be quite useful in practice. Suppose for example that one wants to declare a function which

swaps the values bound to two names. One natural way to do it would be like in the following expression:

```
let x = 0, y = 1
in let f = proc(x,y) let t = x
                     in let d = set x = y
                        in let d = set y = t
                           in 0
   in let d = f(x,y)
      in x + 2 * y
```

However, this does not work under call-by-value parameter passing: the above evaluates to 2 instead of 1. In order for the above to work, one should *not* create bindings for function's parameters to new values obtained after evaluating its arguments, but instead to bind functions' parameters to the already existing locations to which its arguments are bounded. This way, both the argument names and the parameter names of the function after invocation

are bound to the same location, so whatever new value is assigned to one of these is assigned to the other too. This kind of parameter passing is known as *call-by-reference*.

One natural question to ask here is what to do if a function's parameters are call-by-reference and the function is invoked in a context with expressions which are not names as arguments. A language design decision needs to be taken. One possibility would be to generate a runtime error. Another possibility, which is the one that we will consider in our design, would be to automatically convert the calling type of those arguments to call-by-value.

Another important kind of parameter passing is *call-by-need*. Under call-by-need, an argument expression is evaluated only if needed. A typical example of call-by-need is the conditional. Suppose that one wants to define a conditional function `cond` with three arguments expected to evaluate to integers, which returns either its third or its second argument, depending on whether its first argument

evaluates to zero or not. The following defines such a function:

```
let x = 0, y = 3, z = 4,
    cond = proc(a,b,c) if zero?(a) then c else b
in cond(x, y / x, z) + x
```

Like in the expression above, there are situations when one does *not* want to evaluate the arguments of a function at invocation time. In this example, `y / x` would produce a runtime error if `x` is `0`. However, the intended role of the conditional is exactly to avoid evaluating `y / x` if `x` is `0`. There is no way to avoid a runtime error under call-by-value or call-by-reference.

Under call-by-need, the arguments of `cond` are bound to its parameter names *unevaluated* and then evaluated only when their values are needed during the evaluation of `cond`'s body. In the situation above, `a`, `b` and `c` are bound to `x`, `y / x` and `z` all unevaluated (or *frozen*), respectively, and then the body of `cond` is being evaluated. When `zero?(a)` is encountered, the expression

bound to `a`, that is `x`, is evaluated to `0`; this value now *replaces* the previous binding of `a` for later potential use. Then, by the semantics of `it_then_else` which we will soon define formally, `c` needs to be evaluated. It's value, `4`, replaces the previous binding of `c` and it is then returned as the result of `cond`'s invocation. The expression `y / z` is never needed, so it stays unevaluated, thus avoiding the undesired runtime error.

Call-by-need parameter passing is also known as *lazy evaluation*. One can arguably claim that call-by-need is computationally better in practice than call-by-value, because each argument of a function is evaluated *at most once*, while under call-by-value all arguments are evaluated regardless of whether they are needed or not. There are important functional programming languages, like *Haskell*, whose parameter passing style is call-by-need. However, since one does not know how and when the arguments of functions are evaluated, call-by-need parameter passing is typically problematic

in program analysis or verification.

The fourth parameter passing style that we will consider is *call-by-name*, which differs from call-by-need in what the argument expressions are evaluated each time they are used. In the lack of side effects, call-by-need and call-by-name will be behaviorally equivalent, though call-by-need is more efficient because it avoids re-evaluating the same expressions. However, if side effects are present, then call-by-name generates corresponding side effects whenever an argument is encountered, while call-by-need generates the side effects only once.

We will define all four styles of parameter passing in separate modules, and include only those which are desired in each particular programming language design. In order to distinguish them, each parameter will be preceded by its passing-style, e.g., `proc(value x, reference y, need z, value u, name t)`.

Sequential Composition and Loops

One way to obtain sequential composition of statements is by using `let` constructs and dummy names. For example, `Exp1` followed by `Exp2` followed by `Exp3` can be realized by

```
let d = Exp1
in let d = Exp2
   in Exp3
```

The dummy name must not occur free in any of the sequentialized expressions except the first one. Sequential composition makes sense only in the context of side effects. For example, the expression

```
let d = set x = x + y
in let d = set y = x - y
   in let d = set x = x - y
      in Exp
```


occurring in a context where `x` and `y` are already declared, evaluates to `Exp` evaluated in a state in which the values bound to `x` and `y` are swapped.

Since side effects are crucial to almost any useful programming language in current use and since sequential composition is a basic construct of these languages, we will also define it in our framework.

More precisely, we will define a language construct `{<ExpList>}`, where `<ExpList>` is a list of expressions separated by semicolons, whose meaning is that the last expression in the list is evaluated in the state obtained after propagating all the side effects obtained by evaluating the previous ones sequentially.

The expression above then can be written:

```
{ set x = x + y ;
  set y = x - y ;
  set x = x - y ;
  Exp }
```

Exercise 2 *What are the values to which the following two expressions evaluate?*

```
let f = proc(need x) x + x
in let y = 5
   in {
     f(set y = y + 3) ;
     y
   }

let y = 5,
    f = proc(need x) x + x,
    g = proc(reference x) set x = x + 3
in {
  f(g(y));
  y
}
```

What if we change the parrameter passing style to call-by-name?

Like sequential composition, *loops* do not add any computational power to our already existing programming language, because they can be methodologically replaced by recursive functions defined using `letrec`. However, since loops are so frequently used in almost any programming language and are considered basic in most algorithms, we will also provide them as part of our language.

Like with the simple imperative language defined in Lecture 3, the programming language that we will define will have both `while` and `for` loops. Their syntax will be `while <BExp> <Exp>` and `for(<Exp>;<BExp>;<Exp>)<Exp>`. They will be just other expressions, having the expected meaning.

Then in the syntax of our functional programming language, the Collatz conjecture program looks as follows:

```
let n = 178378342647, c = 0
in {
  while not (n equals 1) {
    set c = c + 1 ;
    if even?(n)
    then set n = n / 2
    else set n = 3 * n + 1
  } ;
  c
}
```

Homework 1 Solutions for Programming Language Design (CS322)

Stanley Yong and Grigore Roşu

October 1, 2003

Acknowledgements: The solutions given below are by no means the canonical answers. These solutions are modified forms of the solutions given by your classmates, and I have tried to acknowledge the sources as I went along. Please seek clarification for any doubts.

1 Homework Exercise 1, Lecture 1

Question: The integer computer binary representation over k bits is not a correct implementation of integer numbers as specied above. Which properties are not satisfied?
[10 points]

Answer: The problem is that overflows may occur. Two's complement solved the problems previous representations had: by multiple overflows the properties on integers tend to hold. Since we didn't discuss binary representations for integers in the class, what I expected from you, and this was also stated on the newsgroup, was to assume that expressions are undefined if overflows occur. If both terms of an equality are undefined then we cannot say much about the correctness of an equality, it really depends on the particularities of the implementation of integers. However, the interesting observation is that in the case of associativity, it can be the case that one term is defined while the other is not! Think for example of the situation

$$maxint + (1 + (-1)) = (maxint + 1) + (-1).$$

The first term is always defined while the second generates overflow(s).
For full credit, you were expected to demonstrate explicitly the possibility of having an undefined term on one side of the identity for associativity when the other side is defined.

2 Homework Exercise 1 for Lecture 2:

Question: Argue that the mix-fix notation is equivalent to BNF and CFG. Find an interesting simple example language and express its syntax using the mix-fix notation, BNF, and production-based CFG.
[10 points]

Answer: One should start by understanding the relationships and the differences between the three. There are almost no differences between BNF and CFG, except that, in BNF, one can use the star ($\star$) operator to refer to lists. However, this can be easily simulated in CNF by adding a new non-terminal for those lists. CFG and mix-fix are essentially identical: non-terminals correspond to sorts and productions to mix-fix operations.

An operation definition in mix-fix can be restated in BNF. For instance, the operation + defined in our Peano Naturals:

```
op _+_ :  Nat Nat -> Nat
```

could be restated in BNF form as:

$$\langle Nat \rangle ::= \langle Nat \rangle + \langle Nat \rangle$$

That is, Nat + Nat is a valid derivation for Peano Natural numbers. We realize that a terminal symbol turns out to be a symbolic identifier for an operator in mixfix notation. This occurs with + and a. The syntatic categories are on the other hand sorts.

For full credit, you were expected to show also that the BNF * and + operators were possible to represent in CFG and mix fix.

$$A \rightarrow a+$$

is equivalent to

$$A ::= a \mid aA$$

and in mixfix may be represented as

```
fmod SIG is
  sort A .
  op a : -> A .
  op a_ : A -> A .
endfm
```

also for the Kleene star,

$$A \rightarrow a*$$

is equivalent to

$$A ::= \epsilon \mid aA$$

in mixfix:

```
fmod SIG is
  sort A .
  op epsilon : -> A.
  op a : -> A .
  op __ : A A -> A [assoc id: epsilon] .
endfm
```

3 Homework Exercise 2 for Lecture 2:

Question: Define a Maude module called INT-SET specifying sets of integers with membership, union, intersection and diference (elements in one set and not in the other).
[15 points]

Answer:

```
fmod INT-SET is pr INT .
  sort IntSet .
  subsort Int < IntSet .
  op null : -> IntSet .
  op __ : IntSet IntSet -> IntSet [assoc comm id: null prec 11] .
  op _in_ : Int IntSet -> Bool .
  op _&_ : IntSet IntSet -> IntSet [assoc comm prec 15] .
```

```
  op _diff_ : IntSet IntSet -> IntSet .

  var I : Int .
  vars S1 S2 S3 : IntSet .

  eq I I = I .
  eq I in I S1 = true .
  eq I in S1 = false [owise] .
  eq I S2 & I S3 = I (S2 & S3) .
  eq S1 & S2 = null [owise] .
  eq I S1 diff I S2 = S1 diff S2 .
  eq S1 diff S2 = null [owise] .
endfm
```

The union is naturally achieved by the __ operation.

4 Homework Exercise 1 for Lecture 3:

Question: Show that addition is associative in PEANO-NAT and that multiplication is commutative and associative in PEANO-NAT*, where we also replace mult by its mix-fix variant _*_. These proofs need to be done by induction. Describe also a model/implementation of PEANO-NAT*, where multiplication is implemented in such a way that it is neither commutative nor associative. You can extend the one on strings if you wish.
[15 points]

Answer: To prove the addition is associative in PEANO-NAT is to prove the following equation: (N1 + N2) + N3 = N1 + (N2 + N3).
case 1: N1 = 0 . Then we have (0 + N2) + N3 = N2 + N3 = 0 + (N2 + N3) .
case 2: Assume (m + N2) + N3 = m + (N2 + N3), and N1 = succ(m). Then we have (succ(m) + N2)) + N3 = succ(m + N2) + N3 = succ((m + N2) + N3), and succ(m) + (N2 + N3) = succ(m + (N2 + N3)). Using the assumption, we can get (succ(m) + N2) + N3 = succ(m) + (N2 + N3) . □

To prove the multiplication is commutative, we need to prove M * N = N * M .
case 1: M = 0 . Since 0 * N = 0, We need to prove N * 0 = 0.
case 1.1: N = 0. We have 0 * 0 = 0 .
case 1.2: Assume n * 0 = 0, then succ(n) * 0 = n * 0 + 0 = 0.
case 2: Assume m * N = N * m, we need to prove succ(m) * N = N * succ(m) .
case 2.1: N = 0, succ(m) * 0 = 0 = 0 * succ(m) .
case 2.2: Assume succ(m) * n = n * succ(m), then we get succ(m) * succ(n) = m * succ(n) + succ(n) = m * n + m + succ(n) = m * n + succ(m + n), and succ(n) * succ(m) = n * succ(m) + succ(m) = n * m + n + succ(m) = n * m + succ(n + m). Therefore, succ(m) * succ(n) = succ(n) * succ(m). □

To prove the multiplication is associative, we need to prove (N1 * N2) * N3 = N1 * (N2 * N3).
case 1: N1 = 0. Then we have (0 * N2) * N3 = 0 * N3 = 0 = 0 * (N2 * N3).
case 2: Assume (n * N2) * N3 = n * (N2 * N3), then (succ(n) * N2) * N3 = (n * N2 + N2) * N3, and succ(n) * (N2 * N3) = n * (N2 * N3) + N2 * N3 .

Obviously, to prove the case 2, we only need to prove that (N1 + N2) * N3 = N1 * N3 + N2 * N3.
case 1: N3 = 0. Then the both side of the equation is equal to 0.
case 2: Assume (N1 + N2) * m = N1 * m + N2 * m, then we have (N1 + N2) * succ(m) = (N1 + N2) * m + (N1 + N2), and N1 * succ(m) + N2 * succ(m) = N1 * m + N1 + N2 * m + N2 = N1 * m + N2 * m + (N1 + N2). Therefore, we have (N1 + N2) * succ(m) = N1 * succ(m) + N2 * succ(m). □

One can implement multiplication $x * y$ as "y repeated the length of x times" if x is empty or if the first letter of x is a, and as anything, say yxy if the first letter of x is not a. This indeed satisfies the two Peano axioms of multiplication,

but it is not commutative, because, for example, $a * b = b$ while $b * a = aba$. It is not associative either, because $b * (a * c) = b * c = cbc$, while $(b * a) * c = aba * c = ccc$. Other examples are also possible.

5 Homework Exercise 2 for Lecture 3:

Question: Write an executable specification in Maude 2.0, use binary trees to sort lists of integers. You should define an operation btsort : IntList → IntList, which sorts the argument list of integers, as isort did above. In order to define it, define another operation bt-insert : IntList Tree → Tree, which inserts each integer in the list at its place in the tree, and also use the already defined flatten operation.
[10 points]

Answer:

```
fmod TREE-SORT is pr FLATTEN .
  op btsort : IntList -> IntList .
  op bt-insert : IntList Tree -> Tree .
  var L : IntList . vars T1 T2 : Tree . vars I J : Int .
  eq btsort(L) = flatten((bt-insert(L, empty))) .
  eq bt-insert(I L, T1 J T2) =
     if (I > J)
       then bt-insert(L, T1 J bt-insert(I, T2))
       else bt-insert(L, bt-insert(I, T1) J T2)
     fi .
  eq bt-insert(I L, empty) = bt-insert(L, empty I empty) .
  eq bt-insert(nil, T1) = T1 .
endfm
```

6 Homework Exercise 1 for Lecture 4:

Question: Define another looping statement, namely do_until_: Stmt BExp → Stmt, which executes the statement until the condition holds. Also, define both while and for in terms of do_until_, and then define only the meaning of do_until_. Do you know any programming language which has such a statement?
[10 points]

Answer: Rewrite the LOOPS-SEMANTICS module as what follows:

```
fmod LOOPS-SEMANTICS is protecting LOOPS-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending BLOCK-SEMANTICS .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  op do_until_ : Stmt BExp -> Stmt .
  vars St St1 St2 St3 : Stmt .  var BE : BExp .  var S : State .
  eq state(do St until BE, S) =
    if eval(BE, S)
       then state(do St until BE, state(St, S)) else state(St, S) fi.
  eq while BE St = if BE then do St until BE else skip .
  eq for (St1 ; BE ; St2) St3 =
    St1 ; if BE then do { St3 ; St2} until BE else skip .
endfm
```

Perl has such a statment. Basic also has `do until` statement, but in the form of `do until_ _ loop`. Java and C++ have a similar statement, `do_while_`.

7 Homework Exercise 2 for Lecture 4:

Question: Within the current definition of the simplistic programming language, what happens if a name is used in some expression before it was initialized, that is, before it was assigned a concrete value? Modify the design of the language such that any name is by default automatically initialized with 0.
[10 points]

Answer: If the variable is not initialized, Maude will take the term `State[Name]` as a `Int`. Thus the initial model of `Int` is broken, and the compuation involving the variable can not be evaluated to integer. The final output will be a term instead of a integer.

Add the following line to the *STATE* module to initialize the varaible to 0 by default:

```
eq S[X] = 0 [owise] .
```

8 Homework Exercise 3 for Lecture 4:

Question: Using uninitialized names is bad programming practice, because most programming languages, including C, do not guarantee that names are automatically initialized with a specific value. In this exercise, you are required to define an operation which takes a program and returns the set of all names which are used before initialization. More precisely, you should specify a new module, called UNINITIALIZED, which imports PROG-LANG-SYNTAX and defines sets of names (NameSet) together with an operation uninitialized : Pgm → NameSet which gives the set of names that are used without being initialized. Modularize your definitions as much as possible.
[20 points]

Answer: See the below code. Static analysis is applied, and the output is the set of all possibly uninitialized variables. This solution probably produces an over pessimistic analysis. To do static analysis correctly, one probably needs to implement theorem proving.

Thanks to Feng Chen for his solution. Most of you attempted to solve the dynamic analysis problem, which was fine. However quite a few did not even attempt this exercise.

```
*** ************************ ***
*** Simple calculator language ***
*** ************************ ***

--- ------ ---
--- SYNTAX ---
--- ------ ---

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sort Exp .
  subsorts Int Name < Exp .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod GENERIC-STMT-SYNTAX is
  sort Stmt .
  op skip : -> Stmt .
endfm

fmod ASSIGNMENT-SYNTAX is extending GENERIC-STMT-SYNTAX .
  protecting EXP-SYNTAX .
  op _=_ : Name Exp -> Stmt [prec 40] .
```

```
endfm

fmod SEQ-COMP-SYNTAX is protecting GENERIC-STMT-SYNTAX .
  sort StmtList .
  subsort Stmt < StmtList .
  op __ : StmtList StmtList -> StmtList [assoc] .
  op _;_ : StmtList StmtList -> StmtList [assoc] .
endfm

fmod BLOCK-SYNTAX is extending SEQ-COMP-SYNTAX .
  op {_} : StmtList -> Stmt .
endfm

fmod BEXP-SYNTAX is protecting EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
endfm

fmod LOOPS-SYNTAX is extending BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
endfm

fmod PROG-LANG-SYNTAX is
  extending ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending IF-SYNTAX .
  extending LOOPS-SYNTAX .
  sort Pgm .
  op __ : StmtList Exp -> Pgm .
  op _;_ : StmtList Exp -> Pgm .
endfm

--- --------- ---
--- CHECK ---
--- --------- ---

fmod NAME-SET is pr NAME-SYNTAX .
  sort NameSet .
  subsort Name < NameSet .
  op null : -> NameSet .
  op __ : NameSet NameSet -> NameSet [assoc comm id: null prec 11] .
  op _in_ : Name NameSet -> Bool .
  op _/\_ : NameSet NameSet -> NameSet [assoc comm prec 15] .

  var N : Name .
  vars S S1 S2 S3 : NameSet .

  eq N N S = N S .
  eq N in N S = true .
  eq N in S = false [owise] .
  eq N S1 /\ N S2 = N (S1 /\ S2) .
  eq S1 /\ S2 = null [owise] .
endfm

fmod STATE is protecting NAME-SET .
  sorts State .
  op [_,_] : NameSet NameSet -> State .
  op output : State -> NameSet .
  op _in_ : Name State -> Bool .
  op add1 : Name State -> State .
  op add2 : Name State -> State .
  op merge : State State -> State .
  var N : Name . vars NS1 NS2 NS1' NS2' : NameSet .
  eq output([NS1 , NS2]) = NS2 .
  eq N in [NS1 , NS2] = N in NS1 .
  eq add1(N, [NS1 , NS2]) = [N NS1 , NS2] .
```

```
  eq add2(N, [NS1 , NS2]) = [NS1 , N NS2] .
  eq merge([NS1 , NS2], [NS1', NS2']) = [NS1 /\ NS1' , NS2 NS2'] .
endfm

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op check : Name State -> State .
  var X : Name .  var S : State .
  eq check(X, S) = if X in S then S else add2(X, S) fi .
endfm

fmod EXP-SEMANTICS is protecting EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op check : Exp State -> State .
  vars E E' : Exp .  var I : Int .  var S : State .
  eq check(I, S) = S .
  eq check(E + E', S) = check(E', check(E, S)) .
  eq check(E - E', S) = check(E', check(E, S)) .
  eq check(E * E', S) = check(E', check(E, S)) .
  eq check(E / E', S) = check(E', check(E, S)) .
endfm

fmod GENERIC-STMT-SEMANTICS is protecting GENERIC-STMT-SYNTAX .
  protecting STATE .
  op check : Stmt State -> State .
  eq check(skip, S:State) = S:State .
endfm

fmod ASSIGNMENT-SEMANTICS is protecting ASSIGNMENT-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  extending EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq check(X = E, S) = add1(X, check(E, S)) .
endfm

fmod SEQ-COMP-SEMANTICS is protecting SEQ-COMP-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  op check : StmtList State -> State .
  var St : Stmt .  var Stl : StmtList .  var S : State .
  eq St Stl = St ; Stl .
  eq check(St ; Stl, S) = check(Stl, check(St, S)) .
endfm

fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending SEQ-COMP-SEMANTICS .
  var Stl : StmtList .  var S : State .
  eq check({Stl}, S) = check(Stl, S) .
endfm

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting EXP-SEMANTICS .
  protecting STATE .
  op check : BExp State -> State .
  vars E E' : Exp .  vars BE BE' : BExp .  var S : State .
  eq check(E equals E', S) = check(E', check(E, S)) .
  eq check(zero?(E), S) = check(E, S) .
  eq check(even?(E), S) = check(E, S) .
  eq check(not BE, S) = check(BE, S) .
  eq check(BE and BE', S) = check(BE', check(BE, S)) .
endfm

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending GENERIC-STMT-SEMANTICS .
  var BE : BExp .  vars St St' : Stmt .  var S : State .
  eq check(if BE then St else St', S) =
     merge(check(St, check(BE, S)), check(St', check(BE, S))) .
endfm

fmod LOOPS-SEMANTICS is protecting LOOPS-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending BLOCK-SEMANTICS .
  op for(_;_;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  vars St St1 St2 St3 : Stmt .  var BE : BExp .  var S : State .
  eq for(St1 ; BE ; St2) St3 = St1 ; while BE {St3 ; St2} .
  eq check(while BE St, S) = check(St, check(BE, S)) .
```

```
endfm

fmod UNINITIALIZED is
  protecting PROG-LANG-SYNTAX .
  extending ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending IF-SEMANTICS .
  extending LOOPS-SEMANTICS .
  op uninitialized : Pgm -> NameSet .
  op check : Pgm State -> State .

  var Stl : StmtList .  var E : Exp .
  var pgm : Pgm . var S : State .

  eq uninitialized(pgm) = output(check(pgm, [null, null])) .
  eq Stl E = Stl ; E .
  eq check(Stl ; E, S) = check(E, check(Stl, S)) .
endfm

red uninitialized( skip ; 3 + y) .
red uninitialized( x = 1 ; y = x ; y) .
red uninitialized( x = 1 ; y + 1 = x ; y) .
red uninitialized(
      y = (1 + x) * 2 ;
      z = x * 2 + x * y + y * 2 ;
      x + y + z
) .
red uninitialized(
      x = 17 ;
      y = 1000 ;
      for(i = y ; not zero?(i) ; i = i - 1) {
        p = p * x
      }
      p
) .
red uninitialized(
      x = 0 ;
      for(i = 0 ; not(i equals n); i = i + 1) {
        y = y + x ;
        x = y - x
      }
      y
) .
red uninitialized(
      n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
      c = 0 ;
      while not (n equals 1) {
        c = c + 1 ;
        if even?(n)
        then n = n / 2
        else n = 3 * n + 1
      }
      c
) .
```

CS322 - Programming Language Design

Lecture 6: Designing a Functional Language - Syntax

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

As with the simple imperative programming language discussed in Lecture 4, we start by defining the syntax of our less trivial functional programming language.

There will be two types of expressions: ordinary expressions and boolean expressions. The boolean expressions will be needed to define conditionals and loops. Everything else, including functions and blocks, will be ordinary expressions.

Syntax of Names

We start by defining names, exactly like before. They consist of quoted identifiers as well as all the one letter unquoted constants:

```
fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm
```

Syntax of Generic Expressions

Since almost any item defined in our language will construct expressions, a good and flexible design decision at this incipient stage is to keep the range of possible language extensions open by

defining first *generic expressions*, which will later be made extending by adding concrete expression constructors.

We already know that our expressions will include names and integers, so we already import these. Since our language will be functional, in order to invoke functions we need lists of expressions:

```
fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op `(` : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .                 ***> should be  NzNat
parse x .                 ***> should be  Name
parse 'variable .         ***> should be  Qid
parse 3, x, 'variable .   ***> should be ExpList
```

Syntax of Arithmetic Operators

We consider the same arithmetic operators as for the simple imperative language defined in Lecture 4. One can easily add many others:

```
fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse 3 + x .                                    ***> Should be  Exp
parse 3 + 'variable1 .                           ***> Should be  Exp
parse 3 + 'variable2 + x * (y - z) + 'variable1 .  ***> Should be  Exp
```


Syntax of Boolean Expressions and Conditionals

Boolean expressions will be needed in order to define conditionals and while loops:

```
fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 equals 3 .      ***> Should be  BExp
parse 3 equals 5 .      ***> Should be  BExp
parse 5 equals x .      ***> Should be  BExp
parse 3 + x equals 0 .  ***> Should be  [BExp]
```

Notice that we imported the module GENERIC-EXP-SYNTAX rather than the module ARITH-OPS-SYNTAX!

This is a very important design decision, whose reason comes from our overall modular approach in defining programming languages: one wants to keep the modules introducing new features as disconnected as possible, so they can be easily added in a "plug-and-play" style when designing a new programming language:

```
fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5) then 2 else 3 .          ***> should be Exp
parse if zero?(0) then 2 else 3 .          ***> should be Exp
parse if zero?(x) then y else z .          ***> should be Exp
parse if x equals y
      then x
      else if x equals z then z else y .   ***> should be Exp
```


Syntax of Bindings

Bindings are a crucial feature of any functional programming language. In our language, we need them for both let and letrec. In fact, we need lists of bindings separated by commas, where a binding associates a name to an expression:

```
fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm
```

Notice that bindings were once more defined modularly as generally as possible, by only importing generic expressions.

Syntax of Let

The important language construct `let` can be very easily defined now as taking a list of bindings and an expression (its body) and building another expression. This way one can nest `let` expressions:

```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse let x = 5, y = 6 in x .
parse let x = 1 in let x = 2 in x .
parse let x = 10, y = 0, z = x
      in let x = 5, y = 7 in z .
```

The above expressions all parse correctly to `Exp`. Notice, however, that the last one is not correct semantically because the `x` in binding `z = x` is not declared, but the parser cannot catch this.

Syntax of Functions or Procedures

We learned that there can be different styles of parameter passing. Our flexible design methodology will allow us to define all of them as separate language features. One is then free to add them or not into one's language. For the sake of completeness, we will add them all to our language.

To keep the possibility of adding new calling modes later open, we just define a generic sort called `CallingMode` in a module

```
fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm
```

and then define the syntax of procedural parameters generically for any parameter passing style:

```
fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : CallingMode Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id:()] .
endfm
```

Therefore, a parameter will consist of a calling mode followed by a name, and parameters are separated by commas. A function may have no parameters at all, in which case one uses `()` as usual. The back-quotes tell Maude not to interpret parentheses specially but as ordinary tokens instead.

We define next the four styles of parameter passing, as separate extensions of `CALLING-MODE-SYNTAX`:

```
fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm
```

```
fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm
```

```
fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm
```

```
fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm
```

Putting together all the above, one gets the syntax of procedure declarations. In order to invoke procedures one also needs an

application or *invocation* operator; its precedence is 0:

```
fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse (proc(need x, value y) 0) .
parse (proc(name x, value y) 0) (2,3) .
parse (proc(value x, reference y) (x y)) (proc(value x) 2, 3) .
parse (proc(value x, reference y) (x y)) (proc(value x) y, 3) .
```

Functions are first class citizens in functional languages, so they can be passed as arguments and returned as results by other functions.

Syntax of Letrec

Syntactically, `letrec` is similar to `let`:

```
fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec x = 1
  in letrec x = 2, y = x
     in y
 .   ***> should be Exp
```

Semantically, `letrec` is more complex because it involves the tricky concept of *recursive or circular environment*, which will be explained in detail when we define its semantics formally. The correctly parsed expression above is not correct semantically.

Syntax of Variable Assignments

As in many functional programming languages, we use the syntax `set <Name> = <Exp>` for variable assignment:

```
fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 .
```

So a variable assignment is just another expression. Semantically, it will be defined such that it always evaluates to the integer `1`. It is their *side effects* that make variable assignments interesting and useful in practice.

Syntax of Sequential Composition and Loops

As argued in Lecture 5, blocks and loops do not add any computational power to our language, but we also include them in the language because of their popularity and easy of use:

```
fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm
```

Putting All the Syntax Together

We can now import all the desired modules and generate the syntax module of our language. Quite complex functional programs can be written and parsed now, combining harmoniously all the features (syntactically) defined so far.

```
fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm
```


Homework Exercise 1 *Define a Maude module* `FREE-NAMES` *importing* `PROG-LANG-SYNTAX` *and defining an operation* `free-names : Exp -> NameSet`*, which collects all the names that occur free in an expression.*

Hint. *Define it inductively ovet the structure of the syntax, making sure that you do not forget any important language construct. Also, make sure that you understant the difference between* `let` *and* `letrec` *with respect to how and when they bind the names.*

```
**************************************************
*** Defining a Functional Programming Language ***
**************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse 3 + x .
parse 3 + 'variable1 .
parse 3 + 'variable2 + x * (y - z) + 'variable1 .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
```

```
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 equals 3 .
parse 3 equals 5 .
parse 5 equals x .
parse 3 + x equals 0 .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5) then 2 else 3 .
parse if zero?(0) then 2 else 3 .
parse if zero?(x) then y else z .
parse if x equals y
     then x
     else if x equals z
          then z
          else y .

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let x = 5
  in x
 .
parse
  let x = 5, y = 7
  in x
```

```
 .
parse
  let x = 5
  in let y = x
     in y
 .
parse
  let x = 1
  in let x = 2
     in x
 .
parse
  let x = 10, y = 0, z = x
  in let a = 5, b = 7
        in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : CallingMode Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
```

```
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(need x, value y) 0
 .
parse
  (proc(name x, value y) 0) (2,3)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) 2, 3)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) y, 3)
 .

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec x = 1
  in letrec x = 7, y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
```

```
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm

parse
  let x = 5, y = 7
  in x + y
 .
parse
  let x = 1
  in let x = x + 2
     in x + 1
 .
parse
  let x = 1
  in let y = x + 2
     in x + 1
 .
parse
  let x = 1
  in let z = let y = x + 4
             in y
     in z
 .
```

```
parse
  let x = 1
  in let x = let x = x + 4
             in x
     in x
 .
parse
  let x = 1
  in (x + (let x = 10 in x))
 .
parse
  proc(value x, value y, value z) x * (y - z)
 .
parse
  (proc(value y, value z) y + 5 * z) (1,2)
 .
parse
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
 .
parse
  (proc(value x, value y) x(y)) (proc(value z) 2 * z, 3)
 .
parse
  let x = proc(value x) x in x(x)
 .
parse
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
 .
parse
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
 .
parse
  let y = 1
  in (proc(value x, value y) (x y)) (proc(value x) y, 2)
 .
parse
```

```
  let x = 1
  in let x = 2,
         f = proc (value y, value z) y + x * z
     in f(1,x)
 .
parse
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
 .
parse
  let a = 3
  in let p = proc(value x) x + a, a = 5
     in a * p(2)
 .
parse
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
 .
parse
  let f = proc(value n) n + n
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
 .
parse
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(value x) (p())
---              f = proc(value a) (p())
        in f(2)
 .
parse
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
```

```
                then 0
                else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
 .
parse
  letrec f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
 .
parse
  letrec 'times4 = proc(value x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
 .
parse
  letrec 'even = proc(value x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(value x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
 .
parse
  let x = 1
  in letrec x = 7,
             y = x
     in y
 .
parse
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
 .
parse
```

```
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
 .
parse
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
 .
parse
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
 .
parse
  let f = let c = 0
          in proc()
             let d = set c = c + 1
             in c
  in f() + f()
 .
parse
  let x = 0
  in let f = proc (value x)
             let d = set x = x + 1
             in x
     in f(x) + f(x)
 .
parse
  let x = 0, y = 1
  in let f = proc(value x, value y)
             let t = x
             in let d = set x = y
                in let d = set y = t
                   in 0
     in let d = f(x,y)
        in x + 2 * y
```

```
.
parse
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
.
parse
  let x = 0, y = 3, z = 4,
      f = proc(value a, need b, need c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
.
parse
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
.

parse
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
         'odd = proc() if zero?(x) then 0
                       else let d = set x = x - 1
                            in 'even()
  in 'odd()
.
parse
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
```

```
        in 2 * x + y
.
parse
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
.
parse
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
.
parse
  let x = 3, y = 4,
      f = proc(reference a, reference b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
.
parse
  let f = proc(need x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
.
parse
  let y = 5,
      f = proc(need x) x + x,
```

```
    g = proc(reference x) set x = x + 3
 in {
     f(g(y));
     y
    }
.
parse
 let f = proc(name x) x + x
 in let y = 5
    in {
        f(set y = y + 3) ;
        y
       }
.
parse
 let y = 5,
     f = proc(name x) x + x,
     g = proc(reference x) set x = x + 3
 in {
     f(g(y));
     y
    }
.
parse
 let n = 178378342647, c = 0
 in { while not (n equals 1) {
        set c = c + 1 ;
        if even?(n)
        then set n = n / 2
        else set n = 3 * n + 1
       } ;
       c }
.



-----------------
--- Semantics ---
-----------------

fmod GENERIC-STATE is
 sorts StateAttributeName StateAttribute State .
 op empty : -> State .
 op __ : State State -> State [assoc comm id: empty] .
```

```
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is
  extending GENERIC-STATE .
  protecting LOCATION .
  sorts Index Entry Environment .
  subsort Environment < StateAttribute .
  op env : -> StateAttributeName .
  subsort Entry < Environment .
  op empty : -> Environment .
  op [_,_] : Index Location -> Entry .
  op __ : Environment Environment -> Environment [assoc comm id: empty] .
  op _[_] : Environment Index -> Location .
  op _[_<-_] : Environment Index Location -> Environment .
  vars Ix Ix' : Index .  vars L L' : Location .  var Env : Environment .
  eq ([Ix,L] Env)[Ix] = L .
  eq ([Ix,L] Env)[Ix <- L'] = [Ix,L'] Env .
  eq Env[Ix <- L'] = [Ix,L'] Env [owise] .
endfm

fmod VALUE is protecting GENERIC-STATE .
  sorts Value PreValue .
  subsort Value < PreValue .
  op  eval : PreValue State -> Value .
  op state : PreValue State -> State .
  var V : Value .  var S : State .
  eq  eval(V, S) = V .
  eq state(V, S) = S .
endfm
```

```
fmod CELL is
  protecting LOCATION .
  protecting VALUE .
  sorts Cell Cells .
  subsort Cell < Cells .
  op noCells : -> Cells .
  op [_,_] : Location PreValue -> Cell .
  op __ : Cells Cells -> Cells [assoc comm id: noCells] .
  op _[_] : Cells Location -> PreValue .
  op _[_<*_] : Cells Location PreValue -> Cells .
  vars L L' : Location .  vars Pv Pv' : PreValue .  var Cs : Cells .
  eq ([L,Pv] Cs)[L] = Pv .
  eq ([L,Pv] Cs)[L <* Pv'] = [L,Pv'] Cs .
endfm

fmod STORE is
  extending GENERIC-STATE .
  protecting CELL .
  sort Store .
  subsort Store < StateAttribute .
  op {_,_} : Location Cells -> Store .
  op store : -> StateAttributeName .
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store Location PreValue -> Store .
  op nextLoc : Store -> Location .
  vars L Ln : Location .  var Cs : Cells .  var N : Nat .
  var Pv : PreValue .
  eq {Ln,Cs}[L] = Cs[L] .
  eq {loc(N),Cs}[loc(N) <- Pv] = {loc(N + 1), Cs[loc(N),Pv]} .
  eq {Ln,Cs}[L <- Pv] = {Ln,Cs[L <* Pv]} [owise] .
  eq nextLoc({Ln,Cs}) = Ln .
endfm

fmod STATE is
  protecting STORE .
  protecting ENVIRONMENT .
  op _[_] : State Index -> PreValue .
  op _[_<-_] : State Index Location -> State .
  op _[_<-_] : State Location PreValue -> State .
  op _[_<-_] : State Index PreValue -> State .
  op _[_<*_] : State Index PreValue -> State .
  var S : State .  var Ix : Index .  var L : Location .  var Pv : PreValue .
  eq S[Ix] = S[store][S[env][Ix]] .
```

```
  eq S[Ix <- L ] = S[env <- S[env][Ix <- L]] .
  eq S[ L <- Pv] = S[store <- S[store][L <- Pv]] .
  eq S[Ix <- Pv] = S[env <- S[env][Ix <- nextLoc(S[store])]]
                   [store <- S[store][nextLoc(S[store]) <- Pv]] .
  eq S[Ix <* Pv] = S[store <- S[store][S[env][Ix] <- Pv]] .
endfm

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op  eval : Name State -> Value .
  op state : Name State -> State .
  var X : Name .  var S : State .
  eq  eval(X, S) =  eval(S[idx(X)], S) .
  eq state(X, S) = state(S[idx(X)], S) .
endfm

fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> Value .
  op state : Exp State -> State .
  op  eval : Exp -> Value .
  var I : Int .  var S : State .  vars E E' : Exp .  var El : ExpList .
  eq  eval(I, S) = int(I) .
  eq state(I, S) = S .
  eq eval(E) = eval(E, (env,empty)(store,{loc(0),noCells})) .
endfm

red eval(3) .
***> should be 3
red eval(x) .
***> should be undefined
red eval('variable) .
***> should be undefined
red eval('variable,
          (  env, [idx(x),loc(0)] [idx('variable),loc(1)])
          (store, {loc(2), [loc(0),int(5)] [loc(1),int(4)]})) .
***> should be 4

fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .
```

```
  ops add sub mul div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I), int(I')) = int(I + I') .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
  eq sub(int(I), int(I')) = int(I - I') .
  eq state(E - E', S) = state(E', state(E,S)) .
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I), int(I')) = int(I * I') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I), int(I')) = int(I quo I') .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm

red eval(3 + x, (  env, [idx(x),loc(0)] [idx('n),loc(1)])
             (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
***> should be 8
red eval(3 + 'variable1,
         (  env, [idx(x),loc(0)] [idx('variable1),loc(1)])
         (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
***> should be 7
red eval(3 + 'variable2 + x * (y - z) + 'variable1 ,
        (  env, [idx('variable2),loc(0)]
                [idx(x),loc(1)]
                [idx(y),loc(2)]
                [idx(z),loc(3)]
                [idx('variable1),loc(4)])
        (store, {loc(4), [loc(0), int(0)]
                         [loc(1), int(1)]
                         [loc(2), int(2)]
                         [loc(3), int(3)]
                         [loc(4), int(4)]})) .
***> should be 6

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .  var I : Int .
  eq  eval(E equals E', S) = eval(E, S) == eval(E', state(E, S)) .
  eq state(E equals E', S) = state(E', state(E, S)) .
```

```
  eq  eval(zero?(E), S) = eval(E, S) == int(0) .
  eq state(zero?(E), S) = state(E, S) .
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = evenIntValue?(eval(E, S)) .
  eq evenIntValue?(int(I)) = I rem 2 == 0 .
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) = eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm

red eval(3 equals 3, S) .
***> should be true
red eval(3 equals 5, S) .
***> should be false
red eval(5 equals x, S) .
***> should be false
red eval(3 + x equals 0, S) .
***> should be undefined

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  var S : State .
  eq  eval(if Be then E else E', S) = if eval(Be, S)
       then  eval(E, state(Be, S)) else  eval(E', state(Be, S)) fi .
  eq state(if Be then E else E', S) = if eval(Be, S)
       then state(E, state(Be, S)) else state(E', state(Be, S)) fi .
endfm

red eval(if zero?(5) then 2 else 3, S) .
***> should be 3
red eval(if zero?(0) then 2 else 3, S) .
***> should be 2
red eval(if zero?(x) then y else z,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15)][loc(10),int(3)][loc(12),int(5)]})) .
***> should be 5
red eval(
  if x equals y
  then x
  else if x equals z
       then z
       else y,
```

```
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15)][loc(10),int(3)][loc(12),int(5)]})) .
***> should be 3

fmod BINDINGS-SEMANTICS is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op stateBList : BindingList State -> State .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq stateBList(none, S) = S .
  eq stateBList((X = E, Bl), S) = stateBList(Bl, state(E, S)) .
  eq bindBList (none,S) in S' = S' .
  eq bindBList ((X = E, Bl), S) in S' =
     bindBList (Bl, state(E,S)) in (S'[idx(X) <- eval(E,S)]) .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq  eval(let Bl in E, S) =  eval(E, bindBList (Bl, S) in stateBList(Bl,S)) .
  eq state(let Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S) in stateBList(Bl,S))[store]] .
endfm

red eval(
  let x = 5
  in x
) .
***> should be 5
red eval(
  let x = 5, y = 7
  in x
) .
***> should be 5
red eval(
  let x = 5
  in let y = x
     in y
) .
***> should be 5
red eval(
  let x = 1
```

```
  in let x = 2
     in x
) .
***> should be 2
red eval(
  let x = 10, y = 0, z = x
  in let a = 5, b = 7
        in z
) .
***> should be undefined

fmod PARAMETER-SEMANTICS is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op state : Parameter Exp State -> State .
  op statePList : ParameterList ExpList State -> State .
  op bind : Parameter Exp State State -> State .
  op bindPList : ParameterList ExpList State State -> State .
  vars S S' : State .  var P : Parameter .  var Pl : ParameterList .
  var E : Exp .  var El : ExpList .
  eq statePList((), (), S) = S .
  eq statePList((P,Pl), (E,El), S) =
     statePList(Pl, El, state(P, E, S)) [owise] .
  eq bindPList((), (), S, S') = S' .
  eq bindPList((P,Pl), (E,El), S, S') =
     bindPList(Pl, El, state(P,E,S), bind(P,E,S,S')) .
endfm

fmod CALL-BY-VALUE-SEMANTICS is extending CALL-BY-VALUE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  var X : Name .  var E : Exp .  vars S S' : State .
  eq state(value X, E, S) = state(E, S) .
  eq  bind(value X, E, S, S') = S'[idx(X) <- eval(E, S)] .
endfm

fmod CALL-BY-REFERENCE-SEMANTICS is extending CALL-BY-REFERENCE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  vars X Y : Name .  var E : Exp .  vars S S' : State .
  eq state(reference X, Y, S) = S .
  eq state(reference X, E, S) = state(E, S) [owise] .
  eq  bind(reference X, Y, S, S') = S'[idx(X) <- S[env][idx(Y)]] .
  eq  bind(reference X, E, S, S') = S'[idx(X) <- eval(E, S)] [owise] .
endfm
```

```
fmod CALL-BY-NAME-SEMANTICS is extending CALL-BY-NAME-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op frozen : Exp Environment -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  eq  eval(frozen(E, Env), S) = eval(E, S[env <- Env]) .
  eq state(frozen(E, Env), S) = S[store <- state(E, S[env <- Env])[store]] .
  eq state(name X, E, S) = S .
  eq  bind(name X, E, S, S') = S'[idx(X) <- frozen(E, S[env])] .
endfm

fmod CALL-BY-NEED-SEMANTICS is extending CALL-BY-NEED-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op unfreeze : Exp Environment Location -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  var L : Location .
  eq  eval(unfreeze(E,Env,L), S) = eval(E, S[env <- Env]) .
  eq state(unfreeze(E,Env,L), S) =
     S[store <- state(E, S[env <- Env])[store][L <- eval(E, S[env <- Env])]] .
  eq state(need X, E, S) = S .
  eq  bind(need X, E, S, S') =
      S'[idx(X) <- unfreeze(E, S[env], nextLoc(S'[store]))] .
endfm

fmod CLOSURE is protecting PARAMETER-SEMANTICS .
  sort Closure .
  subsort Closure < Value .
  op closure : ParameterList Exp Environment -> Closure .
  op apply : Closure ExpList State -> Value .
  op state : Closure ExpList State -> State .
endfm

fmod STATIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env])) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env]))[store]] .
endfm

fmod DYNAMIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
```

```
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S))) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)))[store]] .
endfm

fmod PROC-SEMANTICS is protecting PROC-SYNTAX .
  protecting CALL-BY-VALUE-SEMANTICS .
  protecting CALL-BY-REFERENCE-SEMANTICS .
  protecting CALL-BY-NAME-SEMANTICS .
  protecting CALL-BY-NEED-SEMANTICS .
*** the next lets you choose between static vs. dynamic binding
  protecting  STATIC-BINDING .
---  protecting DYNAMIC-BINDING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .  var S : State .
  eq  eval(proc Pl E, S) = closure(Pl, E, S[env]) .
  eq state(proc Pl E, S) = S .
  eq  eval(E El, S) = apply(eval(E,S), El, state(E,S)) .
  eq state(E El, S) = state(eval(E,S), El, state(E,S)) .
endfm

red eval((proc(need x, value y) 0)) .
***> should be closure((need x,value y), 0, empty)

red eval((proc(name x, value y) 0) (2,3)) .
***> should be 0

red eval((proc(value x, reference y) (x(y))) (proc(value x) 2, 3)) .
***> should be 2

red eval((proc(value x, reference y) (x(y))) (proc(value x) y, 3),
        (  env, [idx(y), loc(0)])
        (store, {loc(1), [loc(0),int(1)]})) .
***> should be 1 under static scoping and 3 under dynamic scoping

fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  op mkDanglingBindings : BindingList Location State -> State .
  var X : Name .  var N : Nat .
  var Bl : BindingList .  var E : Exp .  vars S S' : State .
  eq mkDanglingBindings(none, loc(N), S) = S .
  eq mkDanglingBindings((X = E, Bl), loc(N), S) =
```

```
    mkDanglingBindings(Bl, loc(N + 1), S[idx(X) <- loc(N)]) .
 ceq eval(letrec Bl in E, S) = eval(E, bindBList (Bl,S') in stateBList(Bl,S'))
   if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
 ceq state(letrec Bl in E, S) =
    S[store <- state(E, bindBList (Bl, S') in stateBList(Bl, S'))[store]]
   if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
endfm

red eval(
  letrec x = 1
  in letrec x = 7, y = x
    in y
) .
***> should be undefined

fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm

red eval(set x = 3) .
***> should be 1

fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq  eval({E}, S) =  eval(E, S) .
  eq state({E}, S) = state(E, S) .
  eq  eval({El ; E}, S) =  eval(E, state({El}, S)) .
  eq state({El ; E}, S) = state(E, state({El}, S)) .
endfm

red eval({x ; y ; 3 ; x}) .
***> should be undefined

fmod LOOP-SEMANTICS is protecting LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  var S : State .
  eq  eval(while Be E, S) =
    if eval(Be,S) then  eval(while Be E, state(E,state(Be,S)))
    else int(1) fi .
```

```
  eq state(while Be E, S) =
     if eval(Be,S) then state(while Be E, state(E,state(Be,S)))
     else state(Be,S) fi .
endfm

fmod PROG-LANG-SEMANTICS is
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm


red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
```

```
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(value x, value y, value z) x * (y - z)
) .
***> should be closure((value x,value y,value z), x * (y - z), empty)

red eval(
  (proc(value y, value z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(value x, value y) x(y)) (proc(value z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(value x) x in x(x)
) .
***> should be closure(value x, x, empty)

red eval(
```

```
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(value x, value y) (x y)) (proc(value x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (value y, value z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(value x) x + a, a = 5
     in a * p(2)
) .
```

```
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(value n)
          if zero?(n)
          then 1
          else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(value n) n + n
  in let f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(value x) (p())
---            f = proc(value a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc(value 'maker, value x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
```

```
  letrec f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(value x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(value x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(value x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
```

```
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
) .
***> should be 2

red eval(
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
) .
***> should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
) .
***> should be 3

red eval(
  let f = let c = 0
          in proc()
             let d = set c = c + 1
             in c
  in f() + f()
) .
***> should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (value x)
             let d = set x = x + 1
```

```
            in x
    in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(value x, value y)
            let t = x
            in let d = set x = y
               in let d = set y = t
                  in 0
    in let d = f(x,y)
       in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
          if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, need b, need c)
          if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be 4



red eval(
  let x = 0
  in letrec
      'even = proc() if zero?(x)
                  then 1
                  else let d = set x = x - 1
                       in 'odd(),
      'odd  = proc() if zero?(x)
                  then 0
```

```
                  else let d = set x = x - 1
                       in 'even()
    in let d = set x = 7
       in 'odd()
) .
***> should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                  else let d = set x = x - 1
                       in 'odd(),
         'odd = proc() if zero?(x) then 0
                  else let d = set x = x - 1
                       in 'even()
  in 'odd()
) .
***> should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                     if zero?(x)
                     then 0
```

```
                    else 4 + 'times4(x - 1) ;
      'times4(3)
    }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(reference a, reference b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 4

red eval(
  let f = proc(need x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 8

red eval(
```

```
  let f = proc(name x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 11

red eval(
  let y = 5,
      f = proc(name x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 11

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185
```

```
************************************************
*** Defining a Functional Programming Language ***
************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op '(') : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse 3 + x .
parse 3 + 'variable1 .
parse 3 + 'variable2 + x * (y - z) + 'variable1 .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 equals 3 .
parse 3 equals 5 .
parse 5 equals x .
parse 3 + x equals 0 .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5) then 2 else 3 .
parse if zero?(0) then 2 else 3 .
parse if zero?(x) then y else z .
parse if x equals y
      then x
      else if x equals z
           then z
           else y .
```

```
fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let x = 5
  in x
 .
parse
  let x = 5, y = 7
  in x
 .
parse
  let x = 5
  in let y = x
     in y
 .
parse
  let x = 1
  in let x = 2
     in x
 .
parse
  let x = 10, y = 0, z = x
  in let a = 5, b = 7
         in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : CallingMode Name -> Parameter [prec 0] .
  op '(') : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(need x, value y) 0
 .
parse
  (proc(name x, value y) 0) (2,3)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) 2, 3)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) y, 3)
 .

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec x = 1
  in letrec x = 7, y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm

parse
  let x = 5, y = 7
  in x + y
 .
```

```
parse
  let x = 1
  in let x = x + 2
     in x + 1
 .
parse
  let x = 1
  in let y = x + 2
     in x + 1
 .
parse
  let x = 1
  in let z = let y = x + 4
             in y
     in z
 .
parse
  let x = 1
  in let x = let x = x + 4
             in x
     in x
 .
parse
  let x = 1
  in (x + (let x = 10 in x))
 .
parse
  proc(value x, value y, value z) x * (y - z)
 .
parse
  (proc(value y, value z) y + 5 * z) (1,2)
 .
parse
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
 .
parse
  (proc(value x, value y) x(y)) (proc(value z) 2 * z, 3)
 .
parse
  let x = proc(value x) x in x(x)
 .
parse
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
 .
parse
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
 .
parse
  let y = 1
  in (proc(value x, value y) (x y)) (proc(value x) y, 2)
 .
parse
  let x = 1
  in let x = 2,
         f = proc (value y, value z) y + x * z
     in f(1,x)
 .
parse
  let x = 1
```

```
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
 .
parse
  let a = 3
  in let p = proc(value x) x + a, a = 5
     in a * p(2)
 .
parse
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
 .
parse
  let f = proc(value n) n + n
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
 .
parse
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(value x) (p())
---            f = proc(value a) (p())
        in f(2)
 .
parse
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
 .
parse
  letrec f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
 .
parse
  letrec 'times4 = proc(value x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
 .
parse
  letrec 'even = proc(value x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(value x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
 .
```

```
parse
  let x = 1
  in letrec x = 7,
            y = x
     in y
 .
parse
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
 .
parse
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
 .
parse
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
 .
parse
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
 .
parse
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
 .
parse
  let x = 0
  in let f = proc (value x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
 .
parse
  let x = 0, y = 1
  in let f = proc(value x, value y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
 .
parse
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
 .
parse
  let x = 0, y = 3, z = 4,
      f = proc(value a, need b, need c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
 .
parse
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
 .

parse
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
 .
parse
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
 .
parse
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
 .
parse
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
 .
parse
  let x = 3, y = 4,
      f = proc(reference a, reference b)
            {
              set a = a + b ;
              set b = a - b ;
              set a = a - b
            }
  in {
       f(x,y) ;
       x
     }
 .
parse
```

```
  let f = proc(need x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
 .
parse
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
 .
parse
  let f = proc(name x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
 .
parse
  let y = 5,
      f = proc(name x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
 .
parse
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
 .


------------------
--- Semantics ---
------------------

fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
```

```
endfm

fmod ENVIRONMENT is
  extending GENERIC-STATE .
  protecting LOCATION .
  sorts Index Entry Environment .
  subsort Environment < StateAttribute .
  op env : -> StateAttributeName .
  subsort Entry < Environment .
  op empty : -> Environment .
  op [_,_] : Index Location -> Entry .
  op __ : Environment Environment -> Environment [assoc comm id: empty] .
  op _[_] : Environment Index -> Location .
  op _[_<-_] : Environment Index Location -> Environment .
  vars Ix Ix' : Index .  vars L L' : Location .  var Env : Environment .
  eq ([Ix,L] Env)[Ix] = L .
  eq ([Ix,L] Env)[Ix <- L'] = [Ix,L'] Env .
  eq Env[Ix <- L'] = [Ix,L'] Env [owise] .
endfm

fmod VALUE is protecting GENERIC-STATE .
  sorts Value PreValue .
  subsort Value < PreValue .
  op  eval : PreValue State -> Value .
  op state : PreValue State -> State .
  var V : Value .  var S : State .
  eq  eval(V, S) = V .
  eq state(V, S) = S .
endfm

fmod CELL is
  protecting LOCATION .
  protecting VALUE .
  sorts Cell Cells .
  subsort Cell < Cells .
  op noCells : -> Cells .
  op [_,_] : Location PreValue -> Cell .
  op __ : Cells Cells -> Cells [assoc comm id: noCells] .
  op _[_] : Cells Location -> PreValue .
  op _[_<*_] : Cells Location PreValue -> Cells .
  vars L L' : Location .  vars Pv Pv' : PreValue .  var Cs : Cells .
  eq ([L,Pv] Cs)[L] = Pv .
  eq ([L,Pv] Cs)[L <* Pv'] = [L,Pv'] Cs .
endfm

fmod STORE is
  extending GENERIC-STATE .
  protecting CELL .
  sort Store .
  subsort Store < StateAttribute .
  op {_,_} : Location Cells -> Store .
  op store : -> StateAttributeName .
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store Location PreValue -> Store .
  op nextLoc : Store -> Location .
  vars L Ln : Location .  var Cs : Cells .  var N : Nat .
  var Pv : PreValue .
  eq {Ln,Cs}[L] = Cs[L] .
  eq {loc(N),Cs}[loc(N) <- Pv] = {loc(N + 1), Cs[loc(N),Pv]} .
  eq {Ln,Cs}[L <- Pv] = {Ln,Cs[L <* Pv]} [owise] .
  eq nextLoc({Ln,Cs}) = Ln .
endfm

fmod STATE is
  protecting STORE .
  protecting ENVIRONMENT .
  op _[_] : State Index -> PreValue .
  op _[_<-_] : State Index Location -> State .
  op _[_<-_] : State Location PreValue -> State .
  op _[_<-_] : State Index PreValue -> State .
  op _[_<*_] : State Index PreValue -> State .
  var S : State .  var Ix : Index .  var L : Location .  var Pv : PreValue .
  eq S[Ix] = S[store][S[env][Ix]] .
  eq S[Ix <- L ] = S[env <- S[env][Ix <- L]] .
  eq S[ L <- Pv] = S[store <- S[store][L <- Pv]] .
  eq S[Ix <- Pv] = S[env <- S[env][Ix <- nextLoc(S[store])]]
                   [store <- S[store][nextLoc(S[store]) <- Pv]] .
  eq S[Ix <* Pv] = S[store <- S[store][S[env][Ix] <- Pv]] .
endfm

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op  eval : Name State -> Value .
  op state : Name State -> State .
  var X : Name .  var S : State .
  eq  eval(X, S) =  eval(S[idx(X)], S) .
  eq state(X, S) = state(S[idx(X)], S) .
endfm

fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> Value .
  op state : Exp State -> State .
  op  eval : Exp -> Value .
  var I : Int .  var S : State .  vars E E' : Exp .  var El : ExpList .
  eq  eval(I, S) = int(I) .
  eq state(I, S) = S .
  eq eval(E) = eval(E, (env,empty)(store,{loc(0),noCells})) .
endfm

red eval(3) .
***> should be 3
red eval(x) .
***> should be undefined
red eval('variable) .
***> should be undefined
red eval('variable,
           (  env, [idx(x),loc(0)] [idx('variable),loc(1)])
           (store, {loc(2), [loc(0),int(5)] [loc(1),int(4)]})) .
***> should be 4

fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .
  ops add sub mul div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I), int(I')) = int(I + I') .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
  eq sub(int(I), int(I')) = int(I - I') .
  eq state(E - E', S) = state(E', state(E,S)) .
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I), int(I')) = int(I * I') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I), int(I')) = int(I quo I') .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm

red eval(3 + x, (  env, [idx(x),loc(0)] [idx('n),loc(1)])
               (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
***> should be 8
red eval(3 + 'variable1,
           (  env, [idx(x),loc(0)] [idx('variable1),loc(1)])
           (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
***> should be 7
red eval(3 + 'variable2 + x * (y - z) + 'variable1 ,
         (  env, [idx('variable2),loc(0)]
                 [idx(x),loc(1)]
                 [idx(y),loc(2)]
                 [idx(z),loc(3)]
                 [idx('variable1),loc(4)])
         (store, {loc(4), [loc(0), int(0)]
                          [loc(1), int(1)]
                          [loc(2), int(2)]
                          [loc(3), int(3)]
                          [loc(4), int(4)]})) .
***> should be 6

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .  var I : Int .
  eq  eval(E equals E', S) = eval(E, S) == eval(E', state(E, S)) .
  eq state(E equals E', S) = state(E', state(E, S)) .
  eq  eval(zero?(E), S) = eval(E, S) == int(0) .
  eq state(zero?(E), S) = state(E, S) .
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = evenIntValue?(eval(E, S)) .
  eq evenIntValue?(int(I)) = I rem 2 == 0 .
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) = eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm

red eval(3 equals 3, S) .
***> should be true
red eval(3 equals 5, S) .
***> should be false
red eval(5 equals x, S) .
***> should be false
red eval(3 + x equals 0, S) .
***> should be undefined

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  var S : State .
  eq  eval(if Be then E else E', S) = if eval(Be, S)
       then  eval(E, state(Be, S)) else  eval(E', state(Be, S)) fi .
  eq state(if Be then E else E', S) = if eval(Be, S)
       then state(E, state(Be, S)) else state(E', state(Be, S)) fi .
endfm

red eval(if zero?(5) then 2 else 3, S) .
***> should be 3
red eval(if zero?(0) then 2 else 3, S) .
***> should be 2
red eval(if zero?(x) then y else z,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15)][loc(10),int(3)][loc(12),int(5)]})) .
***> should be 5
red eval(
  if x equals y
  then x
  else if x equals z
       then z
       else y,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15)][loc(10),int(3)][loc(12),int(5)]})) .
***> should be 3

fmod BINDINGS-SEMANTICS is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op stateBList : BindingList State -> State .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq stateBList(none, S) = S .
  eq stateBList((X = E, Bl), S) = stateBList(Bl, state(E, S)) .
  eq bindBList (none,S) in S' = S' .
  eq bindBList ((X = E, Bl), S) in S' =
     bindBList (Bl, state(E,S)) in (S'[idx(X) <- eval(E,S)]) .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq  eval(let Bl in E, S) =  eval(E, bindBList (Bl, S) in stateBList(Bl,S)) .
  eq state(let Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S) in stateBList(Bl,S))[store]] .
endfm

red eval(
  let x = 5
  in x
) .
***> should be 5
red eval(
  let x = 5, y = 7
  in x
) .
***> should be 5
red eval(
  let x = 5
  in let y = x
     in y
) .
***> should be 5
red eval(
  let x = 1
  in let x = 2
     in x
) .
***> should be 2
red eval(
  let x = 10, y = 0, z = x
  in let a = 5, b = 7
         in z
) .
***> should be undefined

fmod PARAMETER-SEMANTICS is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op state : Parameter Exp State -> State .
  op statePList : ParameterList ExpList State -> State .
  op bind : Parameter Exp State State -> State .
  op bindPList : ParameterList ExpList State State -> State .
  vars S S' : State .  var P : Parameter .  var Pl : ParameterList .
  var E : Exp .  var El : ExpList .
```

```
  eq statePList((), (), S) = S .
  eq statePList((P,Pl), (E,El), S) =
     statePList(Pl, El, state(P, E, S)) [owise] .
  eq bindPList((), (), S, S') = S' .
  eq bindPList((P,Pl), (E,El), S, S') =
     bindPList(Pl, El, state(P,E,S), bind(P,E,S,S')) .
endfm

fmod CALL-BY-VALUE-SEMANTICS is extending CALL-BY-VALUE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  var X : Name .  var E : Exp .  vars S S' : State .
  eq state(value X, E, S) = state(E, S) .
  eq  bind(value X, E, S, S') = S'[idx(X) <- eval(E, S)] .
endfm

fmod CALL-BY-REFERENCE-SEMANTICS is extending CALL-BY-REFERENCE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  vars X Y : Name .  var E : Exp .  vars S S' : State .
  eq state(reference X, Y, S) = S .
  eq state(reference X, E, S) = state(E, S) [owise] .
  eq  bind(reference X, Y, S, S') = S'[idx(X) <- S[env][idx(Y)]] .
  eq  bind(reference X, E, S, S') = S'[idx(X) <- eval(E, S)] [owise] .
endfm

fmod CALL-BY-NAME-SEMANTICS is extending CALL-BY-NAME-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op frozen : Exp Environment -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  eq  eval(frozen(E, Env), S) = eval(E, S[env <- Env]) .
  eq state(frozen(E, Env), S) = S[store <- state(E, S[env <- Env])[store]] .
  eq state(name X, E, S) = S .
  eq  bind(name X, E, S, S') = S'[idx(X) <- frozen(E, S[env])] .
endfm

fmod CALL-BY-NEED-SEMANTICS is extending CALL-BY-NEED-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op unfreeze : Exp Environment Location -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  var L : Location .
  eq  eval(unfreeze(E,Env,L), S) = eval(E, S[env <- Env]) .
  eq state(unfreeze(E,Env,L), S) =
     S[store <- state(E, S[env <- Env])[store][L <- eval(E, S[env <- Env])]] .
  eq state(need X, E, S) = S .
  eq  bind(need X, E, S, S') =
     S'[idx(X) <- unfreeze(E, S[env], nextLoc(S'[store]))] .
endfm

fmod CLOSURE is protecting PARAMETER-SEMANTICS .
  sort Closure .
  subsort Closure < Value .
  op closure : ParameterList Exp Environment -> Closure .
  op apply : Closure ExpList State -> Value .
  op state : Closure ExpList State -> State .
endfm

fmod STATIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env])) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env]))[store]] .
endfm

fmod DYNAMIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S))) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)))[store]] .
endfm

fmod PROC-SEMANTICS is protecting PROC-SYNTAX .
  protecting CALL-BY-VALUE-SEMANTICS .
  protecting CALL-BY-REFERENCE-SEMANTICS .
  protecting CALL-BY-NAME-SEMANTICS .
  protecting CALL-BY-NEED-SEMANTICS .
*** the next lets you choose between static vs. dynamic binding
  protecting  STATIC-BINDING .
---  protecting DYNAMIC-BINDING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .  var S : State .
  eq  eval(proc Pl E, S) = closure(Pl, E, S[env]) .
  eq state(proc Pl E, S) = S .
  eq  eval(E El, S) = apply(eval(E,S), El, state(E,S)) .
  eq state(E El, S) = state(eval(E,S), El, state(E,S)) .
endfm

red eval((proc(need x, value y) 0)) .
***> should be closure((need x,value y), 0, empty)

red eval((proc(name x, value y) 0) (2,3)) .
***> should be 0

red eval((proc(value x, reference y) (x(y))) (proc(value x) 2, 3)) .
***> should be 2

red eval((proc(value x, reference y) (x(y))) (proc(value x) y, 3),
         (  env, [idx(y), loc(0)])
         (store, {loc(1), [loc(0),int(1)]})) .
***> should be 1 under static scoping and 3 under dynamic scoping

fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  op mkDanglingBindings : BindingList Location State -> State .
  var X : Name .  var N : Nat .
  var Bl : BindingList .  var E : Exp .  vars S S' : State .
  eq mkDanglingBindings(none, loc(N), S) = S .
  eq mkDanglingBindings((X = E, Bl), loc(N), S) =
     mkDanglingBindings(Bl, loc(N + 1), S[idx(X) <- loc(N)]) .
 ceq eval(letrec Bl in E, S) = eval(E, bindBList (Bl,S') in stateBList(Bl,S'))
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
 ceq state(letrec Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S') in stateBList(Bl, S'))[store]]
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
endfm

red eval(
  letrec x = 1
  in letrec x = 7, y = x
     in y
) .
***> should be undefined

fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm

red eval(set x = 3) .
***> should be 1

fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq  eval({E}, S) =  eval(E, S) .
  eq state({E}, S) = state(E, S) .
  eq  eval({El ; E}, S) =  eval(E, state({El}, S)) .
  eq state({El ; E}, S) = state(E, state({El}, S)) .
endfm

red eval({x ; y ; 3 ; x}) .
***> should be undefined

fmod LOOP-SEMANTICS is protecting LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  var S : State .
  eq  eval(while Be E, S) =
     if eval(Be,S) then  eval(while Be E, state(E,state(Be,S)))
     else int(1) fi .
  eq state(while Be E, S) =
     if eval(Be,S) then state(while Be E, state(E,state(Be,S)))
     else state(Be,S) fi .
endfm

fmod PROG-LANG-SEMANTICS is
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm


red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(value x, value y, value z) x * (y - z)
) .
***> should be closure((value x,value y,value z), x * (y - z), empty)

red eval(
   (proc(value y, value z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(value x, value y) x(y)) (proc(value z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(value x) x in x(x)
) .
***> should be closure(value x, x, empty)

red eval(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(value x, value y) (x y)) (proc(value x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (value y, value z) y + x * z
```

```
    in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(value x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(value n) n + n
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(value x) (p())
---            f = proc(value a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(value x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(value x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(value x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***> should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***> should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***> should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***> should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (value x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(value x, value y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, need b, need c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be 4


red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***> should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***> should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(reference a, reference b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 4

red eval(
  let f = proc(need x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8
```

```
red eval(
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 8

red eval(
  let f = proc(name x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 11

red eval(
  let y = 5,
      f = proc(name x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 11

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185
```

CS322 - Programming Language Design

Lecture 7: Designing a Functional Language - Semantics; Part 1

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Once the syntax of a programming language is defined, one can start defining its semantics. Our general philosophy is to define the semantics one a feature-by-feature basis in a style which is as modular as possible, so that it gives us the flexibility to extend the language easily by adding new features. In fact, your second homework assignment will consist of various extensions of our functional programming language.

We will start by defining the important notion of state, first generically and then instantiated to our concrete setting consisting of an environment and a store, and then add semantics for each module defining syntax. Some auxiliary modules are needed occasionally, adding possible semantical choices to the already defined syntax, such as static scoping or dynamic scoping.

Generic State

For the simple imperative language that we defined in Lecture 4, a state containing just pairs associating values to names was sufficient in order to define its semantics. In the case of the functional programming language described in Lecture 5 whose syntax was given in Lecture 6, we have seen that the state should be refined into environment and store.

Since there is a high chance that even more state attributes will be needed in future extensions of the languages, such as ones providing support for data structures and (dynamic) types, to keep our design flexible and expandable, we start by formalizing the informal notion of *state* generically, as a set of pairs of *state attribute names* and *state attributes*. Intuitively, a state attribute can be anything that will ever need to be stored in a state, while a state attribute name allows one to refer to a particular state attribute.

So we first define a state as a set of pairs (the multiple occurrences will be avoided by construction as seen later):

```
fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
```

Then we define state attribute *lookup* and *update* operators, making use of Maude's AC matching:

```
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm
```

Locations

The functional programming language that we wish to define has side effects. That means that the value associated to a given name can change during the evaluation of an expression.

Additionally, as we saw in Lecture 5, under static analysis a function is evaluated to a closure, saying where each name occurring free in that function's body is located, in order to be able to evaluate the body of that function when it is subsequently invoked. Due to side effects, the values associated to names occurring free in functions can change.

This is true in almost any programming language. Think for example of a function `f` in C which refers to a global variable `x` which is assigned the value `0` when `f` is defined, and assume that another function is called before `f` which changes the value of `x` to `1`. Then the value `f` sees associated to `x` is `1`.

The simplest and cleanest way to define and also to implement side effects is to introduce the notion of *location*, which, at this moment, is just an abstract data structure able to store any possible value. Names will be then bound to locations instead of values, so if one wants to change the value associated to a name one just changes the value stored at that name's location. Thus the closure associated to a function needs only to freeze the locations of its free names.

In order to keep the range of possible implementations of our programming language broad, we simply assume that a location can be associated to any natural number:

```
fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm
```

Environments

In the context of our functional programming language, environments are nothing but maps of names to locations. However, environments can be defined generically, for any programming language.

Since some languages prefer the terminology "variable" instead of "name", in order to be general and language-independent we prefer to define environments as mapping *indexes* into locations. If one needs environments later on in order to define a particular programming language, then one just has to declare an appropriate map from names or variables, or whatever else, into indexes. Therefore, indexes provide an *abstract interface* to access the information in an environment.

One simple way to represent such maps is by flattening them into pairs `[<Index>,<Location>]`, which we call *entries*:

```
fmod ENVIRONMENT is
  extending GENERIC-STATE .
  protecting LOCATION .
  sorts Index Entry Environment .
  subsort Environment < StateAttribute .
  op env : -> StateAttributeName .
  subsort Entry < Environment .
  op empty : -> Environment .
  op [_,_] : Index Location -> Entry .
  op __ : Environment Environment -> Environment [assoc comm id: empty] .
```

The environment is an important attribute of a state, to which many further definitions will refer. For that reason, we defined it as a subsort of `StateAttribute` and also associated it an attribute name, `env`, to allow one to easily extract the environment part

from a state. We will ensure via appropriate equations that an environment contains no multiple entries for the same index.

```
  op _[_] : Environment Index -> Location .
  op _[_<-_] : Environment Index Location -> Environment .
  vars Ix Ix' : Index .  vars L L' : Location .  var Env : Environment .
  eq ([Ix,L] Env)[Ix] = L .
  eq ([Ix,L] Env)[Ix <- L'] = [Ix,L'] Env .
  eq Env[Ix <- L'] = [Ix,L'] Env [owise] .
endfm
```

There can be implementations in which the maps from variables or names into indexes are stored in efficient data structures, such as hash tables. Then, in order to find the location of a name one first looks up into the hash table to find the index associated to that name, and then into the environment to find its location.

By *specifying* rather than *implementing (an interpreter of)* a language, we allow all correct implementations *by design*.

Values

Values are stored in locations. However, it is not entirely clear apriori what a value is. In the examples that we have seen so far, values can be integers, closures (values to which functions evaluate), frozen expressions which unfreeze when are first encountered during the evaluation of an expression (call-by-need parameters), or even expressions which are frozen forever (call-by-name).

Therefore, in order to have a design which is as general and modular as possible, we define a generic sort `Value`, which is a subsort of another generic sort called `PreValue`. The sort `PreValue` is intended for those values which need to be stored in locations but which need to be further evaluated in order to become concrete values. How and when these pre-values are evaluated is left open by design, so each new feature added to the language can come with its own decisions.

```
fmod VALUE is protecting GENERIC-STATE .
  sorts Value PreValue .
  subsort Value < PreValue .
  op  eval : PreValue State -> Value .
  op state : PreValue State -> State .
  var V : Value .  var S : State .
  eq  eval(V, S) = V .
  eq state(V, S) = S .
endfm
```

`eval : PreValue State -> Value` evaluates a pre-value to a value. However, evaluating an expression passed to a function in a call-by-need or call-by-name style can have side effects in the calling environment. Therefore, evaluating a pre-value can have side effects! For that reason, we define another operation `state : PreValue State -> State` which collects all the side effects generated by evaluating a pre-value in a given state. Both these operations behave as expected on concrete values.

Stores

Stores historically map locations to values. However, due to the nature of our values, our stores will map locations to pre-values. We realize this by making use of *cells*, which are simple data structures storing a pair `[<Location>,<PreValue>]`:

```
fmod CELL is
  protecting LOCATION .
  protecting VALUE .
  sorts Cell Cells .
  subsort Cell < Cells .
  op noCells : -> Cells .
  op [_,_] : Location PreValue -> Cell .
  op __ : Cells Cells -> Cells [assoc comm id: noCells] .
```

The usual lookup and update operations are defined below. It is important to notice, however, that the update operation, written

`_[_<*_] : Cells Location PreValue -> Cells`, updates a location *only if it already exists in some cell*:

```
  op _[_] : Cells Location -> PreValue .
  op _[_<*_] : Cells Location PreValue -> Cells .
  vars L L' : Location .  vars Pv Pv' : PreValue .  var Cs : Cells .
  eq ([L,Pv] Cs)[L] = Pv .
  eq ([L,Pv] Cs)[L <* Pv'] = [L,Pv'] Cs .
endfm
```

The update operation is left undefined when the updated name is not already in some cell. This behavior is very important when defining the semantics of variable assignment, because a variable can be updated only if it has already been declared.

As a convention, from now on we use the notation `_[_<-_]` for updates in which the updated entities may have not been updated before, in which case new pairs are created, and `_[_<*_]` for updates which change an already existing update.

We can now define stores. In addition to cells, a store provides control on how locations are allocated to (pre-)values. In particular, a store maintains control over the *next available location* and provides an interface operation, called `nextLoc`, which other importing modules can use in order to define the semantics of corresponding language constructs:

```
fmod STORE is
  extending GENERIC-STATE .
  protecting CELL .
  sort Store .
  op {_,_} : Location Cells -> Store .
  op nextLoc : Store -> Location .
```

Stores are crucial state attributes, so we declare them accordingly, together with a corresponding state attribute name:

```
  subsort Store < StateAttribute .
  op store : -> StateAttributeName .
```

The lookup and update operators are declared next, as well as the meaning of all the operators. Notice that the update operation is defined in such a way that the store has full control over the allocated locations. If a value needs to be allocated a new location, then the update must refer to exactly the next available location; importing modules can find that location using `nextLoc`. Otherwise the update must refer to an already allocated location:

```
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store Location PreValue -> Store .
  vars L Ln : Location .  var Cs : Cells .  var N : Nat .
  var Pv : PreValue .
  eq {Ln,Cs}[L] = Cs[L] .
  eq {loc(N),Cs}[loc(N) <- Pv] = {loc(N + 1), Cs[loc(N),Pv]} .
  eq {Ln,Cs}[L <- Pv] = {Ln,Cs[L <* Pv]} [owise] .
  eq nextLoc({Ln,Cs}) = Ln .
endfm
```


States

When defining a programming language, a *state* can be viewed as an abstract data structure holding all the information needed in order to state the meaning of *any* programming language construct.

For our particular language, a state only needs to contain one environment and one store. However, states are also defined generically, so they can be used for any programming language. An important methodological convention that we will follow whenever we define a programming language, says that the auxiliary operators needed to define particular language constructs should be defined directly *on states* as much as it is possible and reasonable, so that the particular representation of the state is *encapsulated*.

This important concept of *information hiding* is good to also follow in software development and engineering. It facilitates data encapsulation and changing data representations without having to

go through the entire code to make appropriate changes.

```
fmod STATE is
  protecting STORE .
  protecting ENVIRONMENT .
  op _[_] : State Index -> PreValue .
  op _[_<-_] : State Index Location -> State .
  op _[_<-_] : State Location PreValue -> State .
  op _[_<-_] : State Index PreValue -> State .
  op _[_<*_] : State Index PreValue -> State .
```

The above form the *interface* of STATE. The first operation is the usual lookup, which returns the (pre-)value associated to an index. This is frequently used in the definition of any programming language. Notice, however, that in order to define it one should lookup first in the environment and then in the store of a state.

The other four operations are update operators. The first three may require to allocate new space in the state, while the fourth just

changes an already exiting value. The first allocates a new entry in the environment binding an index to a location, the second associates a location to a (pre-)value and may potentially require allocating new space, the third creates a new association of an index to a (pre-)value by first creating a new binding in the environment of that index to a *new* location and then storing the (pre-)value at that location, and the fourth just changes the value associated to an index by first looking up the location of that index and then making the change in the store. Formally,

```
  var S : State .  var Ix : Index .  var L : Location .  var Pv : PreValue .
  eq S[Ix] = S[store][S[env][Ix]] .
  eq S[Ix <- L ] = S[env <- S[env][Ix <- L]] .
  eq S[ L <- Pv] = S[store <- S[store][L <- Pv]] .
  eq S[Ix <- Pv] = S[env <- S[env][Ix <- nextLoc(S[store])]]
                    [store <- S[store][nextLoc(S[store]) <- Pv]] .
  eq S[Ix <* Pv] = S[store <- S[store][S[env][Ix] <- Pv]] .
endfm
```

Semantics of Names

We now have all the infrastructure needed in order to start defining the (executable) semantics our programming language. We do it inductively, over the structure of all the language constructs, so we start with names.

What is the meaning of names in our language? Names refer to values, so in a given state they mean a certain value. In order to use the state infrastructure, we first need to define an operator that takes names to indexes:

```
fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
```

We already know that in our language, due to call-by-need and call-by-name parameter passing, names can carry not only values

but also expressions which, once evaluated, can change the state. Therefore, in order to define the meaning of a name in a state, we introduce two new operators and define them as follows:

```
  op  eval : Name State -> Value .
  op state : Name State -> State .
  var X : Name .  var S : State .
  eq  eval(X, S) =  eval(S[idx(X)], S) .
  eq state(X, S) = state(S[idx(X)], S) .
endfm
```

Therefore, the value of a name in a state is calculated by evaluating the pre-value associated to the index of that name in the state. The `eval` operator in the right-hand-side term comes from the module `VALUE`. The operator `state` collects the possible side effects of evaluating the pre-value. Note that the right-hand-side terms evaluate to `V` and `S` if the pre-value of `X` in `S` is a concrete value `V`.

Semantics of Generic Expressions

The meaning of an expression in a state is the value it represents if evaluated in that state. However, due to side effects, its evaluation can change the state. Therefore, we define two operators, one for evaluating the expression and one for collecting the side effects:

```
fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> Value .
  op state : Exp State -> State .
  op  eval : Exp -> Value .
  var I : Int .  var S : State .  vars E E' : Exp .  var El : ExpList .
  eq  eval(I, S) = int(I) .
  eq state(I, S) = S .
  eq eval(E) = eval(E, (env,empty)(store,{loc(0),noCells})) .
endfm
```


Since integers are a special case of expressions, we have to give their meaning: they evaluate to themselves and have no side effects. This is done formally in the module above.

Notice that, in order to keep integers disconnected from other values, we preferred to *wrap* them using the construct `int : Int -> Value`. One could have also defined `Int` as a subsort of `Value`, but doing so is not a good idea because, due to connected component collapsing in parsing, operations on integers may collide with operations of same name defined on other values.

A unary operator `eval : Exp -> Value` was also defined, which is reduced to the binary one with an "initial" state. The initial state contains an empty environment and a store with no cells but ready to allocate location `loc(0)`.

One can now evaluate expressions, either in the initial state or in hand-crafted artificial states. When we predict the value of

expression we forget the wrapper `int`:

```
red eval(3) .           ***> should be 3
red eval(x) .           ***> should be undefined
red eval('variable) .   ***> should be undefined
red eval('variable,
            (  env, [idx(x),loc(0)] [idx('variable),loc(1)])
            (store, {loc(2), [loc(0),int(5)] [loc(1),int(4)]})) .
***> should be 4
```

The last expression, which is just a name, was evaluated in a state whose environment contains an index of `x` pointing to location `loc(0)` and an index of `'variable` pointing to location `loc(1)`, and whose store contains the value `int(5)` at location `loc(0)` and `int(4)` at location `loc(1)`, so the result is `4`.

Semantics of Arithmetic Operators

The meaning of all the arithmetic operators in our language is defined below. Because of side effects, both `eval` and `state` need to be defined. Notice that, in order to correctly propagate the side effects, `eval` evaluates the second expression in the state obtained after evaluating the first one, and `state` sequentializes the side effects of the two expressions:

```
fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .
  ops add sub mul div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I), int(I')) = int(I + I') .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
```

```
  eq sub(int(I), int(I')) = int(I - I') .
  eq state(E - E', S) = state(E', state(E,S)) .
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I), int(I')) = int(I * I') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I), int(I')) = int(I quo I') .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm
```

Since integers are wrapped, corresponding operators on values need to be declared and defined as above. One can now evaluate expressions containing arithmetic operators:

```
red eval(3 + x, (  env, [idx(x),loc(0)] [idx('n),loc(1)])
                (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
red eval(3 + 'variable1,
           (  env, [idx(x),loc(0)] [idx('variable1),loc(1)])
           (store, {loc(100), [loc(0),int(5)][loc(1),int(4)]})) .
```


The states used in these reductions, as well as the one below are artificial, in the sense that they cannot be obtained from the initial state by evaluating expressions. Nevertheless, such states are quite useful in debugging specifications.

```
red eval(3 + 'variable2 + x * (y - z) + 'variable1 ,
         (  env, [idx('variable2),loc(0)]
                 [idx(x),loc(1)]
                 [idx(y),loc(2)]
                 [idx(z),loc(3)]
                 [idx('variable1),loc(4)])
         (store, {loc(4), [loc(0), int(0)]
                          [loc(1), int(1)]
                          [loc(2), int(2)]
                          [loc(3), int(3)]
                          [loc(4), int(4)]})) .
```

Exercise 1 *Say why the states above cannot be obtained during normal evaluations, starting with the initial state.*

Semantics of Boolean Expressions

One needs to define similar `eval` and `state` operators on boolean expressions, because they can involve expressions which need to be evaluated and whose side effects need to be propagated. The following module is straightforward:

```
fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .  var I : Int .
  eq  eval(E equals E', S) = eval(E, S) == eval(E', state(E, S)) .
  eq state(E equals E', S) = state(E', state(E, S)) .
  eq  eval(zero?(E), S) = eval(E, S) == int(0) .
  eq state(zero?(E), S) = state(E, S) .
```



```
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = evenIntValue?(eval(E, S)) .
  eq evenIntValue?(int(I)) = I rem 2 == 0 .
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) = eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm
red eval(3 equals 3, S) .      ***> should be true
red eval(3 equals 5, S) .      ***> should be false
red eval(5 equals x, S) .      ***> should be false
red eval(3 + x equals 0, S) .  ***> should be undefined
```

Homework Exercise 1 *The third reduction above should be also undefined. Explain why the result above,* `false`*, is wrong and give an example of an expression which should normally be undefined but it actually wrongly evaluates to an integer value. Fix this bug.*

CS322 - Programming Language Design

Lecture 8: Designing a Functional Language - Semantics; Part 2

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Semantics of Conditionals

Conditionals can be now easily defined as follows:

```
fmod IF-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  var S : State .
  eq  eval(if Be then E else E', S) = if eval(Be, S)
       then  eval(E, state(Be, S)) else  eval(E', state(Be, S)) fi .
  eq state(if Be then E else E', S) = if eval(Be, S)
       then state(E, state(Be, S)) else state(E', state(Be, S)) fi .
endfm
```

Notice how side effects are propagated in the definition above.

```
red eval(if zero?(x) then y else z,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15)][loc(10),int(3)][loc(12),int(5)]})) .
***> should be 5
```

Semantics of Bindings

Bindings are used by both `let` and `letrec`, but with a different meaning. However, they both need to evaluate all the right-hand-side expressions in some order in a certain state, and for simplicity we assume *sequential order*, so we define `stateBList : BindingList State -> State` which does exactly that.

Moreover, both `let` and `letrec` bind all their names *in parallel* to values obtained after evaluating the corresponding expressions in a certain state. We therefore define the operation `bindBList(_,_)in_ : BindingList State State -> State` which binds each name in the `BindingList` to the corresponding expression evaluated in the first state; the actual binding is created into the second state, to accommodate both `let` and `letrec`:


```
fmod BINDINGS-SEMANTICS is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op stateBList : BindingList State -> State .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .
  var E : Exp .  var Bl : BindingList .
  eq stateBList(none, S) = S .
  eq stateBList((X = E, Bl), S) = stateBList(Bl, state(E, S)) .
  eq bindBList (none,S) in S' = S' .
  eq bindBList ((X = E, Bl), S) in S' =
     bindBList (Bl, state(E,S)) in (S'[idx(X) <- eval(E,S)]) .
endfm
```

Exercise 1 *What happens if one replaces* `state(E,S)` *by just* `S` *in the last equation in* `BINDINGS-SEMANTICS`*? Explain why this is not a fortunate programming language design decision.*

Semantics of Let

The `let <BindingList> in <Exp>` construct can be immediately defined now, by evaluating the expression `<Exp>` in the state obtained after binding all the names to their corresponding values. However, possible side effects must be propagated correctly:

```
fmod LET-SEMANTICS is extending LET-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq eval(let Bl in E, S) =
     eval(E, bindBList (Bl, S) in stateBList(Bl,S)) .
  eq state(let Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S) in stateBList(Bl,S))[store]] .
endfm
```

Notice that the state *after* evaluating the `let` construct forgets all

the bindings that were created in order to evaluate `<Exp>`, but it carries the side effects generated by evaluating the `let`!

```
red eval(let x = 5 in x) .                 ***> should be 5
red eval(let x = 5, y = 7 in x) .          ***> should be 5
red eval(let x = 5 in let y = x in y) .    ***> should be 5
red eval(let x = 1 in let x = 2 in x) .    ***> should be 2
red eval(
  let x = 10, y = 0, z = x
  in let a = 5, b = 7
         in z
) .                                        ***> should be undefined
```

Homework Exercise 1 *Some languages, including* Scheme, *provide a language construct* `let* <BindingList> in <Exp>` *which binds its names* sequentially, *rather than in parallel. For example,* `let* x = 1, y = x + 1 in y` *evaluates to* `2`. *Define* `let*` *and give 5 examples showing the difference between* `let` *and* `let*`.

Semantics of Functions

There are several possibilities which have to be considered when one defines the semantics of functions and function invocations, such as the various *parameter passing styles* and *static versus dynamic scoping*. In order to be flexible in how we design functional programming languages, we have to carefully accommodate all these possibilities so that they *do not conflict* with each other.

There are, however, features that *exclude each other* by definition, such as static versus dynamic scoping. One should not expect to accommodate conflicting features within the same language, but one should be able to put non-conflicting features together easily.

We next consider each function related language construct defined in Lecture 6 and give it appropriate meaning. It is important to

understand that, like one can write different programs implementing the same specification, one can also have different ways to define the same semantics. Our choices below are based on experience with defining programming languages and they aim at as much flexibility as possible in case other language extensions will be desired in the future.

Semantics of Parameters

Without having yet defined the meaning of a function call, we can *anticipate* at this moment that the parameters of a function need to be bound to concrete arguments when a function is called.

We must design our definitions to accommodate the already known fact that different parameters can have different passing styles. What is the exact meaning of a parameter passing? What effect does an argument instantiation have? There are two important

aspects that need to be considered:

1. The *expression passed as an argument may have side effects* in the calling environment. However, the expression may not be evaluated when passed due to the parameter passing style of the corresponding parameter, in which case the side effects are not immediately propagated. In order to properly handle the side effects of arguments, we consider an operation `state : Parameter Exp State -> State`, which, depending on the style of the `<Parameter>`, propagates or not the side effects of `<Exp>`.
2. A *binding of the parameter needs to be generated*, so we need an operator of the form `bind : Parameter Exp State -> State`, where the result state adds to the argument state the binding of the `<Parameter>` to the (pre-)value or the reference of `<Exp>`, depending on the style of the parameter passing. However, there is an important conceptual difference between

   state update and parameter binding when functions are invoked. The state is updated *sequentially* as each argument expression is evaluated, while the bindings are done *in parallel*. That means that we will need to actually define the operation `bind` as `bind : Parameter Exp State State -> State`, where the first `<State>` is the current state at which the corresponding binding should take place, while the second `<State>` contains all the bindings accumulated up to the current parameter. The result `<State>` adds the current binding to the second `<State>` argument of `bind`.

The two operations above, `state` and `bind`, need to be defined for each particular parameter passing style. This will be done when we define the semantics of each parameter passing style. However, we are able at this moment, without any parameter style added to the language, to define generically the meaning of the entire list of parameters. As explained before, the idea is to propagate the side

effects sequentially and the bindings in parallel. We introduce operations `statePList` and `bindPList` that do that:

```
fmod PARAMETER-SEMANTICS is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op state : Parameter Exp State -> State .
  op statePList : ParameterList ExpList State -> State .
  op bind : Parameter Exp State State -> State .
  op bindPList : ParameterList ExpList State State -> State .
  vars S S' : State .  var P : Parameter .  var Pl : ParameterList .
  var E : Exp .  var El : ExpList .
  eq statePList((), (), S) = S .
  eq statePList((P,Pl), (E,El), S) =
     statePList(Pl, El, state(P, E, S)) [owise] .
  eq bindPList((), (), S, S') = S' .
  eq bindPList((P,Pl), (E,El), S, S') =
     bindPList(Pl, El, state(P,E,S), bind(P,E,S,S')) .
endfm
```


The side effects of evaluating the argument expressions are propagated sequentially when the bindings are generated.

Semantics of Call-by-Value

Under call-by-value, the argument expression is simply evaluated, all its side effects are properly propagated in the calling environment, the the binding of parameter to the value obtained after evaluating the expression is generated. The below module specifies *exactly* this meaning formally:

```
fmod CALL-BY-VALUE-SEMANTICS is extending CALL-BY-VALUE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  var X : Name .  var E : Exp .  vars S S' : State .
  eq state(value X, E, S) = state(E, S) .
  eq  bind(value X, E, S, S') = S'[idx(X) <- eval(E, S)] .
endfm
```

Semantics of Call-by-Reference

Call-by-reference makes sense only if the corresponding argument expression is a name. Otherwise one can either generate a runtime error or one can take some default decision. So one should adopt a language design decision here. We prefer to just replace the parameter style to call-by-value if the argument is not a name:

```
fmod CALL-BY-REFERENCE-SEMANTICS is extending CALL-BY-REFERENCE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  vars X Y : Name .  var E : Exp .  vars S S' : State .
  eq state(reference X, Y, S) = S .
  eq state(reference X, E, S) = state(E, S) [owise] .
  eq  bind(reference X, Y, S, S') = S'[idx(X) <- S[env][idx(Y)]] .
  eq  bind(reference X, E, S, S') = S'[idx(X) <- eval(E, S)] [owise] .
endfm
```


Semantics of Call-by-Name

Under call-by-name, the corresponding argument expression is frozen and evaluated each time the parameter needs to be evaluated in the function's body. The crucial aspect here is that the expression is evaluated in the *environment in which the expression has been passed to the function*, and not in the environment in which the function was declared. Even though we have not yet defined the semantics of function calls, we know enough to realize that we need to define a new `<PreValue>` here, which stores the frozen expression together with its environment.

Due to the modular design that we followed when we defined the module `VALUE` because we anticipated such non-standard values that may eventually need to be stored, we can simply and elegantly

solve this problem now by just adding an operator `frozen : Exp Environment -> PreValue` together with its meaning:

```
fmod CALL-BY-NAME-SEMANTICS is extending CALL-BY-NAME-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op frozen : Exp Environment -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  eq  eval(frozen(E, Env), S) = eval(E, S[env <- Env]) .
  eq state(frozen(E, Env), S) = S[store <- state(E, S[env <- Env])[store]] .
  eq state(name X, E, S) = S .
  eq  bind(name X, E, S, S') = S'[idx(X) <- frozen(E, S[env])] .
endfm
```

Note that under call-by-name the argument expression *does not change the state at function invocation time*; it only binds the parameter to the frozen pre-value. However, whenever the pre-value needs to be evaluated, its side effects are propagated in

the proper environment.

Semantics of Call-by-Need

Call-by-need differs from call-by-name in what the frozen *expression is evaluated only once*, the first time its value is needed in order to evaluate the body of the function. Technically speaking, that means that the frozen expression is *unfrozen* and replaced by its value in the store at the appropriate location.

The side effects from evaluating the expression the first time have to be propagated as in the case of call-by-name. However, subsequent evaluations of that expression return directly the value obtained the first time it was evaluated, so the side effects are propagated *only once*. In order to distinguish this new pre-value from the frozen one, we define a new operator, `unfreeze`.

Given a state and a location, one can easily find the pre-value stored at that location. However, given a pre-value it is not at all clear at what location it is stored; the same pre-value can be stored at several locations. Therefore, in order to replace an `unfreeze` pre-value by a proper value, we need to also add the location where the expression should be unfrozen as part of the pre-value:

```
fmod CALL-BY-NEED-SEMANTICS is extending CALL-BY-NEED-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op unfreeze : Exp Environment Location -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  var L : Location .
  eq  eval(unfreeze(E,Env,L), S) = eval(E, S[env <- Env]) .
  eq state(unfreeze(E,Env,L), S) =
     S[store <- state(E, S[env <- Env])[store][L <- eval(E, S[env <- Env])]] .
  eq state(need X, E, S) = S .
  eq  bind(need X, E, S, S') =
      S'[idx(X) <- unfreeze(E, S[env], nextLoc(S'[store]))] .
```



```
endfm
```

Closures and Scoping

A function's body can refer to *unbound names*; these are names which are neither among the function's parameters nor bound within the function's body expression. A major semantical aspect in the design of a programming language is to *decide to what locations those free names are bound*. As we discussed in Lecture 5, there are essentially two possibilities:

1. Under *static (or lexical) scoping (or binding)*, they refer to the corresponding locations in the environment in which the function is defined;
2. Under *dynamic scoping*, they refer to the corresponding locations in the environment in which the function is executed.

We want to support either of these two conflicting scoping

possibilities. In order to support static binding, the notion of *closure* plays a crucial role. A closure essentially *freezes the environment* in which a function was declared. Therefore, *functions evaluate to closures*.

The following defines the syntax of closures and of auxiliary operations needed to define the semantics of function invocation:

```
fmod CLOSURE is protecting PARAMETER-SEMANTICS .
  sort Closure .
  subsort Closure < Value .
  op closure : ParameterList Exp Environment -> Closure .
  op apply : Closure ExpList State -> Value .
  op state : Closure ExpList State -> State .
endfm
```

`apply` is intended to apply a closure to a list of arguments. Notice that these arguments are *not* evaluated, so they are not (pre-)values, because at this moment the parameter styles in

`<ParameterList>` are not known. `state` as usual collects the side effects. The meaning of these operators depends on whether one wants static scoping or dynamic scoping in one's language. We next define them in the context of static scoping:

```
fmod STATIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env])) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env]))[store]] .
endfm
```

Intuitively, the above formally states that, when applying a closure to a list of expressions, one performs the following steps:

1. Evaluates all the argument expressions which need to be evaluated according to the corresponding parameter passing

styles;

2. The side effects are collected and then the parameters are bound according to the parameter passing styles on top of the environment coming with the closure;
3. The body expression is evaluated in the state above.

In the case of dynamic binding, the only change is that step 2 above bounds the parameters on top of the current environment rather than on top of that frozen by the closure:

```
fmod DYNAMIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S))) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)))[store]] .
endfm
```


Semantics of Functions

We have now all the infrastructure to define the semantics of functions and function invocations. One needs to include all the desired styles of parameter passing as well as either STATIC-BINDING or DYNAMIC-BINDING:

```
fmod PROC-SEMANTICS is protecting PROC-SYNTAX .
  protecting CALL-BY-VALUE-SEMANTICS .
  protecting CALL-BY-REFERENCE-SEMANTICS .
  protecting CALL-BY-NAME-SEMANTICS .
  protecting CALL-BY-NEED-SEMANTICS .
*** the next lets us choose between static vs. dynamic binding
  protecting  STATIC-BINDING .
---  protecting DYNAMIC-BINDING .
```

Next we define the semantics of functions and function invocations as expected. Note that the evaluation of function is side effect free:

```
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .  var S : State .
  eq  eval(proc Pl E, S) = closure(Pl, E, S[env]) .
  eq state(proc Pl E, S) = S .
  eq  eval(E El, S) = apply(eval(E,S), El, state(E,S)) .
  eq state(E El, S) = state(eval(E,S), El, state(E,S)) .
endfm
```

Let us next consider some examples. The evaluation of a function is a closure, in the case below one over an empty environment:

```
red eval((proc(need x, value y) 0)) .
***> should be closure((need x,value y), 0, empty)
```

The following evaluates to 0 under both static and dynamic scoping, because the body of the function has no free names:

```
red eval((proc(name x, value y) 0) (2,3)) .
***> should be 0
```

The following shows how functions can be passed as arguments. Follow the semantics and try to understand how it is evaluated:


```
red eval((proc(value x, reference y) (x(y))) (proc(value x) 2, 3)) .
***> should be 2
```

The following shows that we can get different evaluations under different scoping strategies.

```
red eval((proc(value x, reference y) (x(y))) (proc(value x) y, 3),
         (  env, [idx(y), loc(0)])
         (store, {loc(1), [loc(0),int(1)]})) .
***> should be 1 under static scoping and 3 under dynamic scoping
```

Exercise 2 *Follow the application of equations in evaluating the above step by step and understand why we get the different values.*

Homework Exercise 2 *In the semantics of our functional programming language presented so far, we froze the* entire *calling environment in closures and in the pre-values generated by call-by-name and call-by-need parameter passing styles. In the context of large programs, environments can become quite large and consequently freezing entire environments can be prohibitive. Do we need to freeze the entire environment? Which names are needed to be stored? Modify the semantics defined so far in order to freeze only a minimal sub-environment.*

Hint. *Make sure that all the expressions in* `prog-lang2.maude` *will be evaluated properly.*

CS322 - Programming Language Design

Lecture 9: Designing a Functional Language - Semantics; Part 3

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Homework Exercise 1 *Define* call-by-value-result *and add it to our functional programming language. Under call-by-value-result, the actual argument must be a variable. Like in call-by-value, a new location containing the value of the actual parameter is created when the function is invoked, and the formal parameter is bound to that new location. After the body of the function is evaluated, when "returning" to the calling environment, the value at the new location to which the formal parameter is bound is copied back into the location that the actual argument is bound to. This is* almost *like call-by-reference, but there are some subtle differences. Give 2 expressions which evaluate to different values under call-by-reference and under call-by-value-result, respectively.*

Semantics of Letrec

Suppose that one wants to define the factorial function recursively. One apparently natural way to do it is as follows:

```
let f = proc(value n)
          if zero?(n)
          then 1
          else n * f(n - 1)
in f(5)
```

However, the above turns out to be evaluated properly, that is to 120, only under dynamic scoping. Under static scoping it is undefined.

Exercise 1 *Why is it undefined under static scoping?*

Let us next consider the following slightly modified version of the expression above:


```
let f = proc(value n) n + n
in let f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
   in f(5)
```

The above will be defined under both static and dynamic scoping, but it will be evaluated to different values: 40 under static scoping and 120 under dynamic scoping.

We therefore failed to define recursive functions under static scoping. One would, of course, like to define recursive functions as naturally as possible, very close to how we tried above but failed.

Moreover, one would also like to be able to define *mutually recursive* functions as naturally as below:

```
let 'even = proc(value x)
              if zero?(x) then 1
              else 'odd(x - 1),
     'odd = proc(value x)
              if zero?(x) then 0
              else 'even(x - 1)
in 'odd(17)
```

Again, the above evaluates properly under dynamic scoping but it is undefined under static scoping.

In the early years of functional programming, since the advantages of static scoping over dynamic scoping were relatively clear, scientists developed (tricky) techniques to correctly define recursive functions in the context of static scoping, such as the following:


```
let f = proc(value x, value g)
          if zero?(x) then 1
          else x * g(x - 1, g)
in f(5, f)
```

The trick is to "delay" the use of the function's name until it is present in the environment. In the above, `f` evaluates to a closure which is then passed as an argument besides the decreasing `x`. Similarly, in the context of mutually recursive functions one has to pass *all* the mutually recursive functions as arguments:

```
let x = 17,
    'odd = proc(value x, value o, value e)
              if zero?(x) then 0
              else e(x - 1, o, e),
    'even = proc(value x, value o, value e)
              if zero?(x) then 1
              else o(x - 1, o, e)
in 'odd(x, 'odd, 'even)
```

Since both recursion and static scoping are very important concepts in programming languages, they deserve to be supported together without any additional trick or methodology. However, in order to do it properly, scientists have introduced a new language construct `letrec`, as an alternative to `let`. Most functional programming languages today support `letrec`.

With `letrec`, the expressions above can be simply rewritten to:

```
letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(5)
```

and

```
letrec 'even = proc(x) if zero?(x) then 1 else 'odd(x - 1),
        'odd = proc(x) if zero?(x) then 0 else 'even(x - 1)
in 'odd(17)
```


Unlike `let`, which first evaluates the expressions in its bindings, then creates the bindings of names to the corresponding values, and then evaluates its body expression in the new state, `letrec` works as follows:

1. *Creates bindings* of its names to locations which currently contain no values; they will be assigned values at step 3;
2. *Sequentially evaluates the expressions* occurring in its binding in the newly obtained state;
3. *Adds the values* obtained at step 2 to the locations bound at step 1, thus obtaining a new state;
4. *Evaluates its body* in the new state obtained at step 3.

If one does not use the names bound by `letrec` in any of the binding expressions then it is easy to see that it is behaviorally equivalent to `let`.

However, it is crucial to note that those names bound using `letrec` are accessible in the expressions they are bound to! Those of these names associated to function expressions will be therefore bound to closures including their own binding in the environment.

More precisely, if `S'` is the new state obtained at step 3 above when `letrec` is evaluated in a state `S`, then the value associated to a name `X` in `S'` is

- The value of `X` in `S` if `X` is not a name bound by `letrec`;
- A closure whose environment is *the same as* that of `S'` if `X` is bound to a function expression by `letrec` or to an expression which evaluates to a function;
- An integer or an undefined value otherwise.

While this is exactly what we want in the context of recursive functions, one should be very careful when one declares non-functional bindings with `letrec`. For example, the behavior of

the expression

```
let x = 1
in letrec x = 7,
          y = x
   in y
```

is undefined. However, notice that the variable `x` is *not free* in

```
letrec x = 7,
       y = x
in y
```

Instead, it is bound to a location whose value is undefined!

The following module defines the semantics of `letrec` formally:

```
fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  op mkDanglingBindings : BindingList Location State -> State .
  var X : Name .  var N : Nat .
  var Bl : BindingList .  var E : Exp .  vars S S' : State .
  eq mkDanglingBindings(none, loc(N), S) = S .
  eq mkDanglingBindings((X = E, Bl), loc(N), S) =
     mkDanglingBindings(Bl, loc(N + 1), S[idx(X) <- loc(N)]) .
 ceq eval(letrec Bl in E, S) = eval(E, bindBList (Bl,S') in stateBList(Bl,S'))
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
 ceq state(letrec Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S') in stateBList(Bl, S'))[store]]
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
endfm
```

Notice that, for the first time, we used *conditional equations*.

Moreover, we have used the *matching operator* `_:=_`. The meaning of these equations is very intuitive: evaluate the right-hand-side term of `_:=_` in the condition, assign its value to its left-hand-side variable, and then apply the body of the equation.

The situation is actually more complex in Maude. You can read the manual if you are interested, but we do not need the extra-power of conditional equations and matching now. The only reason for which we used them above was to avoid writing the right-hand-side term of `_:=_` twice in each equation. Notice that Maude would not evaluate it twice anyway, because of its internal efficient rewriting algorithms which take advantage of sharing; it was therefore used just for *syntactic sugar* reasons.

Semantics of Variable Assignment

With the already existing infrastructure, which has been developed in such a way to easily accommodate side effects, the semantics of variable assignment can be now trivially defined:

```
fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm

red eval(set x = 3) .
***> should be 1
```


The above semantics reflects the design choice that the value of any variable assignment is one, regardless of whether the assigned name was or was not previously declared.

Exercise 2 *Change the semantics above such that a variable assignment expression is undefined when the assigned name has not been previously declared.*

Answer. In order to check whether a name `X` has been previously declared in a state `S`, one only has to see whether `eval(X, S)` evaluates to a proper value. One would also need to use a conditional equation, where the condition is a membership assertion (see Maude's manual for more on conditional axioms):

```
ceq eval(set X = E, S) = int(1) if eval(X,S) : Value .
```

Semantics of Blocks

The meaning of blocks of expressions is straightforward: each expression is evaluated sequentially, the side effects are properly propagated, and the result value is the *value of the last expression* in the block:

```
fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq  eval({E}, S) =  eval(E, S) .
  eq state({E}, S) = state(E, S) .
  eq  eval({El ; E}, S) =  eval(E, state({El}, S)) .
  eq state({El ; E}, S) = state(E, state({El}, S)) .
endfm
red eval({x ; y ; 3 ; x}) .  ***> should be undefined
```

Notice that we do not allow empty blocks.

Semantics of Loops

The semantics of `while` loops is also straightforward. The only thing which deserves to be mentioned is that, by convention, the value to which a `while` loop evaluates, when it terminates, is `1`:

```
fmod LOOP-SEMANTICS is protecting LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  var S : State .
  eq  eval(while Be E, S) =
     if eval(Be,S) then  eval(while Be E, state(E,state(Be,S)))
     else int(1) fi .
  eq state(while Be E, S) =
     if eval(Be,S) then state(while Be E, state(E,state(Be,S)))
     else state(Be,S) fi .
endfm
```

Putting All the Semantics Together

We can now finally design our functional programming language by putting together all the features that we are interested in having in our language. In this case, we just include everything (but notice that only one static vs. dynamic scoping styles are valid!):

```
fmod PROG-LANG-SEMANTICS is
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm
```


Evaluating Expressions within the Semantics

Due to the fact that Maude is executable, we can now *evaluate expressions directly within the semantics* of our functional programming language.

It is important to understand though, that one could have just defined the entire semantics on paper, mathematically, and one would have obtained, modulo a trivial syntactic translation and formalization, exactly the same definition as the one we presented in the last 4 lectures.

The great advantage of using an executable specification language to define the semantics formally is that one can also have the feeling that one "executes" programs in the defined programming language, the same way as using an interpreter. Moreover, since Maude's rewriting is highly optimized, we get quite an efficient

"interpreter" *for free*: our initial goal was *not* to get an interpreter but to define the semantics, or the mathematical meaning, of all the programming language constructs.

Another important advantage is that, by "testing" the definition of the language on tens of examples, one may discover design errors or partially specified features, which are quite easy to escape when one writes the semantics on paper without machine support.

The examples that follow cover all the language constructs. Some of them are simple, others are tricky. Some of the trickier ones will be modified and included as part of the final exam.

The examples below can also be used as a benchmark to validate changes of the semantics required by adding new features to the language.

Examples with Let and Arithmetic Operators

```
red eval(
  let x = 5, y = 7
  in x + y
) .  ***> should be 12
red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .  ***> should be 4
red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .  ***> should be 2
```

```
red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .  ***> should be 5
red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .  ***> should be 5
red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .  ***> should be 11
```


Adding Functions

```
red eval(
  proc(value x, value y, value z) x * (y - z)
) .
***> should be closure((value x,value y,value z), x * (y - z), empty)

red eval(
  (proc(value y, value z) y + 5 * z) (1,2)
) .  ***> should be 11

red eval(
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
) .  ***> should be 34
```

```
red eval(
  (proc(value x, value y) x(y)) (proc(value z) 2 * z, 3)
) .  ***> should be 6

red eval(
  let x = proc(value x) x in x(x)
) .  ***> should be closure(value x, x, empty)

red eval(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .  ***> should be 1
```


Static vs. Dynamic Scoping

The following expressions have different meanings under static and dynamic scoping:

```
red eval(
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(value x, value y) (x y)) (proc(value x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping
```

```
red eval(
  let x = 1
  in let x = 2,
         f = proc (value y, value z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping
```



```
red eval(
  let a = 3
  in let p = proc(value x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(value n) n + n
```

```
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(value x) (p())
---            f = proc(value a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping
```


Adding Recursion

The first example shows how one can simulate recursion without using `letrec`. It follows the same idea as the one we presented when we defined the semantics of `letrec`. Compare this with the elegance of using `letrec` in the next examples. Notice that one can use `letrec` both under static and under dynamic scoping:

```
red eval(
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12
```

```
red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(value x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12
```



```
red eval(
  letrec 'even = proc(value x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(value x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .  ***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .  ***> should be undefined
```

```
red eval(
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping
```


Variable Assignment

The following examples are intended to show the need and also the effects of variable assignments. Notice the very interesting fact that one can have expressions which are well defined under static scoping but undefined under dynamic scoping. How can that be possible?

```
red eval(
  let c = 0
  in let f = proc()
              let c = c + 1
              in c
     in f() + f()
) .
***> should be 2
```

```
red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***> should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***> should be 3
```



```
red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***> should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (value x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2
```

```
red eval(
  let x = 0, y = 1
  in let f = proc(value x, value y)
              let t = x
              in let d = set x = y
                 in let d = set y = t
                    in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2
```


Call-by-Need

```
red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(value a, need b, need c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be 4
```

Binding Non-Functions in letrec

```
red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***> should be 1
```



```
red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***> should be 0
```

Blocks

```
red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .  ***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .  ***> should be 24
```


Other Examples

Some other interesting examples, in no particular order, are listed below. Understanding them well is your first step towards a good score at final.

```
red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12
```

```
red eval(
  let x = 3, y = 4,
      f = proc(reference a, reference b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 4
```



```
red eval(
  let f = proc(need x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .  ***> should be 8

red eval(
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .  ***> should be 8
```

```
red eval(
  let f = proc(name x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .  ***> should be 11

red eval(
  let y = 5,
      f = proc(name x) x + x,
      g = proc(reference x) set x = x + 3
  in {f(g(y));
      y}
) .  ***> should be 11
```



```
red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185
```

Exercise 3 *Read Chapter 3 in Friedman. As usual, ignore all the Scheme code and other Scheme related comments.*

Homework Exercise 2 (Extra credit: 2 points). *This exercise is intended to show that you understand the relationship between* `letrec` *and static vs. dynamic scoping. Is* `letrec` *needed in the context of dynamic scoping? Why? By the* `let`-equivalent *of an expression we mean the expression in which each occurrence of* `letrec` *is replaced by a* `let`*. Give 2 running examples of expressions which evaluate under static scoping to the same values as their corresponding* `let`*-equivalents under dynamic scoping, and 2 expressions which do* not *evaluate under static scoping to the same values as their corresponding* `let`*-equivalents under dynamic scoping. Your 4 expressions should all be defined under both static and dynamic scoping, and should show that you understand well the core problem. Otherwise you do not get credit for this problem.*

CS322 - Programming Language Design

Homework Assignment 2
(due Th, 02 Oct., by email before the class)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Homework Exercise 1 *Define a Maude module* `FREE-NAMES` *importing* `PROG-LANG-SYNTAX` *and defining an operation* `free-names : Exp -> NameSet`*, which collects all the names that occur free in an expression.*
Hint. *Define it inductively ovet the structure of the syntax, making sure that you do not forget any important language construct. Also, make sure that you understant the difference between* `let` *and* `letrec` *with respect to how and when they bind the names.*

Homework Exercise 2 *The third reduction after defining* `BEXP-SEMANTICS` *should be also undefined. Explain why the result* `false` *is wrong and give an example of an expression which should normally be undefined but it actually wrongly evaluates to an integer value. Fix this bug.*

Homework Exercise 3 *Some languages, including* Scheme*, provide a language construct* `let* <BindingList> in <Exp>` *which binds its names* sequentially*, rather than in parallel. For example,*

`let* x = 1, y = x + 1 in y` *evaluates to* `2`*. Define* `let*` *and give 5 examples showing the difference between* `let` *and* `let*`*.*

Homework Exercise 4 *In the semantics of our functional programming language presented so far, we froze the entire calling environment in closures and in the pre-values generated by call-by-name and call-by-need parameter passing styles. In the context of large programs, environments can become quite large and consequently freezing entire environments can be prohibitive. Do we need to freeze the entire environment? Which names are needed to be stored? Modify the semantics defined so far in order to freeze only a minimal sub-environment.*
Hint. *Make sure that all the expressions in* `prog-lang2.maude` *will be evaluated properly.*

Homework Exercise 5 *Define* call-by-value-result *and add it to our functional programming language. Under call-by-value-result, the actual argument must be a variable. Like in call-by-value, a new*

location containing the value of the actual parameter is created when the function is invoked, and the formal parameter is bound to that new location. After the body of the function is evaluated, when "returning" to the calling environment, the value at the new location to which the formal parameter is bound is copied back into the location that the actual argument is bound to. This is almost *like call-by-reference, but there are some subtle differences. Give 2 expressions which evaluate to different values under call-by-reference and under call-by-value-result, respectively.*

Homework Exercise 6 (Optional, for extra-credit) *This exercise is intended to show that you understand the relationship between* `letrec` *and static vs. dynamic scoping. Is* `letrec` *needed in the context of dynamic scoping? Why? By the* `let`-equivalent *of an expression we mean the expression in which each occurrence of* `letrec` *is replaced by a* `let`*. Give 2 running examples of expressions which evaluate under static scoping to the same values*

as their corresponding `let`*-equivalents under dynamic scoping, and 2 expressions which do* not *evaluate under static scoping to the same values as their corresponding* `let`*-equivalents under dynamic scoping. Your 4 expressions should all be defined under both static and dynamic scoping, and should show that you understand well the core problem. Otherwise you do not get credit for this problem.*

CS322 - Programming Language Design

Lecture 10: Typed Languages - Static Type Checking (Part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

There is nothing in the definition of our functional programming language so far which would forbid one from writing erroneous programs. While the domain or application specific errors are hard or impossible to catch automatically, there is a large class of errors, known as *type errors*, which often can be caught automatically with appropriate program analysis tools.

Static type checkers are special program analysis tools that take programs as inputs and typically report lists of type errors, which may turn into runtime errors or misbehaviors. Type checkers are so popular nowadays and the kind of errors they catch so frequently lead to wrong behaviors, that most compilers today provide a static type checking analysis tool as a front end. The major drawback of static type checkers is that they may *reject correct programs*, so they somehow limit the possibilities of writing programs.

Typed Languages

A typed language defines a set of *types*, as well as a *typing policy* by which types can be associated to values calculated by programs.

By applying this policy recursively, one can mechanically associate a type to each expression in a given program. In order to do it, one needs to inspect the invocations of language constructs and check that each language construct is used properly by analyzing the relationship between its operands, their expected types and the context in which the language construct appears.

If a language construct use is found to violate the expected typing policies, then we say that a *type error* has been found. E.g., think of the expression `3 + (proc(x) x)`. While under certain special circumstances one may wish to accept such programs as correct, in general a typing policy would classify this is as a type error.

When designing a type checking tool for a typed language, one needs to take a decision on *what to do when a type error is found*.

The simplest decision is to reject the program. Better decisions are to also report where the error has been found, as well as possible causes, to users. Even better, the type checker can continue and report subsequent type errors within only one session.

One can even think of taking automatically some correcting measures, such as to convert integers to reals, or kilometers to miles, or to modify the code by introducing runtime checks, etc.

Dynamic Type Checking

At the expense of increasing their runtime overhead, some languages maintain a type together with each computed value in their store. This way, variables are associated not only values but also the types of those values. Before an operation is applied, a *dynamic type checker* validates the application of the operation. In the case of `3 + (proc(x) x)`, before the operation `_+_` is evaluated the dynamic type checker checks whether the two operands are integers; this way a runtime error will be reported.

While dynamic type checking is useful and precise, and many successful programming languages are dynamically typed, such as *Scheme*, *PERL*, *Python*, etc., most programming language designers believe that runtime checks are too expensive in practice.

Untyped Memory Models

If one simply removes the runtime checks for type consistency then one can obviously run into serious difficulties. For example, when adding an integer and a function, since they are both stored as binary numbers at some locations, without a consistency check one would just add the integer with the binary representation of (part of) a particular representation of the function, which is wrong.

Languages without runtime or dynamic type checking are said to have *untyped memory models*. Such languages essentially assume that the operations are applied on the right data, so they can increase their performance by removing the runtime type consistency checks.

Static Type Checking

To take full advantage of the increased execution speed of untyped memory models while still not sacrificing the correct runtime behavior of programs, an additional layer of certification is needed. More precisely, one needs to ensure *before execution* that all the operations will be applied on the expected type of data. This process is known under the terminology *static type checking*.

Languages admitting static type checkers, that ensure that once a program passes the type checker it will never exhibit any type error at runtime, are called *strongly statically typed*. It is quite hard or even impossible (due to undecidability arguments) in practice to devise strongly typed languages where the types are intended to state that a program is free of general purpose errors. For example, it is known to be undecidable to say whether a program will ever perform a division by zero, or if a program terminates.

To keep static type checkers decidable and tractable, one typically has to *restrict* the class of errors that types can exhibit. In the context of our simple functional programming language, like in many other languages, by a *type error* we mean one of the following:

- An application of a non-functional expression to a list of arguments;
- An application of a function expression to a wrong number of arguments or to a list of arguments with types different from the types of function's parameters;
- An assignment of an expression to an existing name, such that the type of the expression is different from the declared type of the name;
- A use of an arithmetic operation on non-integers;
- A use of a conditional statement where the first argument is not a boolean or where the two branches are of different type.

Besides the simplistic typing policy above, we also need to state *how the types of expressions are calculated and propagated* via *typing rules*. This is quite straightforward in our formalisms, because typing rules are nothing but special cases of *conditional equations*. For example, "the type of `x + y` is `integer` if the types of `x` and `y` are `integer`". Similarly, "the type of a conditional is the type of its first branch if the type of its argument is `bool` and the types of its branches coincide".

In order to allow type checking in a language, one first needs to extend the language with *type declarations*, so that one can add typing information to the program. We will see later, when we discuss type inference, that in some situations types can be deduced automatically by examining how names and expressions are used.

Since declarations can occur either in a `let`/`letrec` statement or in a function (the parameters), we will slightly extend the syntax of our language to allow type declarations in addition to and at the

same time with name declarations. For example, we will allow expressions like

```
let integer x = 17 in x
let integer x = 17, integer y = 10 in x + y
```

Besides the *basic types* `integer` and `bool`, we will also introduce *product types* (using the type constructor `_*_`) for a function's parameters. Since our language is functional, with functions as first-class citizens, i.e., can be passed and returned by other functions, we also have to introduce *function types* (using the constructor `_->_`).

Thus, we will be able to write expressions like

```
let integer a = 3
in let (integer -> integer)
         p = proc(integer value x) x + a,
       integer a = 5
   in a * p(2)
```

```
let integer x = 2,
    (integer * integer -> integer)
      f = proc (integer value y, integer value z) y + x * z
in f(1,x)
```

Defining Program Analysis Tools

The modular design of our programming language was intended not only to easily deal with changes in the design of the language, but also to facilitate the definition of *program analysis tools*.

In fact, the definition of a program analysis tool can be seen as very similar in spirit to defining a semantics to a programming language. The meaning a program analysis tool has for a program may, of course, be quite different from that of the semantics of the programming language, because the two may look at two quite different aspects of a program.

However, we will again take advantage of the efficient executable environment provided by Maude through its fast implementation of rewriting, and this way obtain not only executable but also quite efficient program analysis *tools for free*, from just their mathematical rigorous definition.

In today's lecture we will focus on one of the simplest possible program analysis tools, namely a static type checker.

Adding Types to the Syntax

There are very few changes that need to be done in the syntax of our functional programming language in order to allow types. First, of course, one has to introduce the types:

```
fmod TYPE is
  sorts BasicType Type .  subsort BasicType < Type .
  ops integer bool : -> BasicType .
  op nothing : -> Type .
  op _*_ : Type Type -> Type [assoc id: nothing prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
endfm
```

The type of a function without parameters returning an integer will be `nothing -> integer`. Note that there are type expressions which cannot be generated with our current simple language, such as `integer -> bool` or `integer -> integer * integer`.

Besides the above, there are two very minor changes that need to be done to the syntax in order to support types in declarations, one for binding and the other for parameter declarations.

The equality `_=_ : Name Exp -> Binding` used within bindings needs to also allow a type to be specified:

```
  op __=_ : Type Name Exp -> Binding [prec 70] .
```

Similarly, the parameter declaration `__ : CallingMode Name -> Parameter` needs to be changed to

```
  op ___ : Type CallingMode Name -> Parameter [prec 0] .
```

Exercise 1 *We could have designed the syntax of our programing language in a more modular way such that one could have replaced the two changes above by adding one attribute to a "generic" name. The calling mode and the index would be just other attributes of such a generic name. Do this change in the design of our language and then reflect on the increase in modularity attained this way.*

Defining a Static Type Analyzer

The general idea underlying a static type checker is to recursively analyze each language construct occurring in a given program, to prove that its operands satisfy the type constraints, and then to assign a type to the expression created by that new construct.

The type of a name *cannot* be changed by variable assignment constructs, so a name has its declared type in all its occurrences within its scope; but clearly one can differentiate between static and dynamic scoping. We will only consider static scoping.

Homework Exercise 1 (Extra credit). *Define an executable static type checker in Maude for the dynamically scoped version of our language.*

Any software analysis tool has a state that it maintains as it traverses the program to be analyzed. This is also what happened

in the definition of the semantics of the language. Fortunately, the same module `GENERIC-STATE` can be used unchanged.

A type checker ignores the concrete values bound to names, but it needs to *bind names to their types*. We can realize that by defining a special state attribute, called `TypeEnvironment`, which contains a mapping from names to their current types.

The following module can simply "cut-and-paste" the module `ENVIRONMENT` defined by the executable semantics of the language; however, one needs to replace locations by types. The current version of Maude does not provide support for parameterized modules, like OBJ. If it did, then we could have defined a generic environment module as a map parameterized by its domain and target sorts, and then just instantiate it whenever needed.

```
fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
            [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
  op _[_<-_] : TypeEnvironment Index Type -> TypeEnvironment .
  vars Ix Ix' : Index .  vars T T' : Type .  var TEnv : TypeEnvironment .
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv)[Ix <- T'] = [Ix,T'] TEnv .
  eq TEnv[Ix <- T'] = [Ix,T'] TEnv [owise] .
endfm
```


No other information but the type environment is needed by the static type checker. Therefore, we can finalize the infrastructure by defining the following module, which, like for the semantics of language, provides an abstract interface for the subsequent modules:

```
fmod STATE is
  protecting TYPE-ENVIRONMENT .
  op _[_] : State Index -> Type .
  op _[_<-_] : State Index Type -> State .
  var S : State .  var Ix : Index .  var T : Type .
  eq S[Ix] = S[tenv][Ix] .
  eq S[Ix <- T] = S[tenv <- S[tenv][Ix <- T]] .
endfm
```

`S[Ix]` gives the type associated to an index `Ix` in state `S`, while `S[Ix <- T]` updates the type of a name.

Typing of Names

The type of a name is simply the type bound to its index in the current state. Unlike in the definition of the semantics of the language where side effects could modify the values bound to names, *the type bound to name cannot be modified*. Therefore, we only need one operation returning the type of an expression. We start by defining it on names:

```
fmod NAME-STATIC-TYPING is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op type : Name State -> Type .
  var X : Name .  var S : State .
  eq type(X, S) =  S[idx(X)] .
endfm
```


Typing of Generic Expressions

Like before, we also define an operator, simply called `type : Exp -> Type`, as its binary version operator called on the initially empty type environment. Also, we define the type of an integer to be `integer`.

```
fmod GENERIC-EXP-STATIC-TYPING is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-STATIC-TYPING .
  op type : Exp State -> Type .
  op type : Exp -> Type .
  var I : Int .  var S : State .  var E : Exp .  var El : ExpList .
  eq type(I, S) = integer .
  eq type(E) = type(E, (tenv,empty)) .
```

Additionally, we will later need the type of a list of expressions passed to a function, so we are opportunistic and define it now as the product of the individual types:

```
  op typeEList : ExpList State -> Type .
  eq typeEList((), S) = nothing .
  eq typeEList((E,El), S) = type(E,S) * typeEList(El, S) .
endfm
```

Some examples clarify these definitions:

```
red type(3) .          ***> should be integer
red type(x) .          ***> should be undefined
red type('variable) .  ***> should be undefined
red type('variable,
      (tenv, [idx(x),integer] [idx('variable),integer * integer -> integer])) .
***> should be integer * integer -> integer
```


Typing of Arithmetic Operators

From now on, we will use conditional equations as well in our Maude code. Their meaning is rather obvious: they are applied only if their conditions are proved first. One can recursively use other conditional equations in order to prove a condition.

A condition can be multiple, that is, a list of equalities separated by conjunctions (`/\`). Then Maude proves them sequentially, from left to right. It is important to remember that all the variables occurring in a conditional equation must appear in the left-hand-side term of the head of the equation; matching conditions are an exception from this rule, and we will discuss these later in the lecture.

We can now define the *typing rules* for the arithmetic operators simply as the conditional equations. For example, the type of `E +`

E' in a state S is integer if E and E' have the type integer in S.

```
fmod ARITH-OPS-STATIC-TYPING is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  vars E E' : Exp .  var S : State .
 ceq type(E + E', S) = integer
    if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E - E', S) = integer
    if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E * E', S) = integer
    if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E / E', S) = integer
    if type(E,S) = integer /\ type(E',S) = integer .
endfm
```

```
red type(3 + x, (tenv, [idx(x),integer])) .  ***> should be integer
red type(3 + 'variable1,
         (tenv, [idx('variable), integer * integer -> integer])) .
***> should be undefined
```


Typing of Boolean Expressions

The type of a boolean expression is bool, provided that the other boolean expressions or the other expressions it uses have the right types or the right relationship between those:

```
fmod BEXP-STATIC-TYPING is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op type : BExp State -> Type .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
 ceq type(E equals E', S) = bool if type(E, S) = type(E', S) .
 ceq type(zero?(E), S) = bool if type(E, S) = integer .
  eq type(not(Be), S) = type(Be, S) .
 ceq type(even?(E), S) = bool if type(E, S) = integer .
 ceq type(Be and Be', S) = bool if type(Be, S) = bool /\ type(Be', S) = bool .
endfm
```

```
red type(3 equals 5, S) .  ***> should be bool
```

Typing of Conditionals

The type of a conditional is defined to be the type of one of its two branches, provided that the two branches have the same type and that the condition has the type `bool`.

```
fmod IF-STATIC-TYPING is protecting IF-SYNTAX .
  extending BEXP-STATIC-TYPING .
  vars E E' : Exp .  var Be : BExp .  var S : State .
 ceq type(if Be then E else E', S) = type(E, S)
    if type(Be,S) = bool /\ type(E,S) = type(E',S) .
endfm
```

```
red type(if zero?(5) then 2 else 3) .     ***> should be integer
red type(if zero?(x) then y else z,
 (tenv, [idx(x), integer] [idx(y), integer * integer -> integer]
        [idx(z), integer * integer -> integer])) .
***> should be integer * integer -> integer
```


Typing of Bindings

Like for the executable semantics of the language, we define `bindBList(_,_)in_` that creates the bindings of the names and types listed in its first argument, calculated in the second argument state but added within its third argument state. We need this flexibility in order to accommodate both `let` and `letrec`:

```
fmod BINDING-STATIC-TYPING is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  var T : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[idx(X) <- T]) if type(E,S) = T .
endfm
```

Notice that the typing rule of a binding states that the bound expression must have the declared type of the corresponding name.

Typing of Let

The type of a `let` expression is the type of its body expression in the state obtained after binding its names to the corresponding expressions. Notice that the type of a `let` remains undefined if its bindings cannot be properly typed.

```
fmod LET-STATIC-TYPING is extending LET-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq type(let Bl in E, S) = type(E, bindBList (Bl, S) in S) .
endfm
```


Typing of Parameters

In order to calculate the type of a function, we will need to first calculate the product type of its arguments. The operator `typePList` defined below does exactly that.

```
fmod PARAMETER-STATIC-TYPING is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op typePList : ParameterList -> Type .
  var S : State .  var T : Type .  var C : CallingMode .
  var X : Name .  var Pl : ParameterList .
  eq typePList(()) = nothing .
  eq typePList(T C X, Pl) = T * typePList(Pl) .
```

As in the case of `let`, we also need to bind the parameter names to their appropriate types before typing the body of a function. The operation `bindPList` defined below does that:

```
  op bindPList : ParameterList State -> State .
  eq bindPList((), S) = S .
  eq bindPList((T C X, Pl), S) = bindPList(Pl, S[idx(X) <- T]) .
endfm
```

However, note that, unlike in the case of `let`, the consistency between the declared type of the parameter and the type of actual argument is not checked at this moment. It will be checked later, when the typing rule of a function invocation will be defined:

Typing of Functions

The type of a function is simply the function type from its argument product type to the type of its body expression in the state obtained after binding its arguments to their types:


```
fmod PROC-STATIC-TYPING is protecting PROC-SYNTAX .
  protecting PARAMETER-STATIC-TYPING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) =
     typePList(Pl) -> type(E, bindPList(Pl, S)) .
```

When a function is invoked, the type of the result should be the type of its body (extracted from the function's type) provided that the product type of the actual arguments matches the product type of function's parameter (also extracted from function's type):

```
 ceq type(E El, S) = T
    if Tp -> T := type(E, S) /\ typeEList(El, S) = Tp .
endfm
```

The conditional equation above uses a distinctive and powerful feature of Maude, the matching operator `:=`. It works as follows: its right-had-side term is reduced to its normal form; then its

left-hand-side term, regarded as a patters, is matched against the term obtained at the previous step; if the matching succeeds then the variables in the patters are bound accordingly; otherwise the matching does not succeed and so does the condition of the equation.

The following are some clarifying examples:

```
red type(proc(integer need x,
              (integer * integer -> integer) value y) 0) .
***> should be integer * (integer * integer -> integer) -> integer

red type((proc(integer name x,
               (integer -> integer) value y) 0) (2,3)) .
***> should be undefined

red type((proc((integer -> integer) value x,
               integer reference y) (x(y)))
            (proc(integer value x) 2, 3)) .
***> should be integer
```

```
*******************************************************************************
*** Defining a Static Type Checker for a Functional Programming Language ***
*******************************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse 3 + x .
parse 3 + 'variable1 .
parse 3 + 'variable2 + x * (y - z) + 'variable1 .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
```

```
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 equals 3 .
parse 3 equals 5 .
parse 5 equals x .
parse 3 + x equals 0 .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5) then 2 else 3 .
parse if zero?(0) then 2 else 3 .
parse if zero?(x) then y else z .
parse if x equals y
     then x
     else if x equals z
          then z
          else y .

fmod TYPE is
  sorts BasicType Type .
  subsort BasicType < Type .
  ops integer bool : -> BasicType .
  op nothing : -> Type .
  op _*_ : Type Type -> Type [assoc id: nothing prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList
           [assoc id: none prec 71] .
  op __=_ : Type Name Exp -> Binding [prec 70] .
endfm
```

```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let integer x = 5
  in x
 .
parse
  let integer x = 5, integer y = 7
  in x
 .
parse
  let integer x = 5
  in let integer y = x
     in y
 .
parse
  let integer x = 1
  in let integer x = 2
     in x
 .
parse
  let integer x = 10, integer y = 0, integer z = x
  in let integer a = 5, integer b = 7
        in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  protecting TYPE .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op ___ : Type CallingMode Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList
           [assoc id:()] .
endfm
```

```
fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(integer need x, integer value y) 0
 .
parse
  (proc(integer name x, integer value y) 0) (2,3)
 .
parse
  (proc((integer -> integer) value x,
        integer reference y)
    (x(y))) (proc(integer value x) 2, 3)
 .
parse
  (proc((integer -> integer) value x,
        integer reference y)
    (x(y))) (proc(integer value x) y, 3)
 .
```

```
fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec integer x = 1
  in letrec integer x = 7, integer y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



---------------------
--- Type Checking ---
```

```
fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
          [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
  op _[_<-_] : TypeEnvironment Index Type -> TypeEnvironment .
  vars Ix Ix' : Index .  vars T T' : Type .  var TEnv : TypeEnvironment .
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv)[Ix <- T'] = [Ix,T'] TEnv .
  eq TEnv[Ix <- T'] = [Ix,T'] TEnv [owise] .
endfm

fmod STATE is
  protecting TYPE-ENVIRONMENT .
  op _[_] : State Index -> Type .
  op _[_<-_] : State Index Type -> State .
  var S : State .  var Ix : Index .  var T : Type .
  eq S[Ix] = S[tenv][Ix] .
  eq S[Ix <- T] = S[tenv <- S[tenv][Ix <- T]] .
endfm
```

```
fmod NAME-STATIC-TYPING is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op type : Name State -> Type .
  var X : Name .  var S : State .
  eq type(X, S) =  S[idx(X)] .
endfm

fmod GENERIC-EXP-STATIC-TYPING is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-STATIC-TYPING .
  op type : Exp State -> Type .
  op type : Exp -> Type .
  op typeEList : ExpList State -> Type .
  var I : Int .  var S : State .  var E : Exp .  var El : ExpList .
  eq type(I, S) = integer .
  eq type(E) = type(E, (tenv,empty)) .
  eq typeEList((), S) = nothing .
  eq typeEList((E,El), S) = type(E,S) * typeEList(El, S) .
endfm

red type(3) .
***> should be integer
red type(x) .
***> should be undefined
red type('variable) .
***> should be undefined
red type('variable,
     (tenv, [idx(x),integer] [idx('variable),integer * integer -> integer])) .
***> should be integer * integer -> integer

fmod ARITH-OPS-STATIC-TYPING is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  vars E E' : Exp .  var S : State .
 ceq type(E + E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E - E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E * E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E / E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
endfm

red type(3 + x, (tenv, [idx(x),integer])) .
***> should be integer
red type(3 + 'variable1,
       (tenv, [idx('variable), integer * integer -> integer])) .
```

```
***> should be undefined
red type(3 + 'variable2 + x * (y - z) + 'variable1 ,
        (tenv, [idx('variable2), integer]
               [idx(x), integer]
               [idx(y), integer]
               [idx(z), integer]
               [idx('variable1), integer])) .
***> should be integer

fmod BEXP-STATIC-TYPING is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op type : BExp State -> Type .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
 ceq type(E equals E', S) = bool if type(E, S) = type(E', S) .
 ceq type(zero?(E), S) = bool if type(E, S) = integer .
  eq type(not(Be), S) = type(Be, S) .
 ceq type(even?(E), S) = bool if type(E, S) = integer .
 ceq type(Be and Be', S) = bool if type(Be, S) = bool /\ type(Be', S) = bool .
endfm

red type(3 equals 3, S) .
***> should be bool
red type(3 equals 5, S) .
***> should be bool
red type(5 equals x, S) .
***> should be undefined
red type(3 + x equals 0, S) .
***> should be undefined

fmod IF-STATIC-TYPING is protecting IF-SYNTAX .
  extending BEXP-STATIC-TYPING .
  vars E E' : Exp .  var Be : BExp .  var S : State .
 ceq type(if Be then E else E', S) = type(E, S)
    if type(Be,S) = bool /\ type(E,S) = type(E',S) .
endfm

red type(if zero?(5) then 2 else 3) .
***> should be integer
red type(if zero?(0) then 2 else 3, S) .
***> should be integer
red type(if zero?(x) then y else z,
 (tenv, [idx(x), integer]
        [idx(y), integer * integer -> integer]
```

```
       [idx(z), integer * integer -> integer])) .
***> should be integer * integer -> integer
red type(
  if x equals y
  then x
  else if x equals z
       then z
       else y,
 (tenv, [idx(x),integer] [idx(y),integer] [idx(z),integer])) .
***> should be integer

fmod BINDING-STATIC-TYPING is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  var T : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[idx(X) <- T]) if type(E,S) = T .
endfm

fmod LET-STATIC-TYPING is extending LET-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq type(let Bl in E, S) = type(E, bindBList (Bl, S) in S) .
endfm

red type(
  let integer x = 5
  in x
) .
***> should be integer
red type(
  let integer x = 5, integer y = 7
  in x
) .
***> should be integer
red type(
  let integer x = 5
  in let integer y = x
     in y
) .
```

```
***> should be integer
red type(
  let integer x = 1
  in let integer x = 2
     in x
) .
***> should be integer
red type(
  let integer x = 10, integer y = 0, integer z = x
  in let integer a = 5, integer b = 7
        in z
) .
***> should be undefined

fmod PARAMETER-STATIC-TYPING is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op typePList : ParameterList -> Type .
  op bindPList : ParameterList State -> State .
  var S : State .  var T : Type .  var C : CallingMode .  var X : Name .
  var Pl : ParameterList .
  eq typePList(()) = nothing .
  eq typePList(T C X, Pl) = T * typePList(Pl) .
  eq bindPList((), S) = S .
  eq bindPList((T C X, Pl), S) = bindPList(Pl, S[idx(X) <- T]) .
endfm

fmod PROC-STATIC-TYPING is protecting PROC-SYNTAX .
  protecting PARAMETER-STATIC-TYPING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) = typePList(Pl) -> type(E, bindPList(Pl, S)) .
 ceq type(E El, S) = T if Tp -> T := type(E, S) /\ typeEList(El, S) = Tp .
endfm

red type(proc(integer need x, (integer * integer -> integer) value y) 0) .
***> should be integer * (integer * integer -> integer) -> integer

red type((proc(integer name x, integer value y) 0) (2,3)) .
***> should be integer

red type((proc(integer name x, (integer -> integer) value y) 0) (2,3)) .
***> should be undefined
```

```
red type((proc((integer -> integer) value x, integer reference y) (x(y)))
          (proc(integer value x) 2, 3)) .
***> should be integer

red type((proc(integer value x, integer reference y) (x(y)))
          (proc(integer value x) 2, 3)) .
***> should be undefined

red type((proc((integer -> integer) value x, integer reference y) (x(y)))
          (proc(integer value x) y, 3),
        (tenv, [idx(y), integer])) .
***> should be integer

fmod LETREC-STATIC-TYPING is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  op bindBlindlyBList : BindingList State -> State .
  var S : State .  var X : Name .  var E : Exp .
  var T : Type .  var Bl : BindingList .
  eq bindBlindlyBList(none,S) = S .
  eq bindBlindlyBList((T X = E, Bl), S) =
     bindBlindlyBList(Bl, S[idx(X) <- T]) .
  eq type(letrec Bl in E, S) =
     type(E, bindBList(Bl, bindBlindlyBList(Bl,S)) in S) .
endfm

red type(
  letrec integer x = 1
  in letrec integer x = 7, integer y = x
     in y
) .
***> should be integer

red type(
  letrec integer z = 1
  in letrec integer x = 7, integer y = x
     in y
) .
***> should be integer

red type(
  letrec integer z = 1
  in letrec integer z = 7, integer y = x
```

```
    in y
) .
***> should be undefined

fmod VAR-ASSIGNMENT-STATIC-TYPING is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var X : Name .  var E : Exp .  var S : State .
 ceq  type(set X = E, S) = nothing if type(X, S) = type(E, S) .
endfm

red type(set x = 3) .
***> should be undefined

fmod BLOCK-STATIC-TYPING is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq type({E}, S) = type(E, S) .
 ceq type({E ; El}, S) = type({El}, S)
    if type(E, S) == nothing or type(E, S) == integer .
endfm

red type({x ; y ; 3}) .
***> should be undefined
red type({x ; y ; 3 ; x}) .
***> should be undefined

fmod LOOP-STATIC-TYPING is protecting LOOP-SYNTAX .
  extending BEXP-STATIC-TYPING .
  var Be : BExp .  var E : Exp .  var S : State .
 ceq type(while Be E, S) = nothing
    if type(Be, S) = bool /\ type(E,S) = nothing .
endfm

fmod PROG-LANG-STATIC-TYPING is
  extending ARITH-OPS-STATIC-TYPING .
  extending IF-STATIC-TYPING .
  extending LET-STATIC-TYPING .
  extending PROC-STATIC-TYPING .
  extending LETREC-STATIC-TYPING .
  extending VAR-ASSIGNMENT-STATIC-TYPING .
  extending BLOCK-STATIC-TYPING .
  extending LOOP-STATIC-TYPING .
endfm
```

```
red type(
  let integer x = 5, integer y = 7
  in x + y
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = x + 2
     in x + 1
) .
***> should be integer

red type(
  let integer x = 1
  in let integer y = x + 2
     in x + 1
) .
***> should be integer

red type(
  let integer x = 1
  in let integer z = let integer y = x + 4
                     in y
     in z
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = let integer x = x + 4
                     in x
     in x
) .
***> should be integer

red type(
  let integer x = 1
  in (x + (let integer x = 10 in x))
) .
***> should be integer
```

```
red type(
  proc(integer value x, integer value y, integer value z) x * (y - z)
) .
***> should be integer * integer * integer -> integer

red type(
  (proc(integer value y, integer value z) y + 5 * z) (1,2)
) .
***> should be integer

red type(
  let (integer * integer -> integer)
       f = proc(integer value y, integer value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be integer

red type(
  (proc((integer -> integer) value x, integer value y) x(y))
    (proc(integer value z) 2 * z, 3)
) .
***> should be integer

red type(
  let ((integer -> integer) -> (integer -> integer))
       x = proc((integer -> integer) value x) x
  in x(x)
) .
***> should be undefined; cannot be typed

red type(
  let (integer * integer -> integer)
       f = proc(integer value x, integer value y) x + y,
     (integer * integer -> integer)
       g = proc(integer value x, integer value y) x * y,
     ((integer * integer -> integer) * (integer * integer -> integer)
                        * integer * integer -> integer)
       h = proc((integer * integer -> integer) value x,
                (integer * integer -> integer) value y,
                integer value a, integer value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be integer
```

```
red type(
  let integer y = 1
  in let (integer -> integer) f = proc(integer value x) y
     in let integer y = 2
        in f(0)
) .
***> should be integer

red type(
  let integer y = 1
  in (proc((integer -> integer) value x, integer value y) (x y))
      (proc(integer value x) y, 2)
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = 2,
         (integer * integer -> integer)
           f = proc (integer value y, integer value z) y + x * z
     in f(1,x)
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = 2,
         (integer * integer -> integer)
           f = proc(integer value y, integer value z) y + x * z,
         (integer -> integer)
           g = proc(integer value u) u + x
     in f(g(3), 4)
) .
***> should be integer

red type(
  let integer a = 3
  in let (integer -> integer) p = proc(integer value x) x + a,
         integer a = 5
     in a * p(2)
) .
***> should be integer
```

```
red type(
  let (integer -> integer)
       f = proc(integer value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined (but it would be integer under dynamic scoping)

red type(
  let (integer -> integer) f = proc(integer value n) n + n
  in let (integer -> integer)
          f = proc(integer value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be integer

red type(
  let integer a = 0
  in let integer a = 3, (nothing -> integer) p = proc() a
     in let integer a = 5,
            (integer -> integer)
---              f = proc(integer value x) (p())
             f = proc(integer value a) (p())
        in f(2)
) .
***> should be integer

red type(
  let ? 'makemult = proc(? value 'maker, integer value x)
                      if zero?(x)
                      then 0
                      else 4 + 'maker('maker, x - 1)
  in let (integer -> integer)
          'times4 = proc(integer value x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> no way to type this
```

```
red type(
  letrec (integer -> integer)
         f = proc(integer value n)
              if zero?(n)
              then 1
              else n * f(n - 1)
  in f(5)
) .
***> should be integer

red type(
  letrec (integer -> integer)
         'times4 = proc(integer value x)
                    if zero?(x)
                    then 0
                    else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be integer

red type(
  letrec (integer -> integer)
         'even = proc(integer value x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
        (integer -> integer)
          'odd = proc(integer value x)
                    if zero?(x)
                    then 0
                    else 'even(x - 1)
  in 'odd(17)
) .
***> should be integer

red type(
  let integer x = 1
  in letrec integer x = 7,
            integer y = x
     in y
) .
***> should be integer, though its value is undefined
```

```
red type(
  let integer x = 10
  in letrec (integer -> integer)
            f = proc(integer value y) if zero?(y) then x else f(y - 1)
     in let integer x = 20
        in f(5)
) .
***> should be integer

red type(
  let integer c = 0
  in let (nothing -> integer)
          f = proc()
                let integer c = c + 1
                in c
     in f() + f()
) .
***> should be integer

red type(
  let (nothing -> integer)
        f = let integer c = 0
            in proc()
                let integer c = c + 1
                in c
  in f() + f()
) .
***> should be integer; should be undefined under dynamic scoping

red type(
  let integer c = 0
  in let (nothing -> integer)
          f = proc()
                let nothing d = set c = c + 1
                in c
     in f() + f()
) .
***> should be integer

red type(
  let (nothing -> integer)
        f = let integer c = 0
```

```
          in proc()
              let nothing d = set c = c + 1
              in c
  in f() + f()
) .
***> should be integer; should be undefined under dynamic scoping

red type(
  let integer x = 0
  in let (integer -> integer)
         f = proc (integer value x)
              let nothing d = set x = x + 1
              in x
     in f(x) + f(x)
) .
***> should be 2

red type(
  let integer x = 0, integer y = 1
  in let (integer * integer -> integer)
         f = proc(integer value x, integer value y)
              let integer t = x
              in let nothing d = set x = y
                 in let nothing d = set y = t
                    in 0
     in let integer d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red type(
  let integer x = 0, integer y = 3, integer z = 4,
      (integer * integer * integer -> integer)
        f = proc(integer value a, integer value b, integer value c)
             if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be integer, but undefined as a value

red type(
  let integer x = 0, integer y = 3, integer z = 4,
      (integer * integer * integer -> integer)
        f = proc(integer value a, integer need b, integer need c)
```

```
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be integer

red type(
  let integer x = 0
  in letrec
      (nothing -> integer)
        'even = proc() if zero?(x)
                  then 1
                  else let nothing d = set x = x - 1
                      in 'odd(),
      (nothing -> integer)
        'odd  = proc() if zero?(x)
                  then 0
                  else let nothing d = set x = x - 1
                      in 'even()
    in let nothing d = set x = 7
       in 'odd()
) .
***> should be integer

red type(
  letrec integer x = 18,
        (nothing -> integer)
          'even = proc() if zero?(x) then 1
                    else let nothing d = set x = x - 1
                        in 'odd(),
        (nothing -> integer)
          'odd = proc() if zero?(x) then 0
                    else let nothing d = set x = x - 1
                        in 'even()
  in 'odd()
) .
***> should be integer

red type(
  let integer x = 3, integer y = 4
  in let nothing d = set x = x + y
     in let nothing d = set y = x - y
        in let nothing d = set x = x - y
           in 2 * x + y
```

```
) .
***> should be integer

red type(
  let integer x = 3, integer y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be integer

red type(
  let integer 'times4 = 0
---  let (integer -> integer) 'times4 = proc(integer value x) 0
  in {
       set 'times4 = proc(integer value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be undefined
---***> should be integer

red type(
  let integer x = 3, integer y = 4,
      (integer * integer -> nothing)
       f = proc(integer reference a, integer reference b)
           {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
           }
  in {
       f(x,y) ;
       x
     }
) .
***> should be integer

red type(
```

```
  let (integer -> integer) f = proc(integer need x) x + x
  in let integer y = 5
     in {
          f({set y = y + 3 ; 0}) ;
          y
        }
) .
***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer need x) x + x,
      (integer -> integer) g = proc(integer reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
***> should be integer

red type(
  let (integer -> integer)
        f = proc(integer name x) x + x
  in let integer y = 5
     in {
          f({set y = y + 3 ; 0}) ;
          y
        }
) .
***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer name x) x + x,
      (integer -> integer) g = proc(integer reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
***> should be integer

red type(
```

```
  let integer n = 178378342647, integer c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be integer

---------------

***> the following cannot be typed

red type(
  let ? f = proc(integer value x, ? value g)
              if zero?(x)
              then 1
              else x * g(x - 1, g)
  in f(5, f)
) .

red type(
  let integer x = 17,
      ?  'odd = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 0
                  else e(x - 1, o, e),
      ? 'even = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 1
                  else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
```

```
************************************************************************
*** Defining a Static Type Checker for a Functional Programming Language ***
************************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op '(') : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse 3 + x .
parse 3 + 'variable1 .
parse 3 + 'variable2 + x * (y - z) + 'variable1 .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 equals 3 .
parse 3 equals 5 .
parse 5 equals x .
parse 3 + x equals 0 .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5) then 2 else 3 .
parse if zero?(0) then 2 else 3 .
parse if zero?(x) then y else z .
parse if x equals y
      then x
      else if x equals z
           then z
           else y .
```

```
fmod TYPE is
  sorts BasicType Type .
  subsort BasicType < Type .
  ops integer bool : -> BasicType .
  op nothing : -> Type .
  op _*_ : Type Type -> Type [assoc id: nothing prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList
           [assoc id: none prec 71] .
  op __=_ : Type Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let integer x = 5
  in x
 .
parse
  let integer x = 5, integer y = 7
  in x
 .
parse
  let integer x = 5
  in let integer y = x
     in y
 .
parse
  let integer x = 1
  in let integer x = 2
     in x
 .
parse
  let integer x = 10, integer y = 0, integer z = x
  in let integer a = 5, integer b = 7
         in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  protecting TYPE .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op ___ : Type CallingMode Name -> Parameter [prec 0] .
  op '(') : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList
           [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm
```

```
fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(integer need x, integer value y) 0
 .
parse
  (proc(integer name x, integer value y) 0) (2,3)
 .
parse
  (proc((integer -> integer) value x,
        integer reference y)
    (x(y))) (proc(integer value x) 2, 3)
 .
parse
  (proc((integer -> integer) value x,
        integer reference y)
    (x(y))) (proc(integer value x) y, 3)
 .

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec integer x = 1
  in letrec integer x = 7, integer y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
```

```
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


---------------------
--- Type Checking ---
---------------------

fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
          [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
  op _[_<-_] : TypeEnvironment Index Type -> TypeEnvironment .
  vars Ix Ix' : Index .  vars T T' : Type .  var TEnv : TypeEnvironment .
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv)[Ix <- T'] = [Ix,T'] TEnv .
  eq TEnv[Ix <- T'] = [Ix,T'] TEnv [owise] .
endfm

fmod STATE is
  protecting TYPE-ENVIRONMENT .
  op _[_] : State Index -> Type .
  op _[_<-_] : State Index Type -> State .
  var S : State .  var Ix : Index .  var T : Type .
  eq S[Ix] = S[tenv][Ix] .
  eq S[Ix <- T] = S[tenv <- S[tenv][Ix <- T]] .
endfm

fmod NAME-STATIC-TYPING is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op type : Name State -> Type .
  var X : Name .  var S : State .
  eq type(X, S) =  S[idx(X)] .
endfm
```

```
fmod GENERIC-EXP-STATIC-TYPING is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-STATIC-TYPING .
  op type : Exp State -> Type .
  op type : Exp -> Type .
  op typeEList : ExpList State -> Type .
  var I : Int .  var S : State .  var E : Exp .  var El : ExpList .
  eq type(I, S) = integer .
  eq type(E) = type(E, (tenv,empty)) .
  eq typeEList((), S) = nothing .
  eq typeEList((E,El), S) = type(E,S) * typeEList(El, S) .
endfm

red type(3) .
***> should be integer
red type(x) .
***> should be undefined
red type('variable) .
***> should be undefined
red type('variable,
      (tenv, [idx(x),integer] [idx('variable),integer * integer -> integer])) .
***> should be integer * integer -> integer

fmod ARITH-OPS-STATIC-TYPING is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  vars E E' : Exp .  var S : State .
 ceq type(E + E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E - E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E * E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
 ceq type(E / E', S) = integer if type(E,S) = integer /\ type(E',S) = integer .
endfm

red type(3 + x, (tenv, [idx(x),integer])) .
***> should be integer
red type(3 + 'variable1,
         (tenv, [idx('variable), integer * integer -> integer])) .
***> should be undefined
red type(3 + 'variable2 + x * (y - z) + 'variable1 ,
         (tenv, [idx('variable2), integer]
                [idx(x), integer]
                [idx(y), integer]
                [idx(z), integer]
                [idx('variable1), integer])) .
***> should be integer

fmod BEXP-STATIC-TYPING is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op type : BExp State -> Type .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
 ceq type(E equals E', S) = bool if type(E, S) = type(E', S) .
 ceq type(zero?(E), S) = bool if type(E, S) = integer .
  eq type(not(Be), S) = type(Be, S) .
 ceq type(even?(E), S) = bool if type(E, S) = integer .
 ceq type(Be and Be', S) = bool if type(Be, S) = bool /\ type(Be', S) = bool .
endfm

red type(3 equals 3, S) .
***> should be bool
red type(3 equals 5, S) .
***> should be bool
red type(5 equals x, S) .
***> should be undefined
red type(3 + x equals 0, S) .
***> should be undefined

fmod IF-STATIC-TYPING is protecting IF-SYNTAX .
  extending BEXP-STATIC-TYPING .
  vars E E' : Exp .  var Be : BExp .  var S : State .
 ceq type(if Be then E else E', S) = type(E, S)
    if type(Be,S) = bool /\ type(E,S) = type(E',S) .
endfm

red type(if zero?(5) then 2 else 3) .
***> should be integer
red type(if zero?(0) then 2 else 3, S) .
***> should be integer
red type(if zero?(x) then y else z,
 (tenv, [idx(x), integer]
        [idx(y), integer * integer -> integer]
        [idx(z), integer * integer -> integer])) .
***> should be integer * integer -> integer
red type(
  if x equals y
  then x
  else if x equals z
       then z
       else y,
 (tenv, [idx(x),integer] [idx(y),integer] [idx(z),integer])) .
***> should be integer

fmod BINDING-STATIC-TYPING is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  var T : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[idx(X) <- T]) if type(E,S) = T .
endfm

fmod LET-STATIC-TYPING is extending LET-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq type(let Bl in E, S) = type(E, bindBList (Bl, S) in S) .
endfm

red type(
  let integer x = 5
  in x
) .
***> should be integer
red type(
  let integer x = 5, integer y = 7
  in x
) .
***> should be integer
red type(
  let integer x = 5
  in let integer y = x
     in y
) .
***> should be integer
red type(
  let integer x = 1
  in let integer x = 2
     in x
) .
***> should be integer
red type(
  let integer x = 10, integer y = 0, integer z = x
  in let integer a = 5, integer b = 7
         in z
) .
***> should be undefined

fmod PARAMETER-STATIC-TYPING is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-STATIC-TYPING .
  op typePList : ParameterList -> Type .
  op bindPList : ParameterList State -> State .
  var S : State .  var T : Type .  var C : CallingMode .  var X : Name .
  var Pl : ParameterList .
  eq typePList(()) = nothing .
  eq typePList(T C X, Pl) = T * typePList(Pl) .
  eq bindPList((), S) = S .
  eq bindPList((T C X, Pl), S) = bindPList(Pl, S[idx(X) <- T]) .
endfm

fmod PROC-STATIC-TYPING is protecting PROC-SYNTAX .
  protecting PARAMETER-STATIC-TYPING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) = typePList(Pl) -> type(E, bindPList(Pl, S)) .
 ceq type(E El, S) = T if Tp -> T := type(E, S) /\ typeEList(El, S) = Tp .
endfm

red type(proc(integer need x, (integer * integer -> integer) value y) 0) .
***> should be integer * (integer * integer -> integer) -> integer

red type((proc(integer name x, integer value y) 0) (2,3)) .
***> should be integer

red type((proc(integer name x, (integer -> integer) value y) 0) (2,3)) .
***> should be undefined

red type((proc((integer -> integer) value x, integer reference y) (x(y)))
         (proc(integer value x) 2, 3)) .
***> should be integer

red type((proc(integer value x, integer reference y) (x(y)))
         (proc(integer value x) 2, 3)) .
***> should be undefined

red type((proc((integer -> integer) value x, integer reference y) (x(y)))
         (proc(integer value x) y, 3),
        (tenv, [idx(y), integer])) .
***> should be integer

fmod LETREC-STATIC-TYPING is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  op bindBlindlyBList : BindingList State -> State .
  var S : State .  var X : Name .  var E : Exp .
  var T : Type .  var Bl : BindingList .
  eq bindBlindlyBList(none,S) = S .
  eq bindBlindlyBList((T X = E, Bl), S) =
     bindBlindlyBList(Bl, S[idx(X) <- T]) .
  eq type(letrec Bl in E, S) =
     type(E, bindBList(Bl, bindBlindlyBList(Bl,S)) in S) .
endfm

red type(
  letrec integer x = 1
  in letrec integer x = 7, integer y = x
     in y
) .
***> should be integer

red type(
  letrec integer z = 1
  in letrec integer x = 7, integer y = x
     in y
) .
***> should be integer

red type(
  letrec integer z = 1
  in letrec integer z = 7, integer y = x
     in y
) .
***> should be undefined

fmod VAR-ASSIGNMENT-STATIC-TYPING is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var X : Name .  var E : Exp .  var S : State .
 ceq  type(set X = E, S) = nothing if type(X, S) = type(E, S) .
endfm

red type(set x = 3) .
***> should be undefined

fmod BLOCK-STATIC-TYPING is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq type({E}, S) = type(E, S) .
 ceq type({E ; El}, S) = type({El}, S)
    if type(E, S) == nothing or type(E, S) == integer .
endfm

red type({x ; y ; 3}) .
***> should be undefined
red type({x ; y ; 3 ; x}) .
***> should be undefined

fmod LOOP-STATIC-TYPING is protecting LOOP-SYNTAX .
  extending BEXP-STATIC-TYPING .
  var Be : BExp .  var E : Exp .  var S : State .
 ceq type(while Be E, S) = nothing
    if type(Be, S) = bool /\ type(E,S) = nothing .
endfm

fmod PROG-LANG-STATIC-TYPING is
  extending ARITH-OPS-STATIC-TYPING .
  extending IF-STATIC-TYPING .
  extending LET-STATIC-TYPING .
  extending PROC-STATIC-TYPING .
  extending LETREC-STATIC-TYPING .
  extending VAR-ASSIGNMENT-STATIC-TYPING .
  extending BLOCK-STATIC-TYPING .
  extending LOOP-STATIC-TYPING .
endfm

red type(
  let integer x = 5, integer y = 7
  in x + y
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = x + 2
     in x + 1
) .
***> should be integer
```

```
red type(
  let integer x = 1
  in let integer y = x + 2
     in x + 1
) .
***> should be integer

red type(
  let integer x = 1
  in let integer z = let integer y = x + 4
                     in y
     in z
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = let integer x = x + 4
                     in x
     in x
) .
***> should be integer

red type(
  let integer x = 1
  in (x + (let integer x = 10 in x))
) .
***> should be integer

red type(
  proc(integer value x, integer value y, integer value z) x * (y - z)
) .
***> should be integer * integer * integer -> integer

red type(
  (proc(integer value y, integer value z) y + 5 * z) (1,2)
) .
***> should be integer

red type(
  let (integer * integer -> integer)
       f = proc(integer value y, integer value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be integer

red type(
  (proc((integer -> integer) value x, integer value y) x(y))
    (proc(integer value z) 2 * z, 3)
) .
***> should be integer

red type(
  let ((integer -> integer) -> (integer -> integer))
       x = proc((integer -> integer) value x) x
  in x(x)
) .
***> should be undefined; cannot be typed

red type(
  let (integer * integer -> integer)
       f = proc(integer value x, integer value y) x + y,
      (integer * integer -> integer)
       g = proc(integer value x, integer value y) x * y,
      ((integer * integer -> integer) * (integer * integer -> integer)
                                      * integer * integer -> integer)
       h = proc((integer * integer -> integer) value x,
                (integer * integer -> integer) value y,
                integer value a, integer value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be integer

red type(
  let integer y = 1
  in let (integer -> integer) f = proc(integer value x) y
     in let integer y = 2
        in f(0)
) .
***> should be integer

red type(
  let integer y = 1
  in (proc((integer -> integer) value x, integer value y) (x y))
       (proc(integer value x) y, 2)
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = 2,
         (integer * integer -> integer)
           f = proc (integer value y, integer value z) y + x * z
     in f(1,x)
) .
***> should be integer

red type(
  let integer x = 1
  in let integer x = 2,
         (integer * integer -> integer)
           f = proc(integer value y, integer value z) y + x * z,
         (integer -> integer)
           g = proc(integer value u) u + x
     in f(g(3), 4)
) .
***> should be integer

red type(
  let integer a = 3
  in let (integer -> integer) p = proc(integer value x) x + a,
         integer a = 5
     in a * p(2)
) .
***> should be integer

red type(
  let (integer -> integer)
       f = proc(integer value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be undefined (but it would be integer under dynamic scoping)

red type(
  let (integer -> integer) f = proc(integer value n) n + n
  in let (integer -> integer)
          f = proc(integer value n)
                if zero?(n)
                then 1
                else n * f(n - 1)
     in f(5)
) .
***> should be integer

red type(
  let integer a = 0
  in let integer a = 3, (nothing -> integer) p = proc() a
     in let integer a = 5,
            (integer -> integer)
---              f = proc(integer value x) (p())
               f = proc(integer value a) (p())
        in f(2)
) .
***> should be integer

red type(
  let ? 'makemult = proc(? value 'maker, integer value x)
                      if zero?(x)
                      then 0
                      else 4 + 'maker('maker, x - 1)
  in let (integer -> integer)
         'times4 = proc(integer value x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> no way to type this

red type(
  letrec (integer -> integer)
           f = proc(integer value n)
                 if zero?(n)
                 then 1
                 else n * f(n - 1)
  in f(5)
) .
***> should be integer

red type(
  letrec (integer -> integer)
           'times4 = proc(integer value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be integer

red type(
  letrec (integer -> integer)
           'even = proc(integer value x)
                     if zero?(x)
                     then 1
                     else 'odd(x - 1),
         (integer -> integer)
           'odd = proc(integer value x)
                     if zero?(x)
                     then 0
                     else 'even(x - 1)
  in 'odd(17)
) .
***> should be integer

red type(
  let integer x = 1
  in letrec integer x = 7,
            integer y = x
     in y
) .
***> should be integer, though its value is undefined

red type(
  let integer x = 10
  in letrec (integer -> integer)
              f = proc(integer value y) if zero?(y) then x else f(y - 1)
     in let integer x = 20
        in f(5)
) .
***> should be integer

red type(
  let integer c = 0
  in let (nothing -> integer)
           f = proc()
                 let integer c = c + 1
                 in c
     in f() + f()
) .
***> should be integer

red type(
  let (nothing -> integer)
       f = let integer c = 0
           in proc()
                let integer c = c + 1
                in c
  in f() + f()
) .
***> should be integer; should be undefined under dynamic scoping

red type(
  let integer c = 0
  in let (nothing -> integer)
           f = proc()
                 let nothing d = set c = c + 1
                 in c
     in f() + f()
) .
***> should be integer

red type(
  let (nothing -> integer)
       f = let integer c = 0
           in proc()
                let integer c = c + 1
                in c
  in f() + f()
) .
***> should be integer; should be undefined under dynamic scoping

red type(
  let integer x = 0
  in let (integer -> integer)
           f = proc (integer value x)
                 let nothing d = set x = x + 1
                 in x
     in f(x) + f(x)
) .
***> should be 2

red type(
  let integer x = 0, integer y = 1
```

```
  in let (integer * integer -> integer)
          f = proc(integer value x, integer value y)
                let integer t = x
                in let nothing d = set x = y
                   in let nothing d = set y = t
                      in 0
     in let integer d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red type(
  let integer x = 0, integer y = 3, integer z = 4,
      (integer * integer * integer -> integer)
        f = proc(integer value a, integer value b, integer value c)
              if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be integer, but undefined as a value

red type(
  let integer x = 0, integer y = 3, integer z = 4,
      (integer * integer * integer -> integer)
        f = proc(integer value a, integer need b, integer need c)
              if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be integer

red type(
  let integer x = 0
  in letrec
       (nothing -> integer)
         'even = proc() if zero?(x)
                        then 1
                        else let nothing d = set x = x - 1
                             in 'odd(),
       (nothing -> integer)
         'odd  = proc() if zero?(x)
                        then 0
                        else let nothing d = set x = x - 1
                             in 'even()
     in let nothing d = set x = 7
        in 'odd()
) .
***> should be integer

red type(
  letrec integer x = 18,
         (nothing -> integer)
           'even = proc() if zero?(x) then 1
                          else let nothing d = set x = x - 1
                               in 'odd(),
         (nothing -> integer)
            'odd = proc() if zero?(x) then 0
                          else let nothing d = set x = x - 1
                               in 'even()
  in 'odd()
) .
***> should be integer

red type(
  let integer x = 3, integer y = 4
  in let nothing d = set x = x + y
     in let nothing d = set y = x - y
        in let nothing d = set x = x - y
           in 2 * x + y
) .
***> should be integer

red type(
  let integer x = 3, integer y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be integer

red type(
  let integer 'times4 = 0
---  let (integer -> integer) 'times4 = proc(integer value x) 0
  in {
       set 'times4 = proc(integer value x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be undefined
---***> should be integer

red type(
  let integer x = 3, integer y = 4,
      (integer * integer -> nothing)
        f = proc(integer reference a, integer reference b)
            {
              set a = a + b ;
              set b = a - b ;
              set a = a - b
            }
  in {
       f(x,y) ;
       x
     }
) .
***> should be integer

red type(
  let (integer -> integer) f = proc(integer need x) x + x
  in let integer y = 5
     in {
          f({set y = y + 3 ; 0}) ;
          y
        }
) .
***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer need x) x + x,
      (integer -> integer) g = proc(integer reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer name x) x + x
  in let integer y = 5
     in {
          f({set y = y + 3 ; 0}) ;
          y
        }
) .
***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer name x) x + x,
      (integer -> integer) g = proc(integer reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
***> should be integer

red type(
  let integer n = 178378342647, integer c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be integer

---------------

***> the following cannot be typed

red type(
  let ? f = proc(integer value x, ? value g)
              if zero?(x)
              then 1
              else x * g(x - 1, g)
  in f(5, f)
) .

red type(
  let integer x = 17,
      ?  'odd = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 0
                  else e(x - 1, o, e),
      ? 'even = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 1
                  else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
```

CS322 - Programming Language Design

Lecture 11: Typed Languages - Static Type Checking (Part 2) and Dynamic Type Checking

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Notes: The homework problem in Lecture 10 has been transformed into an extra credit problem. The reason for doing so is that I realized that it was too hard for a regular homework problem.

I have also transformed the last problem in Lecture 04 into an extra-credit problem. The `letrec` extract-edit problem in Lecture 09 is simple, so it only gives you 2 extra points.

The policy regarding the extra-credit problems is the following:

- Extra credit problems can be turned in at any time.
- Unless otherwise specified, they represent 5 points from your final score. More precisely, I will add 5 extra points to your final score at the end of the class for each extra credit problem that you solve.
- The regular coursework sums up to 100 points (without the extra-credit problems).

Typing of Letrec

Typing of `letrec` is relatively easy once we have the current infrastructure. In order to find its type, one

- first *blindly* binds all its names to their declared types,
- then checks the type consistency of each of the bound expressions to its corresponding name's type, which can be quickly done by using the operator `bindBList(_,_)_in_` defined in `BINDING-STATIC-TYPING`, and
- then, if each of those are correctly typed, returns the type of its body expression in the new state:


```
fmod LETREC-STATIC-TYPING is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  protecting BINDING-STATIC-TYPING .
  op bindBlindlyBList : BindingList State -> State .
  var S : State .  var X : Name .  var E : Exp .
  var T : Type .  var Bl : BindingList .
  eq bindBlindlyBList(none,S) = S .
  eq bindBlindlyBList((T X = E, Bl), S) =
     bindBlindlyBList(Bl, S[idx(X) <- T]) .
  eq type(letrec Bl in E, S) =
     type(E, bindBList(Bl, bindBlindlyBList(Bl,S)) in S) .
endfm
```

We could have defined `bindBlindlyBList` in the module `BINDING-STATIC-TYPING`, but it would have looked rather artificial there because only `letrec` seems to need it.

```
red type(letrec integer z = 1
         in letrec integer x = 7, integer y = x in y) .  ***> ?
```

Typing of Variable Assignments

A variable assignment statement should not modify the type of the name on which it applies its side effect. Also, even though we decided to evaluate assignment statements to the integer 1, we now type them to nothing, so the type system will signal if one uses them as integers:

```
fmod VAR-ASSIGNMENT-STATIC-TYPING is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var X : Name .  var E : Exp .  var S : State .
 ceq  type(set X = E, S) = nothing if type(X, S) = type(E, S) .
endfm

red type(set x = 3) .
***> should be undefined
```


Typing of Blocks

One can choose among various design decisions for how to type blocks. However, we decide here that the type of a block is the type of its last expression, provided that all the other expressions occurring in the block are used only for their side effects, so their type is nothing:

```
fmod BLOCK-STATIC-TYPING is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-STATIC-TYPING .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq type({E}, S) = type(E, S) .
 ceq type({E ; El}, S) = type({El}, S)
    if type(E, S) == nothing or type(E, S) == integer .
endfm

red type({x ; y ; 3}) .  ***> should be undefined
```

Typing of Loops

In the case of loops, our design decision in typing is to require that they are used entirely for their side effects, meaning that their body should type to `nothing`. If this is the case, then the type of the entire loop is `nothing`.

```
fmod LOOP-STATIC-TYPING is protecting LOOP-SYNTAX .
  extending BEXP-STATIC-TYPING .
  var Be : BExp .  var E : Exp .  var S : State .
 ceq type(while Be E, S) = nothing
    if type(Be, S) = bool /\ type(E,S) = nothing .
endfm
```


Putting All the Typing Together

There is not much to say here. We just put all the typing modules together and get an executable static type checker for our modified language:

```
fmod PROG-LANG-STATIC-TYPING is
  extending ARITH-OPS-STATIC-TYPING .
  extending IF-STATIC-TYPING .
  extending LET-STATIC-TYPING .
  extending PROC-STATIC-TYPING .
  extending LETREC-STATIC-TYPING .
  extending VAR-ASSIGNMENT-STATIC-TYPING .
  extending BLOCK-STATIC-TYPING .
  extending LOOP-STATIC-TYPING .
endfm
```

Examples

One can now type check hypothetical programs written in our designed programming language. There are LOTS of such programs in the file type-checking.maude.maude. Read, execute and understand them; similar problems will be given at final. Let us see some interesting examples next.

```
red type(
  let integer x = 1
  in let integer y = x + 2
     in x + 1
) .  ***> should be integer

red type(
  let integer x = 1
  in let integer z = let integer y = x + 4
```



```
                     in y
     in z
) .  ***> should be integer

red type(
  proc(integer value x, integer value y, integer value z) x * (y - z)
) .  ***> should be integer * integer * integer -> integer

red type(
  (proc(integer value y, integer value z) y + 5 * z) (1,2)
) .  ***> should be integer

red type(
  let ((integer -> integer) -> (integer -> integer))
        x = proc((integer -> integer) value x) x
  in x(x)
) .
***> should be undefined; cannot be typed
```

```
red type(
  let (integer * integer -> integer)
        f = proc(integer value x, integer value y) x + y,
      (integer * integer -> integer)
        g = proc(integer value x, integer value y) x * y,
      ((integer * integer -> integer) * (integer * integer -> integer)
                           * integer * integer -> integer)
        h = proc((integer * integer -> integer) value x,
                 (integer * integer -> integer) value y,
                 integer value a, integer value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .  ***> should be integer

red type(
  let integer y = 1
  in (proc((integer -> integer) value x, integer value y) (x y))
       (proc(integer value x) y, 2)
```



```
) .  ***> should be integer

red type(
  let (integer -> integer)
        f = proc(integer value n)
              if zero?(n)
              then 1
              else n * f(n - 1)
  in f(5)
) .  ***> should be undefined (but it would be integer under dynamic scoping)

red type(
  let integer x = 1
  in letrec integer x = 7,
            integer y = x
     in y
) .  ***> should be integer, though its value is undefined
```

```
red type(
  let integer x = 0, integer y = 3, integer z = 4,
      (integer * integer * integer -> integer)
       f = proc(integer value a, integer value b, integer value c)
             if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be integer, but undefined as a value

red type(
  letrec integer x = 18,
         (nothing -> integer)
           'even = proc() if zero?(x) then 1
                          else let nothing d = set x = x - 1
                               in 'odd(),
         (nothing -> integer)
            'odd = proc() if zero?(x) then 0
                          else let nothing d = set x = x - 1
```



```
                               in 'even()
  in 'odd()
) .  ***> should be integer

red type(
  let integer x = 3, integer y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .  ***> should be integer

red type(
  let integer 'times4 = 0
---  let (integer -> integer) 'times4 = proc(integer value x) 0
  in {
       set 'times4 = proc(integer value x)
                       if zero?(x)
```

```
                    then 0
                    else 4 + 'times4(x - 1) ;
      'times4(3)
    }
) .  ***> should be undefined
---***> should be integer

red type(
  let integer y = 5,
      (integer -> integer) f = proc(integer name x) x + x,
      (integer -> integer) g = proc(integer reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
***> should be integer
```



```
red type(
  let integer n = 178378342647, integer c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .  ***> should be integer

red type(
  let ? f = proc(integer value x, ? value g)
              if zero?(x)
              then 1
              else x * g(x - 1, g)
  in f(5, f)
) .  ***> cannot be typed
```

```
red type(
  let integer x = 17,
      ?  'odd = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 0
                  else e(x - 1, o, e),
      ? 'even = proc(integer value x, ? value o, ? value e)
                  if zero?(x) then 1
                  else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .  ***> cannot be typed
```

Homework Exercise 1 *Define the typing of* `let*` *and add it to the module* `PROG-LANG-STATIC-TYPING`.

Dynamic Type Checking

As we have seen, static type checking may *limit* the way we write programs. Its major advantage is to allow an *untyped memory model* for strongly typed programming languages this way attaining highly *efficient program executions*.

A second important advantage of static type checking is that it gives quick and efficient *feedback* to users, so they can remove certain errors at a relatively early stage. However, this is the role of all software analysis tools after all; a type checker is among the simplest analysis tools.

However, today's computers are becoming extremely fast, making the few extra computing cycles of runtime overhead needed for dynamic type checking less important. Moreover, it is expected that the number of *processors per computer will double every 6*

months or so, which means that a potentially significant computing power could be poorly used. Future programming languages may/should/will take advantage of that extra-power to increase the reliability of software developed using those languages.

One way to use a parallel computing architecture in the design of a programming language, besides the straightforward execution of parts of the program in parallel (if the language provides support for concurrency), is to use a *parallel execution thread which only performs the dynamic type checking*, while the main thread assumes an untyped memory model so it stays fast. The dynamic type checking thread can throw a runtime exception whenever a typing inconsistency is found. This way, one can obtain type safe execution of programs with very low or no overhead at all.

In what follows, we extend our current functional programming language to perform dynamic type checking. If the type safety is violated then the evaluating program terminates and an undefined

value is returned. Notice that the current design *implicitly provides dynamic typing* for the already existing features. Indeed, both expressions

```
let f = proc(value x) x
in f(1,2)

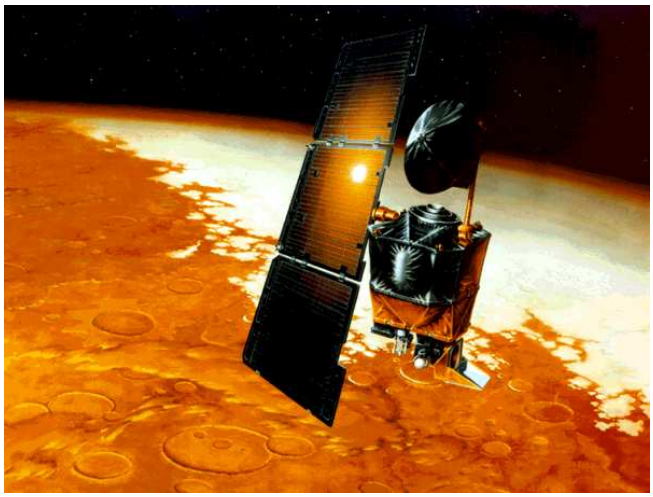
let f = proc(value x) x(x)
in f(1)
```

evaluate to undefined values. In order to non-trivially exemplify dynamic type checking, we will extend our language with *units of measurement*.

Units of Measurement

Numerical values can be more than just "integers" in many important applications. For example, `x = 2` can mean "x is 2 hours", while `y = 3` can mean "y is 3 meters", in which case it is meaningless to calculate `x + y`. There are many applications where physical units play a crucial role. In the context of a programming language providing support for units of measurement, the challenge is to ensure that *units are consistently used* in programs.

Inconsistent use of units of measurement can have disastrous consequences. For example, on 19 June 1985, NASA's space shuttle Discovery flew upside down Maui, in an attempt to point the mirror to a spot 10,023 *nautical miles* above see level; that number was supplied in units of *feet* and then fed into the on-board guidance system, which unfortunately was expecting units in nautical miles. The shuttle was eventually recovered.



As if there was no lesson to be learned from this, on 30 September 1999, NASA's Mars Climate Orbiter spacecraft crashed into Mar's atmosphere due to a software navigation error. Peer review findings indicate that *one programming team used metric units while another used English units* for a key spacecraft operation.

Adding Units of Measurement to a Language

One way to add units of measurement to a language is to add them as new types, so whenever an variable is defined it is also assigned a unit. This is the approach taken in cs497gr. In this class we take a simpler approach, namely to tag each constant with a unit. We use the operation `_:_ : Int Unit -> Int:Unit` for this. Then units can be propagated and associated to each integer value calculated by a program. For example, the expression

```
let y = 1 : meter
in let f = proc(value x) y
   in let y = 2 : second
      in f(0 : meter second)
```

evaluates to `int(1 : meter)` under static scoping and to `int(2 : second)` under dynamic scoping. However, despite the fact that it would evaluate to the numeric value `int(11)` under normal,

dynamically untyped evaluation, the following expression will *not* evaluate, so will be considered undefined, under dynamic typing for units of measurement:

```
let x = 1 : second
in (x + (let x = 10 : meter in x))
```

While one can imagine more or less effective static type checkers for units of measurement, the problem of whether a program is unit safe is in general undecidable. The reason is that one can easily translate almost any mathematical property on integers into a type safety property on units. For example, one can use a conditional of the form

```
if (problem P is true) then (unit safe code)
                       else (unit unsafe code)
```

for some well-known hard to prove problem `P` that is true on integers. A static type checker would have to essentially prove `P` in order to conclude that such code is correct.

To understand the difficulties that a static type checker for units would encounter, let us consider the following familiar example:

```
letrec f = proc(value n)
             if zero?(n)
             then 1 : noUnit
             else n * f(n - 1 : meter)
in let x = f,
       y = proc(value x, value g)
            if x equals 1 : meter
            then x
            else x * g(x - 1 : meter, g),
       n = 10 : meter
   in x(n) + y(n,y)
```

In order to certify the expression `x(n) + y(n,y)`, a static type checker would have to show that they have the same unit, namely `meter^10` in this case. This is clearly *hard to show* mechanically.

In order to add units to our functional programming language, the idea is to replace the previous integers by pairs `<integer> : <unit>`, as well as the previous integer stored values by `int(<integer> : <unit>)`. Then whenever an operation requiring compatible units is evaluated, we first have to check for units consistency. This can be done very elegantly due to the already existing modular definition of the semantics.

Syntax for Attaching Units to Constants

We first define a module, called `INT:UNIT-SYNTAX`, giving the syntax for units of measurement. Starting with some basic units, combined units can be obtained by applying the operations of power and multiplication. To keep the syntax simple, we use concatenation for multiplication:

```
fmod INT:UNIT-SYNTAX is protecting INT .
  sorts Unit Int:Unit .
  op _:_ : Int Unit -> Int:Unit [prec 3] .
  ops meter kilometer foot yard mile : -> Unit .
  ops second minute hour : -> Unit .
  ops gram kilogram : -> Unit .
  ops celsius fahrenheit : -> Unit .
  op newton : -> Unit .
  op noUnit : -> Unit .
  op __ : Unit Unit -> Unit [assoc comm prec 2] .
  op _^_ : Unit Int -> Unit [prec 1] .
endfm
```

Notice that `INT:UNIT` is just a token, and so is `_:_`. `noUnit` stays for the generic unitless unit. When we define the semantics of units, we will see that, for example, `meter^0` is `noUnit`.

The *only* modification that we need to make to the already existing syntax, is to import `INT:UNIT-SYNTAX` instead of `INT` in the module

`GENERIC-EXP-SYNTAX`, and also to define `Int:Unit` instead of `Int` as a subsort of `Exp`:

```
fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT:UNIT-SYNTAX .
  sorts Exp ExpList .
  subsorts Int:Unit Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm
```

Defining a Dynamic Type Checker for Units

Due to the original modular and abstract design, *nothing* needs to change in the already existing state infrastructure, and very little needs to be changed in the already existing executable semantics.

We will need to do the following:

- Add semantics for units; when are two units equal?
- Replace `Int` by `Int:Unit` and similarly variables `var I : Int` by `var I:U : Int` (`I:U` is just a token);
- Modify properly the arithmetic and boolean operators; this is where the entire complexity of the problem goes.

Semantics of Units

Complex units evaluate to *sets* of powered basic units:

```
fmod INT:UNIT-SEMANTICS is protecting INT:UNIT-SYNTAX .
  vars U U' : Unit .  vars N M : Int .
  eq newton = kilogram meter second ^ -2 .
  eq U noUnit = U .
  eq noUnit ^ N = noUnit .
  eq U ^ 0 = noUnit .
  eq U ^ 1 = U .
  eq U U = U ^ 2 .
  eq U (U ^ N) = U ^ (N + 1) .
  eq (U ^ N) (U ^ M) = U ^ (N + M) .
  eq (U U') ^ N = (U ^ N) (U' ^ N) .
  eq (U ^ N) ^ M = U ^ (N * M) .
endfm
```

One can now reduce complex units to their normal forms (modulo AC):

```
red meter second foot meter ^ -1 second ^ -1 .
```

Exercise 1 *Show that the module above yields a canonical AC rewriting system for ground terms, that is, that any complex unit, without variables, reduces to a* unique *normal form, modulo AC.*

The only change needed in the module GENERIC-EXP-SEMANTICS is to import INT:UNIT-SEMANTICS and to replace properly integers by unit tagged integers:

```
fmod GENERIC-EXP-SEMANTICS is protecting INT:UNIT-SEMANTICS .
...
  op int : Int:Unit -> Value .
  var I:U : Int:Unit .  var S : State .
  eq  eval(I:U, S) = int(I:U) .  eq state(I:U, S) = S .
...
endfm
```


Unit Specific Semantics for Arithmetic Operators

The meaning of arithmetic operators needs to be changed to reflect the specific correctness policy of units. In the case of addition and subtraction, the only thing to be checked is that the two expressions *evaluate to integers of same unit*:

```
fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .  vars U U' : Unit .
  ops add mul sub div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I : U), int(I' : U)) = int((I + I') : U) .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
  eq sub(int(I : U), int(I' : U)) = int((I - I') : U) .
  eq state(E - E', S) = state(E', state(E,S)) .
```

If the two expressions evaluate to integers of the same unit, then the result will have the same unit.

In the case of multiplication and division, the units of the two expressions do not matter. We only need to make sure that the two expressions evaluate to integers and that we correctly propagate the resulting unit:

```
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I : U), int(I' : U')) = int((I * I') : U U') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I : U), int(I' : U')) = int((I quo I') : U U' ^ -1) .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm
```


Unit Specific Semantics for Boolean Operators

Boolean expressions also need to be given a different semantics. This because in the context of units, they really have a different meaning. For example, one cannot compare meters and seconds. Additionally, we would like to have several boolean operators which are *unit independent*, such as `zero?` which should work on any unit:

```
fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  op unitEqual? : Value Value -> Bool .
  ops unitZero? unitEven? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
  vars I I' : Int .  var U : Unit .
```

```
  eq  eval(E equals E', S) =
      unitEqual?(eval(E, S), eval(E', state(E, S))) .
  eq unitEqual?(int(I : U), int(I' : U)) = I == I' .
  eq state(E equals E', S) = state(E', state(E, S)) .
  eq  eval(zero?(E), S) = unitZero?(eval(E, S)) .
  eq unitZero?(int(I : U)) = I == 0 .
  eq state(zero?(E), S) = state(E, S) .
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = unitEven?(eval(E, S)) .
  eq unitEven?(int(I : U)) = I rem 2 == 0 .
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) =
     eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm
```


Unit Specific Semantics for Variable Assignment

The last change that we need to do in our language is to say what is the unit of the integer `1` which is returned when a variable assignment is evaluated. Our convention is that it will be attached the unit `noUnit`:

```
fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1 : noUnit) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm
```

The semantics of all the other language constructs remains unchanged. All the features will work correctly now, including static versus dynamic scoping, `letrec`, as well as all the parameter passing styles.

Examples

Please read these examples as well as the ones in the file dynamic-type-checking.maude and understand them. You may be asked to comment on similar examples at final.

```
red eval(
  let x = 1 : second
  in (x + (let x = 10 : hour in x))
) .
***> should be add(int(10 : hour), int(1 : second))

red eval(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1 : meter, 2 : second)
) .
```



```
***> should be sub(add(int(2 : second), int(1 : meter)), int(2 : meter second))

red eval(
  let y = 1 : meter
  in let f = proc(value x) y
     in let y = 2 : second
        in f(0 : meter second)
) .
***> is int(1 : meter) under static and int(2 :second) under dynamic scoping

red eval(
  let x = 1 : meter second
  in let x = 2 : meter ^ -1,
         f = proc (value y, value z) y + x * z
     in f(1 : second, x)
) .
***> should be int(3 : second) under static scoping
***> should be add(int(4 : meter ^ -2), int(1 : second)) under dynamic scoping
```

```
red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3 : meter), 4 : meter)
) .
***> should be add(int(4 : meter ^ 2), int(4 : meter))

red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) (u + x) * u
     in f(g(3 : meter), 4 : meter)
) .
***> should be int(16 : meter ^ 2) under static scoping
```



```
***> should be int(23 : meter ^ 2) under dynamic scoping

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1 : noUnit
               else n * f(n - 1 : meter)
  in f(10 : meter)
) .
***> should be int(3628800 : meter ^ 10)

red eval(
  let f = proc(value x, value g)
            if zero?(x)
            then 1 : noUnit
            else x * g(x - 1 : meter, g)
  in f(10 : meter, f)
) .
```

```
***> should be int(3628800 : meter ^ 10)

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1 : noUnit
               else n * f(n - 1 : meter)
  in let x = f,
         y = proc(value x, value g)
              if x equals 1 : meter
              then x
              else x * g(x - 1 : meter, g),
         n = 10 : meter
     in if x(n) equals y(n,y) then 1 : noUnit else 0 : noUnit
) .
***> should be int(1 : noUnit)

red eval(
```



```
  let x = 3 : second, y = 4 : celsius
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be undefined

red eval(
  let x = 3 : second, y = 4 : second
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be int(24 : meter second ^ 2)

red eval(
```

```
  let n = 178378342647 : meter, c = 0 : noUnit
  in { while not (n equals 1 : meter) {
         set c = c + 1 : noUnit ;
         if even?(n)
         then set n = n / 2 : noUnit
         else set n = 3 : noUnit * n + 1 : meter
       } ;
       c }
) .
***> should be int(185 : noUnit)
```


Homework Exercise 2 *The presented definition of the unit-specific dynamic type checker is a bit too strict, in the sense that it makes no distinction between incompatible units and units which are compatible modulo a simple conversion factor, such as* `mile`*,* `meter`*,* `kilometer`*,* `foot`*, or* `celsius` *and* `fahrenheit`*.*

Relax the given semantics such that conversions are done automatically between compatible units. Convert distances to `foot`*, time to* `second`*, and temperature to* `celsius`*, so we can test your code uniformly. Also, add a new unit,* `fail`*, and attach it to expressions which violate the unit policy. This way, your expression will* always *be evaluated to a value, as far as the non-unit version of your program evaluates to a value under the original semantics.*

```
*****************************************************************************
*** Defining Dynamic Typing for a Functional Programming Language ***
*****************************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod INT:UNIT-SYNTAX is protecting INT .
  sorts Unit Int:Unit .
  op _:_ : Int Unit -> Int:Unit [prec 3] .
  ops meter kilometer foot yard mile : -> Unit .
  ops second minute hour : -> Unit .
  ops gram kilogram : -> Unit .
  ops celsius fahrenheit : -> Unit .
  op newton : -> Unit .
  op noUnit : -> Unit .
  op __ : Unit Unit -> Unit [assoc comm prec 2] .
  op _^_ : Unit Int -> Unit [prec 1] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT:UNIT-SYNTAX .
  sorts Exp ExpList .
  subsorts Int:Unit Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .
parse 3 : meter, x, 'variable .
```

```
fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [assoc comm prec 33] .
  op _-_ : Exp Exp -> Exp [prec 33 gather (E e)] .
  op _*_ : Exp Exp -> Exp [assoc comm prec 31] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse (3 : second) + x .
parse (3 : noUnit) + 'variable1 .
parse (3 : foot) + 'variable2 + x * (y - z) + 'variable1 .
parse (3 : second) * (4 : meter) + (1 : meter) .
parse (3 : second) * (4 : meter) * (1 : second ^ -1) + (1 : meter) .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 : meter equals 3 : second .
parse 3 : meter equals 3 : meter .
parse 3 : meter equals 5 : foot .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5 : second) then 2 : meter else 3 : meter .
parse if zero?(0 : meter) then 2 : second else 3 : mile .
parse if zero?(x) then y else z .
parse if x equals y
     then x
     else if x equals z
          then z
          else y .

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
```

```
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let x = 5 : meter
  in x
 .
parse
  let x = 5 : noUnit, y = 7 : second
  in x
 .
parse
  let x = 5 : mile
  in let y = x
     in y
 .
parse
  let x = 1 : noUnit
  in let x = 2 : meter
     in x
 .
parse
  let x = 10 : celsius, y = 0 : noUnit, z = x
  in let a = 5 : fahrenheit, b = 7 : noUnit
        in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : CallingMode Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id:()] .
```

```
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(need x, value y) 0 : noUnit
 .
parse
  (proc(name x, value y) 0 : noUnit) (2 : second, 3 : meter)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) 2 : meter, 3 : second)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) y, 3 : meter)
 .

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
```

```
endfm

parse
  letrec x = 1 : meter
  in letrec x = 7 : second, y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 : second .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 : celsius ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------
```

```
fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is
  extending GENERIC-STATE .
  protecting LOCATION .
  sorts Index Entry Environment .
  subsort Environment < StateAttribute .
  op env : -> StateAttributeName .
  subsort Entry < Environment .
  op empty : -> Environment .
  op [_,_] : Index Location -> Entry .
  op __ : Environment Environment -> Environment [assoc comm id: empty] .
  op _[_] : Environment Index -> Location .
  op _[_<-_] : Environment Index Location -> Environment .
  vars Ix Ix' : Index .  vars L L' : Location .  var Env : Environment .
  eq ([Ix,L] Env)[Ix] = L .
  eq ([Ix,L] Env)[Ix <- L'] = [Ix,L'] Env .
  eq Env[Ix <- L'] = [Ix,L'] Env [owise] .
endfm

fmod VALUE is protecting GENERIC-STATE .
  sorts Value PreValue .
  subsort Value < PreValue .
  op  eval : PreValue State -> Value .
  op state : PreValue State -> State .
  var V : Value .  var S : State .
```

```
  eq  eval(V, S) = V .
  eq state(V, S) = S .
endfm

fmod CELL is
  protecting LOCATION .
  protecting VALUE .
  sorts Cell Cells .
  subsort Cell < Cells .
  op noCells : -> Cells .
  op [_,_] : Location PreValue -> Cell .
  op __ : Cells Cells -> Cells [assoc comm id: noCells] .
  op _[_] : Cells Location -> PreValue .
  op _[_<*_] : Cells Location PreValue -> Cells .
  vars L L' : Location .  vars Pv Pv' : PreValue .  var Cs : Cells .
  eq ([L,Pv] Cs)[L] = Pv .
  eq ([L,Pv] Cs)[L <* Pv'] = [L,Pv'] Cs .
endfm

fmod STORE is
  extending GENERIC-STATE .
  protecting CELL .
  sort Store .
  subsort Store < StateAttribute .
  op {_,_} : Location Cells -> Store .
  op store : -> StateAttributeName .
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store Location PreValue -> Store .
  op nextLoc : Store -> Location .
  vars L Ln : Location .  var Cs : Cells .  var N : Nat .
  var Pv : PreValue .
  eq {Ln,Cs}[L] = Cs[L] .
  eq {loc(N),Cs}[loc(N) <- Pv] = {loc(N + 1), Cs[loc(N),Pv]} .
  eq {Ln,Cs}[L <- Pv] = {Ln,Cs[L <* Pv]} [owise] .
  eq nextLoc({Ln,Cs}) = Ln .
endfm

fmod STATE is
  protecting STORE .
  protecting ENVIRONMENT .
  op _[_] : State Index -> PreValue .
  op _[_<-_] : State Index Location -> State .
  op _[_<-_] : State Location PreValue -> State .
```

```
  op _[_<-_] : State Index PreValue -> State .
  op _[_<*_] : State Index PreValue -> State .
  var S : State .  var Ix : Index .  var L : Location .  var Pv : PreValue .
  eq S[Ix] = S[store][S[env][Ix]] .
  eq S[Ix <- L ] = S[env <- S[env][Ix <- L]] .
  eq S[ L <- Pv] = S[store <- S[store][L <- Pv]] .
  eq S[Ix <- Pv] = S[env <- S[env][Ix <- nextLoc(S[store])]]
                   [store <- S[store][nextLoc(S[store]) <- Pv]] .
  eq S[Ix <* Pv] = S[store <- S[store][S[env][Ix] <- Pv]] .
endfm

fmod INT:UNIT-SEMANTICS is protecting INT:UNIT-SYNTAX .
  vars U U' : Unit .  vars N M : Int .
  eq newton = kilogram meter second ^ -2 .
  eq U noUnit = U .
  eq noUnit ^ N = noUnit .
  eq U ^ 0 = noUnit .
  eq U ^ 1 = U .
  eq U U = U ^ 2 .
  eq U (U ^ N) = U ^ (N + 1) .
  eq (U ^ N) (U ^ M) = U ^ (N + M) .
  eq (U U') ^ N = (U ^ N) (U' ^ N) .
  eq (U ^ N) ^ M = U ^ (N * M) .
endfm

red meter second foot meter ^ -1 second ^ -1 .

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op  eval : Name State -> Value .
  op state : Name State -> State .
  var X : Name .  var S : State .
  eq  eval(X, S) =  eval(S[idx(X)], S) .
  eq state(X, S) = state(S[idx(X)], S) .
endfm

fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  protecting INT:UNIT-SEMANTICS .
  op int : Int:Unit -> Value .
  op  eval : Exp State -> Value .
  op state : Exp State -> State .
```

```
  op  eval : Exp -> Value .
  var I:U : Int:Unit .  var S : State .  vars E E' : Exp .  var El : ExpList .
  eq  eval(I:U, S) = int(I:U) .
  eq state(I:U, S) = S .
  eq eval(E) = eval(E, (env,empty)(store,{loc(0),noCells})) .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .
parse 3 : meter, x, 'variable .

red eval(3) .
***> should not parse
red eval(x) .
***> should be undefined
red eval('variable) .
***> should be undefined
red eval('variable,
     (  env, [idx(x),loc(0)] [idx('variable),loc(1)])
     (store, {loc(2), [loc(0),int(5 : meter)] [loc(1),int(4 : second)]})) .
***> should be 4 : second

fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .  vars U U' : Unit .
  ops add sub mul div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I : U), int(I' : U)) = int((I + I') : U) .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
  eq sub(int(I : U), int(I' : U)) = int((I - I') : U) .
  eq state(E - E', S) = state(E', state(E,S)) .
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I : U), int(I' : U')) = int((I * I') : U U') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I : U), int(I' : U')) = int((I quo I') : U U' ^ -1) .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm

red eval(3 : second + x,
```

```
    (  env, [idx(x),loc(0)] [idx('n),loc(1)])
    (store, {loc(100), [loc(0),int(5 : second)][loc(1),int(4 : meter)]})) .
***> should be int(8 : second)
red eval(3 : second + 'variable1,
    (  env, [idx(x),loc(0)] [idx('variable1),loc(1)])
    (store, {loc(100), [loc(0),int(5 : second)][loc(1),int(4 : meter)]})) .
***> should be add(int(3 : second), int(4 : meter))
red eval(3 : foot + 'variable2 + x * (y - z) + 'variable1 ,
       (  env, [idx('variable2),loc(0)]
               [idx(x),loc(1)]
               [idx(y),loc(2)]
               [idx(z),loc(3)]
               [idx('variable1),loc(4)])
       (store, {loc(4), [loc(0), int(0 : foot)]
                        [loc(1), int(1 : foot second ^ -1)]
                        [loc(2), int(2 : second)]
                        [loc(3), int(3 : second)]
                        [loc(4), int(4 : foot)]})) .
***> should be int(6 : foot)
red eval((3 : second) * (4 : meter) + (1 : meter)) .
***> should be add(int(12 : meter second), int(1 : meter))
red eval((3 : second) * (4 : meter) * (1 : second ^ -1) + (1 : meter)) .
***> should be int(13 : meter)

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  op unitEqual? : Value Value -> Bool .
  ops unitZero? unitEven? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
  vars I I' : Int .  var U : Unit .
  eq  eval(E equals E', S) = unitEqual?(eval(E, S), eval(E', state(E, S))) .
  eq unitEqual?(int(I : U), int(I' : U)) = I == I' .
  eq state(E equals E', S) = state(E', state(E, S)) .
  eq  eval(zero?(E), S) = unitZero?(eval(E, S)) .
  eq unitZero?(int(I : U)) = I == 0 .
  eq state(zero?(E), S) = state(E, S) .
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = unitEven?(eval(E, S)) .
  eq unitEven?(int(I : U)) = I rem 2 == 0 .
```

```
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) = eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm

red eval(3 : meter equals 3 : second, S) .
***> should be unitEqual?(int(3 : meter), int(3 : second))
red eval(3 : meter equals 3 : meter, S) .
***> should be true
red eval(3 : meter equals 5 : meter, S) .
***> should be false

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  var S : State .
  eq  eval(if Be then E else E', S) = if eval(Be, S)
       then  eval(E, state(Be, S)) else  eval(E', state(Be, S)) fi .
  eq state(if Be then E else E', S) = if eval(Be, S)
       then state(E, state(Be, S)) else state(E', state(Be, S)) fi .
endfm

red eval(if zero?(5 : second) then 2 : meter else 3 : meter, S) .
***> should be int(3 : meter)
red eval(if zero?(0 : meter) then 2 : second else 3 : mile, S) .
***> should be int(2 : second)
red eval(if zero?(x) then y else z,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15 : second)]
                    [loc(10),int(3 : hour)]
                    [loc(12),int(5 : foot)]})) .
***> should be int(5 : foot)
red eval(
  if x equals y
  then x
  else if x equals z
       then z
       else y,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15 : celsius)]
                    [loc(10),int(-3 : celsius)]
                    [loc(12),int(5 : second)]})) .
***> should be undefined
```

```
fmod BINDINGS-SEMANTICS is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op stateBList : BindingList State -> State .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq stateBList(none, S) = S .
  eq stateBList((X = E, Bl), S) = stateBList(Bl, state(E, S)) .
  eq bindBList (none,S) in S' = S' .
  eq bindBList ((X = E, Bl), S) in S' =
     bindBList (Bl, state(E,S)) in (S'[idx(X) <- eval(E,S)]) .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq  eval(let Bl in E, S) =  eval(E, bindBList (Bl, S) in stateBList(Bl,S)) .
  eq state(let Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S) in stateBList(Bl,S))[store]] .
endfm

red eval(
  let x = 5 : meter
  in x
) .  ***> should be int(5 : meter)
red eval(
  let x = 5 : noUnit, y = 7 : second
  in x
) .  ***> should be int(5 : noUnit)
red eval(
  let x = 5 : mile
  in let y = x
     in y
) .  ***> should be int(5 : mile)
red eval(
  let x = 1 : noUnit
  in let x = 2 : meter
     in x
) .  ***> should be int(2 : meter)
red eval(
  let x = 10 : celsius, y = 0 : noUnit, z = x
  in let a = 5 : fahrenheit, b = 7 : noUnit
        in z
```

) . ***> should be undefined

```
fmod PARAMETER-SEMANTICS is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op state : Parameter Exp State -> State .
  op statePList : ParameterList ExpList State -> State .
  op bind : Parameter Exp State State -> State .
  op bindPList : ParameterList ExpList State State -> State .
  vars S S' : State .  var P : Parameter .  var Pl : ParameterList .
  var E : Exp .  var El : ExpList .
  eq statePList((), (), S) = S .
  eq statePList((P,Pl), (E,El), S) =
     statePList(Pl, El, state(P, E, S)) [owise] .
  eq bindPList((), (), S, S') = S' .
  eq bindPList((P,Pl), (E,El), S, S') =
     bindPList(Pl, El, state(P,E,S), bind(P,E,S,S')) .
endfm

fmod CALL-BY-VALUE-SEMANTICS is extending CALL-BY-VALUE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  var X : Name .  var E : Exp .  vars S S' : State .
  eq state(value X, E, S) = state(E, S) .
  eq  bind(value X, E, S, S') = S'[idx(X) <- eval(E, S)] .
endfm

fmod CALL-BY-REFERENCE-SEMANTICS is extending CALL-BY-REFERENCE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  vars X Y : Name .  var E : Exp .  vars S S' : State .
  eq state(reference X, Y, S) = S .
  eq state(reference X, E, S) = state(E, S) [owise] .
  eq  bind(reference X, Y, S, S') = S'[idx(X) <- S[env][idx(Y)]] .
  eq  bind(reference X, E, S, S') = S'[idx(X) <- eval(E, S)] [owise] .
endfm

fmod CALL-BY-NAME-SEMANTICS is extending CALL-BY-NAME-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op frozen : Exp Environment -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  eq  eval(frozen(E, Env), S) = eval(E, S[env <- Env]) .
  eq state(frozen(E, Env), S) = S[store <- state(E, S[env <- Env])[store]] .
  eq state(name X, E, S) = S .
  eq  bind(name X, E, S, S') = S'[idx(X) <- frozen(E, S[env])] .
endfm
```

```
fmod CALL-BY-NEED-SEMANTICS is extending CALL-BY-NEED-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op unfreeze : Exp Environment Location -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  var L : Location .
  eq  eval(unfreeze(E,Env,L), S) = eval(E, S[env <- Env]) .
  eq state(unfreeze(E,Env,L), S) =
     S[store <- state(E, S[env <- Env])[store][L <- eval(E, S[env <- Env])]] .
  eq state(need X, E, S) = S .
  eq  bind(need X, E, S, S') =
      S'[idx(X) <- unfreeze(E, S[env], nextLoc(S'[store]))] .
endfm

fmod CLOSURE is protecting PARAMETER-SEMANTICS .
  sort Closure .
  subsort Closure < Value .
  op closure : ParameterList Exp Environment -> Closure .
  op apply : Closure ExpList State -> Value .
  op state : Closure ExpList State -> State .
endfm

fmod STATIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env])) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env]))[store]] .
endfm

fmod DYNAMIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S))) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)))[store]] .
endfm

fmod PROC-SEMANTICS is protecting PROC-SYNTAX .
  protecting CALL-BY-VALUE-SEMANTICS .
  protecting CALL-BY-REFERENCE-SEMANTICS .
```

```
  protecting CALL-BY-NAME-SEMANTICS .
  protecting CALL-BY-NEED-SEMANTICS .
*** the next lets you choose between static vs. dynamic binding
  protecting  STATIC-BINDING .
---  protecting DYNAMIC-BINDING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .  var S : State .
  eq  eval(proc Pl E, S) = closure(Pl, E, S[env]) .
  eq state(proc Pl E, S) = S .
  eq  eval(E El, S) = apply(eval(E,S), El, state(E,S)) .
  eq state(E El, S) = state(eval(E,S), El, state(E,S)) .
endfm

red eval((proc(need x, value y) 0 : noUnit)) .
***> should be closure((need x,value y), 0 : noUnit, empty)

red eval((proc(name x, value y) 0 : noUnit) (2 : second, 3 : meter)) .
***> should be int(0 : noUnit)

red eval((proc(value x, reference y) (x(y)))
          (proc(value x) 2 : second, 3 : meter)) .
***> should be int(2 : second)

red eval((proc(value x, reference y) (x(y))) (proc(value x) y, 3 : meter),
         (  env, [idx(y), loc(0)])
         (store, {loc(1), [loc(0),int(1 : second)]})) .
***> is int(1 : second) under static and int(3 : meter) under dynamic scoping

fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  op mkDanglingBindings : BindingList Location State -> State .
  var X : Name .  var N : Nat .
  var Bl : BindingList .  var E : Exp .  vars S S' : State .
  eq mkDanglingBindings(none, loc(N), S) = S .
  eq mkDanglingBindings((X = E, Bl), loc(N), S) =
     mkDanglingBindings(Bl, loc(N + 1), S[idx(X) <- loc(N)]) .
 ceq eval(letrec Bl in E, S) = eval(E, bindBList (Bl,S') in stateBList(Bl,S'))
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
 ceq state(letrec Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S') in stateBList(Bl, S'))[store]]
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
endfm
```

```
red eval(
  letrec x = 1 : meter
  in letrec x = 7 : second, y = x
     in y
) .  ***> should be undefined

fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1 : noUnit) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm

red eval(set x = 3 : second) .
***> should be int(1 : noUnit)

fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq  eval({E}, S) =  eval(E, S) .
  eq state({E}, S) = state(E, S) .
  eq  eval({El ; E}, S) =  eval(E, state({El}, S)) .
  eq state({El ; E}, S) = state(E, state({El}, S)) .
endfm

red eval({x ; y ; 3 : celsius ; x}) .
***> should be undefined

fmod LOOP-SEMANTICS is protecting LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  var S : State .
  eq  eval(while Be E, S) =
     if eval(Be,S) then  eval(while Be E, state(E,state(Be,S)))
     else int(1) fi .
  eq state(while Be E, S) =
     if eval(Be,S) then state(while Be E, state(E,state(Be,S)))
     else state(Be,S) fi .
endfm

fmod PROG-LANG-SEMANTICS is
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
```

```
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm



red eval(
  let x = 5 : celsius, y = 7 : fahrenheit
  in x + y
) .
***> should be add(int(7 : fahrenheit), int(5 : celsius))

red eval(
  let x = 5 : celsius, y = 7 : celsius
  in x + y
) .
***> should be int(12 : celsius)

red eval(
  let x = 1 : second
  in (x + (let x = 10 : hour in x))
) .
***> should be add(int(10 : hour), int(1 : second))

red eval(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1 : meter, 2 : second)
) .
***> should be sub(add(int(2 : second), int(1 : meter)), int(2 : meter second))

red eval(
  let y = 1 : meter
  in let f = proc(value x) y
     in let y = 2 : second
        in f(0 : meter second)
) .
***> is int(1 : meter) under static and int(2 :second) under dynamic scoping

red eval(
```

```
  let x = 1 : meter second
  in let x = 2 : meter ^ -1,
         f = proc (value y, value z) y + x * z
     in f(1 : second, x)
) .
***> should be int(3 : second) under static scoping
***> should be add(int(4 : meter ^ -2), int(1 : second)) under dynamic scoping

red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3 : meter), 4 : meter)
) .
***> should be add(int(4 : meter ^ 2), int(4 : meter))

red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) (u + x) * u
     in f(g(3 : meter), 4 : meter)
) .

***> should be int(16 : meter ^ 2) under static scoping
***> should be int(23 : meter ^ 2) under dynamic scoping

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1 : noUnit
               else n * f(n - 1 : meter)
  in f(10 : meter)
) .
***> should be int(3628800 : meter ^ 10)

red eval(
  let f = proc(value x, value g)
            if zero?(x)
            then 1 : noUnit
            else x * g(x - 1 : meter, g)
  in f(10 : meter, f)
```

```
) .
***> should be int(3628800 : meter ^ 10)

red eval(
  letrec f = proc(value n)
             if zero?(n)
             then 1 : noUnit
             else n * f(n - 1 : meter)
  in let x = f,
         y = proc(value x, value g)
             if x equals 1 : meter
             then x
             else x * g(x - 1 : meter, g),
         n = 10 : meter
     in if x(n) equals y(n,y) then 1 : noUnit else 0 : noUnit
) .
***> should be int(1 : noUnit)

red eval(
  let x = 3 : second, y = 4 : celsius
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be undefined

red eval(
  let x = 3 : second, y = 4 : second
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be int(24 : meter second ^ 2)

red eval(
  let n = 178378342647 : meter, c = 0 : noUnit
  in { while not (n equals 1 : meter) {
         set c = c + 1 : noUnit ;
         if even?(n)
         then set n = n / 2 : noUnit
         else set n = 3 : noUnit * n + 1 : meter
```

```
      } ;
      c }
) .
***> should be int(185 : noUnit)
```

```
**********************************************************************
*** Defining Dynamic Typing for a Functional Programming Language ***
**********************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod INT:UNIT-SYNTAX is protecting INT .
  sorts Unit Int:Unit .
  op _:_ : Int Unit -> Int:Unit [prec 3] .
  ops meter kilometer foot yard mile : -> Unit .
  ops second minute hour : -> Unit .
  ops gram kilogram : -> Unit .
  ops celsius fahrenheit : -> Unit .
  op newton : -> Unit .
  op noUnit : -> Unit .
  op __ : Unit Unit -> Unit [assoc comm prec 2] .
  op _^_ : Unit Int -> Unit [prec 1] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT:UNIT-SYNTAX .
  sorts Exp ExpList .
  subsorts Int:Unit Name < Exp < ExpList .
  op '(') : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .
parse 3 : meter, x, 'variable .

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [assoc comm prec 33] .
  op _-_ : Exp Exp -> Exp [prec 33 gather (E e)] .
  op _*_ : Exp Exp -> Exp [assoc comm prec 31] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

parse (3 : second) + x .
parse (3 : noUnit) + 'variable1 .
parse (3 : foot) + 'variable2 + x * (y - z) + 'variable1 .
parse (3 : second) * (4 : meter) + (1 : meter) .
parse (3 : second) * (4 : meter) * (1 : second ^ -1) + (1 : meter) .

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

parse 3 : meter equals 3 : second .
parse 3 : meter equals 3 : meter .
parse 3 : meter equals 5 : foot .

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

parse if zero?(5 : second) then 2 : meter else 3 : meter .
parse if zero?(0 : meter) then 2 : second else 3 : mile .
parse if zero?(x) then y else z .
parse if x equals y
      then x
      else if x equals z
           then z
           else y .

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

parse
  let x = 5 : meter
  in x
 .
parse
  let x = 5 : noUnit, y = 7 : second
  in x
 .
parse
  let x = 5 : mile
  in let y = x
     in y
 .
parse
  let x = 1 : noUnit
  in let x = 2 : meter
     in x
 .
parse
  let x = 10 : celsius, y = 0 : noUnit, z = x
  in let a = 5 : fahrenheit, b = 7 : noUnit
         in z
 .

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op __ : CallingMode Name -> Parameter [prec 0] .
  op '(') : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

parse
  proc(need x, value y) 0 : noUnit
 .
parse
  (proc(name x, value y) 0 : noUnit) (2 : second, 3 : meter)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) 2 : meter, 3 : second)
 .
parse
  (proc(value x, reference y) (x(y))) (proc(value x) y, 3 : meter)
 .

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

parse
  letrec x = 1 : meter
  in letrec x = 7 : second, y = x
     in y
 .

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

parse set x = 3 : second .

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

parse {x ; y ; 3 : celsius ; x} .

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


-----------------
--- Semantics ---
-----------------

fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is
  extending GENERIC-STATE .
  protecting LOCATION .
  sorts Index Entry Environment .
  subsort Environment < StateAttribute .
  op env : -> StateAttributeName .
  subsort Entry < Environment .
  op empty : -> Environment .
  op [_,_] : Index Location -> Entry .
  op __ : Environment Environment -> Environment [assoc comm id: empty] .
  op _[_] : Environment Index -> Location .
  op _[_<-_] : Environment Index Location -> Environment .
  vars Ix Ix' : Index .  vars L L' : Location .  var Env : Environment .
  eq ([Ix,L] Env)[Ix] = L .
  eq ([Ix,L] Env)[Ix <- L'] = [Ix,L'] Env .
  eq Env[Ix <- L'] = [Ix,L'] Env [owise] .
endfm

fmod VALUE is protecting GENERIC-STATE .
  sorts Value PreValue .
  subsort Value < PreValue .
  op  eval : PreValue State -> Value .
  op state : PreValue State -> State .
  var V : Value .  var S : State .
  eq  eval(V, S) = V .
  eq state(V, S) = S .
endfm

fmod CELL is
  protecting LOCATION .
  protecting VALUE .
```

```
  sorts Cell Cells .
  subsort Cell < Cells .
  op noCells : -> Cells .
  op [_,_] : Location PreValue -> Cell .
  op __ : Cells Cells -> Cells [assoc comm id: noCells] .
  op _[_] : Cells Location -> PreValue .
  op _[_<*_] : Cells Location PreValue -> Cells .
  vars L L' : Location .  vars Pv Pv' : PreValue .  var Cs : Cells .
  eq ([L,Pv] Cs)[L] = Pv .
  eq ([L,Pv] Cs)[L <* Pv'] = [L,Pv'] Cs .
endfm

fmod STORE is
  extending GENERIC-STATE .
  protecting CELL .
  sort Store .
  subsort Store < StateAttribute .
  op {_,_} : Location Cells -> Store .
  op store : -> StateAttributeName .
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store Location PreValue -> Store .
  op nextLoc : Store -> Location .
  vars L Ln : Location .  var Cs : Cells .  var N : Nat .
  var Pv : PreValue .
  eq {Ln,Cs}[L] = Cs[L] .
  eq {loc(N),Cs}[loc(N) <- Pv] = {loc(N + 1), Cs[loc(N),Pv]} .
  eq {Ln,Cs}[L <- Pv] = {Ln,Cs[L <* Pv]} [owise] .
  eq nextLoc({Ln,Cs}) = Ln .
endfm

fmod STATE is
  protecting STORE .
  protecting ENVIRONMENT .
  op _[_] : State Index -> PreValue .
  op _[_<-_] : State Index Location -> State .
  op _[_<-_] : State Location PreValue -> State .
  op _[_<-_] : State Index PreValue -> State .
  op _[_<*_] : State Index PreValue -> State .
  var S : State .  var Ix : Index .  var L : Location .  var Pv : PreValue .
  eq S[Ix] = S[store][S[env][Ix]] .
  eq S[Ix <- L ] = S[env <- S[env][Ix <- L]] .
  eq S[ L <- Pv] = S[store <- S[store][L <- Pv]] .
  eq S[Ix <- Pv] = S[env <- S[env][Ix <- nextLoc(S[store])]]
                   [store <- S[store][nextLoc(S[store]) <- Pv]] .
  eq S[Ix <* Pv] = S[store <- S[store][S[env][Ix] <- Pv]] .
endfm

fmod INT:UNIT-SEMANTICS is protecting INT:UNIT-SYNTAX .
  vars U U' : Unit .  vars N M : Int .
  eq newton = kilogram meter second ^ -2 .
  eq U noUnit = U .
  eq noUnit ^ N = noUnit .
  eq U ^ 0 = noUnit .
  eq U ^ 1 = U .
  eq U U = U ^ 2 .
  eq U (U ^ N) = U ^ (N + 1) .
  eq (U ^ N) (U ^ M) = U ^ (N + M) .
  eq (U U') ^ N = (U ^ N) (U' ^ N) .
  eq (U ^ N) ^ M = U ^ (N * M) .
endfm

red meter second foot meter ^ -1 second ^ -1 .

fmod NAME-SEMANTICS is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
```

```
                [idx(y),loc(2)]
                [idx(z),loc(3)]
                [idx('variable1),loc(4)])
        (store, {loc(4), [loc(0), int(0 : foot)]
                         [loc(1), int(1 : foot second ^ -1)]
                         [loc(2), int(2 : second)]
                         [loc(3), int(3 : second)]
                         [loc(4), int(4 : foot)]})) .
***> should be int(6 : foot)
red eval((3 : second) * (4 : meter) + (1 : meter)) .
***> should be add(int(12 : meter second), int(1 : meter))
red eval((3 : second) * (4 : meter) * (1 : second ^ -1) + (1 : meter)) .
***> should be int(13 : meter)

fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op  eval : BExp State -> Bool .
  op state : BExp State -> State .
  op evenIntValue? : Value -> Bool .
  op unitEqual? : Value Value -> Bool .
  ops unitZero? unitEven? : Value -> Bool .
  vars E E' : Exp .  vars Be Be' : BExp .  var S : State .
  vars I I' : Int .  var U : Unit .
  eq  eval(E equals E', S) = unitEqual?(eval(E, S), eval(E', state(E, S))) .
  eq unitEqual?(int(I : U), int(I' : U)) = I == I' .
  eq state(E equals E', S) = state(E', state(E, S)) .
  eq  eval(zero?(E), S) = unitZero?(eval(E, S)) .
  eq unitZero?(int(I : U)) = I == 0 .
  eq state(zero?(E), S) = state(E, S) .
  eq  eval(not(Be), S) = not eval(Be, S) .
  eq state(not(Be), S) = state(Be, S) .
  eq  eval(even?(E), S) = unitEven?(eval(E, S)) .
  eq unitEven?(int(I : U)) = I rem 2 == 0 .
  eq state(even?(E), S) = state(E,S) .
  eq  eval(Be and Be', S) = eval(Be, S) and eval(Be', state(Be, S)) .
  eq state(Be and Be', S) = state(Be', state(Be, S)) .
endfm

red eval(3 : meter equals 3 : second, S) .
***> should be unitEqual?(int(3 : meter), int(3 : second))
red eval(3 : meter equals 3 : meter, S) .
***> should be true
red eval(3 : meter equals 5 : meter, S) .
***> should be false

fmod IF-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  var S : State .
  eq  eval(if Be then E else E', S) = if eval(Be, S)
       then  eval(E, state(Be, S)) else  eval(E', state(Be, S)) fi .
  eq state(if Be then E else E', S) = if eval(Be, S)
       then state(E, state(Be, S)) else state(E', state(Be, S)) fi .
endfm

red eval(if zero?(5 : second) then 2 : meter else 3 : meter, S) .
***> should be int(3 : meter)
red eval(if zero?(0 : meter) then 2 : second else 3 : mile, S) .
***> should be int(2 : second)
red eval(if zero?(x) then y else z,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15 : second)]
                    [loc(10),int(3 : hour)]
                    [loc(12),int(5 : foot)]})) .
***> should be int(5 : foot)
red eval(
  if x equals y
```

```
  op  eval : Name State -> Value .
  op state : Name State -> State .
  var X : Name .  var S : State .
  eq  eval(X, S) =  eval(S[idx(X)], S) .
  eq state(X, S) = state(S[idx(X)], S) .
endfm

fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  protecting INT:UNIT-SEMANTICS .
  op int : Int:Unit -> Value .
  op  eval : Exp State -> Value .
  op state : Exp State -> State .
  op  eval : Exp -> Value .
  var I:U : Int:Unit .  var S : State .  vars E E' : Exp .  var El : ExpList .
  eq  eval(I:U, S) = int(I:U) .
  eq state(I:U, S) = S .
  eq eval(E) = eval(E, (env,empty)(store,{loc(0),noCells})) .
endfm

parse 3 .
parse x .
parse 'variable .
parse 3, x, 'variable .
parse 3 : meter, x, 'variable .

red eval(3) .
***> should not parse
red eval(x) .
***> should be undefined
red eval('variable) .
***> should be undefined
red eval('variable,
      (  env, [idx(x),loc(0)] [idx('variable),loc(1)])
      (store, {loc(2), [loc(0),int(5 : meter)] [loc(1),int(4 : second)]})) .
***> should be 4 : second

fmod ARITH-OPS-SEMANTICS is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .  vars I I' : Int .  vars U U' : Unit .
  ops add sub mul div : Value Value -> Value .
  eq  eval(E + E', S) = add(eval(E, S), eval(E', state(E,S))) .
  eq add(int(I : U), int(I' : U)) = int((I + I') : U) .
  eq state(E + E', S) = state(E', state(E,S)) .
  eq  eval(E - E', S) = sub(eval(E, S), eval(E', state(E,S))) .
  eq sub(int(I : U), int(I' : U)) = int((I - I') : U) .
  eq state(E - E', S) = state(E', state(E,S)) .
  eq  eval(E * E', S) = mul(eval(E, S), eval(E', state(E,S))) .
  eq mul(int(I : U), int(I' : U')) = int((I * I') : U U') .
  eq state(E * E', S) = state(E', state(E,S)) .
  eq  eval(E / E', S) = div(eval(E, S), eval(E', state(E,S))) .
  eq div(int(I : U), int(I' : U')) = int((I quo I') : U U' ^ -1) .
  eq state(E / E', S) = state(E', state(E,S)) .
endfm

red eval(3 : second + x,
      (  env, [idx(x),loc(0)] [idx('n),loc(1)])
      (store, {loc(100), [loc(0),int(5 : second)][loc(1),int(4 : meter)]})) .
***> should be int(8 : second)
red eval(3 : second + 'variable1,
      (  env, [idx(x),loc(0)] [idx('variable1),loc(1)])
      (store, {loc(100), [loc(0),int(5 : second)][loc(1),int(4 : meter)]})) .
***> should be add(int(3 : second), int(4 : meter))
red eval(3 : foot + 'variable2 + x * (y - z) + 'variable1 ,
         (  env, [idx('variable2),loc(0)]
                 [idx(x),loc(1)]
```

```
  then x
  else if x equals z
       then z
       else y,
 (env, [idx(x),loc(1)] [idx(y),loc(10)] [idx(z),loc(12)])
 (store, {loc(100), [loc(1),int(15 : celsius)]
                    [loc(10),int(-3 : celsius)]
                    [loc(12),int(5 : second)]})) .
***> should be undefined

fmod BINDINGS-SEMANTICS is extending BINDING-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op stateBList : BindingList State -> State .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq stateBList(none, S) = S .
  eq stateBList((X = E, Bl), S) = stateBList(Bl, state(E, S)) .
  eq bindBList (none,S) in S' = S' .
  eq bindBList ((X = E, Bl), S) in S' =
     bindBList (Bl, state(E,S)) in (S'[idx(X) <- eval(E,S)]) .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq  eval(let Bl in E, S) =  eval(E, bindBList (Bl, S) in stateBList(Bl,S)) .
  eq state(let Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S) in stateBList(Bl,S))[store]] .
endfm

red eval(
  let x = 5 : meter
  in x
) .  ***> should be int(5 : meter)
red eval(
  let x = 5 : noUnit, y = 7 : second
  in x
) .  ***> should be int(5 : noUnit)
red eval(
  let x = 5 : mile
  in let y = x
     in y
) .  ***> should be int(5 : mile)
red eval(
  let x = 1 : noUnit
  in let x = 2 : meter
     in x
) .  ***> should be int(2 : meter)
red eval(
  let x = 10 : celsius, y = 0 : noUnit, z = x
  in let a = 5 : fahrenheit, b = 7 : noUnit
         in z
) .  ***> should be undefined

fmod PARAMETER-SEMANTICS is protecting PARAMETER-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  op state : Parameter Exp State -> State .
  op statePList : ParameterList ExpList State -> State .
  op bind : Parameter Exp State State -> State .
  op bindPList : ParameterList ExpList State State -> State .
  vars S S' : State .  var P : Parameter .  var Pl : ParameterList .
  var E : Exp .  var El : ExpList .
  eq statePList((), (), S) = S .
  eq statePList((P,Pl), (E,El), S) =
     statePList(Pl, El, state(P, E, S)) [owise] .
```

```
  eq bindPList((), (), S, S') = S' .
  eq bindPList((P,Pl), (E,El), S, S') =
     bindPList(Pl, El, state(P,E,S), bind(P,E,S,S')) .
endfm

fmod CALL-BY-VALUE-SEMANTICS is extending CALL-BY-VALUE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  var X : Name .  var E : Exp .  vars S S' : State .
  eq state(value X, E, S) = state(E, S) .
  eq  bind(value X, E, S, S') = S'[idx(X) <- eval(E, S)] .
endfm

fmod CALL-BY-REFERENCE-SEMANTICS is extending CALL-BY-REFERENCE-SYNTAX .
  extending PARAMETER-SEMANTICS .
  vars X Y : Name .  var E : Exp .  vars S S' : State .
  eq state(reference X, Y, S) = S .
  eq state(reference X, E, S) = state(E, S) [owise] .
  eq  bind(reference X, Y, S, S') = S'[idx(X) <- S[env][idx(Y)]] .
  eq  bind(reference X, E, S, S') = S'[idx(X) <- eval(E, S)] [owise] .
endfm

fmod CALL-BY-NAME-SEMANTICS is extending CALL-BY-NAME-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op frozen : Exp Environment -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  eq  eval(frozen(E, Env), S) = eval(E, S[env <- Env]) .
  eq state(frozen(E, Env), S) = S[store <- state(E, S[env <- Env])[store]] .
  eq state(name X, E, S) = S .
  eq  bind(name X, E, S, S') = S'[idx(X) <- frozen(E, S[env])] .
endfm

fmod CALL-BY-NEED-SEMANTICS is extending CALL-BY-NEED-SYNTAX .
  extending PARAMETER-SEMANTICS .
  op unfreeze : Exp Environment Location -> PreValue .
  var X : Name .  var E : Exp .  vars S S' : State .  var Env : Environment .
  var L : Location .
  eq  eval(unfreeze(E,Env,L), S) = eval(E, S[env <- Env]) .
  eq state(unfreeze(E,Env,L), S) =
     S[store <- state(E, S[env <- Env])[store][L <- eval(E, S[env <- Env])]] .
  eq state(need X, E, S) = S .
  eq  bind(need X, E, S, S') =
      S'[idx(X) <- unfreeze(E, S[env], nextLoc(S'[store]))] .
endfm

fmod CLOSURE is protecting PARAMETER-SEMANTICS .
  sort Closure .
  subsort Closure < Value .
  op closure : ParameterList Exp Environment -> Closure .
  op apply : Closure ExpList State -> Value .
  op state : Closure ExpList State -> State .
endfm

fmod STATIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env])) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)[env <- Env]))[store]] .
endfm

fmod DYNAMIC-BINDING is extending CLOSURE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  var Env : Environment .  var S : State .
  eq apply(closure(Pl,E,Env), El, S) =
     eval(E, bindPList(Pl,El,S,statePList(Pl,El,S))) .
  eq state(closure(Pl,E,Env), El, S) = S[store <-
     state(E, bindPList(Pl,El,S,statePList(Pl,El,S)))[store]] .
endfm

fmod PROC-SEMANTICS is protecting PROC-SYNTAX .
  protecting CALL-BY-VALUE-SEMANTICS .
  protecting CALL-BY-REFERENCE-SEMANTICS .
  protecting CALL-BY-NAME-SEMANTICS .
  protecting CALL-BY-NEED-SEMANTICS .
*** the next lets you choose between static vs. dynamic binding
  protecting  STATIC-BINDING .
---  protecting DYNAMIC-BINDING .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .  var S : State .
  eq  eval(proc Pl E, S) = closure(Pl, E, S[env]) .
  eq state(proc Pl E, S) = S .
  eq  eval(E El, S) = apply(eval(E,S), El, state(E,S)) .
  eq state(E El, S) = state(eval(E,S), El, state(E,S)) .
endfm

red eval((proc(need x, value y) 0 : noUnit)) .
***> should be closure((need x,value y), 0 : noUnit, empty)

red eval((proc(name x, value y) 0 : noUnit) (2 : second, 3 : meter)) .
***> should be int(0 : noUnit)

red eval((proc(value x, reference y) (x(y)))
          (proc(value x) 2 : second, 3 : meter)) .
***> should be int(2 : second)

red eval((proc(value x, reference y) (x(y))) (proc(value x) y, 3 : meter),
         (  env, [idx(y), loc(0)])
         (store, {loc(1), [loc(0),int(1 : second)]})) .
***> is int(1 : second) under static and int(3 : meter) under dynamic scoping

fmod LETREC-SEMANTICS is protecting LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  protecting BINDINGS-SEMANTICS .
  op mkDanglingBindings : BindingList Location State -> State .
  var X : Name .  var N : Nat .
  var Bl : BindingList .  var E : Exp .  vars S S' : State .
  eq mkDanglingBindings(none, loc(N), S) = S .
  eq mkDanglingBindings((X = E, Bl), loc(N), S) =
     mkDanglingBindings(Bl, loc(N + 1), S[idx(X) <- loc(N)]) .
 ceq eval(letrec Bl in E, S) = eval(E, bindBList (Bl,S') in stateBList(Bl,S'))
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
 ceq state(letrec Bl in E, S) =
     S[store <- state(E, bindBList (Bl, S') in stateBList(Bl, S'))[store]]
    if S' := mkDanglingBindings(Bl, nextLoc(S[store]), S) .
endfm

red eval(
  letrec x = 1 : meter
  in letrec x = 7 : second, y = x
     in y
) .  ***> should be undefined

fmod VAR-ASSIGNMENT-SEMANTICS is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var S : State .
  eq  eval(set X = E, S) = int(1 : noUnit) .
  eq state(set X = E, S) = state(E,S)[idx(X) <* eval(E,S)] .
endfm

red eval(set x = 3 : second) .
***> should be int(1 : noUnit)
```

```
fmod BLOCK-SEMANTICS is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq  eval({E}, S) =  eval(E, S) .
  eq state({E}, S) = state(E, S) .
  eq  eval({El ; E}, S) =  eval(E, state({El}, S)) .
  eq state({El ; E}, S) = state(E, state({El}, S)) .
endfm

red eval({x ; y ; 3 : celsius ; x}) .
***> should be undefined

fmod LOOP-SEMANTICS is protecting LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  var S : State .
  eq  eval(while Be E, S) =
     if eval(Be,S) then  eval(while Be E, state(E,state(Be,S)))
     else int(1) fi .
  eq state(while Be E, S) =
     if eval(Be,S) then state(while Be E, state(E,state(Be,S)))
     else state(Be,S) fi .
endfm

fmod PROG-LANG-SEMANTICS is
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm


red eval(
  let x = 5 : celsius, y = 7 : fahrenheit
  in x + y
) .
***> should be add(int(7 : fahrenheit), int(5 : celsius))

red eval(
  let x = 5 : celsius, y = 7 : celsius
  in x + y
) .
***> should be int(12 : celsius)

red eval(
  let x = 1 : second
  in (x + (let x = 10 : hour in x))
) .
***> should be add(int(10 : hour), int(1 : second))

red eval(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1 : meter, 2 : second)
) .
***> should be sub(add(int(2 : second), int(1 : meter)), int(2 : meter second))

red eval(
  let y = 1 : meter
  in let f = proc(value x) y
     in let y = 2 : second
        in f(0 : meter second)
) .
***> is int(1 : meter) under static and int(2 :second) under dynamic scoping

red eval(
  let x = 1 : meter second
  in let x = 2 : meter ^ -1,
         f = proc (value y, value z) y + x * z
     in f(1 : second, x)
) .
***> should be int(3 : second) under static scoping
***> should be add(int(4 : meter ^ -2), int(1 : second)) under dynamic scoping

red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3 : meter), 4 : meter)
) .
***> should be add(int(4 : meter ^ 2), int(4 : meter))

red eval(
  let x = 1 : meter
  in let x = 2 : meter,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) (u + x) * u
     in f(g(3 : meter), 4 : meter)
) .

***> should be int(16 : meter ^ 2) under static scoping
***> should be int(23 : meter ^ 2) under dynamic scoping

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1 : noUnit
               else n * f(n - 1 : meter)
  in f(10 : meter)
) .
***> should be int(3628800 : meter ^ 10)

red eval(
  let f = proc(value x, value g)
            if zero?(x)
            then 1 : noUnit
            else x * g(x - 1 : meter, g)
  in f(10 : meter, f)
) .
***> should be int(3628800 : meter ^ 10)

red eval(
  letrec f = proc(value n)
               if zero?(n)
               then 1 : noUnit
               else n * f(n - 1 : meter)
  in let x = f,
         y = proc(value x, value g)
               if x equals 1 : meter
               then x
               else x * g(x - 1 : meter, g),
         n = 10 : meter
     in if x(n) equals y(n,y) then 1 : noUnit else 0 : noUnit
) .
***> should be int(1 : noUnit)

red eval(
```

```
  let x = 3 : second, y = 4 : celsius
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be undefined

red eval(
  let x = 3 : second, y = 4 : second
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 : meter * x * y }
) .
***> should be int(24 : meter second ^ 2)

red eval(
  let n = 178378342647 : meter, c = 0 : noUnit
  in { while not (n equals 1 : meter) {
         set c = c + 1 : noUnit ;
         if even?(n)
         then set n = n / 2 : noUnit
         else set n = 3 : noUnit * n + 1 : meter
       } ;
       c }
) .
***> should be int(185 : noUnit)
```

CS322 - Programming Language Design

Lecture 12: Typed Languages - Type Inference (part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Checking that operations are applied on arguments of correct types statically has, as we have already discussed, two major benefits:

- Allows efficient implementations of programming languages by assuming untyped memory models and therefore removing the need for runtime consistency checks, and
- Gives rapid feedback to users, so they can correct errors at early stages.

These advantages are typically considered so major in the programming language community, that the *potential drawbacks* of static typing, namely

- Limiting the way programs are written by rejecting programs which do not type-check, and
- Adding typing information may be space and time consuming,

are often ignored.

The first drawback is usually addressed by rewriting the code in a way that passes the type checking procedure. For example, programs passing a function as an argument to itself, which do not type check because of the recursive nature of the type of that function, can be replaced by equivalent programs using `letrec`.

However, there are programming languages or important type systems, such as ones incorporating units of measurement, in which there can be situations where *one cannot rewrite* the code correspondingly. Think, for example, of an expression calculating the geometric mean of an array of "meter" values. In order to do it, one may think of an iterative procedure multiplying all the n elements in the array first and then by taking their n-th root. In order to statically certify that the geometric mean is correctly used as a "meter" value, a type checker would need to prove the the n-th root of the product of n "meter" values is also a "meter" value.

It can be formally proved that it is *impossible* to have a static type

checker able to certify unit consistency in the general case. The obvious alternative, namely to rewrite the programs using units in such a way that they pass some random type checker, may not be plausible in practice. For example, it seems hard to rewrite the geometric mean example such that it passes a simple minded type checker. A better alternative is to help the type checker "understand" the program by providing auxiliary information, such as loop or recursion invariants, via *(type) annotations*.

It is, of course, desirable to require the user to provide *as little type information as possible* as part of a program. In general, depending on the application and domain of interest, there are quite *subtle trade-offs* between the amount of user-provided annotations and the degree of automation of the static analyzer.

In the last two lectures we discussed a simple static type checker which requires the user to provide a type for each declared name. Since some names can be functions taking functions as arguments,

the amount and shape of these type annotations can be quite heavy. In some cases they may even exceed the size of the program itself. In this lecture we address the problem of reducing the amount of required type information by devising automatic procedures that *infer the intended types from how the names are used*.

How much type information can be automatically and efficiently inferred depends again upon the particular domain of interest and its associated type system. In this lecture we will see an extreme fortunate situation, in which *all* the needed type information can be inferred automatically. More precisely, we will discuss a classical procedure that *infers all the types* of all the declared names in our functional programming language. A similar technique is implemented as part of the ML language. By "type", we here mean the language specific types, that is, those that were explicitly provided by users to the static type checker that we discussed.

Type Inference

Programs written in the functional language discussed so far contain all the information that one needs in order to infer the intended type of each name. By carefully collecting and using all this information, one can type check programs without the need for the user to provide any auxiliary type information.

Suppose that one writes the expression `x + y` as part of a larger expression. Then one can infer from here that `x` and `y` are both intended to be integers, because the arithmetic operation `_+_` is defined only on integers! Similarly, if

```
if x then y else z
```

occurs in an expression, then one can deduce, thanks to the typing policy associated to conditionals, that `x` has type `bool` and that `y` and `z` have the same type. Moreover, from

```
if x then y else z + t
```

one can deduce that `z` and `t` are both of type `integer`, implying that `y` is also `integer`. Type inference and type checking work smoothly together, in the sense that one implicitly checks the types while inferring information. For example, if

```
if x then y else z + x
```

is seen then one knows that there is a typing error because the type of `x` *cannot* be both `integer` and `bool`.

The type of functions can also be deduced from the way the functions are used. For example, if one uses `f(x,y)` in some program then one can infer that `f` takes two arguments. Moreover, if `f(x,x)` is seen then one can additionally infer that `f`'s arguments have the same type.

There can be possible that the result type of a function as well as the types of its arguments can all be inferred from the context, without even analyzing the definition of the function. For example,

if one writes

```
f(x,y) + x + let x = y in x
```

then one first infers that `f` must return an `integer` because it is used as an argument of `_+_`, then that `x` is an integer for the same reason, so the type of `f`'s first argument is `integer`, and finally, because of the `let` which is used in a context of an `integer`, that the type of `y` is also `integer`. Therefore, the type of `f` must be `integer * integer -> integer`. In fact, the types of all the names occurring in the expression above were deduced by just analyzing carefully how they are used.

Let us now consider an entire expression which evaluates properly, for example one defining and using a factorial function:

```
letrec f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(5)
```

How can one automatically infer that this expression is type safe? Since `zero?` takes an `integer` and returns a `bool`, one can deduce that the parameter of the function is `integer`. Further, since the `f` occurs in the context of an integer and takes an expression which is supposed to be an integer as argument, `n - 1`, one can deduce that the type of that `f` is `integer -> integer`. Because of the semantics of `letrec`, the bound `f` will have the same type, so the type of the entire expression will be `integer`.

One can infer/check the types differently, obtaining the same result.

Polymorphic Functions

Let us consider a "projection" function which takes two arguments and always returns its first argument, such as `proc(x,y) x`. Without any additional type information, the best one can say about the type of this expression is that it is t_1 `*` t_2 `->` t_1, for some types t_1 and t_2. This function can be now used in any context in which the result type equals that of its first argument, e.g., `(integer -> integer) * integer -> (integer -> integer)`. Such functions are called *polymorphic*. Their type should be thought of as the *most general*, in the sense of the *least constrained*, type that one can associate them.

Let us consider several other polymorphic functions. For example, the type of the function `proc(x,y) x(y)`, which applies its first argument, expected therefore to be a function, to its second argument, is `(`t_1 `->` t_2`) *` t_1 `->` t_2. Also, the type of

```
proc(x,y) if x equals y then 1 else 0
```

is t `*` t `->` `integer`, because `x` and `y` are compared to each other so they must have the same type. However, the function

```
proc(x,y) if x equals x + 1 then x else y
```

is *not* polymorphic anymore, because `x + 1` implies that the type of `x` is `integer`, while the conditional implies that `x` and `y` have the same types, which is also the returning type of the function; the function's type is therefore `integer * integer -> integer`.

How can one infer the most general type of an expression then, by just analyzing it syntactically? The process of doing this is called *type inference* and consists of two steps:

- Collect information under the form of type parametric constraints by recursively analyzing the expression;
- Solve those constraints.

Collecting Type Information

In order to collect type information from an expression, we traverse the expression recursively, assign generic types to certain names and expressions, and then constrain those types. Let us consider, for example., the expression

```
proc(x,y) if x equals x + 1 then x else y
```

and let us manually simulate the type information collecting algorithm.

We have an expression of the form `proc <ParameterList> <Exp>`, so we assume some generic types for its parameters and then calculate the type of the expression. Let t_x and t_y be the generic types of `x` and `y`, respectively. If t_e is the type of function's body expression, then the type of the function will be

t_x `*` t_y `->` t_e.

Let us now calculate the type t_e while gathering type information as we traverse the body of the function.

The body of the function is a conditional, so we can now state a first series of constraints by analyzing the typing policy of the conditional: its condition is of type `bool` and its two branching expressions must have the same type, which will replace t_e. Assuming that t_b, t_1 and t_2 are the types of its condition and branching expressions, respectively, then we can state the type constrains

t_b `=` `bool`, and
t_1 `=` t_2.

We next calculate the types t_b, t_1 and t_2, collecting also the corresponding type information. The types t_1 and t_2 are easy to calculate because they can be just extracted from the type environment: t_x and t_y, respectively. Thus, the type equation t_1 `=`

t_2 above is in fact t_x `=` t_y.

To type the condition `x equals x + 1`, one needs to use the typing policy of `_equals_`: takes two arguments of equal type and returns type `bool`. Thus t_b is the type `bool`, but a new type constraint is generated, namely

t_3 `=` t_4,

where t_3 and t_4 are the types of the two subexpressions of the conditional. t_3 evaluates again to t_x, but in order to evaluate t_4 one needs to apply the typing policy of `_+_`: takes two `integer` arguments and returns an `integer`. This generates the constraint

t_x `=` `integer`.

The constraints

t_3 `=` t_4, and
t_b `=` `bool`,

become vacuous. However, after "walking" through all the expression and analyzing how names were used, we collected the following useful typing information:

t_x `=` t_y,
t_x `=` `integer`, and
the type of the original expression is t_x `*` t_y `->` t_x.

After we learn how to solve such type constraint equational systems we will be able to infer from here that the expression is correctly typed and its type is `integer * integer -> integer`.

Let us now consider the polymorphic function

`proc(x,y) (x(y) + 1)`.

After assigning generic types t_x and t_y to its parameters, while applying the typing policy on `_+_`, one infers that the type of `x(y)`, say $t_{x(y)}$, must be `integer`. When calculating $t_{x(y)}$, by applying the typing policy for function application, saying that the type of

`E(El)` for an expression `E` and a list of expressions `El` is `T` when the type of `El` is `Tp` and the type of `E` is `Tp -> T`, one infers the constraint t_x `=` t_y `->` $t_{x(y)}$. We have thus accumulated the following type information:

$t_{x(y)}$ `=` `integer`,
t_x `=` t_y `->` $t_{x(y)}$, and
the type of the function is t_x `*` t_y `->` `integer`.

We will be able to infer from here that the type of the function is `(`t `->` `integer)` `*` t `->` `integer`, for some generic type t, so the function is *polymorphic in* t.

The Constraint Collecting Technique

One can collect type constraints in different ways. We will do it by recursively traversing the expression to type only once. If the expression is

- a *name* then we return its type from the type environment and collect no auxiliary type constraint;
- a *basic type*, that is, `integer` or `bool`, then we return that type and collect no constraint;
- an *arithmetic operator* then we recursively calculate the types of its operand expressions accumulating the corresponding constraints, then add the type constraints that their type is integer, and then return `integer`;
- a *boolean operator* then we act like before, but return `bool`;

- a *declaration* of a name then we generate a fresh generic type, bind that name to it in the type environment, and then add an appropriate constraint for that type by calculating recursively the type of the bound expression;
- a *function* then generate fresh generic types for its parameters, bind them in the type environment accordingly, calculate the type of the body expression in the new environment accumulating all the constraints, and then return the corresponding function type;
- a *function application* then generate a fresh type t, calculate the type of function expression t_f and that of the argument expressions t_p accumulating all the type constraints, and then add the type constraint t_f `=` t_p `->` t.

Defining a Type Constraint Collector

We will again use the same programming language and analysis tool interface that we used to define the executable semantics and the static and the dynamic type checkers for our functional programming language.

We can just import the syntax of the language unchanged from the executable semantics definition. Also, the module `GENERIC-STATE` remains unchanged.

The `STATE` (module) of our type constraint collector will need to contain three attributes: a *type environment*, a *set of type equations*, and a *counter* needed to generate fresh types.

The constraint collector is defined in the same style as the other tools defined so far, namely inductively over the structure of the syntax. More precisely, we will define an operation


```
op preType :  Exp State -> TypeStatePair
```

which calculates the generic type of an expression in a given state, as well as a new state containing all the accumulated constraints.

We could have defined two distinct operations instead, one returning the type and the other the constraints (which can be regarded as "side effects"), as we did in the definition of the executable semantics. However, `preType` will combine both those operators into only one, thus allowing us to write more compact and more easily understandable Maude code.

We call the type returned by `preType` a *pre-type*, because in order to calculate the final type we need to solve the equational constraints. We will first focus on defining `preType` and then on solving the type constraints.

Defining Types

Like for the type checker, we introduce a specification defining types. Because of the more complex nature of type inference, and especially because of its ability to infer types in the context of polymorphism, we will need more structure on types than before. We declare subsorts `BasicType` and `TypeVar` of the sort `Type`:

```
fmod TYPE is protecting NAT .
  sorts BasicType TypeVar Type .
  subsorts BasicType TypeVar < Type .
  ops integer bool : -> BasicType .
  ops nothing fail : -> Type .
```

The type `fail` will be assigned to expressions which cannot be correctly typed, that is, whose collected type constraints cannot be solved. We also need to define a *type variable constructor* `op t : Nat -> TypeVar`, which will be used to generate fresh types as well

as to represent polymorphic types:

```
  op t : Nat -> TypeVar .
```

The product and function types are defined as before, with the difference that the identity attribute of `_*_` is replaced by the two corresponding equations:

```
  op _*_ : Type Type -> Type [assoc prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
  var T : Type .
  eq T * nothing = T .
  eq nothing * T = T .
endfm
```

The reason for doing so is to eliminate the unnecessary complications due to matching modulo identity.

Defining the Type Environment

The type environment, which is needed to store the bindings of names to type variables, is defined very closely to the type environment of the type checker. The following is cut-and-paste:

```
fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
            [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
```


However, in order to smooth the definition of the operation of adding all the bindings name-type generated by `let` and `letrec`, we define a new operation on type environments which inserts a type environment into another:

```
  op _<++_ : TypeEnvironment TypeEnvironment -> TypeEnvironment .
```

The semantics of type environment operators can be compactly and efficiently defined as follows:

```
  vars Ix Ix' : Index .  vars T T' : Type .
  vars TEnv TEnv' : TypeEnvironment .
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv) <++ ([Ix,T'] TEnv') = ([Ix,T'] TEnv) <++ TEnv' .
  eq TEnv <++ TEnv' = TEnv TEnv' [owise] .
endfm
```

Defining Type Equations

An *type equation* is a commutative pair of types. Sets of equations are defined as state attributes, via the attribute name `eqns`. The set of equations is kept small by removing the useless ones:

```
fmod EQUATIONS is extending GENERIC-STATE .
  protecting TYPE .
  sorts Equation Equations .
  subsorts Equation < Equations < StateAttribute .
  op eqns : -> StateAttributeName .
  op none : -> Equations .
  op _=_ : Type Type -> Equation [comm] .
  op _,_ : Equations Equations -> Equations [assoc comm id: none] .
  var Eq : Equation .  var T : Type .
  eq Eq,Eq = Eq .
  eq (T = T) = none .
endfm
```


Defining a Counter

A counter is needed in order to define fresh generic types. In programming languages providing support for side effects, it is a non-issue to define such a counter. Since Maude is a pure specification language, there is no notion of implicit state that rewrites can modify dynamically. Its only state is the term it reduces, which, in our case, includes also the explicit state of the programming language or analysis tool which is defined. Therefore, we need to define a counter like any other state attribute:

```
fmod TVAR-COUNTER is protecting NAT .
  extending GENERIC-STATE .
  sort Counter .
  op tvarCounter : -> StateAttributeName .
  op counter : Nat -> Counter .
  subsort Counter < StateAttribute .
```

```
  op get : Counter -> Nat .
  op inc : Counter -> Counter .
  var N : Nat .
  eq get(counter(N)) = N .
  eq inc(counter(N)) = counter(N + 1) .
endfm
```

Defining the State of the Constraint Collector

The state of the constraint collector, as well as the state of the entire type inferences, will have three attributes, as included below:

```
fmod STATE is
  protecting TYPE-ENVIRONMENT .
  protecting EQUATIONS .
  protecting TVAR-COUNTER .
```

An initial state is defined as a constant of sort `State`:


```
  op initState : -> State .
  eq initState = (tenv,empty) (eqns,none) (tvarCounter,counter(0)) .
```

Like in the previous Maude definitions, we introduce several operators on `State` which define the interface though which other future modules will interact with the state of the tool that we define. Unlike before, where the state update operations were defined on an item-by-item basis, we now provide operations that can insert a set of equations or an entire type environment into an existing state:

```
  op _[_] : State Index -> Type .
  op _<++_ : State Equations -> State .
  op _<++_ : State TypeEnvironment -> State .
  op _<**_ : State TypeEnvironment -> State .
  var S : State .  var Ix : Index .  var T : Type .
  var Eqns : Equations .  var TEnv : TypeEnvironment .
  eq S[Ix] = S[tenv][Ix] .
```

```
  eq S <++ Eqns = S[eqns <- (S[eqns], Eqns)] .
  eq S <++ TEnv = S[tenv <- (S[tenv] <++ TEnv)] .
  eq S <** TEnv = S[tenv <- TEnv] .
```

The operator _<**_ replaces the entire environment of a state by a different environment. This operation will be needed in order to recover the outer environment after the expression inside a `let`, `letrec`, or `proc` was typed.

Finally, we introduce a new sort and a corresponding constructor for pairs type-state,

```
  sort TypeStatePair .
  op {_,_} : Type State -> TypeStatePair .
```

together with an operation that generates a fresh type variable:

```
  op tvarFresh : State -> TypeStatePair .
  eq tvarFresh(S) = {t(get(S[tvarCounter])),
                     S[tvarCounter <- inc(S[tvarCounter])]} .
endfm
```


Pre-Typing Names

We next define an operation `preType` taking an expression and a state and returning a pair type-state, where the returned type is a temporary type calculated for the expression; that type will be "evaluated" into a concrete type only after solving the type constraints that are collected in the returned state.

We first define the `preType` operation on names. It just returns the current type bound to that name and the state remains unchanged:

```
fmod NAME-TYPE-INFERENCE is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op preType : Name State -> TypeStatePair .
  var X : Name .  var S : State .
  eq preType(X, S) = {S[idx(X)],S} .
endfm
```

Pre-Typing Generic (Lists of) Expressions

There is nothing special to say about how to pre-type integers or empty lists of expressions. However, one has to make sure that constraints are properly accumulated when one pre-types lists of expressions. A one argument `preType` operation is also defined in terms of the binary one on the initial state:

```
fmod GENERIC-EXP-TYPE-INFERENCE is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : Exp State -> TypeStatePair .
  op preType : ExpList State -> TypeStatePair .
  op preType : Exp -> TypeStatePair .
  var I : Int .  vars S S' Sp : State .  vars T Tp : Type .
  vars E E' : Exp .  var El : ExpList .
  eq preType(I, S) = {integer,S} .
  eq preType((),S) = {nothing,S} .
```



```
 ceq preType((E,E',El), S) = {T * Tp, Sp}
    if {T,S'}  := preType(E, S) /\ {Tp,Sp} := preType((E',El), S') .
  eq preType(E) = preType(E, initState) .
endfm
```

Pre-Typing Arithmetic Operations

When we pre-type a built-in operator, there are two things that we should consider, both derived from the typing policy of the language:

1. What are the expected types of its arguments and of its result;
2. How to propagate the type constraints.

Let us only consider the addition only, the others being similar. Consider the expression `E1 + E2` in some state `S`. Then we should first calculate the pre-type of `E1` in `S`; however, note that pre-typing

E1 can generate new type constraints, which need to be propagated properly. If T1 and S1 are the pre-type of and the state after pre-typing E1, then we can move to pre-typing E2 but we should make sure we do it in state S1 ... *plus* the constraint T1 = integer!

The returning pre-type of E1 + E2 is integer, but the constraints after pre-typing E2 *plus* T2 = integer should be now propagated:

```
fmod ARITH-OPS-TYPE-INFERENCE is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  vars S S1 S2 : State .  vars T1 T2 : Type .
 ceq preType(E1 + E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
---> other operators should come here
endfm
```

The reason for which we propagate the constraints sequentially is that one can potentially solve them *on-the-fly*, so the total set of

constraints is kept relatively small and the type information accumulated while pre-typing E1 can be used to pre-type E2 rather than at the end, which may be more effective in practice. We will solve the type constraints only once, at the end.

Exercise 1 *Modify the type inferencer defined in Lectures 12 and 13 such that to perform the type inference* on-the-fly.

Pre-Typing Boolean Expressions

The same idea as for pre-typing arithmetic operators is followed, propagating the appropriate constraints. For example, in the case of _equal_, the constraint is that the pre-types of its arguments should be equal; this constraint can only be propagated after both subexpressions are pre-typed:

```
fmod BEXP-TYPE-INFERENCE is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
```

```
  op preType : BExp State -> TypeStatePair .
  vars Be Be1 Be2 : BExp .
  vars E E1 E2 : Exp .  vars T T1 T2 : Type .  vars S S' S1 S2 : State .
 ceq preType(E1 equals E2, S) = {bool, S2 <++ T1 = T2}
    if {T1,S1} := preType(E1, S) /\ {T2,S2} := preType(E2, S1) .
 ceq preType(zero?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(even?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(not(Be), S) = {bool, S' <++ T = bool}
    if {T,S'} := preType(Be, S) .
 ceq preType(Be1 and Be2, S) = {bool, S2 <++ T2 = bool}
    if {T1,S1} := preType(Be1, S)
    /\ {T2,S2} := preType(Be2, S1 <++ T1 = bool) .
endfm
```


Pre-Typing Conditionals

The following definition is self-explanatory: first pre-type the condition to `Tb`, then by propagating the new constraints plus `Tb = bool` pre-type the left branch to `T1`, then by accumulating the new constraints pre-type the right branch to `T2`, and then return `T1` or `T2` together with the new type constraints enriched with `T1 = T2`:

```
fmod IF-TYPE-INFERENCE is protecting IF-SYNTAX .
  extending BEXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  var Be : BExp .
  vars S Sb S1 S2 : State .  vars T1 T2 Tb : Type .
 ceq preType(if Be then E1 else E2, S) = {T1, S2 <++ T1 = T2}
    if {Tb,Sb} := preType(Be, S)
    /\ {T1,S1} := preType(E1, Sb <++ Tb = bool)
    /\ {T2,S2} := preType(E2, S1) .
endfm
```

Pre-Typing Let

In order to pre-type `let`, we first pre-type each bound expression and carefully collect all the generated type constraints, then we assign fresh generic types to the names to be bound and add new corresponding type constraints, and then finally pre-type the body of the `let`. Let us start with the last step:

```
fmod LET-TYPE-INFERENCE is extending LET-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  var E : Exp .  var Bl : BindingList .  vars S Se Sf : State .
  vars Te Tf : Type .  var X : Name .  var TEnv : TypeEnvironment .
  var Eqns : Equations .
 ceq preType(let Bl in E, S) = {Te, Se <** S[tenv]}
    if {Te,Se} := preType(E, bindLet(Bl, S)) .
```

Notice that the state which is returned after pre-typing `let` *forgets* every binding possibly added in order to pre-type its body, but

retains all the generated type constraints as well as the new value of the fresh type counter! This is simply realized by `Se <** S[tenv]`, which replaces the type environment of `Se`, the state after pre-typing `let`'s body, by the original type environment.

Let us now define `bindLet`, which returns the state in which the body of `let` needs to be defined. In order to do it, we traverse the list of bindings and

1. pre-type each expression, collecting the corresponding constraints,
2. generate a fresh generic type for the bound name,
3. add an appropriate entry in the type environment and a corresponding type constraint.

Because of the semantics of `let`, we define an auxiliary operator which takes two additional arguments, a type environment and a set of equations:

```
  op bindLet : BindingList State -> State .
  op bindLet : BindingList State TypeEnvironment Equations -> State .
  eq bindLet(Bl, S) = bindLet(Bl, S, empty, none) .
 ceq bindLet((X = E, Bl), S, TEnv, Eqns) =
     bindLet(Bl, Sf, TEnv <++ [idx(X),Tf], (Eqns, Tf = Te))
    if {Te,Se} := preType(E,S) /\ {Tf,Sf} := tvarFresh(Se) .
  eq bindLet(none, S, TEnv, Eqns) = (S <++ TEnv) <++ Eqns .
endfm
```

Exercise 2 *Fresh generic types are added for each name bound by a* `let` *construct. Adding new types simplifies a bit our definitions, but, however, these new types are* not *needed. Modify the definition of constraint collector such that you do not define any new types when pre-typing* `let`*.*

Let us simulate pre-typing the expression below in the initial state:

```
let y = let x = 1 in x,
    z = let x = 1 in x
in let x = let x = 1 in x
   in y
```

A generic type `t(0)` will be first generated for the `x` bound in the `let` expression bound to `y`, together with the type constraint `t(0) = integer`; then a type `t(1)` will be generated for `y`, together with the type constraint `t(1) = t(0)`; then another type `t(2)` will be generated for the `x` bound in the expression bound to `z`, and a constraint `t(2) = integer`; a type `t(3)` is then generated for `z` together with `t(3) = t(2)`; two other types are further generated by the inner `let`s together with appropriate type constraint. However, the pre-type bound to `y`, namely `t(1)`, is returned as a pre-type of this expression.

After reduction, `Maude` returns the following result (notice that the

equality operation `_=_` was declared commutative, so Maude may rearrange the order of types in equations:

```
result TypeStatePair: {t(1), (tenv,empty)
  (eqns, integer = t(0), integer = t(2), integer = t(4),
           t(0) = t(1),    t(2) = t(3),    t(4) = t(5))
  (tvarCounter, counter(6))}
```

We will later see how these type constraints can be solved and eventually calculate the type of the expression above, `integer`.

Pre-Typing Parameters

There is not much to say here: a generic fresh type is generated for each parameter and a corresponding binding is added to the type environment; the product type of these fresh types is returned, together with the new state:


```
fmod PARAMETER-TYPE-INFERENCE is protecting PARAMETER-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : ParameterList State -> TypeStatePair .
  vars S Sp Sf : State .  vars Tp Tf : Type .
  var C : CallingMode .  var X : Name .  var Pl : ParameterList .
  eq preType((), S) = {nothing, S} .
 ceq preType((Pl, C X), S) = {Tp * Tf, Sf <++ [idx(X),Tf]}
    if {Tp,Sp} := preType(Pl,S) /\ {Tf,Sf} := tvarFresh(Sp) .
endfm
```

Pre-Typing Functions

As usual, we have to consider the two language constructs related to functions, namely function declaration and function invocation. For function declarations, the returned pre-type is the function type obtained after pre-typing the parameter list of the function using the `preType` operation defined above followed by pre-typing the body expression of the function. However, like for `let`, the

returned state *forgets* the parameter bindings:

```
fmod PROC-TYPE-INFERENCE is protecting PROC-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
  protecting PARAMETER-TYPE-INFERENCE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  vars S Se Sp Sf : State .  vars Tp Te Tf : Type .
 ceq preType(proc Pl E, S) = {Tp -> Te, Se <** S[tenv]}
    if {Tp,Sp} := preType(Pl,S) /\ {Te,Se} := preType(E,Sp) .
```

Function invocations are a bit tricky, because fresh types need to be generated. More precisely, a fresh type is generated for each function invocation, and it is assumed to be the type of the result. Knowing that type, we can generate the appropriate constraints:

```
 ceq preType(E El, S) = {Tf, Sf <++ Te = Tp -> Tf}
    if {Te,Se} := preType(E, S) /\ {Tp,Sp} := preType(El, Se)
    /\ {Tf,Sf} := tvarFresh(Sp) .
endfm
```


It is important to understand the need for the fresh type generated for the result of the function. One may argue that it is not needed, because one can simply pre-type `E`, which should pre-type to a function type `Tp -> T`, then `El` to some `Tp'`, and then return the type `T` and generate the obvious constraint `Tp = Tp'`. However, the problem is that `E` *may not pre-type to a function type*!

Consider, for example, the expression `proc(value x, value y) (x(y))`. When pre-typing `x(y)`, the only thing known about `x` is that it has some generic type, say `t(0)`. `y` has also a generic type, say `t(1)`. Then how about the type of `x(y)`? By generating a fresh type, say `t(2)`, for the result of the function application `x(y)`, we can conclude that the type of `x` should be `t(1) -> t(2)`, so we can generate the appropriate type constraint. Indeed, our Maude definition reduces `preType(proc(value x, value y) (x(y)))` to

```
result TypeStatePair: {t(0) * t(1) -> t(2), (tenv,empty)
     (eqns, t(0) = t(1) -> t(2)) (tvarCounter,counter(3))}
```

Homework Exercise 1 *Define* `preType` *on all the remaining language constructs, namely* `letrec`*, variable assignments, blocks, and* `while` *loops. Follow the modular approach, that is, do it by defining appropriate modules*

`LETREC-TYPE-INFERENCE`*,* `VAR-ASSIGNMENT-TYPE-INFERENCE`*,*

`BLOCK-TYPE-INFERENCE`*, and* `LOOP-TYPE-INFERENCE`*.*

Also, provide 2 working examples for each of the constructs above. Edit the provided `type-inference.maude` *file. Use the examples at the end to test your definitions. When submitting your homework, remove all those examples and add only your 8 new examples. We will test your definitions on all the examples that you provide, plus some of ours. So it is in your interest to provide us with examples as complicated as your definition can handle correctly.*

When pre-typing blocks, assume that all but the last statement in the block should have type `nothing`*.*

```
**************************************************************************
*** Defining Type Inference for a Functional Programming Language ***
**************************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm
```

```
fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op  _=_ :      Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op  __ :      CallingMode Name -> Parameter [prec 0] .
  op `(`) : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList
           [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm
```

```
fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



----------------------
--- Type Inference ---
```

```
fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod TYPE is protecting NAT .
  sorts BasicType TypeVar Type .
  subsorts BasicType TypeVar < Type .
  ops integer bool : -> BasicType .
  op t : Nat -> TypeVar .
  ops nothing fail : -> Type .
  op _*_ : Type Type -> Type [assoc prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
  var T : Type .
  eq T * nothing = T .
  eq nothing * T = T .
endfm

fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
          [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
  op _<++_ : TypeEnvironment TypeEnvironment -> TypeEnvironment .
  vars Ix Ix' : Index .  vars T T' : Type .
  vars TEnv TEnv' : TypeEnvironment .
```

```
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv) <++ ([Ix,T'] TEnv') = ([Ix,T'] TEnv) <++ TEnv' .
  eq TEnv <++ TEnv' = TEnv TEnv' [owise] .
endfm

fmod EQUATIONS is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Equation Equations .
  subsorts Equation < Equations < StateAttribute .
  op eqns : -> StateAttributeName .
  op none : -> Equations .
  op _=_ : Type Type -> Equation [comm] .
  op _,_ : Equations Equations -> Equations [assoc comm id: none] .
  var Eq : Equation .  var T : Type .
  eq Eq,Eq = Eq .
  eq (T = T) = none .
endfm

fmod TVAR-COUNTER is protecting NAT .
  extending GENERIC-STATE .
  sort Counter .
  op tvarCounter : -> StateAttributeName .
  op counter : Nat -> Counter .
  subsort Counter < StateAttribute .
  op get : Counter -> Nat .
  op inc : Counter -> Counter .
  var N : Nat .
  eq get(counter(N)) = N .
  eq inc(counter(N)) = counter(N + 1) .
endfm

fmod STATE is
  protecting TYPE-ENVIRONMENT .
  protecting EQUATIONS .
  protecting TVAR-COUNTER .
  op initState : -> State .
  op _[_] : State Index -> Type .
  op _<++_ : State Equations -> State .
  op _<++_ : State TypeEnvironment -> State .
  op _<**_ : State TypeEnvironment -> State .
  var S : State .  var Ix : Index .  var T : Type .
  var Eqns : Equations .  var TEnv : TypeEnvironment .
```

```
  eq initState = (tenv,empty) (eqns,none) (tvarCounter,counter(0)) .
  eq S[Ix] = S[tenv][Ix] .
  eq S <++ Eqns = S[eqns <- (S[eqns], Eqns)] .
  eq S <++ TEnv = S[tenv <- (S[tenv] <++ TEnv)] .
  eq S <** TEnv = S[tenv <- TEnv] .
  sort TypeStatePair .
  op {_,_} : Type State -> TypeStatePair .
  op tvarFresh : State -> TypeStatePair .
  eq tvarFresh(S) = {t(get(S[tvarCounter])),
                     S[tvarCounter <- inc(S[tvarCounter])]} .
endfm

fmod NAME-TYPE-INFERENCE is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op preType : Name State -> TypeStatePair .
  var X : Name .  var S : State .
  eq preType(X, S) = {S[idx(X)],S} .
endfm

fmod GENERIC-EXP-TYPE-INFERENCE is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : Exp State -> TypeStatePair .
  op preType : ExpList State -> TypeStatePair .
  op preType : Exp -> TypeStatePair .
  var I : Int .  vars S S' Sp : State .  vars T Tp : Type .
  vars E E' : Exp .  var El : ExpList .
  eq preType(I, S) = {integer,S} .
  eq preType((),S) = {nothing,S} .
 ceq preType((E,E',El), S) = {T * Tp, Sp}
    if {T,S'}  := preType(E, S) /\ {Tp,Sp} := preType((E',El), S') .
  eq preType(E) = preType(E, initState) .
endfm

fmod ARITH-OPS-TYPE-INFERENCE is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  vars S S1 S2 : State .  vars T1 T2 : Type .
 ceq preType(E1 + E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
 ceq preType(E1 * E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
```

```
 ceq preType(E1 - E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
 ceq preType(E1 / E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
endfm

fmod BEXP-TYPE-INFERENCE is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op preType : BExp State -> TypeStatePair .
  vars Be Be1 Be2 : BExp .
  vars E E1 E2 : Exp .  vars T T1 T2 : Type .  vars S S' S1 S2 : State .
 ceq preType(E1 equals E2, S) = {bool, S2 <++ T1 = T2}
    if {T1,S1} := preType(E1, S) /\ {T2,S2} := preType(E2, S1) .
 ceq preType(zero?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(even?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(not(Be), S) = {bool, S' <++ T = bool}
    if {T,S'} := preType(Be, S) .
 ceq preType(Be1 and Be2, S) = {bool, S2 <++ T2 = bool}
    if {T1,S1} := preType(Be1, S)
    /\ {T2,S2} := preType(Be2, S1 <++ T1 = bool) .
endfm

fmod IF-TYPE-INFERENCE is protecting IF-SYNTAX .
  extending BEXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  var Be : BExp .
  vars S Sb S1 S2 : State .  vars T1 T2 Tb : Type .
 ceq preType(if Be then E1 else E2, S) = {T1, S2 <++ T1 = T2}
    if {Tb,Sb} := preType(Be, S)
    /\ {T1,S1} := preType(E1, Sb <++ Tb = bool)
    /\ {T2,S2} := preType(E2, S1) .
endfm

fmod LET-TYPE-INFERENCE is extending LET-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op bindLet : BindingList State -> State .
  op bindLet : BindingList State TypeEnvironment Equations -> State .
  var E : Exp .  var Bl : BindingList .  vars S Se Sf : State .
  vars Te Tf : Type .  var X : Name .  var TEnv : TypeEnvironment .
  var Eqns : Equations .
```

```
 ceq preType(let Bl in E, S) = {Te, Se <** S[tenv]}
    if {Te,Se} := preType(E, bindLet(Bl, S)) .
  eq bindLet(Bl, S) = bindLet(Bl, S, empty, none) .
 ceq bindLet((X = E, Bl), S, TEnv, Eqns) =
     bindLet(Bl, Sf, TEnv <++ [idx(X),Tf], (Eqns, Tf = Te))
    if {Te,Se} := preType(E,S) /\ {Tf,Sf} := tvarFresh(Se) .
  eq bindLet(none, S, TEnv, Eqns) = (S <++ TEnv) <++ Eqns .
endfm

red preType(
  let y = let x = 1 in x,
      z = let x = 1 in x
  in let x = let x = 1 in x
     in y
) .

fmod PARAMETER-TYPE-INFERENCE is protecting PARAMETER-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : ParameterList State -> TypeStatePair .
  vars S Sp Sf : State .  vars Tp Tf : Type .
  var C : CallingMode .  var X : Name .  var Pl : ParameterList .
  eq preType((), S) = {nothing, S} .
 ceq preType((Pl, C X), S) = {Tp * Tf, Sf <++ [idx(X),Tf]}
    if {Tp,Sp} := preType(Pl,S) /\ {Tf,Sf} := tvarFresh(Sp) .
endfm

fmod PROC-TYPE-INFERENCE is protecting PROC-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
  protecting PARAMETER-TYPE-INFERENCE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  vars S Se Sp Sf : State .  vars Tp Te Tf : Type .
 ceq preType(proc Pl E, S) = {Tp -> Te, Se <** S[tenv]}
    if {Tp,Sp} := preType(Pl,S) /\ {Te,Se} := preType(E,Sp) .
 ceq preType(E El, S) = {Tf, Sf <++ Te = Tp -> Tf}
    if {Te,Se} := preType(E, S) /\ {Tp,Sp} := preType(El, Se)
    /\ {Tf,Sf} := tvarFresh(Sp) .
endfm

red preType(proc(value x) x) .
red preType(proc(need x, value y) 0) .
red preType((proc(name x, value y) 0) (2,3)) .
red preType((proc(value x, reference y) (x(y))) (proc(value x) 2, 3)) .
red preType((proc(value x, reference y) (x(y))) (proc(value x) y, 3)) .
```

```
fmod LETREC-TYPE-INFERENCE is extending LETREC-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

red preType(
  letrec z = letrec x = 1, y = x in y
  in letrec x = 7, y = x
     in y
) .

fmod VAR-ASSIGNMENT-TYPE-INFERENCE is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod BLOCK-TYPE-INFERENCE is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod LOOP-TYPE-INFERENCE is protecting LOOP-SYNTAX .
  extending BEXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod UNIFICATION is
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op type : Exp -> Type .
--- add your code here
endfm

fmod PROG-LANG-TYPE-INFERENCE is
  extending ARITH-OPS-TYPE-INFERENCE .
  extending IF-TYPE-INFERENCE .
  extending LET-TYPE-INFERENCE .
  extending PROC-TYPE-INFERENCE .
  extending LETREC-TYPE-INFERENCE .
  extending VAR-ASSIGNMENT-TYPE-INFERENCE .
  extending BLOCK-TYPE-INFERENCE .
  extending LOOP-TYPE-INFERENCE .
  extending UNIFICATION .
```

```
endfm

--- you can first preType all the examples below to make sure that
--- your "preType" homework problem is correct, and then you can type them
--- to make sure that your type inference procedure works properly

red type(
  let x = 5, y = 7
  in x + y
) .
red type(
  let x = 1
  in let x = x + 2
     in x + 1
) .
red type(
  let x = 1
  in let y = x + 2
     in x + 1
) .
red type(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
red type(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
red type(
  let x = 1
  in (x + (let x = 10 in x))
) .
red type(
  proc(value x, value y, value z) x * (y - z)
) .
red type(
  (proc(value y, value z) y + 5 * z) (1,2)
) .
red type(
```

```
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
red type(
  (proc(value x, value y) x(y))
    (proc(value z) 2 * z, 3)
) .
red type(
  proc(value x, value y) x
) .
red type(
  proc(value x, value y) (x(y))
) .
red type(
  proc(value x, value y) if x equals x + 1 then x else y
) .
red type(
  let x = proc(value x) x
  in x(x)
) .
red type(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
red type(
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
) .
red type(
  let y = 1
  in (proc(value x, value y) (x y))
       (proc(value x) y, 2)
) .
red type(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z
     in f(1,x)
) .
```

```
red type(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
) .
red type(
  let a = 3
  in let p = proc(value x) x + a,
         a = 5
     in a * p(2)
) .
red type(
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
red type(
  let f = proc(value n) n + n
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
red type(
  let a = 0
  in let a = 3, p = proc() a
     in let  a = 5,
---                f = proc(value x) (p())
               f = proc(value a) (p())
        in f(2)
) .
red type(
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
```

```
) .
red type(
  letrec f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
red type(
  letrec 'times4 = proc(value x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
red type(
  letrec 'even = proc(value x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(value x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
red type(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
red type(
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
red type(
  let c = 0
  in let f = proc()
         let c = c + 1
         in c
```

```
    in f() + f()
) .
red type(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
red type(
  let c = 0
  in let f = proc()
         let d = set c = c + 1
          in c
     in f() + f()
) .
red type(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
red type(
  let x = 0
  in let f = proc ( value x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
red type(
  let x = 0, y = 1
  in let f = proc(value x, value y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
red type(
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
```

```
        if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
red type(
  let x = 0
  in letrec 'even = proc() if zero?(x)
                    then 1
                    else let d = set x = x - 1
                        in 'odd(),
          'odd  = proc() if zero?(x)
                    then 0
                    else let d = set x = x - 1
                        in 'even()
     in let d = set x = 7
        in 'odd()
) .
red type(
  letrec x = 18,
     'even = proc() if zero?(x) then 1
                 else let d = set x = x - 1
                     in 'odd(),
          'odd = proc() if zero?(x) then 0
                    else let d = set x = x - 1
                        in 'even()
  in 'odd()
) .
red type(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
red type(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
red type(
  let 'times4 = 0
  in {
```

```
      set 'times4 = proc(value x)
                    if zero?(x)
                    then 0
                    else 4 + 'times4(x - 1) ;
      'times4(3)
    }
) .
red type(
  let x = 3, y = 4,
      f = proc(reference a, reference b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
      f(x,y) ;
      x
    }
) .
red type(
  let f = proc(need x) x + x
  in let y = 5
     in {
         f({set y = y + 3 ; 0}) ;
         y
       }
) .
red type(
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) {set x = x + 3 ; 0}
  in {
      f(g(y));
      y
    }
) .
red type(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
        set c = c + 1 ;
        if even?(n)
        then set n = n / 2
```

```
        else set n = 3 * n + 1
       } ;
       c }
) .
red type(
  let f = proc(value x, value g)
          if zero?(x)
          then 1
          else x * g(x - 1, g)
  in f(5, f)
) .
red type(
  let    x = 17,
      'odd = proc(value x, value o, value e)
             if zero?(x) then 0
             else e(x - 1, o, e),
     'even = proc(value x, value o, value e)
             if zero?(x) then 1
             else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
```

```
**********************************************************************
*** Defining Type Inference for a Functional Programming Language ***
**********************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sort Name .
  subsort Qid < Name .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op '(') : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op  _=_ :      Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod CALLING-MODE-SYNTAX is
  sort CallingMode .
endfm

fmod PARAMETER-SYNTAX is protecting CALLING-MODE-SYNTAX .
  protecting NAME-SYNTAX .
  sorts Parameter ParameterList .
  subsort Parameter < ParameterList .
  op  __ :      CallingMode Name -> Parameter [prec 0] .
  op '(') : -> ParameterList .
  op _,_ : ParameterList ParameterList -> ParameterList
            [assoc id:()] .
endfm

fmod CALL-BY-VALUE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op value : -> CallingMode .
endfm

fmod CALL-BY-REFERENCE-SYNTAX is extending CALLING-MODE-SYNTAX .
  op reference : -> CallingMode .
endfm

fmod CALL-BY-NAME-SYNTAX is extending CALLING-MODE-SYNTAX .
  op name : -> CallingMode .
endfm

fmod CALL-BY-NEED-SYNTAX is extending CALLING-MODE-SYNTAX .
  op need : -> CallingMode .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  extending PARAMETER-SYNTAX .
  extending CALL-BY-VALUE-SYNTAX .
  extending CALL-BY-REFERENCE-SYNTAX .
  extending CALL-BY-NAME-SYNTAX .
  extending CALL-BY-NEED-SYNTAX .
  op proc__ : ParameterList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


----------------------
--- Type Inference ---
----------------------

fmod GENERIC-STATE is
  sorts StateAttributeName StateAttribute State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op (_,_) : StateAttributeName StateAttribute -> State .
  op _[_] : State StateAttributeName -> StateAttribute [prec 0] .
  op _[_<-_] : State StateAttributeName StateAttribute -> State [prec 0] .
  vars N N' : StateAttributeName . vars A A' : StateAttribute . var S : State .
  eq ((N,A) S)[N] = A .
  eq ((N,A') S)[N <- A] = (N,A) S .
  eq S[N <- A] = S (N,A) [owise] .
endfm

fmod TYPE is protecting NAT .
  sorts BasicType TypeVar Type .
  subsorts BasicType TypeVar < Type .
  ops integer bool : -> BasicType .
  op t : Nat -> TypeVar .
  ops nothing fail : -> Type .
  op _*_ : Type Type -> Type [assoc prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
  var T : Type .
  eq T * nothing = T .
  eq nothing * T = T .
endfm

fmod TYPE-ENVIRONMENT is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Index Entry TypeEnvironment .
  subsort TypeEnvironment < StateAttribute .
  op tenv : -> StateAttributeName .
  subsort Entry < TypeEnvironment .
  op empty : -> TypeEnvironment .
  op [_,_] : Index Type -> Entry .
  op __ : TypeEnvironment TypeEnvironment -> TypeEnvironment
          [assoc comm id: empty] .
  op _[_] : TypeEnvironment Index -> Type .
  op _<++_ : TypeEnvironment TypeEnvironment -> TypeEnvironment .
  vars Ix Ix' : Index .  vars T T' : Type .
  vars TEnv TEnv' : TypeEnvironment .
  eq ([Ix,T] TEnv)[Ix] = T .
  eq ([Ix,T] TEnv) <++ ([Ix,T'] TEnv') = ([Ix,T'] TEnv) <++ TEnv' .
  eq TEnv <++ TEnv' = TEnv TEnv' [owise] .
endfm

fmod EQUATIONS is
  extending GENERIC-STATE .
  protecting TYPE .
  sorts Equation Equations .
  subsorts Equation < Equations < StateAttribute .
  op eqns : -> StateAttributeName .
  op none : -> Equations .
  op _=_ : Type Type -> Equation [comm] .
  op _,_ : Equations Equations -> Equations [assoc comm id: none] .
  var Eq : Equation .  var T : Type .
  eq Eq,Eq = Eq .
  eq (T = T) = none .
endfm

fmod TVAR-COUNTER is protecting NAT .
  extending GENERIC-STATE .
  sort Counter .
  op tvarCounter : -> StateAttributeName .
  op counter : Nat -> Counter .
  subsort Counter < StateAttribute .
  op get : Counter -> Nat .
  op inc : Counter -> Counter .
  var N : Nat .
  eq get(counter(N)) = N .
  eq inc(counter(N)) = counter(N + 1) .
endfm

fmod STATE is
  protecting TYPE-ENVIRONMENT .
  protecting EQUATIONS .
  protecting TVAR-COUNTER .
  op initState : -> State .
  op _[_] : State Index -> Type .
  op _<++_ : State Equations -> State .
  op _<++_ : State TypeEnvironment -> State .
  op _<**_ : State TypeEnvironment -> State .
  var S : State .  var Ix : Index .  var T : Type .
  var Eqns : Equations .  var TEnv : TypeEnvironment .
  eq initState = (tenv,empty) (eqns,none) (tvarCounter,counter(0)) .
  eq S[Ix] = S[tenv][Ix] .
  eq S <++ Eqns = S[eqns <- (S[eqns], Eqns)] .
  eq S <++ TEnv = S[tenv <- (S[tenv] <++ TEnv)] .
  eq S <** TEnv = S[tenv <- TEnv] .
  sort TypeStatePair .
  op {_,_} : Type State -> TypeStatePair .
  op tvarFresh : State -> TypeStatePair .
  eq tvarFresh(S) = {t(get(S[tvarCounter])),
                     S[tvarCounter <- inc(S[tvarCounter])]} .
endfm

fmod NAME-TYPE-INFERENCE is protecting NAME-SYNTAX .
  protecting STATE .
  op idx : Name -> Index .
  op preType : Name State -> TypeStatePair .
  var X : Name .  var S : State .
  eq preType(X, S) = {S[idx(X)],S} .
endfm

fmod GENERIC-EXP-TYPE-INFERENCE is protecting GENERIC-EXP-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : Exp State -> TypeStatePair .
  op preType : ExpList State -> TypeStatePair .
  op preType : Exp -> TypeStatePair .
  var I : Int .  vars S S' Sp : State .  vars T Tp : Type .
  vars E E' : Exp .  var El : ExpList .
  eq preType(I, S) = {integer,S} .
  eq preType((),S) = {nothing,S} .
 ceq preType((E,E',El), S) = {T * Tp, Sp}
    if {T,S'}  := preType(E, S) /\ {Tp,Sp} := preType((E',El), S') .
  eq preType(E) = preType(E, initState) .
endfm

fmod ARITH-OPS-TYPE-INFERENCE is protecting ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  vars S S1 S2 : State .  vars T1 T2 : Type .
 ceq preType(E1 + E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
 ceq preType(E1 * E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
 ceq preType(E1 - E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
 ceq preType(E1 / E2, S) = {integer, S2 <++ T2 = integer}
    if {T1,S1} := preType(E1, S)
    /\ {T2,S2} := preType(E2, S1 <++ T1 = integer) .
endfm
```

```
fmod BEXP-TYPE-INFERENCE is protecting BEXP-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op preType : BExp State -> TypeStatePair .
  vars Be Be1 Be2 : BExp .
  vars E E1 E2 : Exp .  vars T T1 T2 : Type .  vars S S' S1 S2 : State .
 ceq preType(E1 equals E2, S) = {bool, S2 <++ T1 = T2}
    if {T1,S1} := preType(E1, S) /\ {T2,S2} := preType(E2, S1) .
 ceq preType(zero?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(even?(E), S) = {bool, S' <++ T = integer}
    if {T,S'} := preType(E, S) .
 ceq preType(not(Be), S) = {bool, S' <++ T = bool}
    if {T,S'} := preType(Be, S) .
 ceq preType(Be1 and Be2, S) = {bool, S2 <++ T2 = bool}
    if {T1,S1} := preType(Be1, S)
    /\ {T2,S2} := preType(Be2, S1 <++ T1 = bool) .
endfm

fmod IF-TYPE-INFERENCE is protecting IF-SYNTAX .
  extending BEXP-TYPE-INFERENCE .
  vars E1 E2 : Exp .  var Be : BExp .
  vars S Sb S1 S2 : State .  vars T1 T2 Tb : Type .
 ceq preType(if Be then E1 else E2, S) = {T1, S2 <++ T1 = T2}
    if {Tb,Sb} := preType(Be, S)
    /\ {T1,S1} := preType(E1, Sb <++ Tb = bool)
    /\ {T2,S2} := preType(E2, S1) .
endfm

fmod LET-TYPE-INFERENCE is extending LET-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op bindLet : BindingList State -> State .
  op bindLet : BindingList State TypeEnvironment Equations -> State .
  var E : Exp .  var Bl : BindingList .  vars S Se Sf : State .
  vars Te Tf : Type .  var X : Name .  var TEnv : TypeEnvironment .
  var Eqns : Equations .
 ceq preType(let Bl in E, S) = {Te, Se <** S[tenv]}
    if {Te,Se} := preType(E, bindLet(Bl, S)) .
  eq bindLet(Bl, S) = bindLet(Bl, S, empty, none) .
 ceq bindLet((X = E, Bl), S, TEnv, Eqns) =
     bindLet(Bl, Sf, TEnv <++ [idx(X),Tf], (Eqns, Tf = Te))
    if {Te,Se} := preType(E,S) /\ {Tf,Sf} := tvarFresh(Se) .
  eq bindLet(none, S, TEnv, Eqns) = (S <++ TEnv) <++ Eqns .
endfm

red preType(
  let y = let x = 1 in x,
      z = let x = 1 in x
  in let x = let x = 1 in x
     in y
) .

fmod PARAMETER-TYPE-INFERENCE is protecting PARAMETER-SYNTAX .
  protecting NAME-TYPE-INFERENCE .
  op preType : ParameterList State -> TypeStatePair .
  vars S Sp Sf : State .  vars Tp Tf : Type .
  var C : CallingMode .  var X : Name .  var Pl : ParameterList .
  eq preType((), S) = {nothing, S} .
 ceq preType((Pl, C X), S) = {Tp * Tf, Sf <++ [idx(X),Tf]}
    if {Tp,Sp} := preType(Pl,S) /\ {Tf,Sf} := tvarFresh(Sp) .
endfm

fmod PROC-TYPE-INFERENCE is protecting PROC-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
  protecting PARAMETER-TYPE-INFERENCE .
  var Pl : ParameterList .  var E : Exp .  var El : ExpList .
  vars S Se Sp Sf : State .  vars Tp Te Tf : Type .
 ceq preType(proc Pl E, S) = {Tp -> Te, Se <** S[tenv]}
    if {Tp,Sp} := preType(Pl,S) /\ {Te,Se} := preType(E,Sp) .
 ceq preType(E El, S) = {Tf, Sf <++ Te = Tp -> Tf}
    if {Te,Se} := preType(E, S) /\ {Tp,Sp} := preType(El, Se)
    /\ {Tf,Sf} := tvarFresh(Sp) .
endfm

red preType(proc(value x) x) .
red preType(proc(need x, value y) 0) .
red preType((proc(name x, value y) 0) (2,3)) .
red preType((proc(value x, reference y) (x(y))) (proc(value x) 2, 3)) .
red preType((proc(value x, reference y) (x(y))) (proc(value x) y, 3)) .

fmod LETREC-TYPE-INFERENCE is extending LETREC-SYNTAX .
  protecting GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

red preType(
  letrec z = letrec x = 1, y = x in y
  in letrec x = 7, y = x
     in y
) .

fmod VAR-ASSIGNMENT-TYPE-INFERENCE is protecting VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod BLOCK-TYPE-INFERENCE is protecting BLOCK-SYNTAX .
  extending GENERIC-EXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod LOOP-TYPE-INFERENCE is protecting LOOP-SYNTAX .
  extending BEXP-TYPE-INFERENCE .
--- add your code here
endfm

fmod UNIFICATION is
  protecting GENERIC-EXP-TYPE-INFERENCE .
  op type : Exp -> Type .
--- add your code here
endfm

fmod PROG-LANG-TYPE-INFERENCE is
  extending ARITH-OPS-TYPE-INFERENCE .
  extending IF-TYPE-INFERENCE .
  extending LET-TYPE-INFERENCE .
  extending PROC-TYPE-INFERENCE .
  extending LETREC-TYPE-INFERENCE .
  extending VAR-ASSIGNMENT-TYPE-INFERENCE .
  extending BLOCK-TYPE-INFERENCE .
  extending LOOP-TYPE-INFERENCE .
  extending UNIFICATION .
endfm

--- you can first preType all the examples below to make sure that
--- your "preType" homework problem is correct, and then you can type them
--- to make sure that your type inference procedure works properly

red type(
  let x = 5, y = 7
  in x + y
) .

red type(
  let x = 1
  in let x = x + 2
     in x + 1
) .
red type(
  let x = 1
  in let y = x + 2
     in x + 1
) .
red type(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
red type(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
red type(
  let x = 1
  in (x + (let x = 10 in x))
) .
red type(
  proc(value x, value y, value z) x * (y - z)
) .
red type(
  (proc(value y, value z) y + 5 * z) (1,2)
) .
red type(
  let f = proc(value y, value z) y + 5 * z
  in f(1,2) + f(3,4)
) .
red type(
  (proc(value x, value y) x(y))
    (proc(value z) 2 * z, 3)
) .
red type(
  proc(value x, value y) x
) .
red type(
  proc(value x, value y) (x(y))
) .
red type(
  proc(value x, value y) if x equals x + 1 then x else y
) .
red type(
  let x = proc(value x) x
  in x(x)
) .
red type(
  let f = proc(value x, value y) x + y,
      g = proc(value x, value y) x * y,
      h = proc(value x, value y, value a, value b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
red type(
  let y = 1
  in let f = proc(value x) y
     in let y = 2
        in f(0)
) .
red type(
  let y = 1
  in (proc(value x, value y) (x y))
       (proc(value x) y, 2)
) .
red type(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z
     in f(1,x)
) .
red type(
  let x = 1
  in let x = 2,
         f = proc(value y, value z) y + x * z,
         g = proc(value u) u + x
     in f(g(3), 4)
) .
red type(
  let a = 3
  in let p = proc(value x) x + a,
         a = 5
     in a * p(2)
) .
red type(
  let f = proc(value n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
red type(
  let f = proc(value n) n + n
  in let f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
red type(
  let a = 0
  in let a = 3, p = proc() a
     in let  a = 5,
---              f = proc(value x) (p())
                f = proc(value a) (p())
        in f(2)
) .
red type(
  let 'makemult = proc(value 'maker, value x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(value x) ('makemult('makemult,x))
     in 'times4(3)
) .
red type(
  letrec f = proc(value n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
red type(
  letrec 'times4 = proc(value x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
```

```
  in 'times4(3)
) .
red type(
  letrec 'even = proc(value x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(value x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
red type(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
red type(
  let x = 10
  in letrec f = proc(value y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
red type(
  let c = 0
  in let f = proc()
         let c = c + 1
         in c
     in f() + f()
) .
red type(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
red type(
  let c = 0
  in let f = proc()
         let d = set c = c + 1
          in c
     in f() + f()
) .
red type(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
red type(
  let x = 0
  in let f = proc ( value x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
red type(
  let x = 0, y = 1
  in let f = proc(value x, value y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
red type(
  let x = 0, y = 3, z = 4,
      f = proc(value a, value b, value c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
red type(
  let x = 0
  in letrec 'even = proc() if zero?(x)
                           then 1
                           else let d = set x = x - 1
                                in 'odd(),
            'odd  = proc() if zero?(x)
                           then 0
                           else let d = set x = x - 1
                                in 'even()
     in let d = set x = 7
        in 'odd()
) .
red type(
  letrec x = 18,
     'even = proc() if zero?(x) then 1
                    else let d = set x = x - 1
                         in 'odd(),
             'odd = proc() if zero?(x) then 0
                           else let d = set x = x - 1
                                in 'even()
  in 'odd()
) .
red type(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
red type(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
red type(
  let 'times4 = 0
  in {
       set 'times4 = proc(value x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
red type(
  let x = 3, y = 4,
      f = proc(reference a, reference b)
            {
              set a = a + b ;
              set b = a - b ;
              set a = a - b
            }
  in {
       f(x,y) ;
       x
     }
) .
red type(
  let f = proc(need x) x + x
  in let y = 5
     in {
          f({set y = y + 3 ; 0}) ;
          y
        }
) .
red type(
  let y = 5,
      f = proc(need x) x + x,
      g = proc(reference x) {set x = x + 3 ; 0}
  in {
       f(g(y));
       y
     }
) .
red type(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
red type(
  let f = proc(value x, value g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
red type(
  let    x = 17,
      'odd = proc(value x, value o, value e)
               if zero?(x) then 0
               else e(x - 1, o, e),
     'even = proc(value x, value o, value e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
```

CS322 - Programming Language Design

Lecture 13: Typed Languages - Type Inference (part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Last lecture we have seen how one can collect type constraints by traversing the program syntactically and analyzing how names are used. We have defined an operation, called `preType`, which took an expression and returned a pair `{T,S}`, where `T` could be any term of sort `Type` as defined by the specification of types:

```
fmod TYPE is protecting NAT .
  sorts BasicType TypeVar Type .
  subsorts BasicType TypeVar < Type .
  ops integer bool : -> BasicType .
  op t : Nat -> TypeVar .
  ops nothing fail : -> Type .
  op _*_ : Type Type -> Type [assoc prec 1] .
  op _->_ : Type Type -> Type [prec 2] .
  var T : Type .
  eq T * nothing = T .
  eq nothing * T = T .
endfm
```

We refer to terms `t(0)`, `t(1)`, ..., as *type variables*. The state `S` returned by `preType` could contain, among other pieces of information, a set of equations between types involving type variables, which we called *type constraints*.

We should understand the result `{T,S}` of `preType(E)` as follows:

> The type of `E` is `T` in which all the type variables are *substituted* by the types obtained after *solving* the system of equations in `S`. `E` is *not guaranteed* a type *a priori*! If any *conflict* is detected while solving the system then we say that the process of typing `E` failed, and we conclude that the expression is not correctly typed.

If `Eqns` is a set of type equations, that is, a term of sort `Equations`, and `T` is a type potentially involving type variables, then we let `Eqns[T]` denote the type obtained after solving `Eqns` and then substituting the type variables accordingly in `T`; if conflicts are

detected then `Eqns[T]` is the type `fail`.

Once such a magic operation `_[_] : Equations Type -> Type` is defined, we can easily infer the type of an expression:

```
ceq type(E) = Se[eqns][Te] if {Te,Se} := preType(E) .
```

In this lecture we describe a relatively simple equational technique to define the operation `_[_] : Equations Type -> Type`.

Homework Exercise 1 *Formalize in Maude the technique that will be next presented. What you have to do is to add your* `Maude` *code in the module* `UNIFICATION` *in the provided file* `type-inference.maude`. *You can use the examples at the end of the file to test your definition.*

The Type Inference Procedure by Examples

Let us consider the following expression:

```
(proc(value x, value y) x(y))
  (proc(value z) 2 * z * z * 3, 3)
```

After pre-typing, we get the pre-type `t(4)` and a state containing the three equations:

```
integer = t(3),
t(0) = t(1) -> t(2),
t(0) * t(1) -> t(2) = (t(3) -> integer) * integer -> t(4)
```

Why? Were there any other equations which were useless and consequently removed with the two simplifying rules (to see this, remove the two equations before pre-typing):

```
eq Eq, Eq = Eq .
eq (T = T) = none .
```


How can we solve the system of equations above? We can immediately notice that `t(3) = integer`, so we can just substitute `t(3)` accordingly in the other equations, obtaining:

```
t(0) = t(1) -> t(2),
t(0) * t(1) -> t(2) = (integer -> integer) * integer -> t(4).
```

Moreover, since `t(0) = t(1) -> t(2)`, by substitution we obtain

```
(t(1) -> t(2)) * t(1) -> t(2) = (integer -> integer) * integer -> t(4).
```

Both types in the equation above are function types, so their source and target domains must be equal. This observation allows us to generate two other type equations, namely:

```
(t(1) -> t(2)) * t(1) = (integer -> integer) * integer,
t(2) = t(4).
```

Now, substituting `t(2)` by `t(4)` in the first equation we get:

```
(t(1) -> t(4)) * t(1) = (integer -> integer) * integer,
```

that is, an equality of product types, which yields the following:

```
t(1) -> t(4) = integer -> integer,
t(1) = integer.
```

Substituting `t(1)` in the first equation and then equaling the source and target types of the function type, we get

```
integer = integer,
t(4) = integer.
```

The first equation is useless and will be promptly removed by the simplifying rule `(T = T) = none` in `EQUATIONS`. The second equation gives the desired type of our expression, `integer`.

Let us now see an example expression which cannot be typed, e.g.:

```
  let x = proc(value x) x in x(x)
```

After pre-typing, we get the type `t(2)` constrained by:

```
t(1) = t(0) -> t(0),   t(1) = t(1) -> t(2).
```


We can already see that the second equation is problematic because of a recursive definition of `t(1)`, but let us just follow the procedure blindly to see what happens. By substituting `t(1) = t(0) -> t(0)` in the second equation, we get

```
t(0) -> t(0) = (t(0) -> t(0)) -> t(2).
```

Further, this can be split in two other equations,

```
t(0) = t(0) -> t(0),
t(0) = t(2).
```

The first equation is again problematic, but let us ignore it and substitute the second into the first, obtaining

```
t(2) = t(2) -> t(2).
```

Now there is no way to continue, so the type variable `t(2)` cannot be assigned a type. That the expression cannot be typed, so our procedure will simply return the type `fail`.

Unification

The technique that we used to solve the type equations is called *unification*. In general the equational unification problem can be stated as follows, where we only consider the mono-sorted case. Our particular unification problem fits this framework, because we can consider that we have only one sort, `Type`.

Let Σ be a signature over only one sort, let X be a finite set of variables, and let

$$\mathcal{E} = \{t_1 = t'_1, ..., t_n = t'_n \mid t_1, t'_1, ..., t_n, t'_n \in T_\Sigma(X)\}$$

be a finite set of pairs of Σ-terms over variables in X, which from now on we call *equations*. Notice, however, that these are different from the standard equations in equational logic, which are quantified universally.

A map, or a substitution, $\theta : X \to T_\Sigma(X)$ is called a *unifier* of $\mathcal{E}$ if

and only if $\theta^\star(t_i) = \theta^\star(t'_i)$ for all $1 \leq i \leq n$, where $\theta^\star : T_\Sigma(X) \to T_\Sigma(X)$ is the natural extension of $\theta : X \to T_\Sigma(X)$ to entire terms, by substituting each variable x by its corresponding term, $\theta(x)$.

A unifier $\theta : X \to T_\Sigma(X)$ is *more general* than $\varphi : X \to T_\Sigma(X)$ when φ can be obtained from θ by further substitutions; formally, when there is some other map, say $\rho : X \to T_\Sigma(X)$, such that $\varphi = \theta; \rho^\star$, where the composition of function was written sequentially. Let us consider again our special case signature, that of types, and the equations $\mathcal{E}$:

```
t(0) = t(1) -> t(2),
t(0) * t(1) -> t(2) = (t(3) -> t(3)) * (t(4) -> t(4)) -> t(5),
```

Here `t(0)`, ..., `t(5)` are seen as variables in X. These equations can, e.g., be generated when pre-typing the expression:

```
(proc(value x, value y) (x y))
  (proc(value x) x, proc(value y) y)
```

One unifier for these equations, say φ, takes `t(0)` to

`(integer -> integer) -> (integer -> integer)`,

`t(1)`, `t(2)` and `t(3)` to `integer -> integer`, and `t(4)` to `integer`. Another unifier, say θ, takes `t(0)` to

`(t(4) -> t(4)) -> (t(4) -> t(4))`,

and `t(1)`, `t(2)` and `t(3)` to `t(4) -> t(4)`. Then it is clear that θ is *more general* than φ, because $\varphi = \theta; \rho^\star$, where ρ simply takes `t(4)` to `integer`.

If a set of equations $\mathcal{E}$ admits a unifier, they are called *unifiable*. Note that there can be situations, like the ones we have already seen for our special case of signature of types, when a set of equations is not unifiable.

Finding the Most General Unifier

When a set of equations $\mathcal{E}$ is unifiable, its *most general unifier*, written $mgu(\mathcal{E})$ is one which is more general than any other unifier of $\mathcal{E}$. Note that typically there are more than one mgu.

We will next discuss a general procedure for finding an mgu for a set of equations $\mathcal{E}$ over an arbitrary mono-sorted signature Σ. Since we actually need to apply that mgu to the type calculated by the `preType` operation for the expression to type, we will provide a technique that directly calculates $\mathcal{E}[t]$ for any term t, which calculates the term after applying the substitution $mgu(\mathcal{E})$ to t, that is, $(mgu(\mathcal{E}))^\star(t)$.

The technique is in fact quite simple. It can be defined entirely equationally, but in order to make the desired direction clear we write them as rewriting rules (going from left to right). It consists

of iteratively applying the following two steps to $\mathcal{E}[t]$:

1. $(x = u, \mathcal{E})[t] \rightarrow subst(x, u, \mathcal{E})[subst(x, u, t)]$, if x is some variable in X, u is a term in $T_\Sigma(X)$ which does *not* contain any occurrence of x, and $subst(x, u, \mathcal{E})$ and $subst(x, u, t)$ substitute each occurrence of x in $\mathcal{E}$ and t, respective, by u;
2. $(\sigma(t_1, ..., t_k) = \sigma(t'_1, ..., t'_k), \mathcal{E})[t] \rightarrow (t_1 = t'_1, ..., t_k = t'_k, \mathcal{E})[t]$, if $\sigma \in \Sigma$ is some operation of k arguments.

Theorem. *When none of the steps above can be applied anymore, we obtain a potentially modified final set of equations and a final term, say $\mathcal{E}_f[t_f]$. If $\mathcal{E}_f$ is empty then t_f is $(mgu(\mathcal{E}))^\star(t)$, so it can be returned as the type of the original expression. If $\mathcal{E}_f$ is not empty then $\mathcal{E}$ is not unifiable, so the original expression cannot be typed.*

Exercise 1 *Read Chapter 4 in the Friedman book.*

Errata & Challenge

The executable semantics of our functional language seems to have a *bug*. Thanks Jodie and Erin for discovering it! I'll give 2 extra points to the first who discovers and fixes this bug :-)

The bug seems to be related to `letrec` (of course, what else?). The following program

```
letrec f = proc(value x) z + x + 5,
       y = 2,
       a = 3,
       z = let y = 5, a = 6 in y + a
in f(a)
```

is evaluated to `10` instead of ... what?

Project

The CS322 project represents 25% of your grade and it is *individual*! The project consists of defining the executable semantics of a variant of an existing language, namely GNU BC (see `http://www.gnu.org/manual/bc`).

We will ignore some of BC's features, but, more importantly, we will modify some of them and will add some new ones.

These are features that we will *ignore*:

- command line options,
- comments,
- real numbers (we will use rationals instead),
- special variables, such as *ibase* or *scale*,
- the special expressions, except *read()*,
- strings,
- pseudo statements,

- math library functions,
- vectors/arrays,
- auto variables, but we will add local variables instead.

One of the most important features of BC that we will modify is its *dynamic scoping*. We will define it as a statically scoped language.

A feature that we will add to our definition of BC is that of nested functions. In other words, functions can be defined inside other functions.

Also, all variables must be declared before they are used, using the keyword `var`. One or more variables can be declared at the same time:

```
var x, y, z ;
```

Variables can be declared anywhere, including blocks and functions. The scope of a variable declaration spans all the (lexically) subsequent statements, including blocks, sub-blocks, etc.

They can be, of course, shadowed by other variable declarations.

We will have only one implicit type, for rational numbers, and functions cannot take other functions as arguments, so we do not need to worry about types and type checking/inference. We will use Maude's `RAT` module for rational numbers.

As we already know, static scoping brings difficulties in handling recursion. To keep things simple, we assume that all the functions declared in a block see each other's declarations; so a block behaves like a `letrec` with respect to functions.

There will be many clarifying discussions on the newsgroup. Make sure you check it often.

***Warning*: Do not let it for the last two days!**

CS322 - Programming Language Design

Lecture 14: Defining an Object-Oriented Programming Language (part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

During this and the next lecture we will define a simple object-oriented functional programming language, which will have many of the important features of the more advanced OO programming languages.

As for the languages that we defined previously, the very first step is to *understand* the language to be defined. This lecture we will see several *example programs* in our language and will introduce the *basic concepts* of object-oriented programming.

Objects and Classes

An object can be abstractly thought of as an *encapsulated state*, which one can interact with via an *interface*. The state maps, as usual, names to values. However, the "names" forming the state of an object are called *fields*. The interface allowing one to read or write the state of an object consists of *methods*, which are nothing but functions acting on that object's state. The process of invoking an object's method is often regarded as *sending that object a message* containing the method name and the actual arguments.

Object-oriented programming is a programming language paradigm which facilitates defining, handling and coordinating objects.

It is often the case that several objects are intended to have the same structure, that is, the same fields and the same interface to the outside world. For example, one can have several stacks or

several queues in a program, each with its own particular state. To facilitate defining several objects sharing the same structure, object-oriented programming languages provide *classes*. Then objects are built as *instances* of classes:

```
class c extends object
  field a
  method initialize() set a = 5
  method m() a
main
  let x = new c()
  in send x m()
```

The `new` language construct creates an instance object of a class. A default convention in our language is that, whenever an instance is created, the special method `initialize` is immediately invoked; this operation is expected to assign initial values to the fields of the newly created object. The `send` construct sends a message, that is a

method invocation request, to an object. The above evaluates to `5`.

The following class defines two fields, `i` and `j`; its objects will preserve the invariant `i + j = 0`:

```
class c extends object
  field i
  field j
  method initialize(x)
    { set i = x ;
      set j = 0 - x }
  method add(d)
    { set i = i + d ;
      set j = j - d }
  method getstate()
    list(i,j)
```

Let us now consider the following which creates an object `o` and then sends it the message `add(3)`:


```
main
   let a = 0, b = 0, o = new c(5)
   in { set a = send o getstate() ;
        send o add(3) ;
        set b = send o getstate() ;
        list(a,b) }
```

This will evaluate to the list `[5,-5][8,-8]`. Nested lists will be defined as well in our OO language.

Self References

While evaluating their methods, objects can send messages to themselves in order to invoke other methods. For example, the following evaluates to 13:

```
class c extends object
  method initialize() 1
  method m1() send self m2()
  method m2() 13
main
  let o = new c()
  in send o m1()
```


Self References and Recursion

Recursion can be very elegantly supported by the OO paradigm, because it can be very easily and intuitively explained and handled via message passing:

```
class oddeven extends object
  method initialize() 1
  method even(n)
    if zero?(n) then 1 else send self odd(n - 1)
  method odd(n)
    if zero?(n) then 0 else send self even(n - 1)
main
  let o = new oddeven()
  in send o odd(17)
```

So in some sense, the above is equivalent to a `letrec`.

Dynamic Method Dispatch

One of the most pleasant features of OO programming is the capability of objects to potentially send any messages between themselves. Let us consider the following class:

```
class node extends object
  field left
  field right
  method initialize(l,r)
    { set left = l ;
      set right = r }
  method sum() (send left sum()) + (send right sum())
```

Without any apriori knowledge regarding its fields, an object of the class above assumes that both `left` and `right` will be objects providing a method `sum()` in their interface. They can be instances of `node`, a subclass of it (subclasses will be discussed in the sequel),

or of any other class providing a method `sum()`. For example, they can be instances of the following class:

```
class leaf extends object
  field value
  method initialize(v) set value = v
  method sum() value
```

Thus, if an object "knows" that another object is expected to have a certain method as part of its interface, then the former can just send a message invoking that method of the second object. The following will therefore be evaluated to `12`:

```
main
  let o = new node(new node(new leaf(3),
                            new leaf(4)),
                   new leaf(5))
  in send o sum()
```

This dynamic style of method invocation is called *dynamic method dispatch*, and it turns out to be of crucial importance in the context of *subclass polymorphism*, which will be explained later.

Most OO programming languages, including Java, use dynamic method dispatch, which is what we will also consider for our language. However, there are languages using *static method dispatch*, such as C++, which has the advantage that allows more efficient language implementations. Why? However, due to the practical and methodological importance of dynamic method dispatching, languages like C++ provide so called *virtual* classes, whose methods are dynamically dispatched.

Inheritance

There are many situations in which one would like to define a new class by just slightly modifying an already existing class by adding new fields or changing the behavior of some methods. We say that the new class *inherits*, or *extends*, or is a *subclass* of the old one. Consider, e.g., the following class defining two-dimensional points

```
class point extends object
  field x
  field y
  method initialize(initx, inity)
    { set x = initx ; set y = inity }
  method move(dx, dy)
    { set x = x + dx ; set y = y + dy }
  method getLocation() list(x,y)
```

One may want to extend it by defining colored points. So one would

like to only define a new field, say `color`, together with methods to read and write it, preserving the already defined behavior of points:

```
class colorpoint extends point
  field color
  method setColor(c) set color = c
  method getColor()  color
```

Let us now consider the following scenario, in which a point and colored point are created:

```
main
 let p = new point(3,4), cp = new colorpoint(10,20) in
 { send p move(3,4) ;
   send cp setColor(87) ; send cp move(10, 20) ;
   list(send p getLocation(),
        send cp getLocation(), send cp getColor()) }
```

This should return the list `[[6,8],[20,40],87]`.

Let us now consider a more complex situation, in which a subclass

accesses its superclass fields directly:

```
class c1 extends object
  field x
  field y
  method initialize() 1
  method setx1(v) set x = v
  method sety1(v) set y = v
  method getx1() x
  method gety1() y
class c2 extends c1
  field y
  method sety2(v) set y = v
  method getx2() x
  method gety2() y
```

Therefore, subclass' fields can *shadow* some of superclass' fields. The field accessing rule is quite simple: any object instance of the subclass will search for its fields first within the subclass definition

and then within the superclass definition. Thus, the following will evaluate to the list `[101,102,101,999]`:

```
main
  let o2 = new c2() in
  { send o2 setx1(101) ;
    send o2 sety1(102) ;
    send o2 sety2(999) ;
    list(send o2 getx1(), send o2 gety1(),
         send o2 getx2(), send o2 gety2()) }
```

Our language will by *single inheritance*, that is, its classes are allowed to inherit only one class. Many OO programming languages today are single inheritance. While *multiple inheritance* may look like an obvious extension, it is typically problematic in practice (C++ supports it).

Overriding

If a subclass defines a method which has the same name as a method of its superclass (or some ancestor superclass), we say that the new method *overrides* the old one.

The behavior of overridden methods depends on the dispatch style, dynamic or static. Consider for example the following program:

```
class c1 extends object
  method initialize() 1
  method m1() 1
  method m2() send self m1()
class c2 extends c1
  method m1() 2
main
  let o1 = new c1(), o2 = new c2()
  in list(send o1 m1(), send o2 m1(), send o2 m2())
```

Which `m1()` is meant in the expression body of `m2()` within an instance object of `c2`? Under dynamic method dispatch it refers to the `m1()` defined in `c2`, while under static method dispatch to the `m1()` defined in `c1`. So the above evaluates to `[1,2,2]` in the first case and to `[1,2,1]` in the second. What list does the following program evaluate to under dynamic dispatch?

```
class c1 extends object
  method initialize() 1
  method m1() 1
  method m2() 100
  method m3() send self m2()
class c2 extends c1
  method m2() 2
main
  let o1 = new c1(), o2 = new c2()
  in list(send o1 m1(), send o1 m2(), send o1 m3(),
          send o2 m1(), send o2 m2(), send o2 m3())
```


Invoking Superclass Methods

There are situations when one wants to refer to a superclass method, despite the fact that it has been overridden. Consider again the extension of points with colored points. The following is a class definition for points:

```
class point extends object
  field x
  field y
  method initialize(initx, inity)
    { set x = initx ;
      set y = inity }
  method move(dx, dy)
    { set x = x + dx ;
      set y = y + dy }
  method getLocation() list(x,y)
```

The following shows one possible definition of its extension class:

```
class colorpoint extends point
  field color
  method initialize(initx,inity,initcolor)
    { set x = initx ;
      set y = inity ;
      set color = initcolor }
  method setColor(c) set color = c
  method getColor()  color
```

The problem with the above is that it *repeats* the initialization code of the superclass, which may be inconvenient in large examples. In order to solve this problem, we use the language construct `super`, which enforces calling the corresponding method of the superclass:

```
  method initialize(initx,inity,initcolor)
    { super initialize(initx, inity) ;
      set color = initcolor }
```


What value does the following program evaluate to?

```
class c1 extends object
  method initialize() 1
  method m1() send self m2()
  method m2() 13
class c2 extends c1
  method m1() 22
  method m2() 23
  method m3() super m1()
class c3 extends c2
  method m1() 32
  method m2() 33
main
  let o3 = new c3()
  in send o3 m3()
```

A More Complex Example

```
class a extends object
  field i
  field j
  method initialize() 1
  method setup()
    { set i = 15 ;
      set j = 20 ;
      50 }
  method f() send self g()
  method g() i + j
class b extends a
  field j
  field k
  method setup()
    { set j = 100 ;
```



```
      set k = 200 ;
      super setup() ;
      send self h() }
  method g() list(i,j,k)
  method h() super g()
class c extends b
  method g() super h()
  method h() k + j
main
  let p = proc(o)
            let u = send o setup()
            in list(u, send o g(), send o f())
  in list(p(new a()), p(new b()), p(new c()))
```

CS322 - Programming Language Design

Lecture 15: Defining an Object-Oriented Programming Language (part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We next define the object oriented language that we informally discussed in Lecture 14. We define it on top of

A Simpler Definition of a Simpler Functional Language!

More precisely, we consider the following simplifications:

- Combine `eval` and `state` into *only one operation* which returns a pair (value,state), as we did for typed language. The new operation is called `eval`;
- Consider *only call-by-value*.

Exercise 1 *This new language is defined in the file* `new-funct-lang.maude`. *Compare the new definition to the previous one and evaluate all the expressions in the new language. Look at the number of rewrites: it is much smaller. Why?*

The following, for example, is the definition of the semantics of `letrec` in the new style:

```
fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .  extending BINDING-SEMANTICS .
  var X : Name . var Bl : BindingList . var E : Exp . var Env : Env .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(Env)}
    if Env := env(S)
    /\ {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm
```

The operations on state slightly changed. For example, `_<**_ : State StateAttribute` replaces a state attribute, the operation `_[_<- ?]  : State NameList -> State` creates dangling bindings, and `_[_<*_] : State NameList ValueList -> State` updates the values of already existing bindings.

Defining an Object Oriented Language

The definition of our OO (functional) language, which is given in the file `oo-lang.maude`, follows the same pattern as all the other definitions that we have encountered in the class so far, that is:

- *Define the syntax*. We will add new syntax for lists and classes, as well as for creating new objects, sending messages to objects and calling superclass methods.
- *Add state infrastructure*. As informally discussed last lecture, we will need to add infrastructure to store all the classes, for the current object and for the current class (which may be different from the object's!).
- *Define the semantics* of each syntactic construct, potentially defining other helping operations.

Adding Lists

Our previous functional language did not have lists. A very useful method in OO languages is `getState()`, which returns the values of all the fields of an object. In order to do this easily, we add lists to our functional language (trivially) as follows:

```
fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var El : ExpList .  vars S Sl : State .  var Vl : ValueList .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El,S) .
endfm
```

List will be enclosed by square brackets, to distinguish them from the lists of expressions passed to functions. They can be nested.

Syntax of Fields

The syntax of fields is straightforward: a sequence of independent field declarations separated by white spaces. Since the order in which fields are declared does not matter and since a field cannot be declared more than once, fields actually form a *set*:

```
fmod FIELD-SYNTAX is protecting NAME-SYNTAX .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field_ : Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields] .
endfm
```

Everything else being equal, we typically *prefer sets to lists* in Maude, because the matching operations tend to be faster.

Syntax of Methods

Like fields, methods are also separated by white spaces and they also form a *set*. Note that in order to avoid parsing difficulties, it is safe to give the methods and their concatenation higher precedences than all the operations on expressions (remember: the lower the precedence the tighter the binding!):

```
fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method___ : Name NameList Exp -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm
```


Syntax of Classes

Classes also form a *set*. It is often the the case that a class does not declare any new fields, but only uses its superclass' fields. In order to relieve the programmer from having to explicitly state `noFields` in such a class definition, we introduce a special syntax:

```
fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes .
  subsort Class < Classes .
  op noClasses : -> Classes .
  op class_extends__ : Name Name Methods -> Class [prec 115] .
  op class_extends___ : Name Name Fields Methods -> Class [prec 115] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm
```

Syntax of the Other Expression Constructs

The following introduce syntax for the `new`, `send` and `super` expression constructs. The `Name` in the declarations of `new` and `super` is meant to be a *class name*, while that of `send` is meant to be a *method name*. Also, note that the expression argument of `send` is expected to evaluate to an object! All these checks will be automatically done when we define the semantics.

```
fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .  endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .  endfm

fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .  endfm
```


Putting All the Syntax Together

The following shows only the new features that we are adding to the previous definition of the language. Besides the imported modules, it also defines the sort `Program`, which we did not need before, together with a program construct that appends an expression to a set of class definitions:

```
fmod PROG-LANG-SYNTAX is sort Program .
...
  extending LIST-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm
```

Defining OO Auxiliary State Infrastructure

We next define the additional state infrastructure needed in order to define the semantics of objects. So far, no definition was related explicitly to objects! This is quite normal, because *objects are semantic entities*. They do not exist at the syntactic level, but are rather created in order to evaluate expressions. Therefore, like integer values or like closures or like the store itself, objects will live somewhere in the state.

More precisely, objects will have their own state and, since they can be bound to names, they will be visible in the environment. Due to side effects, a name bound to an object may change its value. This suggests that *objects are just like any other value* in the language, in particular can be stored at specific locations.

Objects' most popular intuitive description is as *encapsulated*

states. There are two possibilities to add states to objects:

1. Objects maintain environments pointing to locations in the store where the values associated to their fields can be found;
2. Objects store the values within themselves.

The first easier to define, as we can reuse the already existing infrastructure, and this is the one we will approach next. However, the second is a bit more realistic in practice, because objects may be distributed on different computers and due to locality reasons it is inconvenient to share the same memory space.

Exercise 2 *Modify the subsequent definitions so that objects will maintain their fields values within themselves rather that in the store.*

Defining Object Environments

Objects need to be able to access the value bound to *any field* in the hierarchy of its classes, including the shadowed ones. One may refer to those while evaluating a `super` method, which points to a method of the superclass of the current class in which the expression containing the `super` keyword is evaluated, which may be different from the class of the current object. We define an *object environment* as a set of pairs (class name, environment), that is, a set of environments indexed by their corresponding classes:

```
fmod OBJ-ENVIRONMENT is protecting ENVIRONMENT .
  sort ObjEnv .
  op noObjEnv : -> ObjEnv .
  op (_,_) : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: noObjEnv] .
endfm
```


Defining Objects

We are now ready to define objects as special values, containing a handle to their class name and an object environment:

```
fmod OBJECT is extending OBJ-ENVIRONMENT .
  extending VALUE .
  sort Object .
  subsort Object < Value .
  op o : Name ObjEnv -> Object .
  op class : Object -> Name .
  var Xc : Name .  var OEnv : ObjEnv .
  eq class(o(Xc,OEnv)) = Xc .
endfm
```

Getting the Name of the Superclass

The following defines a straightforward but frequently used operation, which retrieves the superclass' name of a given class:

```
fmod SUPER-CLASS is extending CLASS-SYNTAX .
  op superClass : Name Classes -> [Name] [memo] .
  var Fs : Fields .  var Ms : Methods .
  vars Xc Xc' : Name .  var Cls : Classes .
  eq superClass(Xc, (Cls class Xc extends Xc' Fs Ms)) = Xc' .
endfm
```

This operations is *partial*, because it is not defined for the class `object`. This is realized by declaring its result sort between brackets. Since the classes do not change during evaluation, it makes sense to *memoize*, or cache or hash, this operation, in order to do the matching only once per class.

Defining the State

All the infrastructure needed to define the state is available now. We just make minimal extensions to the already existing state of the functional language, as follows (the dots stay for already existing definitions):

We first include the objects and the auxiliary superclass operation:

```
fmod STATE is
...
  extending OBJECT .
  extending SUPER-CLASS .
```

Next we have to extend the previous location look-up operation, which in the current definition of our functional language is denoted by `_{_} : State Name -> Location`. The current definition contains only the first equation below, because each accessed name

is supposed to have been already bound to a location in the environment. However, now this is *not* the case anymore because the name may refer to a field of the current object. Therefore, in case the environment lookup fails, we have to search for the location of the name into the corresponding object environment:

```
...
  eq (env([X,L] Env) S){X} = L .
  eq S{X} = S{X}{currClass(S)} [owise] .
  eq (obj(o(Xc', (Xc, [X,L] Env) OEnv)) S){X}{Xc} = L .
  eq S{X}{Xc} = S{X}{superClass(Xc,classes(S))} [owise] .
```

The auxiliary operations `currClass` and `classes` refer to the current class and the set of all classes; they are defined below:

```
  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .
```



```
  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .
```

A similar operation for objects also needs to be defined:

```
  op obj : Object -> StateAttribute .
  op obj : State -> Object .
  eq (obj(O) S) <** obj(O') = obj(O') S .
  eq obj(obj(O) S) = O .
endfm
```

We have added all the needed OO infrastructure. A state contains *six attributes* in what follows. The next defines the initial state:

```
  eq initState =
       env(noEnv) store(noStore) nextLoc(0)
       obj(o(object, noObjEnv)) currClass(object)
       classes(noClasses) .
```

Semantics of Names

We have to introduce the extra OO knowledge that the special name `self` refers to the current object. If the name is different from `self`, then it has the same meaning as for the functional programming language:

```
fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .  var O : Object .
  eq eval(self, S) = {obj(S),S} .
  eq eval(X, S) = {S[X], S} [owise] .
endfm
```


Semantics of Fields

Fields do not mean anything by themselves. Their real meaning will be given in the context of classes in the sequel. For now, we define an auxiliary operation needed later extracting the list of all the names occurring in a fields declaration:

```
fmod FIELD-SEMANTICS is protecting FIELD-SYNTAX .
  op names : Fields -> NameList .
  eq names(noFields) = () .
  var Xf : Name .  var Fs : Fields .
  eq names(field Xf Fs) = Xf, names(Fs) .
endfm
```

Semantics of Classes

In order to give the full meaning of classes, we have to specify the three issues below.

(1) What happens if a class does not declare fields (we omit the variable declarations; see the provided code). This is easy, we just reduce it to the class declaring an empty set of fields:

```
fmod CLASS-SEMANTICS is extending CLASS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending FIELD-SEMANTICS .

  eq class Xc extends Xc' Ms = class Xc extends Xc' noFields Ms .
```

(2) How are objects created. This needs to be defined recursively, over the hierarchy of classes. We split it in two cases. If the object is an instance of the class `object` then it contains an empty object environment. If the object is an instance of a class different from

`object`, then we first recursively generate the object environment of a dummy instance of its superclass and then add dangling bindings for the current field names:

```
  op createObject : Name State -> ValueStatePair .
  eq createObject(object,S) = {o(object,noObjEnv), S} .
 ceq createObject(Xc,S) = {o(Xc, (Xc, env(Sf)) OEnv), Sf <** env(Env)}
    if Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms := classes(S)
    /\ {o(Xc',OEnv), S'} := createObject(Xc',S)
    /\ Sf := (S' <** env(noEnv))[names(Fs) <- ?] [owise] .
```

(3) How are methods invoked. Since methods can be overridden, a search needs to be done through the current object's class hierarchy, from subclasses to superclasses, whenever a method is invoked. One can evaluate its actual argument expressions either before the search or after the method is found. We prefer the latter. By convention, we assume that the `initialize` method of

the class `object` returns `0`, so one may omit dummy initializations, but we let any other method undefined within the class `object` (so one gets a runtime error). Notice that, as one evaluates the function invocation, one may change the current class and environment. Therefore, as one returns from a method invocation, one has to recover the calling context properly:

```
 op invokeMethod : Name ExpList State -> ValueStatePair .
ceq invokeMethod(initialize, El, S) = {int(0), S}
   if currClass(S) = object .
ceq invokeMethod(X, El, S) = {Ve, Se <** env(Env)}
   if Xc := currClass(S) /\ Env := env(S)
   /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S)
   /\ {Vl,Sl} := eval(El,S)
   /\ {Ve,Se} := eval(E, Sl[Xl <- Vl]) .
ceq invokeMethod(X, El, S) = {Vm, Sm <** currClass(Xc)}
   if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
   /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) [owise] .
```


Semantics of New

```
fmod NEW-SEMANTICS is extending NEW-SYNTAX .
  extending CLASS-SEMANTICS .
  vars Xc Xc' : Name .  var El : ExpList .  var Vm : Value .
  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(new Xc(El), S) = {O', Sm <** obj(O) <** currClass(Xc')}
    if O := obj(S) /\ Xc' := currClass(S)
    /\ {O',S'} := createObject(Xc,S)
    /\ {Vm,Sm} := invokeMethod(initialize, El,
                     S' <** obj(O') <** currClass(Xc)) .
endfm
```

Semantics of Send

```
fmod SEND-SEMANTICS is extending SEND-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc : Name .  var E : Exp .  var El : ExpList .
  var Vm : Value .  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(send E X(El), S) = {Vm, Sm <** obj(O) <** currClass(Xc)}
    if O := obj(S) /\ Xc := currClass(S)
    /\ {O',S'} := eval(E,S)
    /\ {Vm,Sm} := invokeMethod(X, El,
                    S' <** obj(O') <** currClass(class(O'))) .
endfm
```


Semantics of Super

```
fmod SUPER-SEMANTICS is extending SUPER-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc Xc' : Name .  var El : ExpList .
  var Vm : Value .  vars S Sm : State .
 ceq eval(super X(El), S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) .
endfm
```

Putting All the Semantics Together

```
fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending LIST-SEMANTICS .
...
  extending NEW-SEMANTICS .
  extending SEND-SEMANTICS .
  extending SUPER-SEMANTICS .
  op eval_ : Program -> Value .
  var Cls : Classes .  var E : Exp .  var Ve : Value .  var Se : State .
 ceq eval(Cls main E) = Ve
    if {Ve,Se} := eval(E, initState <** classes(Cls)) .
endfm
```


Exercise 3 *Read Chapter 4 in Friedman for a more detailed exposition of the language and for possible implementations of the language that we formally defined.*

Homework Exercise 1 *The definition presented in this lecture considers* dynamic method dispatch*. Change the definition so that your OO language will work under* static method dispatch*.*

Homework Exercise 2 *Add and define private methods to the OO language defined in this lecture. Take the meaning of private methods from Java.*

```
**************************************************
*** A New Definition of a Funtional Language ***
**************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
```

```
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
```

```
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op `[`] : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: `[`]] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
```

```
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
    (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
```

```
    nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .
  eq eval(X, S) = {S[X], S} .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
```

```
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> ValueStatePair .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
```

```
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
```

```
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
```

```
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
```

```
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
```

```
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
```

```
      in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
```

```
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
```

```
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
```

```
  in letrec
      'even = proc() if zero?(x)
                  then 1
                  else let d = set x = x - 1
                       in 'odd(),
      'odd  = proc() if zero?(x)
                  then 0
                  else let d = set x = x - 1
                       in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                  else let d = set x = x - 1
                       in 'odd(),
         'odd = proc() if zero?(x) then 0
                  else let d = set x = x - 1
                       in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24
```

```
red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
```

```
      f(g(y));
      y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
```

```
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
*********************************************
*** A New Definition of a Funtional Language ***
*********************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op '[') : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: '[']] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .
  eq eval(X, S) = {S[X], S} .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> ValueStatePair .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
```

```
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
```

```
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
            {
              set a = a + b ;
              set b = a - b ;
              set a = a - b
            }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
*******************************
*** Defining an OO Language ***
*******************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
```

```
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
```

```
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting NAME-SYNTAX .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field_ : Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method___ : Name NameList Exp -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes .
  subsort Class < Classes .
  op noClasses : -> Classes .
  op class_extends__ : Name Name Methods -> Class [prec 115] .
  op class_extends___ : Name Name Fields Methods -> Class [prec 115] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm
```

```
fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
```

```
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op `[`] : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: `[`]] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod OBJ-ENVIRONMENT is protecting ENVIRONMENT .
  sort ObjEnv .
  op noObjEnv : -> ObjEnv .
  op (_,_) : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: noObjEnv] .
endfm

fmod OBJECT is extending OBJ-ENVIRONMENT .
  extending VALUE .
  sort Object .
  subsort Object < Value .
  op o : Name ObjEnv -> Object .
  op class : Object -> Name .
  var Xc : Name .  var OEnv : ObjEnv .
  eq class(o(Xc,OEnv)) = Xc .
endfm

fmod SUPER-CLASS is extending CLASS-SYNTAX .
```

```
  op superClass : Name Classes -> [Name] [memo] .
  var Fs : Fields .  var Ms : Methods .
  vars Xc Xc' : Name .  var Cls : Classes .
  eq superClass(Xc, (Cls class Xc extends Xc' Fs Ms)) = Xc' .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending OBJECT .
  extending SUPER-CLASS .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  vars X Xc Xc' : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .
  var OEnv : ObjEnv .  vars Cls Cls' : Classes .  vars O O' : Object .

  eq (env([X,L] Env) S){X} = L .
  eq S{X} = S{X}{currClass(S)} [owise] .
  eq (obj(o(Xc', (Xc, [X,L] Env) OEnv)) S){X}{Xc} = L .
  eq S{X}{Xc} = S{X}{superClass(Xc,classes(S))} [owise] .

  eq S[X] = store(S)[S{X}] .
```

```
  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState =
       env(noEnv) store(noStore) nextLoc(0)
       obj(o(object, noObjEnv)) currClass(object)
       classes(noClasses) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .

  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .

  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .

  op obj : Object -> StateAttribute .
  op obj : State -> Object .
  eq (obj(O) S) <** obj(O') = obj(O') S .
```

```
  eq obj(obj(O) S) = O .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .  var O : Object .
  eq eval(self, S) = {obj(S),S} .
  eq eval(X, S) = {S[X], S} [owise] .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var El : ExpList .  vars S Sl : State .  var Vl : ValueList .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El,S) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
```

```
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> ValueStatePair .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
```

```
   if {Vl,Sl} := eval(exps(Bl), S)
   /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
   if {closure(Xl, E, Env), Sf} := eval(F,S)
   /\ {Vl,Sl} := eval(El,Sf)
   /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
   if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
   /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
   if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
```

```
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod FIELD-SEMANTICS is protecting FIELD-SYNTAX .
  op names : Fields -> NameList .
  eq names(noFields) = () .
  var Xf : Name .  var Fs : Fields .
  eq names(field Xf Fs) = Xf, names(Fs) .
endfm

fmod CLASS-SEMANTICS is extending CLASS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending FIELD-SEMANTICS .
  vars X Xc Xc' : Name .  var Xl : NameList .  var OEnv : ObjEnv .
  var E : Exp .  var El : ExpList .  vars S S' Sf Sl Se Sm : State .
  var Fs : Fields .  var Ms : Methods .  var Cls : Classes .
  vars Ve Vm : Value .  var Vl : ValueList .  var Env : Env .

  eq class Xc extends Xc' Ms = class Xc extends Xc' noFields Ms .

  op createObject : Name State -> ValueStatePair .
  eq createObject(object,S) = {o(object,noObjEnv), S} .
 ceq createObject(Xc,S) = {o(Xc, (Xc, env(Sf)) OEnv), Sf <** env(Env)}
    if Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms := classes(S)
    /\ {o(Xc',OEnv), S'} := createObject(Xc',S)
    /\ Sf := (S' <** env(noEnv))[names(Fs) <- ?] [owise] .

  op invokeMethod : Name ExpList State -> ValueStatePair .
 ceq invokeMethod(initialize, El, S) = {int(0), S}
    if currClass(S) = object .
 ceq invokeMethod(X, El, S) = {Ve, Se <** env(Env)}
    if Xc := currClass(S) /\ Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S)
    /\ {Vl,Sl} := eval(El,S)
    /\ {Ve,Se} := eval(E, Sl[Xl <- Vl]) .
 ceq invokeMethod(X, El, S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) [owise] .
```

```
endfm

fmod NEW-SEMANTICS is extending NEW-SYNTAX .
  extending CLASS-SEMANTICS .
  vars Xc Xc' : Name .  var El : ExpList .  var Vm : Value .
  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(new Xc(El), S) = {O', Sm <** obj(O) <** currClass(Xc')}
    if O := obj(S) /\ Xc' := currClass(S)
    /\ {O',S'} := createObject(Xc,S)
    /\ {Vm,Sm} := invokeMethod(initialize, El,
                  S' <** obj(O') <** currClass(Xc)) .
endfm

fmod SEND-SEMANTICS is extending SEND-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc : Name .  var E : Exp .  var El : ExpList .
  var Vm : Value .  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(send E X(El), S) = {Vm, Sm <** obj(O) <** currClass(Xc)}
    if O := obj(S) /\ Xc := currClass(S)
    /\ {O',S'} := eval(E,S)
    /\ {Vm,Sm} := invokeMethod(X, El,
                  S' <** obj(O') <** currClass(class(O'))) .
endfm

fmod SUPER-SEMANTICS is extending SUPER-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc Xc' : Name .  var El : ExpList .
  var Vm : Value .  vars S Sm : State .
 ceq eval(super X(El), S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending LIST-SEMANTICS .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
```

```
  extending NEW-SEMANTICS .
  extending SEND-SEMANTICS .
  extending SUPER-SEMANTICS .
  op eval_ : Program -> Value .
  var Cls : Classes .  var E : Exp .  var Ve : Value .  var Se : State .
 ceq eval(Cls main E) = Ve
    if {Ve,Se} := eval(E, initState <** classes(Cls)) .
endfm

red eval(
  class c extends object
    field a
    method initialize() set a = 5
    method m() a
  main
    let x = new c()
    in send x m()
) .

red eval(
  class c extends object
    field i
    field j
    method initialize(x)
      {
        set i = x ;
        set j = 0 - x
      }
    method add(d)
      {
        set i = i + d ;
        set j = j - d
      }
    method getstate()
      list(i,j)
  main
    let a = 0, b = 0, o = new c(5)
    in {
         set a = send o getstate() ;
         send o add(3) ;
         set b = send o getstate() ;
         list(a,b)
       }
```

```
) .

red eval(
  class c extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  main
    let o = new c()
    in send o m1()
) .

red eval(
  class oddeven extends object
    method initialize() 1
    method even(n)
      if zero?(n) then 1 else send self odd(n - 1)
    method odd(n)
      if zero?(n) then 0 else send self even(n - 1)
  main
    let o = new oddeven()
    in send o odd(17)
) .

red eval(
  class node extends object
    field left
    field right
    method initialize(l,r)
      {
        set left = l ;
        set right = r
      }
    method sum()
      (send left sum()) + (send right sum())
  class leaf extends object
    field value
    method initialize(v)
      set value = v
    method sum() value
  main
    let o = new node(new node(new leaf(3), new leaf(4)),
                     new leaf(5))
```

```
   in send o sum()
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation()
      list(x,y)
  class colorpoint extends point
    field color
    method setColor(c) set color = c
    method getColor()  color
  main
    let p = new point(3,4), cp = new colorpoint(10,20)
    in {
         send p move(3,4) ;
         send cp setColor(87) ;
         send cp move(10, 20) ;
         list(send p  getLocation(),
              send cp getLocation(),
              send cp getColor())
       }
) .

red eval(
  class c1 extends object
    field x
    field y
    method initialize() 1
    method setx1(v) set x = v
    method sety1(v) set y = v
    method getx1() x
```

```
    method gety1() y
  class c2 extends c1
    field y
    method sety2(v) set y = v
    method getx2() x
    method gety2() y
  main
    let o2 = new c2()
    in {
        send o2 setx1(101) ;
        send o2 sety1(102) ;
        send o2 sety2(999) ;
        list(send o2 getx1(), send o2 gety1(),
            send o2 getx2(), send o2 gety2())
       }
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() send self m1()
  class c2 extends c1
    method m1() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o2 m1(), send o2 m2())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() 100
    method m3() send self m2()
  class c2 extends c1
    method initialize() 1
    method m2() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o1 m2(), send o1 m3(),
          send o2 m1(), send o2 m2(), send o2 m3())
) .
```

```
red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        set x = initx ;
        set y = inity ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
```

```
      set y = y + dy
     }
   method getLocation() list(x,y)
  class colorpoint extends point
   field color
   method initialize(initx,inity,initcolor)
    {
     super initialize(initx, inity) ;
     set color = initcolor
    }
   method setColor(c) set color = c
   method getColor()  color
  main
   let o = new colorpoint(3, 4, 172)
   in list(send o getLocation(), send o getColor())
) .

red eval(
  class c1 extends object
   method initialize() 1
   method m1() send self m2()
   method m2() 13
  class c2 extends c1
   method m1() 22
   method m2() 23
   method m3() super m1()
  class c3 extends c2
   method m1() 32
   method m2() 33
  main
   let o3 = new c3()
   in send o3 m3()
) .

red eval(
  class a extends object
   field i
   field j
   method initialize() 1
   method setup()
    {
     set i = 15 ;
     set j = 20 ;
```

```
      50
    }
  method f() send self g()
  method g() i + j
 class b extends a
  field j
  field k
  method setup()
    {
     set j = 100 ;
     set k = 200 ;
     super setup() ;
     send self h()
    }
  method g() list(i,j,k)
  method h() super g()
 class c extends b
  method g() super h()
  method h() k + j
 main
  let p = proc(o)
          let u = send o setup()
          in list(u, send o g(), send o f())
  in list(p(new a()), p(new b()), p(new c()))
) .


eof

--- the previous examples also work

red eval(
 let x = 5, y = 7
 in x + y
) .
***> should be 12

red eval(
 let x = 1
 in let x = x + 2
    in x + 1
) .
***> should be 4
```

```
red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
```

```
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping
```

```
red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
```

```
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1
```

```
red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3
```

```
red eval(
  let f = let c = 0
          in proc()
              let d = set c = c + 1
              in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
              let d = set x = x + 1
              in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
              let t = x
              in let d = set x = y
                 in let d = set y = t
                    in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
```

```
                then 1
                else let d = set x = x - 1
                     in 'odd(),
      'odd  = proc() if zero?(x)
                then 0
                else let d = set x = x - 1
                     in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                 else let d = set x = x - 1
                      in 'odd(),
         'odd = proc() if zero?(x) then 0
                 else let d = set x = x - 1
                      in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
```

```
  let 'times4 = 0
  in {
      set 'times4 = proc(x)
                      if zero?(x)
                      then 0
                      else 4 + 'times4(x - 1) ;
      'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
```

```
    }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
```

```
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
*******************************
*** Defining an OO Language ***
*******************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  op '(' : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm
```

```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting NAME-SYNTAX .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field_ : Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method___ : Name NameList Exp -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes .
  subsort Class < Classes .
  op noClasses : -> Classes .
  op class_extends__ : Name Name Methods -> Class [prec 115] .
  op class_extends___ : Name Name Fields Methods -> Class [prec 115] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm

fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm
```

```
fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm


-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op '[' : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: '['] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod OBJ-ENVIRONMENT is protecting ENVIRONMENT .
  sort ObjEnv .
  op noObjEnv : -> ObjEnv .
  op (_,_) : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: noObjEnv] .
endfm

fmod OBJECT is extending OBJ-ENVIRONMENT .
  extending VALUE .
  sort Object .
  subsort Object < Value .
  op o : Name ObjEnv -> Object .
  op class : Object -> Name .
  var Xc : Name .  var OEnv : ObjEnv .
  eq class(o(Xc,OEnv)) = Xc .
endfm

fmod SUPER-CLASS is extending CLASS-SYNTAX .
  op superClass : Name Classes -> [Name] [memo] .
  var Fs : Fields .  var Ms : Methods .
  vars Xc Xc' : Name .  var Cls : Classes .
  eq superClass(Xc, (Cls class Xc extends Xc' Fs Ms)) = Xc' .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending OBJECT .
  extending SUPER-CLASS .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  vars X Xc Xc' : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .
  var OEnv : ObjEnv .  vars Cls Cls' : Classes .  vars O O' : Object .

  eq (env([X,L] Env) S){X} = L .
  eq S{X} = S{X}{currClass(S)} [owise] .
  eq (obj(o(Xc', (Xc, [X,L] Env) OEnv)) S){X}{Xc} = L .
  eq S{X}{Xc} = S{X}{superClass(Xc,classes(S))} [owise] .

  eq S[X] = store(S){S{X}} .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .
```

```
  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState =
       env(noEnv) store(noStore) nextLoc(0)
       obj(o(object, noObjEnv)) currClass(object)
       classes(noClasses) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .

  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .

  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .

  op obj : Object -> StateAttribute .
  op obj : State -> Object .
  eq (obj(O) S) <** obj(O') = obj(O') S .
  eq obj(obj(O) S) = O .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .  var O : Object .
  eq eval(self, S) = {obj(S),S} .
  eq eval(X, S) = {S[X], S} [owise] .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var El : ExpList .  vars S Sl : State .  var Vl : ValueList .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El,S) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> ValueStatePair .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod FIELD-SEMANTICS is protecting FIELD-SYNTAX .
  op names : Fields -> NameList .
  eq names(noFields) = () .
  var Xf : Name .  var Fs : Fields .
  eq names(field Xf Fs) = Xf, names(Fs) .
endfm

fmod CLASS-SEMANTICS is extending CLASS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending FIELD-SEMANTICS .
  vars X Xc Xc' : Name .  var Xl : NameList .  var OEnv : ObjEnv .
  var E : Exp .  var El : ExpList .  vars S S' Sf Sl Se Sm : State .
  var Fs : Fields .  var Ms : Methods .  var Cls : Classes .
  vars Ve Vm : Value .  var Vl : ValueList .  var Env : Env .

  eq class Xc extends Xc' Ms = class Xc extends Xc' noFields Ms .

  op createObject : Name State -> ValueStatePair .
  eq createObject(object,S) = {o(object,noObjEnv), S} .
 ceq createObject(Xc,S) = {o(Xc, (Xc, env(Sf)) OEnv), Sf <** env(Env)}
    if Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms := classes(S)
    /\ {o(Xc',OEnv), S'} := createObject(Xc',S)
    /\ Sf := (S' <** env(noEnv))[names(Fs) <- ?] [owise] .

  op invokeMethod : Name ExpList State -> ValueStatePair .
 ceq invokeMethod(initialize, El, S) = {int(0), S}
    if currClass(S) = object .
 ceq invokeMethod(X, El, S) = {Ve, Se <** env(Env)}
    if Xc := currClass(S) /\ Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S)
    /\ {Vl,Sl} := eval(El,S)
    /\ {Ve,Se} := eval(E, Sl[Xl <- Vl]) .
 ceq invokeMethod(X, El, S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) [owise] .
endfm

fmod NEW-SEMANTICS is extending NEW-SYNTAX .
  extending CLASS-SEMANTICS .
  vars Xc Xc' : Name .  var El : ExpList .  var Vm : Value .
  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(new Xc(El), S) = {O', Sm <** obj(O) <** currClass(Xc')}
    if O := obj(S) /\ Xc' := currClass(S)
    /\ {O',S'} := createObject(Xc,S)
    /\ {Vm,Sm} := invokeMethod(initialize, El,
                  S' <** obj(O') <** currClass(Xc)) .
endfm

fmod SEND-SEMANTICS is extending SEND-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc : Name .  var E : Exp .  var El : ExpList .
  var Vm : Value .  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(send E X(El), S) = {Vm, Sm <** obj(O) <** currClass(Xc)}
    if O := obj(S) /\ Xc := currClass(S)
    /\ {O',S'} := eval(E,S)
    /\ {Vm,Sm} := invokeMethod(X, El,
                  S' <** obj(O') <** currClass(class(O'))) .
endfm

fmod SUPER-SEMANTICS is extending SUPER-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc Xc' : Name .  var El : ExpList .
  var Vm : Value .  vars S Sm : State .
 ceq eval(super X(El), S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending LIST-SEMANTICS .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
  extending NEW-SEMANTICS .
  extending SEND-SEMANTICS .
  extending SUPER-SEMANTICS .
  op eval_ : Program -> Value .
  var Cls : Classes .  var E : Exp .  var Ve : Value .  var Se : State .
 ceq eval(Cls main E) = Ve
    if {Ve,Se} := eval(E, initState <** classes(Cls)) .
endfm

red eval(
  class c extends object
    field a
    method initialize() set a = 5
```

```
    method m() a
  main
    let x = new c()
    in send x m()
) .

red eval(
  class c extends object
    field i
    field j
    method initialize(x)
      {
        set i = x ;
        set j = 0 - x
      }
    method add(d)
      {
        set i = i + d ;
        set j = j - d
      }
    method getstate()
      list(i,j)
  main
    let a = 0, b = 0, o = new c(5)
    in {
         set a = send o getstate() ;
         send o add(3) ;
         set b = send o getstate() ;
         list(a,b)
       }
) .

red eval(
  class c extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  main
    let o = new c()
    in send o m1()
) .

red eval(
  class oddeven extends object
    method initialize() 1
    method even(n)
      if zero?(n) then 1 else send self odd(n - 1)
    method odd(n)
      if zero?(n) then 0 else send self even(n - 1)
  main
    let o = new oddeven()
    in send o odd(17)
) .

red eval(
  class node extends object
    field left
    field right
    method initialize(l,r)
      {
        set left = l ;
        set right = r
      }
    method sum()
      (send left sum()) + (send right sum())
  class leaf extends object
    field value
    method initialize(v)
      set value = v
    method sum() value
  main
    let o = new node(new node(new leaf(3), new leaf(4)),
                     new leaf(5))
    in send o sum()
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation()
      list(x,y)
  class colorpoint extends point
    field color
    method setColor(c) set color = c
    method getColor()  color
  main
    let p = new point(3,4), cp = new colorpoint(10,20)
    in {
         send p move(3,4) ;
         send cp setColor(87) ;
         send cp move(10, 20) ;
         list(send p  getLocation(),
              send cp getLocation(),
              send cp getColor())
       }
) .

red eval(
  class c1 extends object
    field x
    field y
    method initialize() 1
    method setx1(v) set x = v
    method sety1(v) set y = v
    method getx1() x
    method gety1() y
  class c2 extends c1
    field y
    method sety2(v) set y = v
    method getx2() x
    method gety2() y
  main
    let o2 = new c2()
    in {
         send o2 setx1(101) ;
         send o2 sety1(102) ;
         send o2 sety2(999) ;
         list(send o2 getx1(), send o2 gety1(),
              send o2 getx2(), send o2 gety2())
       }
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() send self m1()
  class c2 extends c1
    method m1() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o2 m1(), send o2 m2())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() 100
    method m3() send self m2()
  class c2 extends c1
    method initialize() 1
    method m2() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o1 m2(), send o1 m3(),
            send o2 m1(), send o2 m2(), send o2 m3())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        set x = initx ;
        set y = inity ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        super initialize(initx, inity) ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  class c2 extends c1
    method m1() 22
    method m2() 23
    method m3() super m1()
  class c3 extends c2
    method m1() 32
    method m2() 33
  main
    let o3 = new c3()
    in send o3 m3()
) .

red eval(
  class a extends object
    field i
    field j
    method initialize() 1
    method setup()
      {
        set i = 15 ;
        set j = 20 ;
        50
      }
    method f() send self g()
    method g() i + j
  class b extends a
    field j
    field k
    method setup()
      {
        set j = 100 ;
        set k = 200 ;
        super setup() ;
        send self h()
      }
    method g() list(i,j,k)
    method h() super g()
  class c extends b
    method g() super h()
    method h() k + j
```

```
  main
    let p = proc(o)
              let u = send o setup()
              in list(u, send o g(), send o f())
    in list(p(new a()), p(new b()), p(new c()))
) .


eof

--- the previous examples also work

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
```

```
                    in 0
    in let d = f(x,y)
       in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
*********************************
*** Defining an OO Language ***
*********************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
```

```
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
```

```
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting NAME-SYNTAX .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field_ : Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method___ : Name NameList Exp -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes .
  subsort Class < Classes .
  op noClasses : -> Classes .
  op class_extends__ : Name Name Methods -> Class [prec 115] .
  op class_extends___ : Name Name Fields Methods -> Class [prec 115] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm
```

```
fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
```

```
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op `[`] : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: `[`]] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod OBJ-ENVIRONMENT is protecting ENVIRONMENT .
  sort ObjEnv .
  op noObjEnv : -> ObjEnv .
  op (_,_) : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: noObjEnv] .
endfm

fmod OBJECT is extending OBJ-ENVIRONMENT .
  extending VALUE .
  sort Object .
  subsort Object < Value .
  op o : Name ObjEnv -> Object .
  op class : Object -> Name .
  var Xc : Name .  var OEnv : ObjEnv .
  eq class(o(Xc,OEnv)) = Xc .
endfm

fmod SUPER-CLASS is extending CLASS-SYNTAX .
```

```
  op superClass : Name Classes -> [Name] [memo] .
  var Fs : Fields .  var Ms : Methods .
  vars Xc Xc' : Name .  var Cls : Classes .
  eq superClass(Xc, (Cls class Xc extends Xc' Fs Ms)) = Xc' .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending OBJECT .
  extending SUPER-CLASS .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op ? : ->  [ValueStatePair] .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  vars X Xc Xc' : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .
  var OEnv : ObjEnv .  vars Cls Cls' : Classes .  vars O O' : Object .

  eq (env([X,L] Env) S){X} = L .
  eq S{X} = S{X}{currClass(S)} [owise] .
  eq (obj(o(Xc', (Xc, [X,L] Env) OEnv)) S){X}{Xc} = L .
  eq S{X}{Xc} = S{X}{superClass(Xc,classes(S))} [owise] .

  eq S[X] = store(S)[S{X}] .
```

```
  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState =
       env(noEnv) store(noStore) nextLoc(0)
       obj(o(object, noObjEnv)) currClass(object)
       classes(noClasses) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .

  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .

  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .

  op obj : Object -> StateAttribute .
  op obj : State -> Object .
```

```
  eq (obj(O) S) <** obj(O') = obj(O') S .
  eq obj(obj(O) S) = O .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> [ValueStatePair] .
  var X : Name .  var S : State .  var O : Object .
  eq eval(self, S) = {obj(S),S} .
  eq eval(X, S) = {S[X], S} [owise] .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> [ValueStatePair] .
  op  eval : ExpList State -> [ValueListStatePair] .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var El : ExpList .  vars S Sl : State .  var Vl : ValueList .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El,S) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
```

```
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> [ValueStatePair] .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
```

```
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
```

```
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod FIELD-SEMANTICS is protecting FIELD-SYNTAX .
  op names : Fields -> NameList .
  eq names(noFields) = () .
  var Xf : Name .  var Fs : Fields .
  eq names(field Xf Fs) = Xf, names(Fs) .
endfm

fmod CLASS-SEMANTICS is extending CLASS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending FIELD-SEMANTICS .
  vars X Xc Xc' : Name .  var Xl : NameList .  var OEnv : ObjEnv .
  var E : Exp .  var El : ExpList .  vars S S' Sf Sl Se Sm : State .
  var Fs : Fields .  var Ms : Methods .  var Cls : Classes .
  vars Ve Vm : Value .  var Vl : ValueList .  var Env : Env .

  eq class Xc extends Xc' Ms = class Xc extends Xc' noFields Ms .

  op createObject : Name State -> [ValueStatePair] .
  eq createObject(object,S) = {o(object,noObjEnv), S} .
 ceq createObject(Xc,S) = {o(Xc, (Xc, env(Sf)) OEnv), Sf <** env(Env)}
    if Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms := classes(S)
    /\ {o(Xc',OEnv), S'} := createObject(Xc',S)
    /\ Sf := (S' <** env(noEnv))[names(Fs) <- ?] [owise] .

  op invokeMethod : Name ExpList State -> [ValueStatePair] .
 ceq invokeMethod(initialize, El, S) = {int(0), S}
    if currClass(S) = object .
 ceq invokeMethod(X, El, S) = {Ve, Se <** env(Env)}
    if Xc := currClass(S) /\ Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S)
    /\ {Vl,Sl} := eval(El,S)
    /\ {Ve,Se} := eval(E, Sl[Xl <- Vl]) .
 ceq invokeMethod(X, El, S) = ?
    if Xc := currClass(S) /\ Env := env(S)
```

```
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S) [owise] .
 ceq invokeMethod(X, El, S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) [owise] .
endfm

fmod NEW-SEMANTICS is extending NEW-SYNTAX .
  extending CLASS-SEMANTICS .
  vars Xc Xc' : Name .  var El : ExpList .  var Vm : Value .
  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(new Xc(El), S) = {O', Sm <** obj(O) <** currClass(Xc')}
    if O := obj(S) /\ Xc' := currClass(S)
    /\ {O',S'} := createObject(Xc,S)
    /\ {Vm,Sm} := invokeMethod(initialize, El,
                  S' <** obj(O') <** currClass(Xc)) .
endfm

fmod SEND-SEMANTICS is extending SEND-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc : Name .  var E : Exp .  var El : ExpList .
  var Vm : Value .  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(send E X(El), S) = {Vm, Sm <** obj(O) <** currClass(Xc)}
    if O := obj(S) /\ Xc := currClass(S)
    /\ {O',S'} := eval(E,S)
    /\ {Vm,Sm} := invokeMethod(X, El,
                  S' <** obj(O') <** currClass(class(O'))) .
endfm

fmod SUPER-SEMANTICS is extending SUPER-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc Xc' : Name .  var El : ExpList .
  var Vm : Value .  vars S Sm : State .
 ceq eval(super X(El), S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending LIST-SEMANTICS .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
```

```
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
  extending NEW-SEMANTICS .
  extending SEND-SEMANTICS .
  extending SUPER-SEMANTICS .
  op eval_ : Program -> [Value] .
  var Cls : Classes .  var E : Exp .  var Ve : Value .  var Se : State .
 ceq eval(Cls main E) = Ve
    if {Ve,Se} := eval(E, initState <** classes(Cls)) .
endfm

red eval(
  class c extends object
    field a
    method initialize() set a = 5
    method m() a
  main
    let x = new c()
    in send x m()
) .

red eval(
  class c extends object
    field i
    field j
    method initialize(x)
      {
        set i = x ;
        set j = 0 - x
      }
    method add(d)
      {
        set i = i + d ;
        set j = j - d
      }
    method getstate()
      list(i,j)
  main
    let a = 0, b = 0, o = new c(5)
    in {
         set a = send o getstate() ;
```

```
      send o add(3) ;
      set b = send o getstate() ;
      list(a,b)
     }
) .

red eval(
  class c extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  main
    let o = new c()
    in send o m1()
) .

red eval(
  class oddeven extends object
    method initialize() 1
    method even(n)
      if zero?(n) then 1 else send self odd(n - 1)
    method odd(n)
      if zero?(n) then 0 else send self even(n - 1)
  main
    let o = new oddeven()
    in send o odd(17)
) .

red eval(
  class node extends object
    field left
    field right
    method initialize(l,r)
      {
       set left = l ;
       set right = r
      }
    method sum()
      (send left sum()) + (send right sum())
  class leaf extends object
    field value
    method initialize(v)
      set value = v
```

```
   method sum() value
  main
   let o = new node(new node(new leaf(3), new leaf(4)),
                    new leaf(5))
   in send o sum()
) .

red eval(
  class point extends object
   field x
   field y
   method initialize(initx, inity)
     {
      set x = initx ;
      set y = inity
     }
   method move(dx, dy)
     {
      set x = x + dx ;
      set y = y + dy
     }
   method getLocation()
     list(x,y)
  class colorpoint extends point
   field color
   method setColor(c) set color = c
   method getColor()  color
  main
   let p = new point(3,4), cp = new colorpoint(10,20)
   in {
       send p move(3,4) ;
       send cp setColor(87) ;
       send cp move(10, 20) ;
       list(send p  getLocation(),
            send cp getLocation(),
            send cp getColor())
     }
) .

red eval(
  class c1 extends object
   field x
   field y
```

```
    method initialize() 1
    method setx1(v) set x = v
    method sety1(v) set y = v
    method getx1() x
    method gety1() y
  class c2 extends c1
    field y
    method sety2(v) set y = v
    method getx2() x
    method gety2() y
  main
    let o2 = new c2()
    in {
        send o2 setx1(101) ;
        send o2 sety1(102) ;
        send o2 sety2(999) ;
        list(send o2 getx1(), send o2 gety1(),
             send o2 getx2(), send o2 gety2())
       }
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() send self m1()
  class c2 extends c1
    method m1() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o2 m1(), send o2 m2())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() 100
    method m3() send self m2()
  class c2 extends c1
    method initialize() 1
    method m2() 2
  main
```

```
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o1 m2(), send o1 m3(),
            send o2 m1(), send o2 m2(), send o2 m3())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        set x = initx ;
        set y = inity ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
```

```
    }
   method move(dx, dy)
    {
     set x = x + dx ;
     set y = y + dy
    }
   method getLocation() list(x,y)
  class colorpoint extends point
   field color
   method initialize(initx,inity,initcolor)
    {
     super initialize(initx, inity) ;
     set color = initcolor
    }
   method setColor(c) set color = c
   method getColor()  color
  main
   let o = new colorpoint(3, 4, 172)
   in list(send o getLocation(), send o getColor())
) .

red eval(
  class c1 extends object
   method initialize() 1
   method m1() send self m2()
   method m2() 13
  class c2 extends c1
   method m1() 22
   method m2() 23
   method m3() super m1()
  class c3 extends c2
   method m1() 32
   method m2() 33
  main
   let o3 = new c3()
   in send o3 m3()
) .

red eval(
  class a extends object
   field i
   field j
   method initialize() 1
```

```
  method setup()
    {
      set i = 15 ;
      set j = 20 ;
      50
    }
  method f() send self g()
  method g() i + j
 class b extends a
  field j
  field k
  method setup()
    {
      set j = 100 ;
      set k = 200 ;
      super setup() ;
      send self h()
    }
  method g() list(i,j,k)
  method h() super g()
 class c extends b
  method g() super h()
  method h() k + j
 main
  let p = proc(o)
          let u = send o setup()
          in list(u, send o g(), send o f())
  in list(p(new a()), p(new b()), p(new c()))
) .


red eval(
 class a extends object
  field i
  field j
  method initialize() 1
  method setup()
    {
      set i = 15 ;
      set j = 20 ;
      50
    }
  method f() send self g()
```

```
    method g() i + j
  class b extends a
    field j
    field k
    method setup()
      {
        set j = 100 ;
        set k = 200 ;
        super setup() ;
        send self h()
      }
    method g() list(j,k)
    method h() super g()
  class c extends b
    method g() a
    method h() k + j
  main
    let p = proc(o)
              let u = send o setup()
              in list(send o g())
    in list(p(new c()))
) .
***> should be undefined



eof

--- the previous examples also work

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4
```

```
red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
```

```
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
```

```
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---              f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
```

```
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1
```

```
red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3
```

```
red eval(
  let f = let c = 0
          in proc()
              let d = set c = c + 1
              in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
              let d = set x = x + 1
              in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
              let t = x
              in let d = set x = y
                 in let d = set y = t
                    in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
```

```
                else let d = set x = x - 1
                     in 'odd(),
      'odd  = proc() if zero?(x)
                then 0
                else let d = set x = x - 1
                     in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
        'even = proc() if zero?(x) then 1
                  else let d = set x = x - 1
                       in 'odd(),
         'odd = proc() if zero?(x) then 0
                  else let d = set x = x - 1
                       in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
```

```
  in {
      set 'times4 = proc(x)
                      if zero?(x)
                      then 0
                      else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
```

```
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined
```

```
red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                y = 2,
                a = 3,
                z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
*******************************
*** Defining an OO Language ***
*******************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  op '(' : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sort BExp .
  op _equals_ : Exp Exp -> BExp .
  op zero? : Exp -> BExp .
  op even? : Exp -> BExp .
  op not_ : BExp -> BExp .
  op _and_ : BExp BExp -> BExp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : BExp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm
```

```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : BExp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting NAME-SYNTAX .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field_ : Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method___ : Name NameList Exp -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes .
  subsort Class < Classes .
  op noClasses : -> Classes .
  op class_extends__ : Name Name Methods -> Class [prec 115] .
  op class_extends___ : Name Name Fields Methods -> Class [prec 115] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm

fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm
```

```
fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm


-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op '[' : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: '[']] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod OBJ-ENVIRONMENT is protecting ENVIRONMENT .
  sort ObjEnv .
  op noObjEnv : -> ObjEnv .
  op (_,_) : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: noObjEnv] .
endfm

fmod OBJECT is extending OBJ-ENVIRONMENT .
  extending VALUE .
  sort Object .
  subsort Object < Value .
  op o : Name ObjEnv -> Object .
  op class : Object -> Name .
  var Xc : Name .  var OEnv : ObjEnv .
  eq class(o(Xc,OEnv)) = Xc .
endfm

fmod SUPER-CLASS is extending CLASS-SYNTAX .
  op superClass : Name Classes -> [Name] [memo] .
  var Fs : Fields .  var Ms : Methods .
  vars Xc Xc' : Name .  var Cls : Classes .
  eq superClass(Xc, (Cls class Xc extends Xc' Fs Ms)) = Xc' .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending OBJECT .
  extending SUPER-CLASS .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op ? : ->  [ValueStatePair] .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  vars X Xc Xc' : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .
  var OEnv : ObjEnv .  vars Cls Cls' : Classes .  vars O O' : Object .

  eq (env([X,L] Env) S){X} = L .
  eq S{X} = S{X}{currClass(S)} [owise] .
  eq (obj(o(Xc', (Xc, [X,L] Env) OEnv)) S){X}{Xc} = L .
  eq S{X}{Xc} = S{X}{superClass(Xc,classes(S))} [owise] .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
```

```
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState =
       env(noEnv) store(noStore) nextLoc(0)
       obj(o(object, noObjEnv)) currClass(object)
       classes(noClasses) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .

  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .

  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .

  op obj : Object -> StateAttribute .
  op obj : State -> Object .
  eq (obj(O) S) <** obj(O') = obj(O') S .
  eq obj(obj(O) S) = O .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> [ValueStatePair] .
  var X : Name .  var S : State .  var O : Object .
  eq eval(self, S) = {obj(S),S} .
  eq eval(X, S) = {S[X], S} [owise] .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> [ValueStatePair] .
  op  eval : ExpList State -> [ValueListStatePair] .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var El : ExpList .  vars S Sl : State .  var Vl : ValueList .

 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El,S) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op eval : BExp State -> [ValueStatePair] .
  vars E E' : Exp .  vars Be Be' : BExp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' : Exp .  var Be : BExp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  var Be : BExp .  var E : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod FIELD-SEMANTICS is protecting FIELD-SYNTAX .
  op names : Fields -> NameList .
  eq names(noFields) = () .
  var Xf : Name .  var Fs : Fields .
  eq names(field Xf Fs) = Xf, names(Fs) .
endfm

fmod CLASS-SEMANTICS is extending CLASS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending FIELD-SEMANTICS .
  vars X Xc Xc' : Name .  var Xl : NameList .  var OEnv : ObjEnv .
  var E : Exp .  var El : ExpList .  vars S S' Sf Sl Se Sm : State .
  var Fs : Fields .  var Ms : Methods .  var Cls : Classes .
  vars Ve Vm : Value .  var Vl : ValueList .  var Env : Env .

  eq class Xc extends Xc' Ms = class Xc extends Xc' noFields Ms .

  op createObject : Name State -> [ValueStatePair] .
  eq createObject(object,S) = {o(object,noObjEnv), S} .
 ceq createObject(Xc,S) = {o(Xc, (Xc, env(Sf)) OEnv), Sf <** env(Env)}
    if Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms := classes(S)
    /\ {o(Xc',OEnv), S'} := createObject(Xc',S)
    /\ Sf := (S' <** env(noEnv))[names(Fs) <- ?] [owise] .

  op invokeMethod : Name ExpList State -> [ValueStatePair] .
 ceq invokeMethod(initialize, El, S) = {int(0), S}
    if currClass(S) = object .
 ceq invokeMethod(X, El, S) = {Ve, Se <** env(Env)}
    if Xc := currClass(S) /\ Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S)
    /\ {Vl,Sl} := eval(El,S)
    /\ {Ve,Se} := eval(E, Sl[Xl <- Vl]) .
 ceq invokeMethod(X, El, S) = ?
    if Xc := currClass(S) /\ Env := env(S)
    /\ Cls class Xc extends Xc' Fs Ms (method X(Xl) E) := classes(S) [owise] .
 ceq invokeMethod(X, El, S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) [owise] .
endfm

fmod NEW-SEMANTICS is extending NEW-SYNTAX .
  extending CLASS-SEMANTICS .
  vars Xc Xc' : Name .  var El : ExpList .  var Vm : Value .
  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(new Xc(El), S) = {O', Sm <** obj(O) <** currClass(Xc')}
    if O := obj(S) /\ Xc' := currClass(S)
    /\ {O',S'} := createObject(Xc,S)
    /\ {Vm,Sm} := invokeMethod(initialize, El,
                    S' <** obj(O') <** currClass(Xc)) .
endfm

fmod SEND-SEMANTICS is extending SEND-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc : Name .  var E : Exp .  var El : ExpList .
  var Vm : Value .  vars S S' Sm : State .  vars O O' : Object .
 ceq eval(send E X(El), S) = {Vm, Sm <** obj(O) <** currClass(Xc)}
    if O := obj(S) /\ Xc := currClass(S)
    /\ {O',S'} := eval(E,S)
    /\ {Vm,Sm} := invokeMethod(X, El,
                    S' <** obj(O') <** currClass(class(O'))) .
endfm

fmod SUPER-SEMANTICS is extending SUPER-SYNTAX .
  extending CLASS-SEMANTICS .
  vars X Xc Xc' : Name .  var El : ExpList .
  var Vm : Value .  vars S Sm : State .
 ceq eval(super X(El), S) = {Vm, Sm <** currClass(Xc)}
    if Xc := currClass(S) /\ Xc' := superClass(Xc, classes(S))
    /\ {Vm,Sm} := invokeMethod(X, El, S <** currClass(Xc')) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending LIST-SEMANTICS .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
  extending NEW-SEMANTICS .
  extending SEND-SEMANTICS .
  extending SUPER-SEMANTICS .
  op eval_ : Program -> [Value] .
  var Cls : Classes .  var E : Exp .  var Ve : Value .  var Se : State .
 ceq eval(Cls main E) = Ve
    if {Ve,Se} := eval(E, initState <** classes(Cls)) .
endfm
```

```
red eval(
  class c extends object
    field a
    method initialize() set a = 5
    method m() a
  main
    let x = new c()
    in send x m()
) .

red eval(
  class c extends object
    field i
    field j
    method initialize(x)
      {
        set i = x ;
        set j = 0 - x
      }
    method add(d)
      {
        set i = i + d ;
        set j = j - d
      }
    method getstate()
      list(i,j)
  main
    let a = 0, b = 0, o = new c(5)
    in {
         set a = send o getstate() ;
         send o add(3) ;
         set b = send o getstate() ;
         list(a,b)
       }
) .

red eval(
  class c extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  main
    let o = new c()
    in send o m1()
) .

red eval(
  class oddeven extends object
    method initialize() 1
    method even(n)
      if zero?(n) then 1 else send self odd(n - 1)
    method odd(n)
      if zero?(n) then 0 else send self even(n - 1)
  main
    let o = new oddeven()
    in send o odd(17)
) .

red eval(
  class node extends object
    field left
    field right
    method initialize(l,r)
      {
        set left = l ;
        set right = r
      }
    method sum()
      (send left sum()) + (send right sum())
  class leaf extends object
    field value
    method initialize(v)
      set value = v
    method sum() value
  main
    let o = new node(new node(new leaf(3), new leaf(4)),
                     new leaf(5))
    in send o sum()
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation()
      list(x,y)
  class colorpoint extends point
    field color
    method setColor(c) set color = c
    method getColor()  color
  main
    let p = new point(3,4), cp = new colorpoint(10,20)
    in {
         send p move(3,4) ;
         send cp setColor(87) ;
         send cp move(10, 20) ;
         list(send p  getLocation(),
              send cp getLocation(),
              send cp getColor())
       }
) .

red eval(
  class c1 extends object
    field x
    field y
    method initialize() 1
    method setx1(v) set x = v
    method sety1(v) set y = v
    method getx1() x
    method gety1() y
  class c2 extends c1
    field y
    method sety2(v) set y = v
    method getx2() x
    method gety2() y
  main
    let o2 = new c2()
    in {
         send o2 setx1(101) ;
         send o2 sety1(102) ;
         send o2 sety2(999) ;
         list(send o2 getx1(), send o2 gety1(),
              send o2 getx2(), send o2 gety2())
       }
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() send self m1()
  class c2 extends c1
    method m1() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o2 m1(), send o2 m2())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() 1
    method m2() 100
    method m3() send self m2()
  class c2 extends c1
    method initialize() 1
    method m2() 2
  main
    let o1 = new c1(), o2 = new c2()
    in list(send o1 m1(), send o1 m2(), send o1 m3(),
            send o2 m1(), send o2 m2(), send o2 m3())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        set x = initx ;
        set y = inity ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class point extends object
    field x
    field y
    method initialize(initx, inity)
      {
        set x = initx ;
        set y = inity
      }
    method move(dx, dy)
      {
        set x = x + dx ;
        set y = y + dy
      }
    method getLocation() list(x,y)
  class colorpoint extends point
    field color
    method initialize(initx,inity,initcolor)
      {
        super initialize(initx, inity) ;
        set color = initcolor
      }
    method setColor(c) set color = c
    method getColor()  color
  main
    let o = new colorpoint(3, 4, 172)
    in list(send o getLocation(), send o getColor())
) .

red eval(
  class c1 extends object
    method initialize() 1
    method m1() send self m2()
    method m2() 13
  class c2 extends c1
    method m1() 22
    method m2() 23
    method m3() super m1()
  class c3 extends c2
    method m1() 32
    method m2() 33
  main
    let o3 = new c3()
    in send o3 m3()
) .

red eval(
  class a extends object
    field i
    field j
    method initialize() 1
    method setup()
      {
        set i = 15 ;
        set j = 20 ;
        50
      }
    method f() send self g()
    method g() i + j
  class b extends a
    field j
    field k
    method setup()
      {
        set j = 100 ;
        set k = 200 ;
        super setup() ;
        send self h()
      }
    method g() list(i,j,k)
```

```
    method h() super g()
  class c extends b
    method g() super h()
    method h() k + j
  main
    let p = proc(o)
              let u = send o setup()
              in list(u, send o g(), send o f())
    in list(p(new a()), p(new b()), p(new c()))
) .


red eval(
  class a extends object
    field i
    field j
    method initialize() 1
    method setup()
      {
        set i = 15 ;
        set j = 20 ;
        50
      }
    method f() send self g()
    method g() i + j
  class b extends a
    field j
    field k
    method setup()
      {
        set j = 100 ;
        set k = 200 ;
        super setup() ;
        send self h()
      }
    method g() list(j,k)
    method h() super g()
  class c extends b
    method g() a
    method h() k + j
  main
    let p = proc(o)
              let u = send o setup()
              in list(send o g())
    in list(p(new c()))
) .
***> should be undefined



eof

--- the previous examples also work

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4
```

```
red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
```

```
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping
```

```
red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2
```

```
red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
            {
              set a = a + b ;
              set b = a - b ;
              set a = a - b
            }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                y = 2,
                a = 3,
                z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

CS322 - Programming Language Design

Lecture 16: Defining a Typed Object-Oriented Language (part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We start investigating the relationship between objects and types in functional programming. The main idea is that an object `o` can be thought of having a type `c`, if an only if `c` is either the class for which `o` was created as an instance or a superclass of that class.

In order the make our OO language more interesting from the type checking point of view, we extend it with several features which can be encountered in many other OO languages, such as

- *type/class declarations*,
- *abstract classes*,
- *casting*, and
- *subtype polymorphism*.

Before defining our type checker rigorously, which will happen in the next lecture, we need to first understand all these features and also how the type checking procedure will work.

An Example in the Extended OO Language

The following OO program implements trees that can have binary nodes and leafs storing integers, together with methods `sum` and `equal`.

Both classes `node` and `leaf` will extend the following *abstract class*:

```
abstract class tree extends object
  method int initialize() 1
  abstractmethod int sum()
  abstractmethod bool equal(tree t)
```

Classes which are not abstract are called *concrete*. The major difference between abstract classes and concrete classes is that objects can be created (using `new`) only for concrete classes. An abstract class can contain *abstract methods*, which do not have a body, but can also contain *concrete* methods, which do have a

body. Notice that all the methods declare the "types" of their arguments as well as the types of their returned values.

The role of abstract classes is to provide static information about their future concrete subclasses. For example, the tree abstract class says that each concrete subclass of `tree` is supposed to override the methods `sum` and `equal`.

Let us now declare a first concrete subclass of `tree`, staying for binary nodes, which has two fields `left` and `right`:

```
class node extends tree
  field tree left
  field tree right
```

Notice that the fields are declared with a type, which is the abstract class `tree`. This says that in any instance of the class `node`, the two fields are expected to contain objects that are instances of `tree` (this is not possible though because `tree` is an

abstract class so it cannot have instance objects) ... or subclasses of it. This leads us to the following very important concept in OO programing languages:

> *Subtype polymorphism:* An object whose creation class is `c` can be used in any context where an object of class `c` or its superclasses is expected.

Therefore, any objects of subclasses of `tree`, including `node`, will be allowed to be assigned to these fields.

Let us next define the methods of `node`. When one creates a node object, one most likely wants to also initialize its `left` and `right` fields to other objects. Thus, one would want to have a binary `initialize` method as follows:

```
method void initialize(tree l, tree r)
  { set left = l ;
    set right = r }
```


Due to subtype polymorphism, `initialize` can be called with any objects of classes extending `tree`. Notice, however, that `initialize` in `node` has a different arity than the same method in its superclass, `tree`! We will therefore allow the `initialize` method to have different types across different classes in the hierarchy, and this creates a type safety leak as we will see later, but this will be the only exception. And there will be no safety leak if one *does not use* `initialize` explicitly (but only implicitly, when creating new objects).

The following three classes are straightforward:

```
method tree getleft() left
method tree getright() right
method int sum() (send left sum()) + (send right sum())
```

Note that `sum` overrides the corresponding abstract method in `tree`. The next class overrides the other abstract class in `tree`, called `equal`, whose role is to test whether two trees are equal.

Each subclass of `tree` is expected to define its own `equal` concrete method, because what it means for two trees to be equal is dependent on the particular structure of the tree:

```
method bool equal(tree t)
  if instanceof t node
  then if send left equal(send cast t node getleft())
       then send right equal(send cast t node getright())
       else false
  else false
```

There are two new language constructs in the method above, namely `instanceof` and `cast`. Their behavior is as follows:

- `instanceof <Exp> <Name>` evaluates the `<Exp>` to an object and then returns true if and only if that object is an instance of `<Name>` or of any of its superclasses.
- `cast <Exp> <Name>` evaluates the `<Exp>` to an object and then

  returns *that object* when it is an instance of `<Name>` or of any of its superclasses, and a runtime error otherwise.

Casting objects is, in my view, a *misleading terminology*. This is because any object has a definite class, set when it was instantiated, and nothing can change that class during the lifetime of the object.

Similarly, we can now define the `leaf` concrete subclass of `tree`:

```
class leaf extends tree
  field int value
  method void initialize(int v) set value = v
  method int sum() value
  method int getvalue() value
  method bool equal(tree t)
    if instanceof t leaf
    then zero?(value - (send cast t leaf getvalue()))
    else false
```

One can now evaluate the following in the context of these classes:

```
main
  let o1 = new node(new node(new leaf(3),
                             new leaf(4)),
                    new leaf(5))
  in list(send o1 sum(), if send o1 equal(o1) then 100 else 200)
```

A definition of this extension of the language can be very easily obtained from `oo-lang.maude`. One should first make sure that an object is never created for an abstract class, by checking whether the class specified in a `new` statement is concrete or not. Then one should add syntax and semantics for `instanceof` and `cast`, most likely something in the style below, where `superOrEqual` tests whether a class is equal to or a superclass of another class (should be this memoized?):


```
...
  ceq eval(instanceof E Xc, S) =
      {bool(superOrEqual(Xc, class(O), classes(S))), Se}
     if {O,Se} := eval(E,S) .
...
  ceq eval(cast E Xc, S) = {O,Se}
     if {O,Se} := eval(E, S)
     /\ superOrEqual(Xc, class(O), classes(S)) .
...
```

Homework Exercise 1 *Modify the definition of the OO language in* `oo-lang.maude` *to support type declarations and the other features described above. You do not need to check the types at runtime.*

Type Checking the OO Language

We next focus on the principles underlying our type checking tool that will be formally defined in the next lecture. The role of the type checker is to ensure that certain kinds of errors will never occur at runtime. Besides the other errors considered by the type checker of our functional language, we are now also considering:

- sending a message to an object which does not provide the corresponding method;
- sending a message with the wrong number and type of arguments;
- creating an object for an abstract class;
- creating an object for a concrete subclass of an abstract class, in which one or more of the abstract methods were not overridden by concrete methods.

Technique: Abstract Interpretation

As we have seen so far in the class, all our definitions have almost the same structure, in the sense that they *give meaning* to the syntactic programming language constructs. One particular kind of meaning that one can associate to an expression is, of course, its value. More generally, making abstraction of functions, the meaning of an expression can really be viewed as a function from integer values associated to its free names into an integer value associated to itself. Something of the form:

$$valueMeaning(E) := [free(E) \to Int] \to Int$$

We have rigorously defined the meaning of each expression inductively, where the map $free(E) \to Int$ was provided implicitly by the state that was carried as an extra argument of `eval`.

In the context of the typed functional language, what we did was to think of a type as the value of an expression. And what we really defined inductively over the structure of expressions was a type specific meaning of expressions:

$$typeMeaning(E) := [free(E) \to Type] \to Type$$

Therefore, the concrete values bound to names were abstracted away by their types. It is then not surprising that we used the same definitional structure for defining a type checker as we used for defining the semantics of the language. And in general, almost if not all software analysis tools should follow the same pattern.

The technique underlying all these definitions is called *abstract interpretation*, It is typically defined quite mathematically, but in this course we keep it simple and just describe it intuitively in the context of our programming languages:

- Select a set of *abstract values* that you are interested to analyze;
- Define the *abstract meaning* of a program as a map

$$absMeaning(E) := [free(E) \to absValue] \to absValue$$

Once one fixes the abstract values of interest and *understands* how they propagate, then one can state it formally following the same pattern used to define the semantics of the programming languages. The problem may sometimes be quite non-trivial, because in order to apply the abstraction on a certain language construct one may need to do quite involved reasoning (think of type inference, for example).

Types of Objects

So we now have to define the abstract values of our new analysis tool, the OO type checker. While the types of the other expressions are like before, that is, elements in the infinite set of well formed terms over the type constructors, the types of objects are to be considered differently.

What is the type of an object? The class that the object was created as an instance for? The answer is certainly no, because otherwise we could not handle subtype polymorphism. The type of an object, which is its abstract value in what follows, is a *set of possible classes*, namely all those for which it can be an instance of (in the sense of `instanceof`). Thus, we say that an object `o` *has type* `c` if and only if its class is either `c` or a subclass of `c`.

What we will do is to follow the structure of the executable

semantics and adapt it to "evaluate" expressions to OO types.

Processing Classes

The typing policy that must be enforced when processing the class declarations is the following:

- Each concrete class has only concrete methods;
- Each method overriding a superclass method *has the same type* as the overridden method. This is because one cannot know statically to which method in the chain of classes of an object a method invocation refers to (because of subtype polymorphism), so the best one can do is to enforce that overriding methods preserve the types (modulo subtype polymorphism, of course). One exception here will be the `initialize` method, which is allowed to have different types from superclasses to subclasses.

Processing Methods

When processing methods, one must check that the following typing policy is not violated:

- Nothing to worry about in the case of abstract methods;
- The concrete methods have to be type checked to make sure that they have their declared type. In order to do it, the typed parameters are added to the environment and then the body of the method is evaluated. The obtained type must be either the declared type or a subtype of it.

Processing Expressions

Nothing changes with respect to type checking the already known expressions, except that one should now consider the *subtype polymorphism* aspect overall. More precisely, the declared type of a bound name can be equal to or a supertype of the actual type obtained by type checking (or evaluating) the bound expression.

Processing the `new` Construct

When processing `new`, we will check the following policy:

- The corresponding class should have been declared;
- The corresponding class is concrete;
- The initialization is *type safe*, that is, the number and the types of arguments match the declared ones, modulo the policy of subtype polymorphism;
- The corresponding class type is then the returned type after evaluating the `new` construct.

Processing Method Invocation

The typing policy for method invocation is the following:

- The expression to which the message is sent must evaluate to an object type, that is, its corresponding class;
- After retrieving that class's info, check that the method's name is accessible from that class; we don't have to worry about that method being abstract, because we already know that we are not in an abstract class and that each abstract method has been overridden;
- After retrieving the corresponding method declaration, check that its declared type matches the type of the type of the argument expressions, by type checking them; this has also to be done modulo the subtype polymorphism.

Processing `instanceof` and `cast`

In the case of `instanceof`, the only thing one needs to do is to check that its first argument evaluates to an object type and that its second argument is defined as a class.

The situation is more problematic for `cast`, because it is *impossible* to guarantee statically that it will never generate a runtime error! Why? The best we can do is to *reject casts that will always fail*.

How can we detect those casts that will always fail? This is relatively easy, by *intersecting* the OO type of the object (i.e., a set) to which its first argument evaluates (if it does not evaluate to an object then issue a type error) with the set obtained by taking the union of the second class and its subclasses. If the intersection is *empty* then the cast will always fail, so issue a type error. Otherwise we cannot say anything.

CS322 - Programming Language Design

Lecture 17: Defining a Typed Object-Oriented Language (part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Let us now define formally the type checker for the OO language that we discussed in Lecture 16. As before, based on the principles of *abstract interpretation*, we inductively go over the structure of the programs/expressions and define the "meaning" of each language construct with respect to the abstract values, which in our case are types with subtype polymorphism.

As before, we will start with defining the syntax of the language then the needed state infrastructure, and finally the mathematical semantics of the type checker. In order to define the semantics, we just have to formalize the typing policy discussed in Lecture 16. As usual, due to the executable capabilities of Maude, we will obtain an type checking interpreter for free.

Syntax of Arithmetic/Boolean Expressions

Like before, we define syntax for names, generic expressions, arithmetic operators, etc. However, a subtle change is the syntax is that we now allow boolean expressions to be just like any other expressions, so they can be taken as arguments and returned by functions, and we add two special boolean constants, `true` and `false`. Note that this decision slightly increases the complexity of the type checker, but it allows more flexibility in writing programs.

The new syntax for boolean expressions, conditionals and loops becomes:


```
fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
  ops true false : -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm
```

Type Constructors

Types can now also be *class names*. In addition to these, we also need to introduce *list types*. Note that, in order to avoid parsing conflicts with the multiplication and list operations which is also are correctly defined on names (regarded as expressions), we use `_*'_` and `list'` for type constructors:

```
fmod TYPE is protecting NAME .
  sorts BasicType Type .
  subsorts BasicType Name < Type .
  ops int bool void : -> BasicType .
  op _->_ : Type Type -> Type [prec 2] .
  op list' : Type -> Type .
  op _*'_ : Type Type -> Type [assoc prec 1] .
--- to distinguish them from _*_ and list which also work on names
endfm
```


Syntax of Bindings

When we type checked our functional language, we required the user to provide types for all bound names. However, this is a bit too strong, because in the case of `let`, the type of bound name will be the type of the binding expression. From now we let it optional to declare types in bindings:

```
fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE .
  sorts Binding BindingList .  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
  op __=_ : Type Name Exp -> Binding [prec 70] .
endfm
```

Programs with `letrec` untyped bindings will *not* type check. Why?

Syntax for Parameters

Typed parameters will be needed both for function declarations, and for concrete and abstract method declarations. We define them exactly like in the case of our typed functional language:

```
fmod PARAMETER-SYNTAX is protecting TYPE .
  sorts Param ParamList .
  subsort Param < ParamList .
  op __ : Type Name -> Param [prec 3] .
  op `(` : -> ParamList .
  op _,_ : ParamList ParamList -> ParamList [assoc id: ()] .
endfm
```

No other syntactic changes are needed in the syntax of the corresponding features borrowed from the typed functional language.

Syntax of Methods

In order to define the syntax for classes, we first need to define syntax for fields and methods. Fields can be borrowed from the previous OO language, but methods need to be changed in order to accommodate *abstract methods*:

```
fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting PARAMETER-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method____ : Type Name ParamList Exp -> Method [prec 105] .
  op abstract`method___ : Type Name ParamList -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm
```

Syntax of Classes

Classes also need to be slightly changed, in order to accommodate *abstract classes*. Also, to ease other definitions later, we prefer to add special syntax for the class body, which contains fields and methods:

```
fmod CLASS-SYNTAX is protecting FIELD-SYNTAX . protecting METHOD-SYNTAX .
  sorts Class Classes ClassSpecifier ClassBody .
  subsort Class < Classes .
  subsort Methods < ClassBody .
  op noClasses : -> Classes .
  op __extends__ : ClassSpecifier Name Name ClassBody -> Class [prec 115] .
  ops class abstract`class : -> ClassSpecifier .
  op __ : Fields Methods -> ClassBody [prec 112] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm
```


Syntax for `instanceof` and `cast`

Nothing changes in the syntax of `new`, `send` and `super`, but we need to add new syntax for `instanceof` and `cast`:

```
fmod INSTANCEOF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op instanceof__ : Exp Name -> Exp [prec 1] .
endfm

fmod CAST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op cast__ : Exp Name -> Exp [prec 1] .
endfm
```

We can now *put all the syntax together* in a module `PROG-SYNTAX` as we did before and thus finalize the syntax of our typed OO language.

Defining the Semantics of the OO Type Checker

In what follows, we will only discuss *part of* the formal definition.

Homework Exercise 1 *Complete the definition of the semantics of the OO Type Checker discussed in this lecture. Extend the provided file* `oo-type-checking.maude`*. Your definition should type-check the program provided at the end of the file in order to be considered. You need to define all the auxiliary infrastructure as well as the meaning of* `cast`*. All the auxiliary infrastructure will be informally discussed in what follows. Feel free to modify the provided code however you wish (it may even contain - planted or not - errors).*

Exercise 1 *Read Chapter 6 in the Friedman book for how to* implement *OO type checkers, if you are interested. In this course we focus on how to define them mathematically.*

Defining the Auxiliary Infrastructure

Let us discuss the infrastructure operations that you will need to define formally as part of your homework assignment. The very first thing that will be needed is the notion of *type environment*:

```
fmod TYPE-ENVIRONMENT is
  sort TEnv .
*** add your code here
endfm
```

Then a series of *helping operations* needs to be defined. You can place all the definitions in only module. All these operations are *partial* and they can be *memoized* for efficiency. The meaning of these operations will be explained as we encounter them in the definition of the type checker:

```
fmod AUX-CLASS-OPS is extending CLASS-SYNTAX .
  extending TYPE-ENVIRONMENT .
  op superClass : Name Classes -> [Name] [memo] .
  op concreteClass : Name Classes -> [Bool] [memo] .
  op methodsOnPath : Name Classes -> [Methods] [memo] .
  op fieldTEnv : Name Classes -> [TEnv] [memo] .
  op subtypeOf : Type Type Classes -> [Bool] [memo] .
  op declType : Name Name Classes -> [Type] [memo] .
*** add your code here
endfm
```


The module STATE also needs to be completed with definitions for all the state infrastructure operations, which are by now quite familiar:

```
fmod STATE is extending AUX-CLASS-OPS .
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  op _[_] : State Name -> Type .
  op _[_<-_] : State Name Type -> State .
  op initState : -> State .
*** add your code here
...
endfm
```

Type Checking Expressions

Type checking the non-OO expression constructs does not change much from what we had before. However, we have to consider now the fact that boolean expressions are treated like the other expression. Below there are some code fragments.

Generic expressions:

```
  eq type(I, S) = int .
  eq type(X, S) = S[X] .
 ceq type((E,El), S) = type(E,S) *' type(El, S) if El =/= () .
```

Lists:

```
  eq type(list(E), S) = list'(type(E,S)) .
 ceq type(list(E,El), S) = list'(T)
    if T := type(E,S) /\ type(list(El),S) = list'(T) [owise] .
```


Arithmetic and boolean expressions:

```
 ceq type(E + E', S) = int if type(E,S) = int /\ type(E',S) = int .
...
 ceq type(zero?(E), S) = bool if type(E, S) = int .
 ceq type(Be and Be', S) = bool
    if type(Be, S) = bool /\ type(Be', S) = bool .
  eq type(true, S) = bool .
  eq type(false, S) = bool .
```

Assignments, blocks and loops:

```
 ceq type(set X = E, S) = void if type(E,S) = S[X] .
...
  eq type({E}, S) = type(E,S) .
 ceq type({E ; El}, S) = type({El}, S) if type(E,S) = void .
...
 ceq type(while Be E, S) = void
    if type(Be, S) = bool /\ type(E, S) = void .
```

Type Checking Bindings

Since bindings are not required to come with a declared type, we need to treat two different cases. Note, however, that if there is a declared type, then the *subtype polymorphism* rule applies:

```
fmod BINDING-TYPE-CHECKING is extending BINDING-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  vars T T' : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- T'])
    if T' := type(E, S) /\ subtypeOf(T', T, classes(S)) .
  eq bindBList((X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- type(E,S)]) .
endfm
```


Type Checking Functions

There is nothing different in the definition of the type checker for functions, except that the *subtype polymorphism* rule needs to also be considered: a function can be invoked on arguments whose types can be subtypes of the declared types:

```
fmod PROC-TYPE-CHECKING is extending PROC-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  var Pl : ParamList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) = typePList(Pl) -> type(E, bindPList(Pl, S)) .
 ceq type(E El, S) = T
    if Tp -> T := type(E, S)
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```

Type Checking Methods

As discussed in the previous lecture, methods need to be type checked at class-time definition, to ensure that that they indeed return the promised type, modulo subtype polymorphism:

```
fmod METHOD-TYPE-CHECKING is extending METHOD-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  extending GENERIC-EXP-TYPE-CHECKING .
  op type-check : Methods State -> Bool .
  var S : State .  var Ms : Methods .  var T : Type .  var Xm : Name .
  var Pl : ParamList .  var E : Exp .
  eq type-check(noMethods, S) = true .
  eq type-check((Ms abstract method T Xm(Pl)), S) = type-check(Ms, S) .
 ceq type-check((Ms method T Xm(Pl) E), S) = type-check(Ms, S)
    if subtypeOf(type(E, bindPList(Pl, S)), T, classes(S)) .
endfm
```


In the definition above we assumed that the state already contains the type bindings of the fields of the method's class, as well as all the needed fields from its superclasses. This will be done when the class is type checked, from where the above operation is called and where the type safety with respect to overriding will also be checked.

Type Checking Classes

An operation `type-check : Classes State -> Bool` is defined next, which says whether a set of classes is correctly typed:

```
fmod CLASS-TYPE-CHECKING is extending CLASS-SYNTAX .
  protecting METHOD-TYPE-CHECKING .
  op type-check : Classes State -> Bool .
```

One of the very first typing policies for classes says that *no concrete class should declare abstract methods*. This can be simply done with the following equation, speculating AC matching:

```
eq type-check((Cls class Xc extends Xc'
                    Fs Ms abstract method T Xm(Pl)), S) = false .
```

Another important safety policy says that any method overriding another method should have *exactly* the same type as the overridden method. In our Maude framework, one rapid way to do this, which would probably be considered inefficient in the context of implementing a programming language, is as follows:

```
ceq type-check((Cls CS Xc extends Xc' Cb), S) = false
   if Ms (method T Xm(Pl) E) (method T' Xm(Pl') E')
        := methodsOnPath(Xc, classes(S))
   /\ Xm =/= initialize
   /\ typePList(Pl) -> T =/= typePList(Pl') -> T' .
```

Here, the auxiliary operation `methodsOnPath`, which you will define

as part of your homework, returns the *set* of all the methods defined on the path to the specified class current. If there are two methods having the same name, which is different from `initialize` (because this is the only accepted exception to the typing rule), and different declared types then the typing policy is violated.

If all the method declarations respect the type safety requirements when they are considered together, the only thing left to check is that each individual method respects its declared type. In order to do that, we need to type the body of each method in each class in a state which is aware of the current class (this is needed because of potential `super` invocations within the method body) and which sees no other type bindings than the class (and implicitly its superclass hierarchy) field declarations:

```
eq type-check((Cls CS Xc extends Xc' Fs Ms), S) =
   type-check(Cls, S) and type-check(Ms,
     S <** currClass(Xc) <** tenv(fieldTEnv(Xc, classes(S)))) [owise] .
```

Type Checking new

The type policy of new is that a new object can be created only for concrete classes, and in order to check this you need to define the operation concreteClass, and that the corresponding initialize method is type compatible with the expression list of new, modulo, of course, subtype polymorphism:

```
fmod NEW-TYPE-CHECKING is extending NEW-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xc : Name . var El : ExpList . var S : State . vars Tp T : Type .
 ceq type(new Xc(El), S) = Xc  if concreteClass(Xc, classes(S))
    /\ Tp -> T := declType(initialize, Xc, classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```

declType traverses the class hierarchy to find the latest declaration of the specified method.

Type Checking send

This is similar to (part of) the definition of new. Notice that subtype polymorphism is again taken into account:

```
fmod SEND-TYPE-CHECKING is extending SEND-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var E : Exp .  var El : ExpList .
  vars Tp T : Type .  var S : State .
 ceq type(send E Xm(El), S) = T
    if Tp -> T := declType(Xm, type(E,S), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```

Type Checking `super`

The difference between `send` and `super` is that the latter searches for the corresponding method starting with the superclass of the current class (that's why we need to store the current class in the state!):

```
fmod SUPER-TYPE-CHECKING is extending SUPER-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var El : ExpList .
  vars Tp T : Type .  var S : State .
  ceq type(super Xm(El), S) = T
    if Tp -> T :=
       declType(Xm, superClass(currClass(S), classes(S)), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```


Type Checking `instanceof`

As said in Lecture 16, the only thing we need to check here is its argument type is an existing class:

```
fmod INSTANCEOF-TYPE-CHECKING is extending INSTANCEOF-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  vars Xc Xc' Xc'' : Name .  var S : State .
  var Cls : Classes .  var CS : ClassSpecifier .  var Cb : ClassBody .
  ceq type(instanceof E Xc, S) = bool
     if Xc' := type(E,S) /\ Cls CS Xc' extends Xc'' Cb := classes(S) .
endfm
```

One can also check that the class `Xc` is declared.

Type Checking `cast`

This is left entirely to you as part of the homework. Reread the explanations in Lecture 16 on how `cast` is conservatively typed.

```
fmod CAST-TYPE-CHECKING is extending CAST-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var Xc : Name .  var S : State .  var T : Type .
***> add your code here
endfm
```


Putting the Type Checker Together

The type of a program makes sense only if its classes type check properly. If this is the case, then the type of the main expression is returned, calculated in the state containing all the correctly typed classes.

```
fmod PROG-LANG-TYPE-CHECKING is extending PROG-LANG-SYNTAX .
  op type_ : Program -> [Type] .
  var Cls : Classes .  var E : Exp .  var Te : Type .  vars S Se : State .
 ceq type(Cls main E) = type(E, S)
    if S := initState <** classes(Cls)
    /\ type-check(Cls, S) .
endfm
```

Homework Exercise 2 (Extra credit: 3 points). *Add* interfaces *to the typed OO language presented in the last two lectures. Your need to provide* two *definitions: one for the semantics of the language (as an extension of the homework problem in Lecture 16), and one for the typer checker (as an extension of the homework problem in this lecture). Read Exercise 6.12 for the informal definition and the syntax of interfaces.*

```
**************************************
*** Defining a typed OO Language ***
**************************************

--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  ops tree equal getleft getright getvalue : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op `(`) : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
```

```
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
  ops true false : -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod TYPE is
  protecting NAME .
  sorts BasicType Type .
  subsorts BasicType Name < Type .
  ops int bool void : -> BasicType .
  op _*'_ : Type Type -> Type [assoc prec 1] .
--- to distinguish it from _*_ which works on names as expressions
  op _->_ : Type Type -> Type [prec 2] .
  op list' : Type -> Type .
--- to distinguish it from list which works on names as expressions
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
  op __=_ : Type Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PARAMETER-SYNTAX is protecting TYPE .
  sorts Param ParamList .
```

```
  subsort Param < ParamList .
  op __ : Type Name -> Param [prec 3] .
  op `(`) : -> ParamList .
  op _,_ : ParamList ParamList -> ParamList [assoc id: ()] .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  protecting PARAMETER-SYNTAX .
  op proc__ : ParamList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 1] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting TYPE .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field__ : Type Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields prec 110] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting PARAMETER-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
```

```
  op noMethods : -> Methods .
  op method____ : Type Name ParamList Exp -> Method [prec 105] .
  op abstract`method___ : Type Name ParamList -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes ClassSpecifier ClassBody .
  subsort Class < Classes .
  subsort Methods < ClassBody .
  op noClasses : -> Classes .
  op __extends__ : ClassSpecifier Name Name ClassBody -> Class [prec 115] .
  ops class abstract`class : -> ClassSpecifier .
  op __ : Fields Methods -> ClassBody [prec 112] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm

fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm

fmod INSTANCEOF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op instanceof__ : Exp Name -> Exp [prec 1] .
endfm

fmod CAST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op cast__ : Exp Name -> Exp [prec 1] .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
```

```
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  extending CAST-SYNTAX .
  extending INSTANCEOF-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm

parse(
  abstract class tree extends object
    method int initialize() 1
    abstract method int sum()
    abstract method bool equal(tree t)

  class node extends tree
    field tree left
    field tree right
    method void initialize(tree l, tree r)
      {
        set left = l ;
        set right = r
      }
    method tree getleft() left
    method tree getright() right
    method int sum() (send left sum()) + (send right sum())
    method bool equal(tree t)
      if instanceof t node
      then if send left equal(send cast t node getleft())
           then send right equal(send cast t node getright())
           else false
      else false

  class leaf extends tree
    field int value
    method void initialize(int v) set value = v
```

```
   method int sum() value
   method int getvalue() value
   method bool equal(tree t)
    if instanceof t leaf
    then zero?(value - (send cast t leaf getvalue()))
    else false

  main
   let o1 = new node(new leaf(3), new leaf(4)),
       o2 = new leaf(5),
       o  = new leaf(5)
   in let o3 = new node(o1, o2)
      in list(send o1 sum(),
              send o2 sum(),
              send o3 sum(),
              if send o1 equal(o2) then 100 else 200,
              if send o3 equal(o3) then 100 else 200)
) .



---------------------
--- Type Checking ---
---------------------

fmod TYPE-ENVIRONMENT is
  sort TEnv .
*** add your code here
endfm

fmod AUX-CLASS-OPS is extending CLASS-SYNTAX .
  extending TYPE-ENVIRONMENT .
  op superClass : Name Classes -> [Name] [memo] .
  op concreteClass : Name Classes -> [Bool] [memo] .
  op methodsOnPath : Name Classes -> [Methods] [memo] .
  op fieldTEnv : Name Classes -> [TEnv] [memo] .
  op subtypeOf : Type Type Classes -> [Bool] [memo] .
  op declType : Name Name Classes -> [Type] [memo] .
*** add your code here
endfm

fmod STATE is
  extending AUX-CLASS-OPS .
```

```
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  op _[_] : State Name -> Type .
  op _[_<-_] : State Name Type -> State .
  op initState : -> State .

*** add your code here

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  var S : State .  vars Xc Xc' : Name .
  vars Cls Cls' : Classes .  vars TEnv TEnv' : TEnv .
  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .
  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .
  op tenv : TEnv -> StateAttribute .
  op tenv : State -> TEnv .
  eq (tenv(TEnv) S) <** tenv(TEnv') = tenv(TEnv') S .
  eq tenv(tenv(TEnv) S) = TEnv .
endfm

fmod GENERIC-EXP-TYPE-CHECKING is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op type : ExpList State -> Type .
  var I : Int .  var X : Name .  var S : State .
  var E : Exp .  var El : ExpList .
  eq type(I, S) = int .
  eq type(X, S) = S[X] .
  eq type((), S) = void .
 ceq type((E,El), S) = type(E,S) *' type(El, S) if El =/= () .
endfm

fmod LIST-TYPE-CHECKING is extending LIST-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
```

```
  var E : Exp .  var El : ExpList .  var S : State .  var T : Type .
  eq type(list(E), S) = list'(type(E,S)) .
 ceq type(list(E,El), S) = list'(T)
    if T := type(E,S) /\ type(list(El),S) = list'(T) [owise] .
endfm

fmod ARITH-OPS-TYPE-CHECKING is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars E E' : Exp .  var S : State .
 ceq type(E + E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E - E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E * E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E / E', S) = int if type(E,S) = int /\ type(E',S) = int .
endfm

fmod BEXP-TYPE-CHECKING is extending BEXP-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars E E' Be Be' : Exp .  var S : State .
 ceq type(E equals E', S) = bool if type(E,S) = type(E',S) .
 ceq type(zero?(E), S) = bool if type(E, S) = int .
 ceq type(even?(E), S) = bool if type(E, S) = int .
 ceq type(not(Be), S) = bool if type(Be, S) = bool .
 ceq type(Be and Be', S) = bool if type(Be, S) = bool /\ type(Be', S) = bool .
  eq type(true, S) = bool .
  eq type(false, S) = bool .
endfm

fmod IF-TYPE-CHECKING is extending IF-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars Be E E' : Exp .  var S : State .
 ceq type(if Be then E else E', S) = type(E, S)
    if type(Be,S) = bool /\ type(E,S) = type(E',S) .
endfm

fmod BINDING-TYPE-CHECKING is extending BINDING-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  vars T T' : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- T'])
    if T' := type(E, S)
```

```
    /\ subtypeOf(T', T, classes(S)) .
  eq bindBList((X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- type(E,S)]) .
endfm

fmod LET-TYPE-CHECKING is extending LET-SYNTAX .
  extending BINDING-TYPE-CHECKING .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq type(let Bl in E, S) = type(E, bindBList(Bl, S) in S) .
endfm

fmod PARAMETER-TYPE-CHECKING is protecting PARAMETER-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  op typePList : ParamList -> Type .
  op bindPList : ParamList State -> State .
  var S : State .  var T : Type .  var X : Name .  var Pl : ParamList .
  eq typePList(()) = void .
  eq typePList(T X, Pl) = T *' typePList(Pl) .
  eq bindPList((), S) = S .
  eq bindPList((T X, Pl), S) = bindPList(Pl, S[X <- T]) .
endfm

fmod PROC-TYPE-CHECKING is extending PROC-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  var Pl : ParamList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) = typePList(Pl) -> type(E, bindPList(Pl, S)) .
 ceq type(E El, S) = T
    if Tp -> T := type(E, S)
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod LETREC-TYPE-CHECKING is extending LETREC-SYNTAX .
  extending BINDING-TYPE-CHECKING .
  op bindBlindlyBList : BindingList State -> State .
  var S : State .  var X : Name .  var E : Exp .
  var T : Type .  var Bl : BindingList .
  eq bindBlindlyBList(none,S) = S .
  eq bindBlindlyBList((T X = E, Bl), S) =
     bindBlindlyBList(Bl, S[X <- T]) .
  eq type(letrec Bl in E, S) =
     type(E, bindBList(Bl, bindBlindlyBList(Bl,S)) in S) .
endfm
```

```
fmod VAR-ASSIGNMENT-TYPE-CHECKING is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var X : Name .  var E : Exp .  var S : State .
 ceq type(set X = E, S) = void if type(E,S) = S[X] .
endfm

fmod BLOCK-TYPE-CHECKING is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var El : ExpList; .  var S : State .
  eq type({E}, S) = type(E,S) .
 ceq type({E ; El}, S) = type({El}, S) if type(E,S) = void .
endfm

fmod LOOP-TYPE-CHECKING is extending LOOP-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Be E : Exp .  var S : State .
 ceq type(while Be E, S) = void if type(Be, S) = bool /\ type(E, S) = void .
endfm

fmod METHOD-TYPE-CHECKING is extending METHOD-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  extending GENERIC-EXP-TYPE-CHECKING .
  op type-check : Methods State -> Bool .
  var S : State .  var Ms : Methods .  var T : Type .  var Xm : Name .
  var Pl : ParamList .  var E : Exp .
  eq type-check(noMethods, S) = true .
  eq type-check((Ms abstract method T Xm(Pl)), S) = type-check(Ms, S) .
 ceq type-check((Ms method T Xm(Pl) E), S) = type-check(Ms, S)
    if subtypeOf(type(E, bindPList(Pl, S)), T, classes(S)) .
endfm

fmod CLASS-TYPE-CHECKING is extending CLASS-SYNTAX .
  protecting METHOD-TYPE-CHECKING .
  op type-check : Classes State -> Bool .
  var Cls : Classes .  vars X Xc Xc' Xm : Name .
  var Pl Pl' : ParamList .  vars E E' : Exp .
  vars T T' : Type .  var Cb : ClassBody .
  var CS : ClassSpecifier .  var Fs : Fields .
  var Ms : Methods .  var S : State .
  eq type-check((Cls class Xc extends Xc'
                   Fs Ms abstract method T Xm(Pl)), S) = false .
 ceq type-check((Cls CS Xc extends Xc' Cb), S) = false
```

```
    if Ms (method T Xm(Pl) E) (method T' Xm(Pl') E')
        := methodsOnPath(Xc, classes(S))
    /\ Xm =/= initialize
    /\ typePList(Pl) -> T =/= typePList(Pl') -> T' .
  ceq type-check((Cls CS Xc extends Xc' Cb), S) = false
    if Ms (abstract method T Xm(Pl)) (method T' Xm(Pl') E')
        := methodsOnPath(Xc, classes(S))
    /\ Xm =/= initialize
    /\ typePList(Pl) -> T =/= typePList(Pl') -> T' .
  eq type-check((Cls CS Xc extends Xc' Fs Ms), S) =
     type-check(Cls, S) and type-check(Ms,
       S <** currClass(Xc) <** tenv(fieldTEnv(Xc, classes(S)))) [owise] .
endfm

fmod NEW-TYPE-CHECKING is extending NEW-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xc : Name .  var El : ExpList .
  var S : State .  vars Tp T : Type .
  ceq type(new Xc(El), S) = Xc
    if concreteClass(Xc, classes(S))
    /\ Tp -> T := declType(initialize, Xc, classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod SEND-TYPE-CHECKING is extending SEND-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var E : Exp .  var El : ExpList .
  vars Tp T : Type .  var S : State .
  ceq type(send E Xm(El), S) = T
    if Tp -> T := declType(Xm, type(E,S), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod SUPER-TYPE-CHECKING is extending SUPER-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var El : ExpList .
  vars Tp T : Type .  var S : State .
  ceq type(super Xm(El), S) = T
    if Tp -> T :=
       declType(Xm, superClass(currClass(S), classes(S)), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```

```
fmod INSTANCEOF-TYPE-CHECKING is extending INSTANCEOF-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  vars Xc Xc' Xc" : Name .  var S : State .
  var Cls : Classes .  var CS : ClassSpecifier .  var Cb : ClassBody .
  ceq type(instanceof E Xc, S) = bool
     if Xc' := type(E,S) /\ Cls CS Xc' extends Xc" Cb := classes(S) .
endfm

fmod CAST-TYPE-CHECKING is extending CAST-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var Xc : Name .  var S : State .  var T : Type .
***> add your code here
endfm

fmod PROG-LANG-TYPE-CHECKING is extending PROG-LANG-SYNTAX .
  extending LIST-TYPE-CHECKING .
  extending ARITH-OPS-TYPE-CHECKING .
  extending IF-TYPE-CHECKING .
  extending LET-TYPE-CHECKING .
  extending PROC-TYPE-CHECKING .
  extending LETREC-TYPE-CHECKING .
  extending VAR-ASSIGNMENT-TYPE-CHECKING .
  extending BLOCK-TYPE-CHECKING .
  extending LOOP-TYPE-CHECKING .
  extending NEW-TYPE-CHECKING .
  extending SEND-TYPE-CHECKING .
  extending SUPER-TYPE-CHECKING .
  extending INSTANCEOF-TYPE-CHECKING .
  extending CAST-TYPE-CHECKING .

  extending CLASS-TYPE-CHECKING .
  op type_ : Program -> [Type] .
  var Cls : Classes .  var E : Exp .  var Te : Type .  vars S Se : State .
 ceq type(Cls main E) = type(E, S)
    if S := initState <** classes(Cls)
    /\ type-check(Cls, S) .
endfm

red type(
  abstract class tree extends object
    method int initialize() 1
    abstract method int sum()
    abstract method bool equal(tree t)
```

```
class node extends tree
  field tree left
  field tree right
  method void initialize(tree l, tree r)
    {
      set left = l ;
      set right = r
    }
  method tree getleft() left
  method tree getright() right
  method int sum() (send left sum()) + (send right sum())
  method bool equal(tree t)
    if instanceof t node
    then if send left equal(send cast t node getleft())
         then send right equal(send cast t node getright())
         else false
    else false

class leaf extends tree
  field int value
  method void initialize(int v) set value = v
  method int sum() value
  method int getvalue() value
  method bool equal(tree t)
    if instanceof t leaf
    then zero?(value - (send cast t leaf getvalue()))
    else false

main
  let o1 = new node(new leaf(3), new leaf(4)),
      o2 = new leaf(5),
      o  = new leaf(5)
  in let o3 = new node(o1, o2)
     in list(send o1 sum(),
             send o2 sum(),
             send o3 sum(),
             if send o1 equal(o2) then 100 else 200,
             if send o3 equal(o3) then 100 else 200)
) .
```

```
************************************
*** Defining a typed OO Language ***
************************************

--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  ops initialize self object : -> Name .
  ops c1 c2 c3 m1 m2 m3 o1 o2 o3 : -> Name .
  ops setx1 setx2 sety1 sety2 getx1 getx2 gety1 gety2 : -> Name .
  ops add getstate : -> Name .
  ops oddeven odd even : -> Name .
  ops node leaf left right sum value : -> Name .
  ops point colorpoint color initx inity initcolor move dx dy cp
      getLocation getColor setLocation setColor : -> Name .
  op setup : -> Name .
  ops tree equal getleft getright getvalue : -> Name .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  op '(') : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: () prec 100] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
  ops true false : -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod TYPE is
  protecting NAME .
  sorts BasicType Type .
  subsorts BasicType Name < Type .
  ops int bool void : -> BasicType .
  op _*'_ : Type Type -> Type [assoc prec 1] .
--- to distinguish it from _*_ which works on names as expressions
  op _->_ : Type Type -> Type [prec 2] .
  op list' : Type -> Type .
--- to distinguish it from list which works on names as expressions
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting TYPE .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
  op __=_ : Type Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PARAMETER-SYNTAX is protecting TYPE .
  sorts Param ParamList .
  subsort Param < ParamList .
  op __ : Type Name -> Param [prec 3] .
  op '(') : -> ParamList .
  op _,_ : ParamList ParamList -> ParamList [assoc id: ()] .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  protecting PARAMETER-SYNTAX .
  op proc__ : ParamList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 1] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod FIELD-SYNTAX is protecting TYPE .
  sorts Field Fields .
  subsort Field < Fields .
  op noFields : -> Fields .
  op field__ : Type Name -> Field .
  op __ : Fields Fields -> Fields [assoc comm id: noFields prec 110] .
endfm

fmod METHOD-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  protecting PARAMETER-SYNTAX .
  sorts Method Methods .
  subsort Method < Methods .
  op noMethods : -> Methods .
  op method____ : Type Name ParamList Exp -> Method [prec 105] .
  op abstract'method___ : Type Name ParamList -> Method [prec 105] .
  op __ : Methods Methods -> Methods [assoc comm id: noMethods prec 110] .
endfm

fmod CLASS-SYNTAX is
  protecting FIELD-SYNTAX .
  protecting METHOD-SYNTAX .
  sorts Class Classes ClassSpecifier ClassBody .
  subsort Class < Classes .
  subsort Methods < ClassBody .
  op noClasses : -> Classes .
  op __extends__ : ClassSpecifier Name Name ClassBody -> Class [prec 115] .
  ops class abstract'class : -> ClassSpecifier .
  op __ : Fields Methods -> ClassBody [prec 112] .
  op __ : Classes Classes -> Classes [assoc comm id: noClasses prec 120] .
endfm

fmod NEW-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op new__ : Name ExpList -> Exp .
endfm

fmod SEND-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op send___ : Exp Name ExpList -> Exp .
endfm

fmod SUPER-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op super__ : Name ExpList -> Exp .
endfm

fmod INSTANCEOF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op instanceof__ : Exp Name -> Exp [prec 1] .
endfm

fmod CAST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op cast__ : Exp Name -> Exp [prec 1] .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending CLASS-SYNTAX .
  extending NEW-SYNTAX .
  extending SEND-SYNTAX .
  extending SUPER-SYNTAX .
  extending CAST-SYNTAX .
  extending INSTANCEOF-SYNTAX .
  sort Program .
  op _main_ : Classes Exp -> Program [prec 125] .
endfm

parse(
  abstract class tree extends object
    method int initialize() 1
    abstract method int sum()
    abstract method bool equal(tree t)

  class node extends tree
    field tree left
    field tree right
    method void initialize(tree l, tree r)
      {
        set left = l ;
        set right = r
      }
    method tree getleft() left
    method tree getright() right
    method int sum() (send left sum()) + (send right sum())
    method bool equal(tree t)
      if instanceof t node
      then if send left equal(send cast t node getleft())
           then send right equal(send cast t node getright())
           else false
      else false

  class leaf extends tree
    field int value
    method void initialize(int v) set value = v
    method int sum() value
    method int getvalue() value
    method bool equal(tree t)
      if instanceof t leaf
      then zero?(value - (send cast t leaf getvalue()))
      else false

  main
    let o1 = new node(new leaf(3), new leaf(4)),
        o2 = new leaf(5),
        o  = new leaf(5)
    in let o3 = new node(o1, o2)
       in list(send o1 sum(),
               send o2 sum(),
               send o3 sum(),
               if send o1 equal(o2) then 100 else 200,
               if send o3 equal(o3) then 100 else 200)
) .


---------------------
--- Type Checking ---
---------------------

fmod TYPE-ENVIRONMENT is
  sort TEnv .
*** add your code here
endfm

fmod AUX-CLASS-OPS is extending CLASS-SYNTAX .
  extending TYPE-ENVIRONMENT .
  op superClass : Name Classes -> [Name] [memo] .
  op concreteClass : Name Classes -> [Bool] [memo] .
  op methodsOnPath : Name Classes -> [Methods] [memo] .
  op fieldTEnv : Name Classes -> [TEnv] [memo] .
  op subtypeOf : Type Type Classes -> [Bool] [memo] .
  op declType : Name Name Classes -> [Type] [memo] .
*** add your code here
endfm

fmod STATE is
  extending AUX-CLASS-OPS .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  op _[_] : State Name -> Type .
```

```
  op _[_<-_] : State Name Type -> State .
  op initState : -> State .

*** add your code here

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  var S : State .  vars Xc Xc' : Name .
  vars Cls Cls' : Classes .  vars TEnv TEnv' : TEnv .
  op classes : Classes -> StateAttribute .
  op classes : State -> Classes .
  eq (classes(Cls) S) <** classes(Cls') = classes(Cls') S .
  eq classes(classes(Cls) S) = Cls .
  op currClass : Name -> StateAttribute .
  op currClass : State -> Name .
  eq (currClass(Xc) S) <** currClass(Xc') = currClass(Xc') S .
  eq currClass(currClass(Xc) S) = Xc .
  op tenv : TEnv -> StateAttribute .
  op tenv : State -> TEnv .
  eq (tenv(TEnv) S) <** tenv(TEnv') = tenv(TEnv') S .
  eq tenv(tenv(TEnv) S) = TEnv .
endfm

fmod GENERIC-EXP-TYPE-CHECKING is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op type : ExpList State -> Type .
  var I : Int .  var X : Name .  var S : State .
  var E : Exp .  var El : ExpList .
  eq type(I, S) = int .
  eq type(X, S) = S[X] .
  eq type((), S) = void .
 ceq type((E,El), S) = type(E,S) *' type(El, S) if El =/= () .
endfm

fmod LIST-TYPE-CHECKING is extending LIST-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var El : ExpList .  var S : State .  var T : Type .
  eq type(list(E), S) = list'(type(E,S)) .
 ceq type(list(E,El), S) = list'(T)
    if T := type(E,S) /\ type(list(El),S) = list'(T) [owise] .
endfm

fmod ARITH-OPS-TYPE-CHECKING is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars E E' : Exp .  var S : State .
 ceq type(E + E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E - E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E * E', S) = int if type(E,S) = int /\ type(E',S) = int .
 ceq type(E / E', S) = int if type(E,S) = int /\ type(E',S) = int .
endfm

fmod BEXP-TYPE-CHECKING is extending BEXP-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars E E' Be Be' : Exp .  var S : State .
 ceq type(E equals E', S) = bool if type(E,S) = type(E',S) .
 ceq type(zero?(E), S) = bool if type(E, S) = int .
 ceq type(even?(E), S) = bool if type(E, S) = int .
 ceq type(not(Be), S) = bool if type(Be, S) = bool .
 ceq type(Be and Be', S) = bool if type(Be, S) = bool /\ type(Be', S) = bool .
  eq type(true, S) = bool .
  eq type(false, S) = bool .
endfm

fmod IF-TYPE-CHECKING is extending IF-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  vars Be E E' : Exp .  var S : State .
```

```
  eq type({E}, S) = type(E,S) .
 ceq type({E ; El}, S) = type({El}, S) if type(E,S) = void .
endfm

fmod LOOP-TYPE-CHECKING is extending LOOP-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Be E : Exp .  var S : State .
 ceq type(while Be E, S) = void if type(Be, S) = bool /\ type(E, S) = void .
endfm

fmod METHOD-TYPE-CHECKING is extending METHOD-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  extending GENERIC-EXP-TYPE-CHECKING .
  op type-check : Methods State -> Bool .
  var S : State .  var Ms : Methods .  var T : Type .  var Xm : Name .
  var Pl : ParamList .  var E : Exp .
  eq type-check(noMethods, S) = true .
  eq type-check((Ms abstract method T Xm(Pl)), S) = type-check(Ms, S) .
 ceq type-check((Ms method T Xm(Pl) E), S) = type-check(Ms, S)
    if subtypeOf(type(E, bindPList(Pl, S)), T, classes(S)) .
endfm

fmod CLASS-TYPE-CHECKING is extending CLASS-SYNTAX .
  protecting METHOD-TYPE-CHECKING .
  op type-check : Classes State -> Bool .
  var Cls : Classes .  vars X Xc Xc' Xm : Name .
  var Pl Pl' : ParamList .  vars E E' : Exp .
  vars T T' : Type .  var Cb : ClassBody .
  var CS : ClassSpecifier .  var Fs : Fields .
  var Ms : Methods .  var S : State .
  eq type-check((Cls class Xc extends Xc'
                   Fs Ms abstract method T Xm(Pl)), S) = false .
 ceq type-check((Cls CS Xc extends Xc' Cb), S) = false
    if Ms (method T Xm(Pl) E) (method T' Xm(Pl') E')
         := methodsOnPath(Xc, classes(S))
    /\ Xm =/= initialize
    /\ typePList(Pl) -> T =/= typePList(Pl') -> T' .
 ceq type-check((Cls CS Xc extends Xc' Cb), S) = false
    if Ms (abstract method T Xm(Pl)) (method T' Xm(Pl') E')
         := methodsOnPath(Xc, classes(S))
    /\ Xm =/= initialize
    /\ typePList(Pl) -> T =/= typePList(Pl') -> T' .
  eq type-check((Cls CS Xc extends Xc' Fs Ms), S) =
     type-check(Cls, S) and type-check(Ms,
       S <** currClass(Xc) <** tenv(fieldTEnv(Xc, classes(S)))) [owise] .
endfm

fmod NEW-TYPE-CHECKING is extending NEW-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xc : Name .  var El : ExpList .
  var S : State .  vars Tp T : Type .
 ceq type(new Xc(El), S) = Xc
    if concreteClass(Xc, classes(S))
    /\ Tp -> T := declType(initialize, Xc, classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod SEND-TYPE-CHECKING is extending SEND-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var E : Exp .  var El : ExpList .
  vars Tp T : Type .  var S : State .
 ceq type(send E Xm(El), S) = T
    if Tp -> T := declType(Xm, type(E,S), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm
```

```
 ceq type(if Be then E else E', S) = type(E, S)
    if type(Be,S) = bool /\ type(E,S) = type(E',S) .
endfm

fmod BINDING-TYPE-CHECKING is extending BINDING-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  op bindBList(_,_)in_ : BindingList State State -> State .
  vars S S' : State .  var X : Name .  var E : Exp .
  var Bl : BindingList .  vars T T' : Type .
  eq bindBList(none,S) in S' = S' .
 ceq bindBList((T X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- T'])
    if T' := type(E, S)
    /\ subtypeOf(T', T, classes(S)) .
  eq bindBList((X = E, Bl), S) in S' =
     bindBList(Bl, S) in (S'[X <- type(E,S)]) .
endfm

fmod LET-TYPE-CHECKING is extending LET-SYNTAX .
  extending BINDING-TYPE-CHECKING .
  var E : Exp .  var Bl : BindingList .  var S : State .
  eq type(let Bl in E, S) = type(E, bindBList(Bl, S) in S) .
endfm

fmod PARAMETER-TYPE-CHECKING is protecting PARAMETER-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  op typePList : ParamList -> Type .
  op bindPList : ParamList State -> State .
  var S : State .  var T : Type .  var X : Name .  var Pl : ParamList .
  eq typePList(()) = void .
  eq typePList(T X, Pl) = T *' typePList(Pl) .
  eq bindPList((), S) = S .
  eq bindPList((T X, Pl), S) = bindPList(Pl, S[X <- T]) .
endfm

fmod PROC-TYPE-CHECKING is extending PROC-SYNTAX .
  protecting PARAMETER-TYPE-CHECKING .
  var Pl : ParamList .  var E : Exp .  var El : ExpList .
  var S : State .  vars T Tp : Type .
  eq type(proc Pl E, S) = typePList(Pl) -> type(E, bindPList(Pl, S)) .
 ceq type(E El, S) = T
    if Tp -> T := type(E, S)
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod LETREC-TYPE-CHECKING is extending LETREC-SYNTAX .
  extending BINDING-TYPE-CHECKING .
  op bindBlindlyBList : BindingList State -> State .
  var S : State .  var X : Name .  var E : Exp .
  var T : Type .  var Bl : BindingList .
  eq bindBlindlyBList(none,S) = S .
  eq bindBlindlyBList((T X = E, Bl), S) =
     bindBlindlyBList(Bl, S[X <- T]) .
  eq type(letrec Bl in E, S) =
     type(E, bindBList(Bl, bindBlindlyBList(Bl,S)) in S) .
endfm

fmod VAR-ASSIGNMENT-TYPE-CHECKING is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var X : Name .  var E : Exp .  var S : State .
 ceq type(set X = E, S) = void if type(E,S) = S[X] .
endfm

fmod BLOCK-TYPE-CHECKING is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var El : ExpList; .  var S : State .
```

```
fmod SUPER-TYPE-CHECKING is extending SUPER-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var Xm : Name .  var El : ExpList .
  vars Tp T : Type .  var S : State .
  ceq type(super Xm(El), S) = T
    if Tp -> T :=
       declType(Xm, superClass(currClass(S), classes(S)), classes(S))
    /\ subtypeOf(type(El, S), Tp, classes(S)) .
endfm

fmod INSTANCEOF-TYPE-CHECKING is extending INSTANCEOF-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  vars Xc Xc' Xc'' : Name .  var S : State .
  var Cls : Classes .  var CS : ClassSpecifier .  var Cb : ClassBody .
  ceq type(instanceof E Xc, S) = bool
     if Xc' := type(E,S) /\ Cls CS Xc' extends Xc'' Cb := classes(S) .
endfm

fmod CAST-TYPE-CHECKING is extending CAST-SYNTAX .
  extending GENERIC-EXP-TYPE-CHECKING .
  var E : Exp .  var Xc : Name .  var S : State .  var T : Type .
***> add your code here
endfm

fmod PROG-LANG-TYPE-CHECKING is extending PROG-LANG-SYNTAX .
  extending LIST-TYPE-CHECKING .
  extending ARITH-OPS-TYPE-CHECKING .
  extending IF-TYPE-CHECKING .
  extending LET-TYPE-CHECKING .
  extending PROC-TYPE-CHECKING .
  extending LETREC-TYPE-CHECKING .
  extending VAR-ASSIGNMENT-TYPE-CHECKING .
  extending BLOCK-TYPE-CHECKING .
  extending LOOP-TYPE-CHECKING .
  extending NEW-TYPE-CHECKING .
  extending SEND-TYPE-CHECKING .
  extending SUPER-TYPE-CHECKING .
  extending INSTANCEOF-TYPE-CHECKING .
  extending CAST-TYPE-CHECKING .

  extending CLASS-TYPE-CHECKING .
  op type_ : Program -> [Type] .
  var Cls : Classes .  var E : Exp .  var Te : Type .  vars S Se : State .
 ceq type(Cls main E) = type(E, S)
    if S := initState <** classes(Cls)
    /\ type-check(Cls, S) .
endfm

red type(
  abstract class tree extends object
    method int initialize() 1
    abstract method int sum()
    abstract method bool equal(tree t)

  class node extends tree
    field tree left
    field tree right
    method void initialize(tree l, tree r)
      {
        set left = l ;
        set right = r
      }
    method tree getleft() left
    method tree getright() right
    method int sum() (send left sum()) + (send right sum())
    method bool equal(tree t)
```

```
      if instanceof t node
      then if send left equal(send cast t node getleft())
           then send right equal(send cast t node getright())
           else false
      else false

  class leaf extends tree
    field int value
    method void initialize(int v) set value = v
    method int sum() value
    method int getvalue() value
    method bool equal(tree t)
      if instanceof t leaf
      then zero?(value - (send cast t leaf getvalue()))
      else false

  main
    let o1 = new node(new leaf(3), new leaf(4)),
        o2 = new leaf(5),
        o  = new leaf(5)
    in let o3 = new node(o1, o2)
       in list(send o1 sum(),
               send o2 sum(),
               send o3 sum(),
               if send o1 equal(o2) then 100 else 200,
               if send o3 equal(o3) then 100 else 200)
) .
```

CS322 - Programming Language Design

Lecture 18: Continuation-based Definition of a Functional Language (Part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

All our previous definitions were based on the notion of *state*, which contained the entire *data context* needed in order to evaluate the "current" expression. When defining the meaning of an expression, we traversed it inductively, evaluated each subexpression in appropriate data contexts, collected the side effects, and eventually calculated the value of the current expression.

When evaluating an expression, we had to implicitly maintain a *control context*, to know from where to continue our execution after evaluating a subexpression. For example, the Maude definition of the semantics of multiplication was:

```
ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
   if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
```

That means that in order to evaluate `E * E'`, Maude started the evaluation of `E` in the current data context, but also had to maintain, as part of the obligations in the condition, the control context, that is, how to continue after the evaluation of `E` is done.

As a consequence of storing the control context implicitly in Maude, the program will make Maude run out of memory:

```
letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(8000)
```

The same will happen in any implementation of a programing language based on a state which only provides the data context but not the control context. If we follow the control flow of the evaluation of this expression ignoring the details, we can see that it essentially follows the following pattern, which will be the same for any data-context-only based implementations:


```
f(5) => 5 * f(4)
     => 5 * (4 * f(3))
     => 5 * (4 * (3 * f(2)))
     => 5 * (4 * (3 * (2 * f(1))))
     => 5 * (4 * (3 * (2 * (1 * f(0)))))
     => 5 * (4 * (3 * (2 * (1 * 1))))
     => 5 * (4 * (3 * (2 * 1)))
     => 5 * (4 * (3 * 2))
     => 5 * (4 * 6)
     => 5 * 24
     => 120
```

Compare this execution with the execution of the following "iterative" version of the factorial function, which evaluates without any problem in about *4 seconds* to a number of *47,838* digits:

```
letrec f = proc(n,m)
             if zero?(n)
             then m
             else f(n - 1, m * n)
in f(13000, 1)
```

For these experiments I have used a slightly modified version of the executable semantics of the functional language, provided in the file new-funct-lang2.maude. This is the version which we extended with objects, so boolean expressions are considered just ordinary expressions. If we follow the control flow of the evaluation of the expression above, ignoring again the details, we can see the following pattern:

```
f(5,1) => f(4,5) => f(3,20) => f(2,60) => f(1,120) => f(0,120) => 120
```


Control Behavior

The second definition of factorial was faster because *no control context had to be recorded* implicitly (or internally) by Maude. This happened because each time the value needed to be returned by the function was equal to the value returned by its recursive call. One can distinguish two types of control behaviors:

- *Recursive control behavior*. When additional control information must be recorded with each recursive call, and this information must be retained until the call returns.
- *Iterative control behavior*. When only a bounded amount of memory is needed for control information.

Note that it may be possible that programs run out of memory even under iterative control behavior. E.g., a recursive function may need additional space to store local names, bound with `let` or

`letrec`. However, note that this is *data context* information and has nothing to do with the control context information.

In fact, the iterative version of the factorial function will also run relatively quickly out of memory (for `n=20,000` on a 4GB PC) in our current Maude semantics. Why?

Homework Exercise 1 Extra credit (max 10 points). *Define a garbage collector for our functional language. Discuss with me first if you want to do it.*

The first version of the factorial required Maude to record control information because the recursive function is called *in an operand position*, so after the evaluation of recursive call, we still have to finish the local computation (evaluating the other operands and execute the outer calls).

In the next few lectures we will learn how to handle control contexts in a similar manner as we handled data contexts. The

underlying guiding principle will be following:

> **Calling of functions does not create control context, but only the evaluation of the their arguments.**

As we used environments to record data contexts, we will use *continuations* to record control contexts.

We will define a Maude executable semantics of the functional language discussed so far which passes a continuation as part of its state, then we will add to the language several control context sensitive features, such as *exceptions* and *concurrency*.

Continuation-based Executable Semantics

Our goal next will be define an executable semantics of the simple functional language which, unlike our previous semantics, e.g.,

```
...
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
...
```

will *not* require Maude to record implicitly/internally any information regarding the control context during the evaluation of an expression. In order to do this, we will need to ensure that, at any moment, the state in which an expression is evaluated *contains all the information needed to complete the entire computation*.

Suppose that we want to evaluate the expression `x * (y + z)` in a state in which `x`, `y`, and `z` are bound to `3`, `4`, and `5`, respectively. From the perspective of a *continuation based semantics*, it is not correct now to evaluate `x`, `y` and `z` and then evaluate the expression.

Instead, one must choose one of the two top subexpressions, say `x`, as the "current expression" to be evaluated next, and store the remaining *computation obligations* somewhere in the state, more precisely in its *continuation component*.

It is important to note that the evaluation of an expression may modify the environment. It is therefore crucial to also *freeze the current environment* in the continuations whenever it is needed to be recovered in order to evaluate the remaining expressions.

Arithmetic Operations Example

Keeping the notation informal, this intuition corresponds to reducing the evaluation of `x * (y + z)` in state `S` to the evaluation of `x` only, but in state `S` whose continuation component, say `C`, is modified to (let `Env` be the environment of `S`):

```
[_ | (y + z) | Env] -> * -> C
```

with the following meaning: once `x` is evaluated, pass its value to the continuation (the underscore), then pick the next expression to evaluate, which is `y + z`; once that is available then, with the two values available, continue to the next computation obligation, the multiplication (which expects a pair of values; that's the reason for which we grouped them within the square brackets), and so on.

Thus, the next step is to evaluate in `Env` the expression `y + z` in a state whose continuation is

```
[3,_ | () | Env] -> * -> C
```

By applying the same principle, choose `y` to evaluate next, and record the appropriate continuation:

```
[_ | z | Env] -> + -> [3,_ | () | Env] -> * -> C
```

`y` evaluates to `4`, so the next step is to evaluate `z` in a state whose continuation is:

```
[4,_ | () | Env] -> + -> [3,_ | () | Env] -> * ->
C
```

Further, `z` evaluates to 5, so the continuation becomes:

```
[4,5 | () | Env] -> + -> [3,_ | () | Env] -> * ->
C
```

Once a list of expressions is completely evaluated and its values are

all in the continuation, the continuation itself can be then evaluated, by applying the immediately next operator (or closure) in the continuation data structure. In this case that operator is `+` and we get:

```
9 -> [3,_ | () | Env] -> * -> C
```

and further

```
[3,9 | () |Env] -> * -> C
```

which evaluates to

```
27 -> C.
```

The rest of the computation will be driven by whatever control context `C` contains, which is in fact nothing but the context in which the original expression `x * (y + z)` has been called.

Conditionals

In the case of conditionals, one should choose the condition as the current expression and store the rest in the continuation structure for future processing. Thus, evaluating `if b then x else y + x` in a state where `b` is bound to `false`, `x` to `1` and `y` to `2`, and whose continuation is `C` and environment `Env`, reduces to evaluating `b` in a state whose continuation is

```
if(x, y + x, Env) -> C.
```

Once `b` is evaluated to `false`, the continuation will become

```
bool(false) -> if(x, y + x, Env) -> C.
```

which will further be reduced to evaluating the expression `y + x` in a state whose environment will be set to `Env`, though not really needed in our simple case because evaluating `b` does not modify the environment, and whose continuation will be simply `C`.

The Factorial Example

Let us consider evaluating the factorial expression in a state where `x` and `y` are bound to `1` and `3`, respectively, using a continuation-based semantics.

```
letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(x+y)
```

We only focus on the continuation specific steps, ignoring the environment specific ones. After processing the `let` construct, which we do not show here but will define formally in the next lecture, the goal is to evaluate `f(x+y)` in a state which contains the recursive definition of `f` and some current continuation, `C`.

Then we pick the list of subexpressions involved, `f, x+y`, and

evaluate it in the continuation:

`fn -> C`,

where `fn` is a continuation entry constructor saying that the next list of values is expected to be a function and its evaluated arguments (we will therefore define the evaluation of lists, also in a continuation based style). After a few steps, this reduces to evaluating the continuation

`[closure(n, <body>, Env), 4] -> fn -> C`,

which further reduces to evaluating the `<body>` of the factorial function in a state whose environment is `Env[n <- 4]` and whose continuation is ... just `C`. By applying the continuation-based meaning of the conditional, this further reduces to evaluating `n * f(n - 1)`, which further (in a few steps) reduces to evaluating

`[closure(n, <body>, Env), 3] -> fn -> [4,_ | () | Env] -> * -> C`.

Continuing this procedure, we eventually get to the continuation

```
1 -> [1,_ | () | Env] -> * -> [2,_ | () | Env] ->
* -> [3,_ | () | Env] -> * -> [4,_ | () | Env] ->
* -> C,
```

which can now iteratively be evaluated to

```
6 -> [4,_ | () | Env] -> * -> C,
```

further to `[4,6 | () | Env] -> * -> C` and then to `24 -> C`.

Therefore, the *explicit* continuation grows now instead of the *implicit* control flow maintained by Maude, so this executable semantics also suffers from running out of memory. Nothing can fix badly written programs, like the one above. There are, however, techniques to *transform* badly written programs into better ones, which we will investigate soon. Note though that the state is not saved as part of this explicit control context, as it used to be in the `Maude` implicit one.

However, if we consider the other definition of factorial, then the continuation will be bounded in size. The evaluation of this program will unnecessary quickly run out of memory anyway though, because of the lack of garbage collecting.

```
**************************************************
*** A New Definition of a Funtional Language ***
**************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
```

```
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
```

```
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op `[`] : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: `[`]] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
```

```
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
    (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
     nextLoc(N + 1) S)[Xl <- Vl] .
```

```
  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .
  eq eval(X, S) = {S[X], S} .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
```

```
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' Be Be' : Exp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' Be : Exp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
```

```
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
```

```
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E Be : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm



red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
```

```
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
```

```
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping
```

```
red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
```

```
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
```

```
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
```

```
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
              let d = set c = c + 1
              in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
              let d = set x = x + 1
              in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
              let t = x
              in let d = set x = y
                 in let d = set y = t
                    in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
```

```
      'even = proc() if zero?(x)
                then 1
                else let d = set x = x - 1
                     in 'odd(),
      'odd  = proc() if zero?(x)
                then 0
                else let d = set x = x - 1
                     in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                  else let d = set x = x - 1
                       in 'odd(),
         'odd = proc() if zero?(x) then 0
                  else let d = set x = x - 1
                       in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24
```

```
red eval(
  let 'times4 = 0
  in {
      set 'times4 = proc(x)
                    if zero?(x)
                    then 0
                    else 4 + 'times4(x - 1) ;
      'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
```

```
      y
    }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
```

```
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                y = 2,
                a = 3,
                z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(13000, 1)

) .

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(8000)
) .
```

```
**********************************************
*** A New Definition of a Funtional Language ***
**********************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


------------------
--- Semantics ---
------------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op '[') : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: '[']] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S){S{X}] .

  eq S[() <- []] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* []] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod NAME-SEMANTICS is extending NAME-SYNTAX .
  protecting STATE .
  op  eval : Name State -> ValueStatePair .
  var X : Name .  var S : State .
  eq eval(X, S) = {S[X], S} .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending NAME-SEMANTICS .
  op int : Int -> Value .
  op  eval : Exp State -> ValueStatePair .
  op  eval : ExpList State -> ValueListStatePair .
  op  eval : Exp -> Value .
  var I : Int .  vars S Se Sl : State .  vars E E' : Exp .  var El : ExpList .
  var Ve : Value .  var Vl : ValueList .
  eq  eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {[],S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E / E', S) = {int(Ie quo Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' Be Be' : Exp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' Be : Exp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm
```

```
fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  vars S Se : State .  var Ve : Value .
 ceq eval(set X = E, S) = {int(1), Se[X <* Ve]}
    if {Ve,Se} := eval(E,S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var E : Exp .  var El : ExpList; .  vars S Se : State .  var Ve : Value .
  eq eval({E}, S) = eval(E,S) .
 ceq eval({E ; El}, S) = eval({El}, Se) if {Ve,Se} := eval(E,S) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E Be : Exp .  vars S Sb Se : State .  var Ve : Value .
 ceq eval(while Be E, S) = eval(while Be E, Se)
    if {bool(true), Sb} := eval(Be,S) /\ {Ve,Se} := eval(E,Sb) .
 ceq eval(while Be E, S) = {int(1), Sb}
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm
```

```
red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
```

```
    in let x = 20
       in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(13000, 1)

) .

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(8000)
) .
```

CS322 - Programming Language Design

Lecture 19: Continuation-based Definition of a Functional Language (Part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

This lecture we define formally the continuation-based semantics of our functional programming language. As usual, there will be an abstract notion of state, which will contain all the required information to evaluate an expression. However, this time, the state will contain information regarding both the *data context* and the *control context*.

This lecture we try a different style of defining the semantics of a language, in which all the information needed is in the state, including the expression to be evaluated! Therefore, one can see our next definitions as defining a *state transition system*, telling how the next state is generated.

In what follows, only *part of the definition* will be provided, with explanations on how the remaining part should be defined.

Homework Exercise 1 *Complete the continuation-based definition of the language discussed in this lecture. Edit the provided file,* `continuations-semantics.maude`*. Being able to run all the provided examples is a good sign that your definition is correct.*

The state ingredients needed to define the semantics include as usual *locations*, *environments*, *values*, and *stores*. In addition to these, we will add *continuations* in order to maintain the control context.

Locations, Values, the Environment and the Store

Since the current language is *call-by-value*, we can simplify several definitions later on by manipulating lists of expressions, values and locations, respectively. The modules `LOCATION`, `ENVIRONMENT`, `VALUE` and `STORE` are modified to reflect that. Operators are therefore defined as follows:

```
  op _[_<-_] : Env NameList LocationList -> Env .
  op _[_<-_] : Store LocationList ValueList -> Store .
...
```

You do not need to redefine these.

Continuations

In this class we define semantics of languages or analysis tools *modularly*, meaning that we introduce the language features one by one, together with the additional particular infrastructure (as auxiliary operations).

The module `CONTINUATIONS` defines the sort `Continuation`, together with the syntax of some common operations on continuations. The meaning of these operations will be defined later, in the module `EVAL`, when all the needed state infrastructure is available.


```
fmod CONTINUATION is
  extending VALUE .
  extending ENVIRONMENT .
  extending GENERIC-EXP-SYNTAX .
  sort Continuation .
  op stop : -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op _->_ : LocationList Continuation -> Continuation .
  op [_|_] ->_ : Exp Env Continuation -> Continuation .
endfm
```

The intended meaning is as follows:

- `stop : -> Continuation` is the initial state of the continuation data structure. Whenever a value is "sent" to it (see the next operation), it means that the evaluation is finished and that value will be output.
- `op _->_ : ValueList Continuation -> Continuation` "tells" the continuation that a list of values is available to be further processed; this is typically the result of evaluating a list of expressions.
- `op _->_ : LocationList Continuation -> Continuation` "tells" the continuation that the next values must be stored at the specified locations.
- `op [_|_] ->_ :  Exp Env Continuation -> Continuation` says that the evaluation process should continue by evaluating the given expression in the given environment.

The State Infrastructure

As before, a state is a set of state attributes.

```
fmod STATE is extending (see the file)
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
```

The state will now dynamically contain *4 or 5 attributes*, depending upon whether a list of expressions is to be evaluated or a continuation. The 5 attributes are the following:

```
  op nextLoc : Nat -> StateAttribute .
  op env : Env -> StateAttribute .
  op store : Store -> StateAttribute .
  op exps_ : ExpList -> StateAttribute .
  op cont : Continuation -> StateAttribute .
```

- `nextLoc` gives the next available location in the store.
- `env` gives the current environment.
- `store` keeps the current store.
- `exps`, when occurs in a state, it keeps a list of expressions to be evaluated. Their evaluation is of course performed in a continuation style, and their values will be passed to the continuation to distribute them properly.
- `cont` gives the current continuation structure; we will add constructors for continuations later, on a feature by feature basis.

The *initial state* can be defined now:

```
op initState : -> State .
eq initState = nextLoc(0) env(noEnv) store(noStore) cont(stop) .
```


The Main Two Evaluation Operators

We first declare the two evaluation operators needed, one for evaluating lists of expressions and the other for evaluating, or applying, continuations.

```
fmod EVAL is extending STATE .
  op evalExps : State -> Value .
  op evalCont : State -> Value .
  op eval : Exp -> Value .
```

A unary `eval` is also declared as usual, and defined as follows:

```
  eq eval(E) = evalExps(exps(E) initState) .
```

The main operators `evalExps` and `evalCont` will be rigorously defined defined by each feature added to the language. However, we can now define the meaning of the global continuation constructors:

```
  eq eval(E) = evalExps(exps(E) initState) .
  eq evalCont(cont(V -> stop) S) = V .
  eq evalCont(cont(Vl -> Ll -> C) store(St) S) =
     evalCont(cont(C) store(St[Ll <- Vl]) S) .
  eq evalCont(cont([E | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont(C) env(Env) S) .
endfm
```

Note that, like it will also be the case in the rest of the definition, the definition of `evalCont` is triggered by the `cont(...)` attribute, while that of `evalExps` by the `exps(...)` attribute, which can store a list of expressions. *AC matching* will be intensively speculated in today's language definition.

Continuation-based Semantics of Generic Expressions

The continuation based meaning of an expression is as follows:

> Evaluate the expression to a value, and then pass that value to the continuation for further processing.

The principle extends to lists of expressions.

As in the previous semantics, we need to add a value constructor for integer values, called `int` as before. To keep the code shorter, we merge the semantics of names within the semantics of generic expressions:

```
fmod GENERIC-EXP-SEMANTICS is extending (see file)
  op int : Int -> Value .
  eq evalExps(exps(I) cont(C) S) = evalCont(cont(int(I) -> C) S) .
  eq evalExps(exps(X) cont(C) env([X,L] Env) store([L,V] St) S) =
     evalCont(cont(V -> C) env([X,L] Env) store([L,V] St) S) .
```

An auxiliary operator is needed in order to handle lists of expressions. This is because of the continuation style principle, enforcing us to evaluate the expressions one by one. Note that the evaluation of an expression can change the environment and that all the expressions need to be evaluated in the same original environment, so that needs to be frozen:

```
  op [_|_|_] ->_ : ValueList ExpList Env Continuation -> Continuation .
  eq evalExps(exps() S) = evalCont(S) .
 ceq evalExps(exps(E,El) cont(C) env(Env) S) =
     evalExps(exps(E) cont([noValues | El | Env] -> C) env(Env) S)
    if El =/= () .
  eq evalCont(cont(V -> [Vl | () | Env] -> C) S) =
     evalCont(cont((Vl,V) -> C) S) .
  eq evalCont(cont(V -> [Vl | E,El | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont([Vl,V | El | Env] -> C) env(Env) S) .
```

The list of values is then passed to the continuation.

Continuation-based Semantics of Arithmetic Operators

The following is self explanatory. The argument expressions are evaluated, after telling the continuation how to interpret the next list of values: "`+ -> C`". Note that a corresponding semantics for the continuation needs to also be provided:

```
fmod ARITH-OPS-SEMANTICS is extending (see file)
  op +'->_ : Continuation -> Continuation .
  eq evalExps(exps(E + E') cont(C) S) =
     evalExps(exps((E,E')) cont(+ -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> + -> C) S) =
     evalCont(cont(int(I + J) -> C) S) .
---> define subtraction, multiplication, division
endfm
```

Complete the module.

Continuation-based Semantics of Boolean Operators

The next is very similar to the continuation-based semantics of arithmetic operators:

```
fmod BEXP-SEMANTICS is extending (see file)
  op bool : Bool -> Value .
  op equals‘->_ : Continuation -> Continuation .
  vars E E’ Be Be’ : Exp .  var S : State .  vars I J : Int .
  var C : Continuation .  vars B B’ : Bool .
  eq evalExps(exps(E equals E’) cont(C) S) =
     evalExps(exps((E,E’)) cont(equals -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> equals -> C) S) =
     evalCont(cont(bool(I == J) -> C) S) .
---> define zero?, even?, not, and
endfm
```

Complete this module.

Continuation-based Semantics of Conditionals

One must *not* evaluate both the branching subexpressions of a conditional, so we need to pass them both to the continuation to choose the appropriate one after the condition is evaluated. They must be evaluated in the same environment as the conditional, so that also needs to be passed:

```
fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  op if(_,_,_) ->_ : Exp Exp Env Continuation -> Continuation .
  vars Be E E’ : Exp .  var C : Continuation .
  var S : State .  vars Env Env’ : Env .  var B : Bool .
  eq evalExps(exps(if Be then E else E’) cont(C) env(Env) S) =
     evalExps(exps(Be) cont(if(E,E’,Env) -> C) env(Env) S) .
  eq evalCont(cont(bool(B) -> if(E,E’,Env) -> C) env(Env’) S) =
     evalExps(exps(if B then E else E’ fi) cont(C) env(Env) S) .
endfm
```

Continuation-based Semantics of Let

The semantics of let can be very elegantly given as follows:

```
fmod LET-SEMANTICS is extending (see file)
 ceq evalExps(exps(let Bl in E) nextLoc(N) env(Env) cont(C) S) =
     evalExps(exps(El) nextLoc(N + M) env(Env)
              cont(Ll -> [E | Env[Xl <- Ll]] -> C) S)
    if Xl := names(Bl) /\ El := expressions(Bl)
    /\ M := #(Xl) /\ Ll := locs(N, M) .
endfm
```

So as many locations as bound names are first generated in Ll, then the continuation is told to evaluate the body of the let in the proper environment, after first placing the values of the binding expressions into the corresponding locations.

Continuation-based Semantics for Procedures

We first need to add *closures* as values and to define a corresponding continuation construct for function invocations:

```
fmod PROC-SEMANTICS is (see file)
  op closure : NameList Exp Env -> Value .
  op fn`->_ : Continuation -> Continuation .
```

A procedure expression is simply evaluated to its closure:

```
  eq evalExps(exps(proc(Xl) E) cont(C) env(Env) S) =
     evalCont(cont(closure(Xl,E,Env) -> C) env(Env) S) .
```

Function application is a bit trickier (see also let):

```
  eq evalExps(exps(F(El)) cont(C) S) =
     evalExps(exps((F,El)) cont(fn -> C) S) .
--->  add a corresponding evalCont for the above equations
endfm
```

Continuation-based Semantics of Assignment

Among the remaining features, we only discuss variable assignment. You should define the continuation-based semantics of `letrec`, blocks and loops as part of HW5.

```
fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var C : Continuation .  var S : State .  var L : Location .
  eq evalExps(exps(set X = E) env([X,L] Env) cont(C) S) =
     evalExps(exps(E) cont(L -> int(1) -> C) env([X,L] Env) S) .
endfm
```

So the continuation is first told to return 1 as the result of the assignment, but before that to place the result of evaluating `E` into the location `L` associated to `X` in the environment.

```
*****************************************************************
*** Continuation Based Semantics of a Funtional Language ***
*****************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
```

```
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
```

```
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLocations : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList
           [assoc id: noLocations] .
  op loc : Nat -> Location .
  op locs : Nat Nat -> LocationList .
  vars N M : Nat .
  eq locs(N,0) = noLocations .
  eq locs(N,M) = loc(N), locs(N + 1, M - 1) .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLocations] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
```

```
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq St[noLocations <- noValues] = St .
  eq ([L,V] St)[L,Ll <- V',Vl] = ([L,V'] St)[Ll <- Vl] .
  eq St[L,Ll <- V',Vl] = (St [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  extending VALUE .
  extending ENVIRONMENT .
  extending GENERIC-EXP-SYNTAX .
  sort Continuation .
  op stop : -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op _->_ : LocationList Continuation -> Continuation .
  op [_|_] ->_ : Exp Env Continuation -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  extending GENERIC-EXP-SYNTAX .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op initState : -> State .
  eq initState = nextLoc(0) env(noEnv) store(noStore) cont(stop) .
  op nextLoc : Nat -> StateAttribute .
  op env : Env -> StateAttribute .
```

```
  op store : Store -> StateAttribute .
  op exps_ : ExpList -> StateAttribute .
  op cont : Continuation -> StateAttribute .
endfm

fmod EVAL is extending STATE .
  op evalExps : State -> Value .
  op evalCont : State -> Value .
  op eval : Exp -> Value .
  var E : Exp .  var Vl : ValueList .  var St : Store .
  var S : State .  var C : Continuation .  var Ll : LocationList .
  var Env Env' : Env .  var V : Value .
  eq eval(E) = evalExps(exps(E) initState) .
  eq evalCont(cont(V -> stop) S) = V .
  eq evalCont(cont(Vl -> Ll -> C) store(St) S) =
     evalCont(cont(C) store(St[Ll <- Vl]) S) .
  eq evalCont(cont([E | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont(C) env(Env) S) .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending EVAL .
  op [_|_|_] ->_ : ValueList ExpList Env Continuation -> Continuation .
  op int : Int -> Value .
  op #_ : ExpList -> Nat .
  var E : Exp .  var El : ExpList .  var I : Int .
  var C : Continuation .  var S : State .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  var X : Name .
  var L : Location .  var St : Store .
  eq #() = 0 .
  eq #(E,El) = 1 + #(El) .
  eq evalExps(exps(I) cont(C) S) = evalCont(cont(int(I) -> C) S) .
  eq evalExps(exps(X) cont(C) env([X,L] Env) store([L,V] St) S) =
     evalCont(cont(V -> C) env([X,L] Env) store([L,V] St) S) .
  eq evalExps(exps() S) = evalCont(S) .
 ceq evalExps(exps(E,El) cont(C) env(Env) S) =
     evalExps(exps(E) cont([noValues | El | Env] -> C) env(Env) S)
    if El =/= () .
  eq evalCont(cont(V -> [Vl | () | Env] -> C) S) =
     evalCont(cont((Vl,V) -> C) S) .
  eq evalCont(cont(V -> [Vl | E,El | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont([Vl,V | El | Env] -> C) env(Env) S) .
endfm
```

```
fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .
  var C : Continuation .  vars I J : Int .
  op +`->_ : Continuation -> Continuation .
  eq evalExps(exps(E + E') cont(C) S) =
     evalExps(exps((E,E')) cont(+ -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> + -> C) S) =
     evalCont(cont(int(I + J) -> C) S) .
---> define subtraction, multiplication, division
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  op equals`->_ : Continuation -> Continuation .
  vars E E' Be Be' : Exp .  var S : State .  vars I J : Int .
  var C : Continuation .  vars B B' : Bool .
  eq evalExps(exps(E equals E') cont(C) S) =
     evalExps(exps((E,E')) cont(equals -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> equals -> C) S) =
     evalCont(cont(bool(I == J) -> C) S) .
---> define zero?, even?, not, and
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  op if(_,_,_) ->_ : Exp Exp Env Continuation -> Continuation .
  vars Be E E' : Exp .  var C : Continuation .
  var S : State .  vars Env Env' : Env .  var B : Bool .
  eq evalExps(exps(if Be then E else E') cont(C) env(Env) S) =
     evalExps(exps(Be) cont(if(E,E',Env) -> C) env(Env) S) .
  eq evalCont(cont(bool(B) -> if(E,E',Env) -> C) env(Env') S) =
     evalExps(exps(if B then E else E' fi) cont(C) env(Env) S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op expressions_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
```

```
  eq expressions(X = E, Bl) = E, expressions(Bl) .
  eq expressions(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
 ceq evalExps(exps(let Bl in E) nextLoc(N) env(Env) cont(C) S) =
     evalExps(exps(El) nextLoc(N + M) env(Env)
              cont(Ll -> [E | Env[Xl <- Ll]] -> C) S)
    if Xl := names(Bl) /\ El := expressions(Bl)
    /\ M := #(Xl) /\ Ll := locs(N, M) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  op fn`->_ : Continuation -> Continuation .
  vars Xl : NameList .  vars E F : Exp .  var C : Continuation .
  vars Env Env' : Env .  var S : State .  var El : ExpList .
  var Vl : ValueList .  var St : Store .  vars N M : Nat .
  var Ll : LocationList .
  eq evalExps(exps(proc(Xl) E) cont(C) env(Env) S) =
     evalCont(cont(closure(Xl,E,Env) -> C) env(Env) S) .
  eq evalExps(exps(F(El)) cont(C) S) =
     evalExps(exps((F,El)) cont(fn -> C) S) .
--->  add a corresponding evalCont for the above equation
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
---> to be defined ... (hint: look at let)
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var C : Continuation .  var S : State .  var L : Location .
  eq evalExps(exps(set X = E) env([X,L] Env) cont(C) S) =
     evalExps(exps(E) cont(L -> int(1) -> C) env([X,L] Env) S) .
endfm
```

```
fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
---> to be defined
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
---> to be defined
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm

--- if the following evaluate properly, then your definition is most
--- likely correct.

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2
```

```
red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6
```

```
red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping
```

```
red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
          if zero?(n)
          then 1
          else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
```

```
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                if zero?(x)
                then 0
                else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined
```

```
red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
             let d = set c = c + 1
             in c
  in f() + f()
) .
```

```
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
             let d = set x = x + 1
             in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
             let t = x
             in let d = set x = y
                in let d = set y = t
                   in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
          if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
```

```
      in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
         'odd = proc() if zero?(x) then 0
                       else let d = set x = x - 1
                            in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1) ;
       'times4(3)
     }
```

```
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
```

```
        if even?(n)
        then set n = n / 2
        else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

--------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
                if zero?(x) then 0
                else e(x - 1, o, e),
      'even = proc(x, o, e)
                if zero?(x) then 1
                else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined
```

```
red eval( letrec f = proc(x) z + x + 5,
              y = 2,
              a = 3,
              z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

-------------

red eval(
  letrec f = proc(n,m)
             if zero?(n)
             then m
             else f(n - 1, m * n)
  in f(100, 1)

) .

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(100)
) .
```

```
************************************************************
*** Continuation Based Semantics of a Funtional Language ***
************************************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


------------------
--- Semantics ---
------------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLocations : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList
           [assoc id: noLocations] .
  op loc : Nat -> Location .
  op locs : Nat Nat -> LocationList .
  vars N M : Nat .
  eq locs(N,0) = noLocations .
  eq locs(N,M) = loc(N), locs(N + 1, M - 1) .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLocations] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq St[noLocations <- noValues] = St .
  eq ([L,V] St)[L,Ll <- V',Vl] = ([L,V'] St)[Ll <- Vl] .
  eq St[L,Ll <- V',Vl] = (St [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  extending VALUE .
  extending ENVIRONMENT .
  extending GENERIC-EXP-SYNTAX .
  sort Continuation .
  op stop : -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op _->_ : LocationList Continuation -> Continuation .
  op [_|_] ->_ : Exp Env Continuation -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  extending GENERIC-EXP-SYNTAX .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op initState : -> State .
  eq initState = nextLoc(0) env(noEnv) store(noStore) cont(stop) .
  op nextLoc : Nat -> StateAttribute .
  op env : Env -> StateAttribute .
  op store : Store -> StateAttribute .
  op exps_ : ExpList -> StateAttribute .
  op cont : Continuation -> StateAttribute .
endfm

fmod EVAL is extending STATE .
  op evalExps : State -> Value .
  op evalCont : State -> Value .
  op eval : Exp -> Value .
  var E : Exp .  var Vl : ValueList .  var St : Store .
  var S : State .  var C : Continuation .  var Ll : LocationList .
  var Env Env' : Env .  var V : Value .
  eq eval(E) = evalExps(exps(E) initState) .
  eq evalCont(cont(V -> stop) S) = V .
  eq evalCont(cont(Vl -> Ll -> C) store(St) S) =
     evalCont(cont(C) store(St[Ll <- Vl]) S) .
  eq evalCont(cont([E | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont(C) env(Env) S) .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending EVAL .
  op [_|_|_] ->_ : ValueList ExpList Env Continuation -> Continuation .
  op int : Int -> Value .
  op #_ : ExpList -> Nat .
  var E : Exp .  var El : ExpList .  var I : Int .
  var C : Continuation .  var S : State .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  var X : Name .
  var L : Location .  var St : Store .
  eq #() = 0 .
  eq #(E,El) = 1 + #(El) .
  eq evalExps(exps(I) cont(C) S) = evalCont(cont(int(I) -> C) S) .
  eq evalExps(exps(X) cont(C) env([X,L] Env) store([L,V] St) S) =
     evalCont(cont(V -> C) env([X,L] Env) store([L,V] St) S) .
  eq evalExps(exps() S) = evalCont(S) .
 ceq evalExps(exps(E,El) cont(C) env(Env) S) =
     evalExps(exps(E) cont([noValues | El | Env] -> C) env(Env) S)
   if El =/= () .
  eq evalCont(cont(V -> [Vl | () | Env] -> C) S) =
     evalCont(cont((Vl,V) -> C) S) .
  eq evalCont(cont(V -> [Vl | E,El | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont([Vl,V | El | Env] -> C) env(Env) S) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var S : State .
  var C : Continuation .  vars I J : Int .
  op +'->_ : Continuation -> Continuation .
  eq evalExps(exps(E + E') cont(C) S) =
     evalExps(exps((E,E')) cont(+ -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> + -> C) S) =
     evalCont(cont(int(I + J) -> C) S) .
---> define subtraction, multiplication, division
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  op equals'->_ : Continuation -> Continuation .
  vars E E' Be Be' : Exp .  var S : State .  vars I J : Int .
  var C : Continuation .  vars B B' : Bool .
  eq evalExps(exps(E equals E') cont(C) S) =
     evalExps(exps((E,E')) cont(equals -> C) S) .
  eq evalCont(cont((int(I),int(J)) -> equals -> C) S) =
     evalCont(cont(bool(I == J) -> C) S) .
---> define zero?, even?, not, and
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  op if(_,_,_) ->_ : Exp Exp Env Continuation -> Continuation .
  vars Be E E' : Exp .  var C : Continuation .
  var S : State .  vars Env Env' : Env .  var B : Bool .
  eq evalExps(exps(if Be then E else E') cont(C) env(Env) S) =
     evalExps(exps(Be) cont(if(E,E',Env) -> C) env(Env) S) .
  eq evalCont(cont(bool(B) -> if(E,E',Env) -> C) env(Env') S) =
     evalExps(exps(if B then E else E' fi) cont(C) env(Env) S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op expressions_ : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq expressions(X = E, Bl) = E, expressions(Bl) .
  eq expressions(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
```

```
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
 ceq evalExps(exps(let Bl in E) nextLoc(N) env(Env) cont(C) S) =
     evalExps(exps(El) nextLoc(N + M) env(Env)
              cont(Ll -> [E | Env[Xl <- Ll]] -> C) S)
    if Xl := names(Bl) /\ El := expressions(Bl)
    /\ M := #(Xl) /\ Ll := locs(N, M) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  op fn'->_ : Continuation -> Continuation .
  vars Xl : NameList .  vars E F : Exp .  var C : Continuation .
  vars Env Env' : Env .  var S : State .  var El : ExpList .
  var Vl : ValueList .  var St : Store .  vars N M : Nat .
  var Ll : LocationList .
  eq evalExps(exps(proc(Xl) E) cont(C) env(Env) S) =
     evalCont(cont(closure(Xl,E,Env) -> C) env(Env) S) .
  eq evalExps(exps(F(El)) cont(C) S) =
     evalExps(exps((F,El)) cont(fn -> C) S) .
--->  add a corresponding evalCont for the above equation
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
---> to be defined ... (hint: look at let)
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var C : Continuation .  var S : State .  var L : Location .
  eq evalExps(exps(set X = E) env([X,L] Env) cont(C) S) =
     evalExps(exps(E) cont(L -> int(1) -> C) env([X,L] Env) S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
---> to be defined
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
---> to be defined
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
endfm

--- if the following evaluate properly, then your definition is most
--- likely correct.

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
```

```
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
```

```
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
```

```
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
```

```
              in c
    in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
    in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
    in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
    in let d = f(x,y)
       in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
```

```
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
                if zero?(x) then 0
                else e(x - 1, o, e),
      'even = proc(x, o, e)
                if zero?(x) then 1
                else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(100, 1)

) .

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(100)
) .
```

CS322 - Programming Language Design

Lecture 20: Exceptions. Continuation-Passing Style (CPS) Transformation (Part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Exceptions

We next show that *exceptions* can be very elegantly and easily handled in the presence of *continuations*. This is because continuations are structures to encode the *control context* of a program, and exceptions represent a control-related feature of a programming language.

We consider a slightly modified version of the continuation-based semantics in `continuations-semantics.maude`, namely one in which *continuation items* are stacked generically on top of an existing continuation. The reason for this slight change is that we want to *discard continuation items when an exception is thrown*. The new version of continuation-based semantics, together with part of the definition of exceptions, is given in file `exceptions.maude`.

Homework Exercise 1 *Complete the definition of the language in* `exceptions.maude`*. First add definitions for the features that you were supposed to define as part of the homework exercise in Lecture 19 (this must be trivial and gives you a chance to familiarize yourself with the new language definition), and then add the* definition of exceptions*. Also, enrich your previous continuation-based definition of division to* throw an exception *(-1) when division by zero is encountered.*

We next discuss exceptions and how they can be integrated in a language whose semantics is based on continuations. The syntax for exceptions adopted in what follows is:

```
fmod EXCEPTIONS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op try_catch_ : Exp Exp -> Exp .
  op throw_ : Exp -> Exp .
endfm
```


`try <Exp1> catch <Exp2>` and `throw(<Exp>)` work as follows:

- First evaluates `<Exp2>`; this is expected to evaluate to a closure denoting a function taking one argument;
- Then evaluates `<Exp1>`. There are two cases now:
  - If `<Exp1>` evaluates normally to a value `V`, then the result of `throw <Exp1> catch <Exp2>` is simply `V`, that is, the closure associated to the second expressions is discarded;
  - If the evaluation of `<Exp1>` encounters a `throw(<Exp>)` expression, then the expression `<Exp>` is evaluated in its environment and its value is passed as an *argument* to the closure denoting `<Exp2>`. The result value of this function invocation is the result of `throw <Exp1> catch <Exp2>`.

Let us consider the following inefficient program calculating the position at which an element occurs the first time in a list, or -1 in case the element is not in the list:

```
letrec f = proc(n,l)
             if null?(l) then -1
             else if n equals car(l)
                  then 1
                  else let p = f(n, cdr(l))
                       in if p equals -1 then -1  <--- inefficient
                          else p + 1
in f(5, list(2, 6, 7, 2, 6, 5, 7))
```

This is inefficient because of the test `p equals -1` which needs to be performed at every recursive call.

In a language with exceptions, one can throw an exception when the end of the list is reached, and then the computation will be recovered from the place where the function was invoked, within the *correct data and control contexts*:

```
let f = proc(n,l)
          letrec b = proc(l)
                       if null?(l) then throw(-1)
                       else if n equals car(l)
                            then 1 else 1 + b(cdr(l))
          in try b(l) catch proc(x) x
in f(5, list(2, 6, 7, 2, 6, 5, 7))
```

Implicit vs. Explicit Exceptions

The exception above is an *explicit* exception, because a `throw` command is explicitly used to throw an exception. There are situations, however, when exception throwing commands do not explicitly occur in programs but they are implicit. A typical example is "division by zero". For example, the expression

```
try
  let x = 6 / 0
  in x
catch proc(x) x
```

evaluates to -1 because `6/0` implicitly throws an exception. Our convention is that it throws the expression `-1`. As part of your HW exercise you have to define both general purpose explicit exceptions and particular implicit exceptions for division by zero. Note that lists are added to `exceptions.maude`.

Exceptions and Continuations

We next briefly discuss how exceptions can be added to a continuation-based definition of language. Since we want an arithmetic operation to potentially throw implicit exceptions, the module `EXCEPTIONS-SEMANTICS` needs to be defined *before* `ARITH-OPS-SEMANTICS`. The major idea is the following:

> *Since in a continuation-based definition of a language the continuation contains all the needed information to continue the computation, an exception throwing statement just needs to discard the continuation items one-by-one from the current control context until the first try/catch is encountered. Then the thrown value is passed to the catch closure and the computation naturally continues from there.*

Defining Exceptions with Continuations

In addition to defining three important continuation items,

```
op try : Exp Env -> ContinuationItem .
op catch : Value -> ContinuationItem .
op throw : -> ContinuationItem .
```

one needs only 6 equations, performing the following tasks:

1. When `try E catch E'` is encountered, in the spirit of continuations one starts evaluating `E'` and place `try(E,Env)` in the continuation, where `Env` is the current environment;
2. Once `E'` is evaluated, i.e., its value `Closure` (which is expected to be a closure with one argument) is returned on top of `try(E,Env)` in the continuation, start evaluating `E` in `Env` and place `catch(Closure)` in the continuation to know where to stop discarding continuation items if an exception is thrown;

3. If a value `V` is returned as a result of evaluating `E` then discard the continuation item `catch(Closure)` above and pass `V` to the remaining continuation;
4. When a `throw(E)` statement is encountered, in the continuation spirit one should place the continuation item `throw` in the continuation and evaluate `E`;
5. Once a value `V` of `E` is calculated and therefore found in the continuation, one starts discarding continuation items that occur immediately under `throw` and are different from `catch(Closure)`; this can be done with two equations using `[owise]`, where the next one may come first;
6. If the continuation matches `cont(values(V) -> throw -> catch(Closure) -> ...)`, then one invokes `Closure` on `V` and discard `throw`. With the existing machinery, the continuation transforms into `cont(values(Closure,V) -> fn -> ...)`.

Continuation-Passing Style Transformation

As we have discussed, the behavior of programs at runtime can be characterized as

- *Recursive control behavior* when additional information must be recorded with each recursive call and that information needs to be retained until the call returns; and as
- *Iterative control behavior* when only a limited amount of space is required for storing control information.

Very efficient compilation can be developed for programs admitting iterative control behavior, because function calls, recursive or not, can be reduced to just memory allocation for arguments followed by simple jumps in RAM.

In what follows we will define a procedure, called *CPS* as an abbreviation for *continuation-passing style*, that transforms any

program in our functional language into a program which has iterative control behavior. The trick is to

> *Collect and pass all the control information needed to continue the execution of the program as an additional argument to every function.*

The CPS transformation can be seen as a pre-compilation stage, which arranges the programs in a form admitting very efficient compilation. We do not discuss compilation issues in this class, but because of its practical and theoretical importance we will discuss and define the CPS transformation procedure.

Tail Positions

Recall that the main reason for which the expression

```
letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
in f(8000)
```

could not be evaluated while the expression

```
letrec f = proc(n,m)
             if zero?(n)
             then m
             else f(n - 1, m * n)
in f(13000, 1)
```

could be evaluated, was that the former had to implicitly store control context to "remember" from where to continue the

computation once it returns from a recursive call. The second one did not need to store this information, because the result of the recursive call is the result of the entire computation. We call such calls, whose result is the result of the whole call, *tail calls*. Let us define these formally.

An expression is called *simple* if and only if it does not contain any function call. Primitive operations, such as addition, car, cdr, etc., are not considered function calls. A function declaration is always simple and a function invocation is never simple.

A subexpression is in *tail position* if and only if it has the property that if it is evaluated, then its result becomes the result of the entire expression. An expression may have more than one tail positions; e.g., the conditional. For our functional language (to keep the presentation simple, we do not consider side effects), the tail positions are given by the following easy to remember rules, where `Tail` denotes a tail position:

```
if <Exp> then Tail else Tail
let/letrec <BindingList> in Tail
proc(<ParamList>) Tail
```

Function calls on tail positions are called *tail calls*. An expression is in *tail form* if and only if it has only simple expressions in non-tail positions. Our goal next is to transform any functional program into an equivalent one which is in tail form.

Exercise 1 *Read Section 8.1 in Friedman for more on tails forms.*

The Factorial Example

Let us consider the recursive behavior version of the factorial. The CPS procedure will *automatically* transform it into the following equivalent tail-form expression:

```
letrec f = proc (n,k)
  if zero?(n)
  then k(1)
  else f(n - 1, proc(v)(k(v * n)))
in f(100, proc(x) x)
```

The idea is to add one more argument to each function, called *the continuation argument*, with the continuation-based intuition that it collects all the control information needed in order for a function call to be moved from a non-tail position to a tail-position.

The CPS Procedure Informally

The CPS program transformation will be defined by means of two transformations defined mutually recursively:

- One for detecting the function declarations in simple expressions and transforming them by adding the continuation parameter, and
- Another for collecting and passing the continuation to function calls.

We denote these operations simply by `[<simple expression>]` and `[<expression> -> <continuation>]`. Note that the argument `<continuation>` is nothing but a function declaration.

`[_]` and `[_->_]` will be formally defined in the next lecture. We next only discuss their definitions intuitively.

`[<simple expression>]` traverses the simple expression and transforms any function declaration `proc(Xl) E` into

```
proc(Xl, K) [E -> K]
```

where `K` is a fresh name, reflecting the intuition that each function will take a continuation argument to which it will

"pass" its result once calculated.

The operation `[<expression> -> <continuation function>]` is the difficult one to define. With the intuition that the expression must be modified to one which is in tail form and which passes its value to the continuation, we can split its definition into the following four possibilities.

(1) If `E` is a *simple expression* then

`[E -> K]` rewrites to `K([E])`.

Therefore, the simple expression is first transformed and then "passed" to the continuation.

(2) If `E` is *not* simple but all its direct subexpressions are simple, in our language meaning that `E` is a function call, say `F(El)`, then

`[F(El) -> K]` rewrites to `[F]([El],K)`.

Therefore, all the direct simple subexpressions are first transformed and then the continuation is "passed" as an additional argument.

(3) If `E` has some direct subexpression on a *non-tail position* which is *not* simple, then in the spirit of the continuation-passing style, that expression can be selected to be evaluated next but a proper continuation needs to be generated. Suppose that `E` is `C{E'}` for some *context* `C`. Then

`[C{E'} -> K]` rewrites to `[E' -> proc(v)[C{v} -> K]]`,

where `v` is a fresh name. Note that the right-hand term contains two applications of `[_->_]`. However, the inner one is applied to an expression which is simpler than the original one, because one of its non-simple subexpressions has been replaced by a name. This will ensure that the mutual recursion terminates.

The three rules above reduce any transformation to one in which the only non-simple expressions are on tail positions:

(4) If none of the above transformations can be applied anymore then the only thing left to do is to apply the transformation to the

direct subexpressions occurring on tail positions and to also propagate `[_]` appropriately on the non-tail simple direct subexpressions. There are two concrete cases to analyze:

`[if E then E1 else E2 -> K]` rewrites to
`if [E] then [E1 -> K] else [E2 -> K]`,

and

`[let/letrec X1=E1,...,Xn=En in E -> K]` rewrites to
`let/letrec X1=[E1],...,Xn=[En] in [E -> K]`,

Note that `E` in the former and `E1,...,En` in the later are all simple, otherwise rule *(3)* would apply instead.

```
*****************************************
*** Funtional Language with Exceptions ***
*****************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod EXCEPTIONS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op try_catch_ : Exp Exp -> Exp .
  op throw_ : Exp -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
```

```
  op _and_ : Exp Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  op car : Exp -> Exp .
  op cdr : Exp -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
```

```
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending EXCEPTIONS-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLocations : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList
           [assoc id: noLocations] .
  op loc : Nat -> Location .
  op locs : Nat Nat -> LocationList .
  vars N M : Nat .
  eq locs(N,0) = noLocations .
  eq locs(N,M) = loc(N), locs(N + 1, M - 1) .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
```

```
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLocations] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq St[noLocations <- noValues] = St .
  eq ([L,V] St)[L,Ll <- V',Vl] = ([L,V'] St)[Ll <- Vl] .
  eq St[L,Ll <- V',Vl] = (St [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  extending VALUE .
  extending ENVIRONMENT .
  extending GENERIC-EXP-SYNTAX .
  sorts ContinuationItem Continuation .
  op stop : -> Continuation .
  op _->_ : ContinuationItem Continuation -> Continuation .
  op values_ : ValueList -> ContinuationItem .
  op locations : LocationList -> ContinuationItem .
  op frozenExp : Exp Env -> ContinuationItem .
```

```
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  extending GENERIC-EXP-SYNTAX .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op initState : -> State .
  eq initState = nextLoc(0) env(noEnv) store(noStore) cont(stop) .
  op nextLoc : Nat -> StateAttribute .
  op env : Env -> StateAttribute .
  op store : Store -> StateAttribute .
  op exps_ : ExpList -> StateAttribute .
  op cont : Continuation -> StateAttribute .
endfm

fmod EVAL is extending STATE .
  op evalExps : State -> [Value] .
  op evalCont : State -> [Value] .
  op eval : Exp -> Value .
  var E : Exp .  var Vl : ValueList .  var St : Store .
  var S : State .  var C : Continuation .  var Ll : LocationList .
  var Env Env' : Env .  var V : Value .
  eq eval(E) = evalExps(exps(E) initState) .
  eq evalCont(cont(values(V) -> stop) S) = V .
  eq evalCont(cont(values(Vl) -> locations(Ll) -> C) store(St) S) =
     evalCont(cont(C) store(St[Ll <- Vl]) S) .
  eq evalCont(cont(frozenExp(E,Env) -> C) env(Env') S) =
     evalExps(exps(E) cont(C) env(Env) S) .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending EVAL .
  op [_|_|_] : ValueList ExpList Env -> ContinuationItem .
  op int : Int -> Value .
  op #_ : ExpList -> Nat .
  var E : Exp .  var El : ExpList .  var I : Int .
  var C : Continuation .  var S : State .  vars Env Env' : Env .
```

```
  var V : Value .  var Vl : ValueList .  var X : Name .
  var L : Location .  var St : Store .
  eq #() = 0 .
  eq #(E,El) = 1 + #(El) .
  eq evalExps(exps(I) cont(C) S) =
     evalCont(cont(values(int(I)) -> C) S) .
  eq evalExps(exps(X) cont(C) env([X,L] Env) store([L,V] St) S) =
     evalCont(cont(values(V) -> C) env([X,L] Env) store([L,V] St) S) .
  eq evalExps(exps() S) = evalCont(S) .
 ceq evalExps(exps(E,El) cont(C) env(Env) S) =
     evalExps(exps(E) cont([noValues | El | Env] -> C) env(Env) S)
    if El =/= () .
  eq evalCont(cont(values(V) -> [Vl | () | Env] -> C) S) =
     evalCont(cont(values(Vl,V) -> C) S) .
  eq evalCont(cont(values(V) -> [Vl | E,El | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont([Vl,V | El | Env] -> C) env(Env) S) .
endfm

fmod EXCEPTIONS-SEMANTICS is extending EXCEPTIONS-SYNTAX .
  extending PROC-SEMANTICS .
  op try : Exp Env -> ContinuationItem .
  op catch : Value -> ContinuationItem .
  op throw : -> ContinuationItem .
---> define exceptions here
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending EXCEPTIONS-SEMANTICS .
  vars E E' : Exp .  var S : State .
  var C : Continuation .  vars I J : Int .
  ops + - * / : -> ContinuationItem .
  eq evalExps(exps(E + E') cont(C) S) =
     evalExps(exps((E,E')) cont(+ -> C) S) .
  eq evalCont(cont(values((int(I),int(J))) -> + -> C) S) =
     evalCont(cont(values(int(I + J)) -> C) S) .
---> define subtraction, multiplication, division
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  ops equals zero? even? not and : -> ContinuationItem .
  vars E E' Be Be' : Exp .  var S : State .  vars I J : Int .
```

```
  var C : Continuation .  vars B B' : Bool .
  eq evalExps(exps(E equals E') cont(C) S) =
     evalExps(exps((E,E')) cont(equals -> C) S) .
  eq evalCont(cont(values(int(I),int(J)) -> equals -> C) S) =
     evalCont(cont(values(bool(I == J)) -> C) S) .
---> define zero?, even?, not, and
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending BEXP-SEMANTICS .
  ops list car cdr null? : -> ContinuationItem .
  var E : Exp .  var El : ExpList .  var C : Continuation .
  var S : State .  var Vl : ValueList .  var V : Value .
  eq evalExps(exps(list(El)) cont(C) S) =
     evalExps(exps(El) cont(list -> C) S) .
  eq evalCont(cont(values(Vl) -> list -> C) S) =
     evalCont(cont(values([Vl]) -> C) S) .
  eq evalCont(cont(list -> C) S) =
     evalCont(cont(values([noValues]) -> C) S) .
  eq evalExps(exps(car(E)) cont(C) S) =
     evalExps(exps(E) cont(car -> C) S) .
  eq evalCont(cont(values([V,Vl]) -> car -> C) S) =
     evalCont(cont(values(V) -> C) S) .
  eq evalExps(exps(cdr(E)) cont(C) S) =
     evalExps(exps(E) cont(cdr -> C) S) .
  eq evalCont(cont(values([V,Vl]) -> cdr -> C) S) =
     evalCont(cont(values([Vl]) -> C) S) .
  eq evalExps(exps(null?(E)) cont(C) S) =
     evalExps(exps(E) cont(null? -> C) S) .
  eq evalCont(cont(values([Vl]) -> null? -> C) S) =
     evalCont(cont(values(bool(Vl == noValues)) -> C) S) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  op if(_,_,_) : Exp Exp Env -> ContinuationItem .
  vars Be E E' : Exp .  var C : Continuation .
  var S : State .  vars Env Env' : Env .  var B : Bool .
  eq evalExps(exps(if Be then E else E') cont(C) env(Env) S) =
     evalExps(exps(Be) cont(if(E,E',Env) -> C) env(Env) S) .
  eq evalCont(cont(values(bool(B)) -> if(E,E',Env) -> C) env(Env') S) =
     evalExps(exps(if B then E else E' fi) cont(C) env(Env) S) .
endfm
```

```
fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op expressions_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq expressions(X = E, Bl) = E, expressions(Bl) .
  eq expressions(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
 ceq evalExps(exps(let Bl in E) nextLoc(N) env(Env) cont(C) S) =
     evalExps(exps(El) nextLoc(N + M) env(Env)
              cont(locations(Ll) -> frozenExp(E, Env[Xl <- Ll]) -> C) S)
    if Xl := names(Bl) /\ El := expressions(Bl)
    /\ M := #(Xl) /\ Ll := locs(N, M) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  op fn : -> ContinuationItem .
  vars Xl : NameList .  vars E F : Exp .  var C : Continuation .
  vars Env Env' : Env .  var S : State .  var El : ExpList .
  var Vl : ValueList .  var St : Store .  vars N M : Nat .
  var Ll : LocationList .
  eq evalExps(exps(proc(Xl) E) cont(C) env(Env) S) =
     evalCont(cont(values(closure(Xl,E,Env)) -> C) env(Env) S) .
  eq evalExps(exps(F(El)) cont(C) S) =
     evalExps(exps((F,El)) cont(fn -> C) S) .
--->  add a corresponding evalCont for the above equation
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
```

```
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
---> to be defined ... (hint: look at let)
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var C : Continuation .  var S : State .  var L : Location .
  eq evalExps(exps(set X = E) env([X,L] Env) cont(C) S) =
     evalExps(exps(E) env([X,L] Env)
            cont(locations(L) -> values(int(1)) -> C) S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
---> to be defined
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  op while(_,_,_) : Exp Exp Env -> ContinuationItem .
---> to be defined
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending LIST-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
  extending EXCEPTIONS-SEMANTICS .
endfm

--- if the following evaluate properly, then your definition is most
--- likely correct.  Look especially at the last three examples.


red eval(
```

```
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
```

```
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
```

```
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
```

```
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                  if zero?(x)
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
```

```
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
```

```
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
             let d = set c = c + 1
             in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
             let d = set x = x + 1
             in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
             let t = x
             in let d = set x = y
                in let d = set y = t
                   in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
```

```
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
         'odd = proc() if zero?(x) then 0
                       else let d = set x = x - 1
                            in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11
```

```
red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
```

```
      }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
```

```
    'even = proc(x, o, e)
              if zero?(x) then 1
              else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

----------

red eval(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(100, 1)

) .

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
```

```
  in f(100)
) .

red eval(
  letrec f = proc(n,l)
             if null?(l) then -1
             else if n equals car(l)
                  then 1
                  else let p = f(n, cdr(l))
                       in if p equals -1 then -1
                          else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

red eval(
  let f = proc(n,l)
          letrec b = proc(l)
                     if null?(l) then throw(-1)
                     else if n equals car(l)
                          then 1
                          else 1 + b(cdr(l))
          in try b(l) catch proc(x) x
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

red eval(
  try
    let x = 6 / 0
    in x
  catch proc(x) x
) .

red eval(
  try
    let n = 1028, c = 0
    in { while not (n equals 1) {
           set c = c + 1 ;
           if n equals 5 then set n = n / (n - n) else set n = n ;
           if even?(n)
           then set n = n / 2
           else set n = 3 * n + 1
         } ;
         c }
```

```
  catch proc(x) x
) .
```

```
****************************************
*** Funtional Language with Exceptions ***
****************************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod EXCEPTIONS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op try_catch_ : Exp Exp -> Exp .
  op throw_ : Exp -> Exp .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : Exp Exp -> Exp .
  op zero? : Exp -> Exp .
  op even? : Exp -> Exp .
  op not_ : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  op car : Exp -> Exp .
  op cdr : Exp -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is protecting BEXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending LIST-SYNTAX .
  extending EXCEPTIONS-SYNTAX .
  extending ARITH-OPS-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLocations : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList
             [assoc id: noLocations] .
  op loc : Nat -> Location .
  op locs : Nat Nat -> LocationList .
  vars N M : Nat .
  eq locs(N,0) = noLocations .
  eq locs(N,M) = loc(N), locs(N + 1, M - 1) .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLocations] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq St[noLocations <- noValues] = St .
  eq ([L,V] St)[L,Ll <- V',Vl] = ([L,V'] St)[Ll <- Vl] .
  eq St[L,Ll <- V',Vl] = (St [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  extending VALUE .
  extending ENVIRONMENT .
  extending GENERIC-EXP-SYNTAX .
  sorts ContinuationItem Continuation .
  op stop : -> Continuation .
  op _->_ : ContinuationItem Continuation -> Continuation .
  op values_ : ValueList -> ContinuationItem .
  op locations : LocationList -> ContinuationItem .
  op frozenExp : Exp Env -> ContinuationItem .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  extending GENERIC-EXP-SYNTAX .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op initState : -> State .
  eq initState = nextLoc(0) env(noEnv) store(noStore) cont(stop) .
  op nextLoc : Nat -> StateAttribute .
  op env : Env -> StateAttribute .
  op store : Store -> StateAttribute .
  op exps_ : ExpList -> StateAttribute .
  op cont : Continuation -> StateAttribute .
endfm

fmod EVAL is extending STATE .
  op evalExps : State -> [Value] .
  op evalCont : State -> [Value] .
  op eval : Exp -> Value .
  var E : Exp .  var Vl : ValueList .  var St : Store .
  var S : State .  var C : Continuation .  var Ll : LocationList .
  var Env Env' : Env .  var V : Value .
  eq eval(E) = evalExps(exps(E) initState) .
  eq evalCont(cont(values(V) -> stop) S) = V .
  eq evalCont(cont(values(Vl) -> locations(Ll) -> C) store(St) S) =
     evalCont(cont(C) store(St[Ll <- Vl]) S) .
  eq evalCont(cont(frozenExp(E,Env) -> C) env(Env') S) =
     evalExps(exps(E) cont(C) env(Env) S) .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending EVAL .
  op [_|_|_] : ValueList ExpList Env -> ContinuationItem .
  op int : Int -> Value .
  op #_ : ExpList -> Nat .
  var E : Exp .  var El : ExpList .  var I : Int .
  var C : Continuation .  var S : State .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  var X : Name .
  var L : Location .  var St : Store .
  eq #() = 0 .
  eq #(E,El) = 1 + #(El) .
  eq evalExps(exps(I) cont(C) S) =
     evalCont(cont(values(int(I)) -> C) S) .
  eq evalExps(exps(X) cont(C) env([X,L] Env) store([L,V] St) S) =
     evalCont(cont(values(V) -> C) env([X,L] Env) store([L,V] St) S) .
  eq evalExps(exps() S) = evalCont(S) .
 ceq evalExps(exps(E,El) cont(C) env(Env) S) =
     evalExps(exps(E) cont([noValues | El | Env] -> C) env(Env) S)
   if El =/= () .
  eq evalCont(cont(values(V) -> [Vl | () | Env] -> C) S) =
     evalCont(cont(values(Vl,V) -> C) S) .
  eq evalCont(cont(values(V) -> [Vl | E,El | Env] -> C) env(Env') S) =
     evalExps(exps(E) cont([Vl,V | El | Env] -> C) env(Env) S) .
endfm

fmod EXCEPTIONS-SEMANTICS is extending EXCEPTIONS-SYNTAX .
  extending PROC-SEMANTICS .
  op try : Exp Env -> ContinuationItem .
  op catch : Value -> ContinuationItem .
  op throw : -> ContinuationItem .
---> define exceptions here
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending EXCEPTIONS-SEMANTICS .
  vars E E' : Exp .  var S : State .
  var C : Continuation .  vars I J : Int .
  ops + - * / : -> ContinuationItem .
  eq evalExps(exps(E + E') cont(C) S) =
     evalExps(exps((E,E')) cont(+ -> C) S) .
  eq evalCont(cont(values((int(I),int(J))) -> + -> C) S) =
     evalCont(cont(values(int(I + J)) -> C) S) .
---> define subtraction, multiplication, division
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  ops equals zero? even? not and : -> ContinuationItem .
  vars E E' Be Be' : Exp .  var S : State .  vars I J : Int .
  var C : Continuation .  vars B B' : Bool .
  eq evalExps(exps(E equals E') cont(C) S) =
     evalExps(exps((E,E')) cont(equals -> C) S) .
  eq evalCont(cont(values(int(I),int(J)) -> equals -> C) S) =
     evalCont(cont(values(bool(I == J)) -> C) S) .
---> define zero?, even?, not, and
endfm
```

```
fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending BEXP-SEMANTICS .
  ops list car cdr null? : -> ContinuationItem .
  var E : Exp .  var El : ExpList .  var C : Continuation .
  var S : State .  var Vl : ValueList .  var V : Value .
  eq evalExps(exps(list(El)) cont(C) S) =
     evalExps(exps(El) cont(list -> C) S) .
  eq evalCont(cont(values(Vl) -> list -> C) S) =
     evalCont(cont(values([Vl]) -> C) S) .
  eq evalCont(cont(list -> C) S) =
     evalCont(cont(values([noValues]) -> C) S) .
  eq evalExps(exps(car(E)) cont(C) S) =
     evalExps(exps(E) cont(car -> C) S) .
  eq evalCont(cont(values([V,Vl]) -> car -> C) S) =
     evalCont(cont(values(V) -> C) S) .
  eq evalExps(exps(cdr(E)) cont(C) S) =
     evalExps(exps(E) cont(cdr -> C) S) .
  eq evalCont(cont(values([V,Vl]) -> cdr -> C) S) =
     evalCont(cont(values([Vl]) -> C) S) .
  eq evalExps(exps(null?(E)) cont(C) S) =
     evalExps(exps(E) cont(null? -> C) S) .
  eq evalCont(cont(values([Vl]) -> null? -> C) S) =
     evalCont(cont(values(bool(Vl == noValues)) -> C) S) .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  op if(_,_,_) : Exp Exp Env -> ContinuationItem .
  vars Be E E' : Exp .  var C : Continuation .
  var S : State .  vars Env Env' : Env .  var B : Bool .
  eq evalExps(exps(if Be then E else E') cont(C) env(Env) S) =
     evalExps(exps(Be) cont(if(E,E',Env) -> C) env(Env) S) .
  eq evalCont(cont(values(bool(B)) -> if(E,E',Env) -> C) env(Env') S) =
     evalExps(exps(if B then E else E' fi) cont(C) env(Env) S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op expressions_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq expressions(X = E, Bl) = E, expressions(Bl) .
  eq expressions(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
 ceq evalExps(exps(let Bl in E) nextLoc(N) env(Env) cont(C) S) =
     evalExps(exps(El) nextLoc(N + M) env(Env)
              cont(locations(Ll) -> frozenExp(E, Env[Xl <- Ll]) -> C) S)
    if Xl := names(Bl) /\ El := expressions(Bl)
    /\ M := #(Xl) /\ Ll := locs(N, M) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  op fn : -> ContinuationItem .
  vars Xl : NameList .  vars E F : Exp .  var C : Continuation .
  vars Env Env' : Env .  var S : State .  var El : ExpList .
  var Vl : ValueList .  var St : Store .  vars N M : Nat .
  var Ll : LocationList .
  eq evalExps(exps(proc(Xl) E) cont(C) env(Env) S) =
     evalCont(cont(values(closure(Xl,E,Env)) -> C) env(Env) S) .
  eq evalExps(exps(F(El)) cont(C) S) =
     evalExps(exps((F,El)) cont(fn -> C) S) .
--->  add a corresponding evalCont for the above equation
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var Bl : BindingList .  var E : Exp .  vars N M : Nat .
  var Env : Env .  var C : Continuation .  var S : State .
  var El : ExpList .  var Ll : LocationList .  var Xl : NameList .
---> to be defined ... (hint: look at let)
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var C : Continuation .  var S : State .  var L : Location .
  eq evalExps(exps(set X = E) env([X,L] Env) cont(C) S) =
     evalExps(exps(E) env([X,L] Env)
              cont(locations(L) -> values(int(1)) -> C) S) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
---> to be defined
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-SEMANTICS .
  op while(_,_,_) : Exp Exp Env -> ContinuationItem .
---> to be defined
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending LIST-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .
  extending EXCEPTIONS-SEMANTICS .
endfm

--- if the following evaluate properly, then your definition is most
--- likely correct.  Look especially at the last three examples.


red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) x * (y - z)
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping
```

```
red eval(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2
```

```
red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if zero?(x)
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if zero?(x)
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if zero?(x) then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if zero?(x) then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if zero?(x)
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n equals 1) {
         set c = c + 1 ;
         if even?(n)
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined
```

```
red eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(100, 1)

) .

red eval(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(100)
) .

red eval(
  letrec f = proc(n,l)
               if null?(l) then -1
               else if n equals car(l)
                    then 1
                    else let p = f(n, cdr(l))
                         in if p equals -1 then -1
                            else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

red eval(
  let f = proc(n,l)
            letrec b = proc(l)
                         if null?(l) then throw(-1)
                         else if n equals car(l)
                              then 1
                              else 1 + b(cdr(l))
            in try b(l) catch proc(x) x
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

red eval(
  try
    let x = 6 / 0
    in x
  catch proc(x) x
) .

red eval(
  try
    let n = 1028, c = 0
    in { while not (n equals 1) {
           set c = c + 1 ;
           if n equals 5 then set n = n / (n - n) else set n = n ;
           if even?(n)
           then set n = n / 2
           else set n = 3 * n + 1
         } ;
         c }
  catch proc(x) x
) .
```

CS322 - Programming Language Design

Lecture 21: Continuation-Passing Style (CPS) Transformation (Part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We now formally define the CPS procedure which transform any functional program into an equivalent tail-form program. The intuition underlying the CPS transformed functions is the following:

> *They do not "return" values anymore to the calling context; instead, they just pass the result to the "continuation" argument, which "knows" how to continue the remaining of the computation.*

Several functional languages are compiled after an initial CPS transformation. However, despite the fact that even very serious languages, such as New Jersey's Standard ML, use CPS before compilation, there is *not* a full agreement on whether the CPS transformation really brings a huge benefit in compilation.

The reason is that the additional continuation argument can grow quite big and thus may become a serious bottle-neck of the entire technique, unless lambda-optimizations are performed on it to keep

it compact.

There is an interesting research initiative in using continuations in *internet applications*, such as client-server systems, based on the idea that a server maintains a continuation for each client, knowing how to continue the service.

The Two Mutually Recursive Transformations

Let us recall how `[_]` and `[_->_]` are mutually recursively defined.

`[_]` works only on simple expressions. It propagates to function declarations, which are modified by adding the continuation parameter:

`[proc(Xl) E]` rewrites to `proc(Xl, K) [E -> K]`.

`[_->_]` takes any expression and a continuation, and modifies the expression to "send" its result to the continuation. There are 4 cases, considered in the following order:

(1) `[E -> K]` rewrites to `K([E])` if `E` is *simple*.

(2) `[F(El) -> K]` rewrites to `[F]([El],K)` if `F` and `El` are *simple*

(3) `[C{E} -> K]` rewrites to `[E -> proc(v)[C{v} -> K]]` if `E` is *not simple* and occurs on a *non-tail* position in a context `C{_}`.

(4) Otherwise propagate the continuation in the tail positions:

`[if E then E1 else E2 -> K]` rewrites to
`if [E] then [E1 -> K] else [E2 -> K]`,

and

`[let/letrec X1=E1,...,Xn=En in E -> K]` rewrites to
`let/letrec X1=[E1],...,Xn=[En] in [E -> K]`,

Defining Simple Expressions

Simple expressions are those expressions intended to have an efficient and controlled evaluation, which means that their evaluation does not need to invoke any function. Note that they can contain function declarations and that the body of the declared functions can invoke functions!

One way to check whether an expression is simple or not is to define a corresponding predicate inductively over the structure of expressions. However, in Maude we can do it more elegantly by taking advantage of Maude's capability of efficiently calculating the *least sort* of a term. The least sort of a term is always calculated *by default*, as shown by the following examples:


```
Maude> parse 3 + 4 .
NzNat: 3 + 4
Maude> parse 3 - 2 .
Int: 3 - 2
Maude> parse 3 + f(x) .
Exp: 3 + f x
```

The least sort of a term makes sense because of *operator overloading*, that is, because the same operator can have several arities. For example, `_-_` is defined between integers, rationals, expressions, etc. In the case of `3 - 2`, since both `3` and `2` are integers, `Maude` inferred that the least sort this expression can have is `Int`. This is related to the concept of *most concrete abstract value* of a concrete value in *abstract interpretation*.

Therefore, what we have to do is to first define appropriate sorts for simple expressions and for lists of simple expressions, `SExp` and `SExpList`, respectively, together with the sort hierarchy

```
subsorts Int Name < SExp < Exp SExpList < ExpList
```

and then to overload all the operations that propagate the property of *simple expression*, such as:

```
  op _,_ : SExpList SExpList -> SExpList [ditto] .
  op _,_ :  ExpList  ExpList ->  ExpList [ditto] .
...
  op _+_ : SExp SExp -> SExp [ditto] .
  op _+_ :  Exp  Exp ->  Exp [ditto] .
...
  op zero? : SExp -> SExp .
  op list_ : SExpList -> SExp .
  op if_then_else_ : SExp SExp SExp -> SExp .
...
```


Of course, all these are defined *modularly* as usual. Note that function declarations are always simple and function invocations are always non-simple, so we declare them as follows:

```
  op proc__ : NameList Exp -> SExp .
  op __ : Exp ExpList -> Exp [prec 1] .
```

Bindings can also be simple, namely when all the expressions involved are simple:

```
  subsort SBinding < Binding SBindingList < BindingList .
  op none : -> SBindingList .
  op _,_ : SBindingList SBindingList -> SBindingList [assoc id: none prec 71] .
  op _,_ :  BindingList  BindingList ->  BindingList [ditto] .
  op _=_ : Name SExp -> SBinding [prec 70] .
  op _=_ : Name  Exp ->  Binding [ditto] .
```

`let` and `letrec` are simple if and only if all their subexpressions are simple:

```
  op let_in_ : SBindingList SExp -> SExp .
  op let_in_ :  BindingList  Exp ->  Exp .
...
  op letrec_in_ : SBindingList SExp -> SExp .
  op letrec_in_ :  BindingList  Exp ->  Exp .
```

All the above are defined in the file `cps.maude`. We will next define part of the CPS transformation.

Homework Exercise 1 *Complete the definition of the CPS transformation in* `cps.maude`*. If all the examples at the end of the file can be evaluated properly, then your definition is most likely correct. Note that the CPS transformation is first applied and then the resulting programs are executed.*

The CPS State

Unlike programming languages, most specification languages, including Maude, do not provide the users a *state* in the usual sense and do not have *side effects*. That's the reason for which we always need to carry a state in our definitions.

In the case of the CPS transformation, since we need to generate *fresh names* for continuations and function parameters, we need to also maintain a state containing the current indexes for the different types of fresh names. This state will be called `CpsState`, and, like before, it will have attributes of sort `CpsStateAttribute`.

```
fmod CPS-STATE is ... (see cps.maude)
  sorts CpsStateAttribute CpsState .
  subsort CpsStateAttribute < CpsState .
  op empty : -> CpsState .
  op __ : CpsState CpsState -> CpsState [assoc comm id: empty] .
```

As usual, the CPS state needs to be passed by most of the operators that will be defined, so we need appropriate sorts and constructors:

```
sorts SExpListCpsStatePair ExpCpsStatePair .
op {_,_} : SExpList CpsState -> SExpListCpsStatePair .
op {_,_} : ExpList CpsState -> ExpCpsStatePair .
```

The initial CPS state is empty:

```
op initCpsState : -> CpsState .
eq initCpsState = empty .
```


The next is the core of the CPS state. The state should allow us to generate fresh names of the form `'k0`, `'k1`, `'k3`, etc., or `'v0`, `'v1`, `'v3`, etc.. Therefore, for each `Qid` prefix, we maintain a current counter. An operation `genSym` is defined to generate new names:

```
op nextSym : Qid Nat -> CpsStateAttribute .
op genSym : Qid CpsState -> ExpCpsStatePair .
var Q : Qid .  var N : Nat .  var S : CpsState .
eq genSym(Q, nextSym(Q,N) S) =
   {qid(string(Q) + string(N,10)), nextSym(Q, N + 1) S} .
eq genSym(Q, S) = {qid(string(Q) + string(0,10)), nextSym(Q, 1) S} .
endfm
```

The CPS Basic Operations

The program transformers `[_]` and `[_->_]` will most certainly need to generate fresh names for function parameters and continuations. Therefore, they modify the CPS state, so they need to pass the CPS state. Consequently, they will need to be defined as follows:

```
fmod CPS-BASIC is ... (see cps.maude)
  op [_->_,_] : Exp Exp CpsState -> [ExpCpsStatePair] .
  op [_,_] : SExpList CpsState -> [SExpListCpsStatePair] .
```

Once these operators are properly (mutually recursively) defined, we can calculate the CPS of a program as follows:

```
  op cps : Exp -> Exp .
 ceq cps(E) = proc(Xk) E'
    if {Xk,Sk} := genSym('k,initCpsState)
    /\ {E',S'} := [E -> Xk, Sk] .
```


Here, `Xk` is a fresh name starting with `'k` generated by `genSym`, and `E` and `E'` are any expressions. `cps(E)` will generate a function which takes a continuation as argument, and *passes it the value of* `E`. In order to evaluate `E` via the CPS transformation we will have to evaluate

```
cps(E)(proc(x) x)
```

That is exactly how we will define the operation `evalCps` later.

Since the simple expressions come for free due to least sort calculation, we can actually get rid of the first of the four cases in the definition of `[_->_,_]` as follows, where `Es` and `Es'` are declared as variables of sort `SExp` (i.e., as simple expressions):

```
 ceq [Es -> K, S] = {K(Es'),S'} if {Es',S'} := [Es,S] .
```

The above takes care of the situation when the expression to apply the continuation to is simple, but there are several other situations in which we will need to find a non-simple subexpression of an expression. Therefore, we define the following now, and then close the module CPS-BASIC:

```
  op nonSimple : Exp -> Bool .
  eq nonSimple(Es) = false .
  eq nonSimple(E) = true [owise] .
endfm
```

What is left to do now is to inductively traverse each language construct and to define the two mutually recursive program transformers.

CPS Transformation of Generic Expressions

In the case of generic expressions, which also define lists of expressions, we need to only define the first transformer. The second will be defined on each expression constructor separately in the sequel:

```
fmod GENERIC-EXP-CPS is extending CPS-BASIC .
  vars S S' Sl : CpsState .  vars Els Els' : SExpList .
  vars Es Es' : SExp .  var X : Name .  var I : Int .
  eq [X,S] = {X,S} .
  eq [I,S] = {I,S} .
  eq [(),S] = {(),S} .
 ceq [(Els,Es),S] = {(Els',Es'), S'}
    if Els =/= () /\ {Els',Sl} := [Els,S] /\ {Es',S'} := [Es,Sl] .
endfm
```

CPS Transformation of Arithmetic Operators

We only discuss addition, but you will have to define subtraction and multiplication as part of the HW exercise:

```
fmod ARITH-OPS-CPS is ... (see cps.maude)
 ceq [E1 + E2 -> K, S] = [E1 -> proc(V) Ep, Sp]
    if nonSimple(E1) /\ {V,Sv} := genSym('v,S)
    /\ {Ep,Sp} := [V + E2 -> K, Sv] .
 ceq [Es1 + Es2, S] = {Es1' + Es2', S'}
    if {(Es1',Es2'), S'} := [(Es1,Es2), S] .
---> define subtraction and multiplication only (no division)
endfm
```

Since `_+_` is commutative, `nonSimple(E1)` will match either of the two subexpressions which is *not* simple. If both are simple then there is nothing to do here, because the whole expression is simple and the rule in `CPS-BASIC` will take care of it. From now on, `E`, `E'`,

`E1`, `E1'`, etc., stay for *any expressions*, and `Es`, `Es'`, `Es1`, `Es1'`, etc., stay for *simple expressions*.

Therefore, if there is a non-simple subexpression it will be selected, then a fresh name `V` will be generated which will be the parameter of the new continuation (which is supposed to pass the value of `E1` after evaluation), and then the body of the continuation function, `V + E2`, is CPS processed. Note that `E2` may itself be non-simple, in which case the equation above will be applied one more time.

CPS Transformation of Boolean Operators

```
fmod BEXP-CPS is extending BEXP-SYNTAX .
  protecting GENERIC-EXP-CPS .
  vars E E1 E2 Ep : Exp .  vars K Es Es1 Es2 Es' Es1' Es2' : SExp .
  vars S Sp Sv S' : CpsState .  var V : Name .
 ceq [zero?(E) -> K, S] = [E -> proc(V)(K(zero?(V))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [zero?(Es), S] = {zero?(Es'), S'} if {Es', S'} := [Es, S] .
---> define the other operations in BEXP-SYNTAX
endfm
```


CPS Transformation of Lists

```
fmod LIST-CPS is extending LIST-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Els Els' : SExpList .  vars E E' Ep : Exp .  var El : ExpList .
  vars K Es Es' Es1 Es1' Es2 Es2' : SExp .  var V : Name .
  var S S' Sv Sp : CpsState .
 ceq [list(Els,E,El) -> K, S] = [E -> proc(V) Ep, Sp]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [list(Els,V,El) -> K, Sv] .
 ceq [list(Els), S] = {list(Els'),S'} if {Els',S'} := [Els,S] .
 ceq [car(E) -> K, S] = [E -> proc(V)(K(car(V))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [car(Es), S] = {car(Es'), S'} if {Es',S'} := [Es, S] .
---> define the other operations in LIST-SYNTAX
endfm
```

CPS Transformation of Conditionals

There are two situations to analyze in the case of conditionals, one when the condition is not simple and another in which the condition is simple. You should complete the module below:

```
fmod IF-CPS is protecting IF-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Be E1 E2 E1' E2' Ep : Exp .
  vars K Bes Bes' Es1 Es2 Es1' Es2' : SExp .
  vars S Sp Sb Sv S1 S1' S' : CpsState .  var V : Name .
 ceq [if Be then E1 else E2 -> K, S] = ...
 ceq [if Bes then E1 else E2 -> K, S] = ...
---> complete this module
endfm
```


CPS Transformation of Bindings

If all the binding expressions are simple, then we need to apply the first transformation to them when we define the CPS on `let` and `letrec` later on. So it will be useful to have the following simple binding transformation application defined now:

```
fmod BINDING-CPS is protecting BINDING-SYNTAX .
  extending CPS-BASIC .
  sort SBindingListCpsStatePair .
  op {_,_} : SBindingList CpsState -> SBindingListCpsStatePair .
  op [_,_] : SBindingList CpsState -> [SBindingListCpsStatePair] .
  vars S S' Sbl : CpsState .  var X : Name .  vars Es Es' : SExp .
  vars Bls Bls' : SBindingList .
  eq [none, S] = {none, S} .
 ceq [(Bls, X = Es), S] = {(Bls', X = Es'), S'}
    if {Bls',Sbl} := [Bls, S] /\ {Es',S'} := [Es,Sbl] .
endfm
```

CPS Transformation of `let`

If there is any non-simple binding expression, then that expression needs to CPS processed. Note, however, that if the continuation is not a variable then we have to generate a wrapping `let`. Why?

Also, notice that the first non-simple binding expression can be easily found by matching, where `Bls` is a list of simple binding lists:

```
fmod LET-CPS is ... (see cps.maude)
  vars Bls Bls' : SBindingList .  vars X V Xk : Name .
  var E E' Ep : Exp .  vars S Sp Sv Sbl S' Sk : CpsState .
  var K Es Es' : SExp .  var Bl : BindingList .
 ceq [let Bls, X = E', Bl in E -> Xk, S] = [E' -> proc(V) Ep, Sp]
    if nonSimple(E') /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [let Bls, X = V, Bl in E -> Xk, Sv] .
 ceq [let Bls, X = E', Bl in E -> K, S] = {let Xk = K in Ep, Sp}
    if nonSimple(E') /\ {Xk,Sk} := genSym('k,S)
```



```
    /\ {Ep, Sp} := [let Bls, X = E', Bl in E -> Xk, Sk] [owise] .
 ceq [let Bls in E -> K, S] = {let Bls' in E', S'} if ...
 ceq [let Bls in Es, S] = ...
---> complete this module
endfm
```

CPS Transformation of Functions

Several cases need to be separately analyzed when defining the CPS transformations on functions. They are self explanatory and listed below:

```
fmod PROC-CPS is ... (see cps.maude)
  vars F E Ep : Exp .  var El : ExpList .  var V : Name .
  vars S Sv Sk Sp S' : CpsState .  vars K Fs Fs' : SExp .
  vars Els Els' : SExpList .  var Xl : NameList .
 ceq [F(El) -> K, S] = ...
 ceq [Fs(Els,E,El) -> K, S] = ...
 ceq [Fs(Els) -> K, S] = {Fs'(Els',K), S'}
    if {(Fs',Els'), S'} := [(Fs,Els), S] .
 ceq [proc(Xl) E, S] = ...
 ceq [Fs(Els), S] = ...
---> complete this module
endfm
```


You should define the CPS transform of `letrec` as well. It should not be too different from that of `let`.

Defining CPS Evaluation

Now we can define the `evalCps` procedure which first transforms an expression in CPS and then evaluates it using the standard executable semantics of our simple functional language:

```
fmod EVAL-CPS is
  protecting PROG-LANG-SEMANTICS .
  protecting PROG-LANG-CPS .
  op evalCps : Exp -> Value .
  vars E : Exp .
  eq evalCps(E) = eval(cps(E) (proc(x) x)) .
endfm
```

An Example CPS Transformation

You may be asked at the final to apply the CPS transformation by hand to a simple program like the one below, which removes all the elements of a list which are equal to certain integer number:

```
red cps(
  letrec r = proc(n,l)
               if null?(l) then emptylist
               else if n equals car(l)
                    then r(n,cdr(l))
                    else cons(car(l), r(n,cdr(l)))
  in r(3, list(3,1,3,2,3,3,3,4,3,5,3,3))
) .
```


The result is the following, which also the result that your CPS definition should return:

```
result SExp:
  proc ('k0)
    letrec r = proc (n,l,'k1)
                 if null?(l) then 'k1(emptylist)
                 else if n equals car(l)
                 then r(n,cdr(l),'k1)
                 else r(n,cdr(l),proc('v0) 'k1(cons(car(l),'v0))) in
    r(3,list (3,1,3,2,3,3,3,4,3,5,3,3),'k0)
```

```
**************************
*** CPS Transformation ***
**************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts SExp Exp SExpList ExpList .
  subsorts Int Name < SExp < Exp SExpList < ExpList .
  subsort NameList < SExpList .
  op _,_ : SExpList SExpList -> SExpList [ditto] .
  op _,_ :  ExpList  ExpList ->  ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : SExp SExp -> SExp [ditto] .
  op _+_ :  Exp  Exp ->  Exp [ditto] .
  op _-_ : SExp SExp -> SExp [ditto] .
  op _-_ :  Exp  Exp ->  Exp [ditto] .
  op _*_ : SExp SExp -> SExp [ditto] .
  op _*_ :  Exp  Exp ->  Exp [ditto] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : SExp SExp -> SExp [comm] .
  op _equals_ :  Exp  Exp ->  Exp [ditto] .
  op zero? : SExp -> SExp .
  op zero? :  Exp ->  Exp .
  op even? : SExp -> SExp .
  op even? :  Exp ->  Exp .
```

```
  op not_ : SExp -> SExp .
  op not_ :  Exp ->  Exp .
  op _and_ : SExp SExp -> SExp [comm] .
  op _and_ :  Exp  Exp ->  Exp [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : SExpList -> SExp .
  op list_ :  ExpList ->  Exp .
  op car : SExp -> SExp .
  op car :  Exp ->  Exp .
  op cdr : SExp -> SExp .
  op cdr :  Exp ->  Exp .
  op cons : SExp SExp -> SExp .
  op cons :  Exp  Exp ->  Exp .
  op emptylist : -> SExp .
  op null? : SExp -> SExp .
  op null? :  Exp ->  Exp .
  op number? : SExp -> SExp .
  op number? :  Exp ->  Exp .
endfm

fmod IF-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op if_then_else_ : SExp SExp SExp -> SExp .
  op if_then_else_ :  Exp  Exp  Exp ->  Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding SBinding BindingList SBindingList .
  subsort SBinding < Binding SBindingList < BindingList .
  op none : -> SBindingList .
  op _,_ : SBindingList SBindingList -> SBindingList [assoc id: none prec 71] .
  op _,_ :  BindingList  BindingList ->  BindingList [ditto] .
  op _=_ : Name SExp -> SBinding [prec 70] .
  op _=_ : Name  Exp ->  Binding [ditto] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : SBindingList SExp -> SExp .
  op let_in_ :  BindingList  Exp ->  Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
```

```
  op proc__ : NameList Exp -> SExp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : SBindingList SExp -> SExp .
  op letrec_in_ :  BindingList  Exp ->  Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
```

```
  sorts Value ValueList .
  subsort Value < ValueList .
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .
```

```
  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- noValues] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* noValues] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending STATE .
  op int : Int -> Value .
  op eval : Exp State -> [ValueStatePair] .
  op eval : ExpList State -> [ValueListStatePair] .
  op eval : Exp -> [Value] .
```

```
  var X : Name . var I : Int . vars S Se Sl : State . var Ve : Value .
  var Vl : ValueList .  vars E E' : Exp .  var El : ExpList .
  eq eval(X, S) = {S[X], S} .
  eq eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {noValues,S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' Be Be' : Exp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending BEXP-SEMANTICS .
  var El : ExpList .  vars S S' Sl : State .  var I : Int .
  var Vl : ValueList .  var V : Value .  vars E E' : Exp .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El, S) .
 ceq eval(car(E), S) = {V,Sl} if {[V,Vl],Sl} := eval(E,S) .
 ceq eval(cdr(E), S) = {[Vl],Sl} if {[V,Vl],Sl} := eval(E,S) .
```

```
 ceq eval(cons(E,E'), S) = {[V,Vl],Sl}  if {(V,[Vl]),Sl} := eval((E,E'),S) .
  eq eval(emptylist, S) = {[noValues],S} .
 ceq eval(null?(E), S) = {bool(Vl == noValues), Sl} if {[Vl],Sl} := eval(E,S) .
 ceq eval(number?(E), S) = {bool(true),S'} if {int(I),S'} := eval(E,S) .
 ceq eval(number?(E), S) = {bool(false),S'} if {V,S'} := eval(E,S) [owise] .
endfm

fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' Be : Exp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
```

```
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending BEXP-SEMANTICS .
  extending LIST-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
endfm



-----------
--- CPS ---
-----------

fmod CPS-STATE is
  extending GENERIC-EXP-SYNTAX .
  extending CONVERSION .
  sorts CpsStateAttribute CpsState .
  subsort CpsStateAttribute < CpsState .
  sorts SExpListCpsStatePair ExpCpsStatePair .
  op {_,_} : SExpList CpsState -> SExpListCpsStatePair .
  op {_,_} : ExpList CpsState -> ExpCpsStatePair .
  op empty : -> CpsState .
  op __ : CpsState CpsState -> CpsState [assoc comm id: empty] .
  op initCpsState : -> CpsState .
  op nextSym : Qid Nat -> CpsStateAttribute .
```

```
  op genSym : Qid CpsState -> ExpCpsStatePair .
  var Q : Qid .  var N : Nat .  var S : CpsState .
  eq initCpsState = empty .
  eq genSym(Q, nextSym(Q,N) S) =
     {qid(string(Q) + string(N,10)), nextSym(Q, N + 1) S} .
  eq genSym(Q, S) = {qid(string(Q) + string(0,10)), nextSym(Q, 1) S} .
endfm

fmod CPS-BASIC is protecting CPS-STATE .
  extending PROC-SYNTAX .
  op nonSimple : Exp -> Bool .
  op cps : Exp -> Exp .
  op [_->_,_] : Exp Exp CpsState -> [ExpCpsStatePair] .
  op [_,_] : SExpList CpsState -> [SExpListCpsStatePair] .
  vars E E' : Exp .  vars S S' Sk : CpsState .
  vars Es Es' K : SExp .  var Xk : Name .
  eq nonSimple(Es) = false .
  eq nonSimple(E) = true [owise] .
 ceq cps(E) = proc(Xk) E'
    if {Xk,Sk} := genSym('k,initCpsState) /\ {E',S'} := [E -> Xk, Sk] .
 ceq [Es -> K, S] = {K(Es'),S'} if {Es',S'} := [Es,S] .
endfm

fmod GENERIC-EXP-CPS is extending CPS-BASIC .
  vars S S' Sl : CpsState .  vars Els Els' : SExpList .
  vars Es Es' : SExp .  var X : Name .  var I : Int .
  eq [X,S] = {X,S} .
  eq [I,S] = {I,S} .
  eq [(),S] = {(),S} .
 ceq [(Els,Es),S] = {(Els',Es'), S'}
    if Els =/= () /\ {Els',Sl} := [Els,S] /\ {Es',S'} := [Es,Sl] .
endfm

fmod ARITH-OPS-CPS is extending ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-CPS .
  vars E1 E2 Ep : Exp .  vars K Es1 Es2 Es1' Es2' : SExp .
  vars S Sp Sv S' : CpsState .  var V : Name .
 ceq [E1 + E2 -> K, S] = [E1 -> proc(V) Ep, Sp]
    if nonSimple(E1) /\ {V,Sv} := genSym('v,S)
    /\ {Ep,Sp} := [V + E2 -> K, Sv] .
 ceq [Es1 + Es2, S] = {Es1' + Es2', S'}
    if {(Es1',Es2'), S'} := [(Es1,Es2), S] .
---> define subtraction and multiplication only (no division)
```

```
endfm

fmod BEXP-CPS is extending BEXP-SYNTAX .
  protecting GENERIC-EXP-CPS .
  vars E E1 E2 Ep : Exp .  vars K Es Es1 Es2 Es' Es1' Es2' : SExp .
  vars S Sp Sv S' : CpsState .  var V : Name .
 ceq [zero?(E) -> K, S] = [E -> proc(V)(K(zero?(V))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [zero?(Es), S] = {zero?(Es'), S'} if {Es', S'} := [Es, S] .
---> define the other operations in BEXP-SYNTAX
endfm

fmod LIST-CPS is extending LIST-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Els Els' : SExpList .  vars E E' Ep : Exp .  var El : ExpList .
  vars K Es Es' Es1 Es1' Es2 Es2' : SExp .  var V : Name .
  var S S' Sv Sp : CpsState .
 ceq [list(Els,E,El) -> K, S] = [E -> proc(V) Ep, Sp]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [list(Els,V,El) -> K, Sv] .
 ceq [list(Els), S] = {list(Els'),S'} if {Els',S'} := [Els,S] .
 ceq [car(E) -> K, S] = [E -> proc(V)(K(car(V))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [car(Es), S] = {car(Es'), S'} if {Es',S'} := [Es, S] .
---> define the other operations in LIST-SYNTAX
endfm

fmod IF-CPS is protecting IF-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Be E1 E2 E1' E2' Ep : Exp .
  vars K Bes Bes' Es1 Es2 Es1' Es2' : SExp .
  vars S Sp Sb Sv S1 S1' S' : CpsState .  var V : Name .
 ceq [if Be then E1 else E2 -> K, S] = ...
 ceq [if Bes then E1 else E2 -> K, S] = ...
---> complete this module
endfm

fmod BINDING-CPS is protecting BINDING-SYNTAX .
  extending CPS-BASIC .
  sort SBindingListCpsStatePair .
  op {_,_} : SBindingList CpsState -> SBindingListCpsStatePair .
  op [_,_] : SBindingList CpsState -> [SBindingListCpsStatePair] .
  vars S S' Sbl : CpsState .  var X : Name .  vars Es Es' : SExp .
```

```
  vars Bls Bls' : SBindingList .
  eq [none, S] = {none, S} .
 ceq [(Bls, X = Es), S] = {(Bls', X = Es'), S'}
    if {Bls',Sbl} := [Bls, S] /\ {Es',S'} := [Es,Sbl] .
endfm

fmod LET-CPS is extending LET-SYNTAX .
  extending BINDING-CPS .
  vars Bls Bls' : SBindingList .  vars X V Xk : Name .
  var E E' Ep : Exp .  vars S Sp Sv Sbl S' Sk : CpsState .
  var K Es Es' : SExp .  var Bl : BindingList .
 ceq [let Bls, X = E', Bl in E -> Xk, S] = [E' -> proc(V) Ep, Sp]
    if nonSimple(E') /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [let Bls, X = V, Bl in E -> Xk, Sv] .
 ceq [let Bls, X = E', Bl in E -> K, S] = {let Xk = K in Ep, Sp}
    if nonSimple(E') /\ {Xk,Sk} := genSym('k,S)
    /\ {Ep, Sp} := [let Bls, X = E', Bl in E -> Xk, Sk] [owise] .
 ceq [let Bls in E -> K, S] = {let Bls' in E', S'}
    if ...
 ceq [let Bls in Es, S] = ...
---> complete this module
endfm

fmod PROC-CPS is extending PROC-SYNTAX .
  extending CPS-BASIC .
  vars F E Ep : Exp .  var El : ExpList .  var V : Name .
  vars S Sv Sk Sp S' : CpsState .  vars K Fs Fs' : SExp .
  vars Els Els' : SExpList .  var Xl : NameList .
 ceq [F(El) -> K, S] = ...
 ceq [Fs(Els,E,El) -> K, S] = ...
 ceq [Fs(Els) -> K, S] = {Fs'(Els',K), S'}
    if {(Fs',Els'), S'} := [(Fs,Els), S] .
 ceq [proc(Xl) E, S] = ...
 ceq [Fs(Els), S] = ...
---> complete this module
endfm

fmod LETREC-CPS is ...
---> define this module; how different is it from LET-CPS?
endfm

fmod PROG-LANG-CPS is
  extending ARITH-OPS-CPS .
```

```
  extending BEXP-CPS .
  extending LIST-CPS .
  extending IF-CPS .
  extending LET-CPS .
  extending PROC-CPS .
  extending LETREC-CPS .
endfm

fmod EVAL-CPS is
  protecting PROG-LANG-SEMANTICS .
  protecting PROG-LANG-CPS .
  op evalCps : Exp -> Value .
  vars E : Exp .
  eq evalCps(E) = eval(cps(E) (proc(x) x)) .
endfm

red [proc(x,y,z) x + y + z, initCpsState] .
red [f(x) -> k, initCpsState] .
red [f(x - 1) -> k, initCpsState] .
red [(proc(x)e) (x * y) -> k, initCpsState] .
red [g(proc(a)(f(a,b))) -> k, initCpsState] .
red [proc(x,y) (f(x,y)) -> k, initCpsState] .

red [f(x) + y -> k, initCpsState] .
red [f(g(x)) + y -> k, initCpsState] .
red [f(x) + g(y) -> k, initCpsState] .
red [f(a,b) + g(proc(a)(f(a,b))) -> k, initCpsState] .

red [if f(x) + 1 then y else z -> k, initCpsState] .
red [let z = f(x), u = z in g(z,u) -> k, initCpsState] .


red evalCps(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red evalCps(
  let x = 1
  in let x = x + 2
     in x + 1
) .
```

```
***> should be 4

red evalCps(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red evalCps(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red evalCps(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red evalCps(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red evalCps(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red evalCps(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red evalCps(
```

```
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red evalCps(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red evalCps(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red evalCps(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red evalCps(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red evalCps(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping
```

```
red evalCps(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red evalCps(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red evalCps(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red evalCps(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red evalCps(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
```

```
    in 'times4(3)
) .
***> should be 12

red evalCps(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red evalCps(
  letrec 'times4 = proc(x)
                   if zero?(x)
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red evalCps(
  letrec 'even = proc(x)
                 if zero?(x)
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if zero?(x)
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red evalCps(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined
```

```
red evalCps(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red evalCps(
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
) .
***>  should be 2

red evalCps(
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red evalCps(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

---------------

red evalCps(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
```

```
***> should be 120

red evalCps(
  let x = 17,
      'odd = proc(x, o, e)
               if zero?(x) then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if zero?(x) then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red evalCps(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red evalCps(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red evalCps( letrec f = proc(x) z + x + 5,
               y = 2,
               a = 3,
               z = let y = 5, a = 6 in y + a
         in f(a)
) .
***> should be 19

----------

red evalCps(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(100, 1)
) .
```

```
red evalCps(
  letrec f = proc(n)
             if zero?(n)
             then 1
             else n * f(n - 1)
  in f(100)
) .

red evalCps(
  letrec f = proc(n,l)
             if null?(l) then -1
             else if n equals car(l)
                  then 1
                  else let p = f(n, cdr(l))
                       in if p equals -1 then -1
                          else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

red evalCps(
  letrec r = proc(n,l)
             if null?(l) then emptylist
             else if n equals car(l)
                  then r(n,cdr(l))
                  else cons(car(l), r(n,cdr(l)))
  in r(3, list(3,1,3,2,3,3,3,4,3,5,3,3))
) .

red evalCps(
  letrec (a = proc(n,o,l)
              if null?(l)
              then emptylist
              else cons(b(n,o,car(l)), a(n,o,cdr(l)))),
         (b = proc(n,o,x)
              if number?(x)
              then if x equals o then n else x
              else a(n,o,x))
  in (a(7, 3, list(1, list(3,2,3), 3)))
) .
```

```
*************************
*** CPS Transformation ***
*************************

--------------
--- Syntax ---
--------------

fmod NAME-SYNTAX is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME-SYNTAX .
  protecting INT .
  sorts SExp Exp SExpList ExpList .
  subsorts Int Name < SExp < Exp SExpList < ExpList .
  subsort NameList < SExpList .
  op _,_ : SExpList SExpList -> SExpList [ditto] .
  op _,_ :  ExpList  ExpList ->  ExpList [ditto] .
endfm

fmod ARITH-OPS-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : SExp SExp -> SExp [ditto] .
  op _+_ :  Exp  Exp ->  Exp [ditto] .
  op _-_ : SExp SExp -> SExp [ditto] .
  op _-_ :  Exp  Exp ->  Exp [ditto] .
  op _*_ : SExp SExp -> SExp [ditto] .
  op _*_ :  Exp  Exp ->  Exp [ditto] .
endfm

fmod BEXP-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op _equals_ : SExp SExp -> SExp [comm] .
  op _equals_ :  Exp  Exp ->  Exp [ditto] .
  op zero? : SExp -> SExp .
  op zero? :  Exp ->  Exp .
  op even? : SExp -> SExp .
  op even? :  Exp ->  Exp .
  op not_ : SExp -> SExp .
  op not_ :  Exp ->  Exp .
  op _and_ : SExp SExp -> SExp [comm] .
  op _and_ :  Exp  Exp ->  Exp [ditto] .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : SExpList -> SExp .
  op list_ :  ExpList ->  Exp .
  op car : SExp -> SExp .
  op car :  Exp ->  Exp .
  op cdr : SExp -> SExp .
  op cdr :  Exp ->  Exp .
  op cons : SExp SExp -> SExp .
  op cons :  Exp  Exp ->  Exp .
  op emptylist : -> SExp .
  op null? : SExp -> SExp .
  op null? :  Exp ->  Exp .
  op number? : SExp -> SExp .
  op number? :  Exp ->  Exp .
endfm

fmod IF-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op if_then_else_ : SExp SExp SExp -> SExp .
  op if_then_else_ :  Exp  Exp  Exp ->  Exp .
endfm

fmod BINDING-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  sorts Binding SBinding BindingList SBindingList .
  subsort SBinding < Binding SBindingList < BindingList .
  op none : -> SBindingList .
  op _,_ : SBindingList SBindingList -> SBindingList [assoc id: none prec 71] .
  op _,_ :  BindingList  BindingList ->  BindingList [ditto] .
  op _=_ : Name SExp -> SBinding [prec 70] .
  op _=_ : Name  Exp ->  Binding [ditto] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : SBindingList SExp -> SExp .
  op let_in_ :  BindingList  Exp ->  Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> SExp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : SBindingList SExp -> SExp .
  op letrec_in_ :  BindingList  Exp ->  Exp .
endfm

fmod PROG-LANG-SYNTAX is
  extending ARITH-OPS-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
endfm


------------------
--- Semantics ---
------------------

fmod LOCATION is
  protecting INT .
  sorts Location .
  op loc : Nat -> Location .
endfm

fmod ENVIRONMENT is protecting NAME-SYNTAX .
  protecting LOCATION .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env Name Location -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  eq ([X,L] Env)[X <- L'] = [X,L'] Env .
  eq Env[X <- L] = Env [X,L] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noValues : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noValues] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_] : Store Location -> Value .
  op _[_<-_] : Store Location Value -> Store .
  var L : Location .  var St : Store .  vars V V' : Value .
  eq ([L,V] St)[L] = V .
  eq ([L,V] St)[L <- V'] = [L,V'] St .
  eq St[L <- V'] = St [L,V'] [owise] .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .

  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _<**_ : State StateAttribute -> State [gather (E e)] .

  sorts ValueStatePair ValueListStatePair LocationStatePair .
  subsort ValueStatePair < ValueListStatePair .
  op {_,_} : Value State -> ValueStatePair .
  op {_,_} : ValueList State -> ValueListStatePair .

  op _{_} : State Name -> Location .
  op _{_}{_} : State Name Name -> Location .
  op _[_] : State Name -> Value .
  op _[_<-_] : State NameList ValueList -> State .
  op _[_<*_] : State NameList ValueList -> State .
  op _[_<- ?] : State NameList -> State .
  op initState : -> State .

  vars S S' : State .  var L : Location .  var N : Nat .
  var X : Name .  var Xl : NameList .  vars Env Env' : Env .
  var V : Value .  var Vl : ValueList .  vars St St' : Store .

  eq (env([X,L] Env) S){X} = L .

  eq S[X] = store(S)[S{X}] .

  eq S[() <- noValues] = S .
  eq (env(Env) store(St) nextLoc(N) S)[(X,Xl) <- (V,Vl)] =
     (env(Env[X <- loc(N)]) store(St[loc(N) <- V])
      nextLoc(N + 1) S)[Xl <- Vl] .

  eq S[() <* noValues] = S .
  eq S[(X,Xl) <* (V,Vl)] = (S <** store(store(S)[S{X} <- V]))[Xl <* Vl] .

  eq S[() <- ?] = S .
  eq (env(Env) nextLoc(N) S)[(X,Xl) <- ?] =
     (env(Env[X <- loc(N)]) nextLoc(N + 1) S)[Xl <- ?] .
  eq initState = env(noEnv) store(noStore) nextLoc(0) .

  op nextLoc : Nat -> StateAttribute .

--- the following are generic and could be done via
--- a proper instantiation of a parameterized module
  op env : Env -> StateAttribute .
  op env : State -> Env .
  eq (env(Env) S) <** env(Env') = env(Env') S .
  eq env(env(Env) S) = Env .

  op store : Store -> StateAttribute .
  op store : State -> Store .
  eq (store(St) S) <** store(St') = store(St') S .
  eq store(store(St) S) = St .
endfm

fmod GENERIC-EXP-SEMANTICS is extending GENERIC-EXP-SYNTAX .
  extending STATE .
  op int : Int -> Value .
  op eval : Exp State -> [ValueStatePair] .
  op eval : ExpList State -> [ValueListStatePair] .
  op eval : Exp -> [Value] .
  var X : Name . var I : Int . vars S Se Sl : State . var Ve : Value .
  var Vl : ValueList .  vars E E' : Exp .  var El : ExpList .
  eq eval(X, S) = {S[X], S} .
  eq eval(I, S) = {int(I),S} .
 ceq eval(E) = Ve if {Ve,Se} := eval(E, initState) .
  eq eval((),S) = {noValues,S} .
 ceq eval((E,E',El), S) = {(Ve,Vl), Sl}
    if {Ve,Se} := eval(E,S) /\ {Vl,Sl} := eval((E',El),Se) .
endfm

fmod ARITH-OPS-SEMANTICS is extending ARITH-OPS-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  vars S Se Se' : State .  vars Ie Ie' : Int .
 ceq eval(E + E', S) = {int(Ie + Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E - E', S) = {int(Ie - Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(E * E', S) = {int(Ie * Ie'),Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
endfm

fmod BEXP-SEMANTICS is extending BEXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  vars E E' Be Be' : Exp .  vars S Sb Sb' Se Se' : State .
  vars Ie Ie' : Int .  vars B B' : Bool .
  op bool : Bool -> Value .
 ceq eval(E equals E', S) = {bool(Ie == Ie'), Se'}
    if {int(Ie),Se} := eval(E,S) /\ {int(Ie'),Se'} := eval(E',Se) .
 ceq eval(zero?(E), S) = {bool(Ie == 0), Se} if {int(Ie),Se} := eval(E,S) .
 ceq eval(even?(E), S) = {bool(Ie rem 2 == 0), Se} if
    {int(Ie), Se} := eval(E, S) .
 ceq eval(not(Be), S) = {bool(not(B)),Sb} if {bool(B),Sb} := eval(Be,S) .
 ceq eval(Be and Be', S) = {bool(B and B'), Sb'}
    if {bool(B),Sb} := eval(Be, S) /\ {bool(B'),Sb'} := eval(Be',Sb) .
endfm

fmod LIST-SEMANTICS is extending LIST-SYNTAX .
  extending BEXP-SEMANTICS .
  var El : ExpList .  vars S S' Sl : State .  var I : Int .
  var Vl : ValueList .  var V : Value .  vars E E' : Exp .
 ceq eval(list(El), S) = {[Vl],Sl} if {Vl,Sl} := eval(El, S) .
 ceq eval(car(E), S) = {V,Sl} if {[V,Vl],Sl} := eval(E,S) .
 ceq eval(cdr(E), S) = {[Vl],Sl} if {[V,Vl],Sl} := eval(E,S) .
 ceq eval(cons(E,E'), S) = {[V,Vl],Sl}  if {(V,[Vl]),Sl} := eval((E,E'),S) .
  eq eval(emptylist, S) = {[noValues],S} .
 ceq eval(null?(E), S) = {bool(Vl == noValues), Sl} if {[Vl],Sl} := eval(E,S) .
 ceq eval(number?(E), S) = {bool(true),S'} if {int(I),S'} := eval(E,S) .
 ceq eval(number?(E), S) = {bool(false),S'} if {V,S'} := eval(E,S) [owise] .
endfm
```

```
fmod IF-SEMANTICS is extending IF-SYNTAX .
  extending BEXP-SEMANTICS .
  vars E E' Be : Exp .  vars S Sb : State .
 ceq eval(if Be then E else E', S) = eval(E, Sb)
    if {bool(true), Sb} := eval(Be,S) .
 ceq eval(if Be then E else E', S) = eval(E',Sb)
    if {bool(false), Sb} := eval(Be,S) .
endfm

fmod BINDING-SEMANTICS is extending BINDING-SYNTAX .
  op names_ : BindingList -> NameList .
  op exps_  : BindingList -> ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq names(X = E, Bl) = X, names(Bl) .
  eq names(none) = () .
  eq exps(X = E, Bl) = E, exps(Bl) .
  eq exps(none) = () .
endfm

fmod LET-SEMANTICS is extending LET-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var E : Exp .  var Bl : BindingList .  vars S Se Sl : State .
  var Ve : Value .  var Vl : ValueList .
 ceq eval(let Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S)
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <- Vl]) .
endfm

fmod PROC-SEMANTICS is extending PROC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op closure : NameList Exp Env -> Value .
  var Xl : NameList .  vars E F : Exp .  vars S Se Sl Sf : State .
  var El : ExpList .  var Env : Env .
  vars V Ve : Value .  var Vl : ValueList .
  eq eval(proc(Xl) E, S) = {closure(Xl, E, env(S)), S} .
 ceq eval(F(El), S) = {Ve, Se <** env(env(S))}
    if {closure(Xl, E, Env), Sf} := eval(F,S)
    /\ {Vl,Sl} := eval(El,Sf)
    /\ {Ve,Se} := eval(E, (Sl <** env(Env))[Xl <- Vl]) .
endfm

fmod LETREC-SEMANTICS is extending LETREC-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  extending BINDING-SEMANTICS .
  var X : Name .  var Bl : BindingList .  var E : Exp .
  vars S Se Sl : State .  var Ve : Value .  var Vl : ValueList .
 ceq eval(letrec Bl in E, S) = {Ve, Se <** env(env(S))}
    if {Vl,Sl} := eval(exps(Bl), S[names(Bl) <- ?])
    /\ {Ve,Se} := eval(E, Sl[names(Bl) <* Vl]) .
endfm

fmod PROG-LANG-SEMANTICS is extending PROG-LANG-SYNTAX .
  extending ARITH-OPS-SEMANTICS .
  extending BEXP-SEMANTICS .
  extending LIST-SEMANTICS .
  extending IF-SEMANTICS .
  extending LET-SEMANTICS .
  extending PROC-SEMANTICS .
  extending LETREC-SEMANTICS .
endfm


-----------
--- CPS ---
-----------


fmod CPS-STATE is
  extending GENERIC-EXP-SYNTAX .
  extending CONVERSION .
  sorts CpsStateAttribute CpsState .
  subsort CpsStateAttribute < CpsState .
  sorts SExpListCpsStatePair ExpCpsStatePair .
  op {_,_} : SExpList CpsState -> SExpListCpsStatePair .
  op {_,_} : ExpList CpsState -> ExpCpsStatePair .
  op empty : -> CpsState .
  op __ : CpsState CpsState -> CpsState [assoc comm id: empty] .
  op initCpsState : -> CpsState .
  op nextSym : Qid Nat -> CpsStateAttribute .
  op genSym : Qid CpsState -> ExpCpsStatePair .
  var Q : Qid .  var N : Nat .  var S : CpsState .
  eq initCpsState = empty .
  eq genSym(Q, nextSym(Q,N) S) =
     {qid(string(Q) + string(N,10)), nextSym(Q, N + 1) S} .
  eq genSym(Q, S) = {qid(string(Q) + string(0,10)), nextSym(Q, 1) S} .
endfm

fmod CPS-BASIC is protecting CPS-STATE .
  extending PROC-SYNTAX .
  op nonSimple : Exp -> Bool .
  op cps : Exp -> Exp .
  op [_->_,_] : Exp Exp CpsState -> [ExpCpsStatePair] .
  op [_,_] : SExpList CpsState -> [SExpListCpsStatePair] .
  vars E E' : Exp .  vars S S' Sk : CpsState .
  vars Es Es' K : SExp .  var Xk : Name .
  eq nonSimple(Es) = false .
  eq nonSimple(E) = true [owise] .
 ceq cps(E) = proc(Xk) E'
    if {Xk,Sk} := genSym('k,initCpsState) /\ {E',S'} := [E -> Xk, Sk] .
 ceq [Es -> K, S] = {K(Es'),S'} if {Es',S'} := [Es,S] .
endfm

fmod GENERIC-EXP-CPS is extending CPS-BASIC .
  vars S S' Sl : CpsState .  vars Els Els' : SExpList .
  vars Es Es' : SExp .  var X : Name .  var I : Int .
  eq [X,S] = {X,S} .
  eq [I,S] = {I,S} .
  eq [(),S] = {(),S} .
 ceq [(Els,Es),S] = {(Els',Es'), S'}
    if Els =/= () /\ {Els',Sl} := [Els,S] /\ {Es',S'} := [Es,Sl] .
endfm

fmod ARITH-OPS-CPS is extending ARITH-OPS-SYNTAX .
  protecting GENERIC-EXP-CPS .
  vars E1 E2 Ep : Exp .  vars K Es1 Es2 Es1' Es2' : SExp .
  vars S Sp Sv S' : CpsState .  var V : Name .
 ceq [E1 + E2 -> K, S] = [E1 -> proc(V) Ep, Sp]
    if nonSimple(E1) /\ {V,Sv} := genSym('v,S)
    /\ {Ep,Sp} := [V + E2 -> K, Sv] .
 ceq [Es1 + Es2, S] = {Es1' + Es2', S'}
    if {(Es1',Es2'), S'} := [(Es1,Es2), S] .
---> define subtraction and multiplication only (no division)
endfm

fmod BEXP-CPS is extending BEXP-SYNTAX .
  protecting GENERIC-EXP-CPS .
  vars E E1 E2 Ep : Exp .  vars K Es Es1 Es2 Es' Es1' Es2' : SExp .
  vars S Sp Sv S' : CpsState .  var V : Name .
 ceq [zero?(E) -> K, S] = [E -> proc(V)(K(zero?(V)))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [zero?(Es), S] = {zero?(Es'), S'} if {Es', S'} := [Es, S] .
---> define the other operations in BEXP-SYNTAX
endfm

fmod LIST-CPS is extending LIST-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Els Els' : SExpList .  vars E E' Ep : Exp .  var El : ExpList .
  vars K Es Es' Es1 Es1' Es2 Es2' : SExp .  var V : Name .
  var S S' Sv Sp : CpsState .
 ceq [list(Els,E,El) -> K, S] = [E -> proc(V) Ep, Sp]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [list(Els,V,El) -> K, Sv] .
 ceq [list(Els), S] = {list(Els'),S'} if {Els',S'} := [Els,S] .
 ceq [car(E) -> K, S] = [E -> proc(V)(K(car(V)))), Sv]
    if nonSimple(E) /\ {V,Sv} := genSym('v,S) .
 ceq [car(Es), S] = {car(Es'), S'} if {Es',S'} := [Es, S] .
---> define the other operations in LIST-SYNTAX
endfm

fmod IF-CPS is protecting IF-SYNTAX .
  extending GENERIC-EXP-CPS .
  vars Be E1 E2 E1' E2' Ep : Exp .
  vars K Bes Bes' Es1 Es2 Es1' Es2' : SExp .
  vars S Sp Sb Sv S1 S1' S' : CpsState .  var V : Name .
 ceq [if Be then E1 else E2 -> K, S] = ...
 ceq [if Bes then E1 else E2 -> K, S] = ...
---> complete this module
endfm

fmod BINDING-CPS is protecting BINDING-SYNTAX .
  extending CPS-BASIC .
  sort SBindingListCpsStatePair .
  op {_,_} : SBindingList CpsState -> SBindingListCpsStatePair .
  op [_,_] : SBindingList CpsState -> [SBindingListCpsStatePair] .
  vars S S' Sbl : CpsState .  var X : Name .  vars Es Es' : SExp .
  vars Bls Bls' : SBindingList .
  eq [none, S] = {none, S} .
 ceq [(Bls, X = Es), S] = {(Bls', X = Es'), S'}
    if {Bls',Sbl} := [Bls, S] /\ {Es',S'} := [Es,Sbl] .
endfm

fmod LET-CPS is extending LET-SYNTAX .
  extending BINDING-CPS .
  vars Bls Bls' : SBindingList .  vars X V Xk : Name .
  var E E' Ep : Exp .  vars S Sp Sv Sbl S' Sk : CpsState .
  var K Es Es' : SExp .  var Bl : BindingList .
 ceq [let Bls, X = E', Bl in E -> Xk, S] = [E' -> proc(V) Ep, Sp]
    if nonSimple(E') /\ {V,Sv} := genSym('v,S)
    /\ {Ep, Sp} := [let Bls, X = V, Bl in E -> Xk, Sv] .
 ceq [let Bls, X = E', Bl in E -> K, S] = {let Xk = K in Ep, Sp}
    if nonSimple(E') /\ {Xk,Sk} := genSym('k,S)
    /\ {Ep, Sp} := [let Bls, X = E', Bl in E -> Xk, Sk] [owise] .
 ceq [let Bls in E -> K, S] = {let Bls' in E', S'}
    if ...
 ceq [let Bls in Es, S] = ...
---> complete this module
endfm

fmod PROC-CPS is extending PROC-SYNTAX .
  extending CPS-BASIC .
  vars F E Ep : Exp .  var El : ExpList .  var V : Name .
  vars S Sv Sk Sp S' : CpsState .  vars K Fs Fs' : SExp .
  vars Els Els' : SExpList .  var Xl : NameList .
 ceq [F(El) -> K, S] = ...
 ceq [Fs(Els,E,El) -> K, S] = ...
 ceq [Fs(Els) -> K, S] = {Fs'(Els',K), S'}
    if {(Fs',Els'), S'} := [(Fs,Els), S] .
 ceq [proc(Xl) E, S] = ...
 ceq [Fs(Els), S] = ...
---> complete this module
endfm

fmod LETREC-CPS is ...
---> define this module; how different is it from LET-CPS?
endfm

fmod PROG-LANG-CPS is
  extending ARITH-OPS-CPS .
  extending BEXP-CPS .
  extending LIST-CPS .
  extending IF-CPS .
  extending LET-CPS .
  extending PROC-CPS .
  extending LETREC-CPS .
endfm

fmod EVAL-CPS is
  protecting PROG-LANG-SEMANTICS .
  protecting PROG-LANG-CPS .
  op evalCps : Exp -> Value .
  vars E : Exp .
  eq evalCps(E) = eval(cps(E) (proc(x) x)) .
endfm

red [proc(x,y,z) x + y + z, initCpsState] .
red [f(x) -> k, initCpsState] .
red [f(x - 1) -> k, initCpsState] .
red [(proc(x)e) (x * y) -> k, initCpsState] .
red [g(proc(a)(f(a,b))) -> k, initCpsState] .
red [proc(x,y) (f(x,y)) -> k, initCpsState] .

red [f(x) + y -> k, initCpsState] .
red [f(g(x)) + y -> k, initCpsState] .
red [f(x) + g(y) -> k, initCpsState] .
red [f(a,b) + g(proc(a)(f(a,b))) -> k, initCpsState] .

red [if f(x) + 1 then y else z -> k, initCpsState] .
red [let z = f(x), u = z in g(z,u) -> k, initCpsState] .


red evalCps(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red evalCps(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red evalCps(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red evalCps(
  let x = 1
  in let z = let y = x + 4
             in y
```

```
    in z
) .
***> should be 5

red evalCps(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red evalCps(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red evalCps(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red evalCps(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red evalCps(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red evalCps(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red evalCps(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red evalCps(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red evalCps(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red evalCps(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red evalCps(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red evalCps(
  let f = proc(n)
            if zero?(n)
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red evalCps(
  let f = proc(n) n + n
  in let f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red evalCps(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red evalCps(
  let 'makemult = proc('maker, x)
                    if zero?(x)
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red evalCps(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

red evalCps(
  letrec 'times4 = proc(x)
                     if zero?(x)
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red evalCps(
  letrec 'even = proc(x)
                   if zero?(x)
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if zero?(x)
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red evalCps(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red evalCps(
  let x = 10
  in letrec f = proc(y) if zero?(y) then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red evalCps(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red evalCps(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red evalCps(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if zero?(a) then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

---------------

red evalCps(
  let f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red evalCps(
  let x = 17,
      'odd = proc(x, o, e)
                if zero?(x) then 0
                else e(x - 1, o, e),
      'even = proc(x, o, e)
                if zero?(x) then 1
                else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red evalCps(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red evalCps(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red evalCps( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red evalCps(
  letrec f = proc(n,m)
               if zero?(n)
               then m
               else f(n - 1, m * n)
  in f(100, 1)
) .

red evalCps(
  letrec f = proc(n)
               if zero?(n)
               then 1
               else n * f(n - 1)
  in f(100)
) .

red evalCps(
  letrec f = proc(n,l)
               if null?(l) then -1
               else if n equals car(l)
                    then 1
                    else let p = f(n, cdr(l))
                         in if p equals -1 then -1
                            else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .
```

```
red evalCps(
  letrec r = proc(n,l)
               if null?(l) then emptylist
               else if n equals car(l)
                    then r(n,cdr(l))
                    else cons(car(l), r(n,cdr(l)))
  in r(3, list(3,1,3,2,3,3,3,4,3,5,3,3))
) .

red evalCps(
  letrec (a = proc(n,o,l)
               if null?(l)
               then emptylist
               else cons(b(n,o,car(l)), a(n,o,cdr(l)))),
         (b = proc(n,o,x)
               if number?(x)
               then if x equals o then n else x
               else a(n,o,x))
  in (a(7, 3, list(1, list(3,2,3), 3)))
) .
```

CS322 - Programming Language Design

Lecture 22: Operational Semantics

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

By an *operational semantics* of a programming language, one typically understands a collection of rules specifying how its expressions and statements are evaluated/executed. These rules say how a possible implementation of a programming language should "operate".

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics.

For example, *all* our Maude definitions discussed so far in the class reflect operational semantics of languages or analysis tools, where the operational engine is based on an efficient implementation of equational reasoning by rewriting. We often called our semantics *equational* or/and *executable*, to keep it distinct from the more usual operational semantics formalism discussed in this lecture.

A Simple Language and its Executable Semantics

Let us next consider a very simple non-procedural imperative language in the style of the first language that we discussed in Lecture 4, namely one which has arithmetic and boolean expressions, conditionals and while loops. To keep the definition simple, we do not modularize it for this example language. Formally, its syntax is given by the following module:

```
fmod SYNTAX is protecting QID .
  protecting INT .  protecting BOOL .
  sorts Name AExp BExp Stmt Pgm .
  subsort Qid < Name .  subsorts Int Name < AExp .
  subsort Bool < BExp .
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
--- arithmetic expressions
  op _+_ : AExp AExp -> AExp [ditto] .
```



```
  op _-_ : AExp AExp -> AExp [ditto] .
  op _*_ : AExp AExp -> AExp [ditto] .
  op _/_ : AExp AExp -> AExp [prec 31] .
--- boolean expressions
  op _<=_ : AExp AExp -> BExp [ditto] .
  op _>=_ : AExp AExp -> BExp [ditto] .
  op _equals_ : AExp AExp -> BExp .
  op _and_ : BExp BExp -> BExp [ditto] .
  op _or_ : BExp BExp -> BExp [ditto] .
  op not_ : BExp -> BExp [ditto] .
--- statements
  op skip : -> Stmt .
  op _=_ : Name AExp -> Stmt [prec 40] .
  op _;_ : Stmt Stmt -> Stmt [assoc] .
  op {_} : Stmt -> Stmt .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
--- programs
```

```
  op _;_ : Stmt AExp -> Pgm .
endfm
```

A program is a sequence of statements followed by an expression. The expression is evaluated in the state obtained after evaluating all the statements and its result is returned as the result of the evaluation of the entire program.

In order to give the executable semantics of a language, we need to first add the state infrastructure, which for such a simple language is straightforward (a map from names to integer values):

```
fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
```



```
  vars X : Index .  vars I I' : Int .  var S : State .
  eq ([X,I] S)[X] = I .
  eq S[X] = 0 [owise] .
  eq ([X,I'] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm
```

We can now add the executable semantics of the language.

```
fmod SEMANTICS is protecting SYNTAX .
  protecting STATE .
  subsort Name < Index .
  var X : Name .  var S : State .  vars A A' : AExp .
  var I : Int .  vars St St' : Stmt .  vars B B' : BExp .
--- arithmetic expressions
  op eval : AExp State -> Int .
  eq eval(X, S) = S[X] .
  eq eval(I, S) = I .
  eq eval(A + A', S) = eval(A, S) + eval(A', S) .
```

```
  eq eval(A - A', S) = eval(A, S) - eval(A', S) .
  eq eval(A * A', S) = eval(A, S) * eval(A', S) .
  eq eval(A / A', S) = eval(A, S) quo eval(A', S) .
--- boolean expressions
  op eval : BExp State -> Bool .
  eq eval(true, S) = true .  eq eval(false, S) = false .
  eq eval(A <= A', S) = eval(A, S) <= eval(A', S) .
  eq eval(A >= A', S) = eval(A, S) >= eval(A', S) .
  eq eval(A equals A', S) = eval(A, S) == eval(A', S) .
  eq eval(B and B', S) = eval(B, S) and eval(B', S) .
  eq eval(B or B', S) = eval(B, S) or eval(B', S) .
  eq eval(not B, S) = not eval(B, S) .
--- statements
  op state : Stmt State -> State .
  eq state(skip, S:State) = S:State .
  eq state(X = A, S) = S[X <- eval(A,S)] .
  eq state(St ; St', S) = state(St', state(St, S)) .
  eq state({St'}, S) = state(St', S) .
```



```
  eq state(if B then St else St', S) =
     if eval(B, S) then state(St, S) else state(St', S) fi .
  eq state(while B St, S) =
     if eval(B, S) then state(while B St, state(St, S)) else S fi .
--- programs
  op eval : Pgm -> Int .
  eq eval(St' ; A) = eval(A, state(St', empty)) .
endfm
```

From now on in this lecture, we call the executable semantics above, as well as all the similar semantics that we gave so far for languages or analysis tools, *equational semantics*. In what follows we define the more usual *structural operational semantics* of this language and show that it is entirely equivalent to the equational semantics.

Structural Operational Semantics (SOS)

Traditionally, the operational semantics of a language, typically called *structural* (and abbreviated *SOS*) because it is defined inductively over the structure of the syntax, is given by defining a *transition relation* on *configurations*. A configuration consists of a term over the syntax of the language and a state.

For this simple imperative language, a *state* is a map from names to integer numbers $\texttt{Name} \to \texttt{Int}$. We let σ, σ', etc., denote states. If σ is a state and x a name, then we let $\sigma[x]$ or $\sigma(x)$ denote the integer value to which σ maps x. Moreover, if x is a name and i an integer, then we let $\sigma[x \leftarrow i]$ denote the function $\texttt{Name} \to \texttt{Int}$ defined as follows:

$$\sigma[x \leftarrow i](y) = \begin{cases} \sigma(x) & \text{if } x \neq y, \\ i & \text{if } x = y. \end{cases}$$

We also let $\emptyset$ denote the initial state. One has two options at this point: one is to consider that $\emptyset[x]$ is some default initial value for any x, another is to consider states as *partial* functions and thus let $\emptyset[x]$ undefined.

SOS Rules for Arithmetic Expressions

We introduce a relation $\langle \_, \_ \rangle \to \_$ on triples of arithmetic expressions, states and integers, with the following intuition: $\langle a, \sigma \rangle \to i$, also called a *transition*, states that the arithmetic expression a evaluates/executes/transits to the integer i in state σ. The SOS for arithmetic expressions can then be given as a

collection of *parametric rules* of the form:

$$\frac{\mathit{transition}_1,\ \mathit{transition}_2,\ \ldots, \mathit{transition}_k}{\mathit{transition}}$$

The intuition here is that *transition* is possible whenever $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$ are possible. We may also say that *transition* is derivable, or can be inferred, from $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$. This reflects the fact that, like our previous equational semantics, SOS can also be viewed as a logic system, where one can deduce possible behaviors of programs.

If $k = 0$, then we simply write

$$\frac{\cdot}{\mathit{transition}}$$

In the case of our simple language, the transition relation is going to be deterministic, in the sense that whenever $\langle a, \sigma\rangle \to i_1$ and $\langle a, \sigma\rangle \to i_2$ can be deduced, then $i_1 = i_2$, because our language is deterministic. However, in the context of concurrent languages, as

we will see later, $\langle a, \sigma\rangle \to i$ states that a *may possibly* evaluate to i in state σ, but it may also evaluate to other integers.

As we did when we defined the equational semantics, we need to inductively define the transition relation for each language construct for arithmetic expressions. Since `Name` and Int are subsorts of `AExp`, we start by introducing the following two SOS rules, one for names and the other for integers:

$$\frac{\cdot}{\langle x, \sigma\rangle \to \sigma[x]} \tag{1}$$

$$\frac{\cdot}{\langle i, \sigma\rangle \to i} \tag{2}$$

We next give the SOS rules for the arithmetic operations of addition, subtraction, multiplication and division:

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 + a_2, \sigma\rangle \to i} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \quad (3)$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 - a_2, \sigma\rangle \to i} \text{ where } i \text{ is } i_1 \text{ minus } i_2 \quad (4)$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 * a_2, \sigma\rangle \to i} \text{ where } i \text{ is } i_1 \text{ times } i_2 \quad (5)$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 / a_2, \sigma\rangle \to i} \text{ where } i \text{ is the quotient of } i_1 \text{ by } i_2 \quad (6)$$

Note that all these rules are *parametric*, that it, they can be viewed as collections of concrete *instance rules*. A possible instance of rule (3) can be the following, which, of course, seems problematic:

$$\frac{\langle 1, \sigma\rangle \to 1,\ \langle 2, \sigma\rangle \to 9}{\langle 1 + 2, \sigma\rangle \to 10}$$

The rule above is indeed a correct instance of (3). However, one will never be able to infer $\langle 2, \sigma\rangle \to 9$, so this rule cannot be applied in a correct inference.

The following is a correct inference, where x and y are any names and σ is any state with $\sigma[x] = \sigma[y] = 1$:

$$\frac{\dfrac{\cdot}{\langle x, \sigma\rangle \to 1},\ \dfrac{\dfrac{\cdot}{\langle y, \sigma\rangle \to 1},\ \dfrac{\cdot}{\langle x, \sigma\rangle \to 1}}{\langle y * x, \sigma\rangle \to 1},\ \dfrac{\cdot}{\langle 2, \sigma\rangle \to 2}}{\langle y * x + 2, \sigma\rangle \to 3}}{\langle x - (y * x + 2), \sigma\rangle \to -2}$$

The proof above can be regarded as an upside-down tree, with dots as leaves and instances of SOS rules as nodes. This way, we have a way to mathematically derive facts about programs within their

semantics. We will later see how one can actually prove behavioral equivalence of programs.

SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma\rangle \to t$, where b is a boolean expression and t is a truth value in the set $\{true, false\}$:

$$\frac{\cdot}{\langle true, \sigma\rangle \to true} \tag{7}$$

$$\frac{\cdot}{\langle false, \sigma\rangle \to false} \tag{8}$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 \leq a_2, \sigma\rangle \to true} \text{ where } i_1 \text{ less than or equal to } i_2 \tag{9}$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 \leq a_2, \sigma\rangle \to false} \text{ where } i_1 \text{ larger than } i_2 \tag{10}$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 \geq a_2, \sigma\rangle \to true} \text{ for } i_1 \text{ larger than or equal to } i_2 \tag{11}$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 \geq a_2, \sigma\rangle \to false} \text{ where } i_1 \text{ smaller than } i_2 \tag{12}$$

$$\frac{\langle a_1, \sigma\rangle \to i,\ \langle a_2, \sigma\rangle \to i}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \to true} \tag{13}$$

$$\frac{\langle a_1, \sigma\rangle \to i_1,\ \langle a_2, \sigma\rangle \to i_2}{\langle a_1 \texttt{ equals } a_2, \sigma\rangle \to false} \text{ where } i_1 \text{ different from } i_2 \tag{14}$$

$$\frac{\langle b_1, \sigma\rangle \to true,\ \langle b_2, \sigma\rangle \to true}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \to true} \tag{15}$$

$$\frac{\langle b_1, \sigma\rangle \to t_1,\ \langle b_2, \sigma\rangle \to t_2}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \to \mathit{false}} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathit{false} \tag{16}$$

$$\frac{\langle b_1, \sigma\rangle \to \mathit{false}\ \langle b_2, \sigma\rangle \to \mathit{false}}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \mathit{false}} \tag{17}$$

$$\frac{\langle b_1, \sigma\rangle \to t_1,\ \langle b_2, \sigma\rangle \to t_2}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \mathit{true}} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathit{true} \tag{18}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \texttt{not } b, \sigma\rangle \to \mathit{true}} \tag{19}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{true}}{\langle \texttt{not } b, \sigma\rangle \to \mathit{false}} \tag{20}$$

SOS Rules for Statements

Statements in our simple imperative language change the state, so we need to introduce a new transition relation of the form $\langle s, \sigma\rangle \to \sigma'$, where s is a statement and σ, σ' are states. The SOS rules for statements are then:

$$\frac{\cdot}{\langle \texttt{skip}, \sigma\rangle \to \sigma} \tag{21}$$

$$\frac{\langle a, \sigma\rangle \to i}{\langle x = a, \sigma\rangle \to \sigma[x \leftarrow i]} \tag{22}$$

$$\frac{\langle s_1, \sigma\rangle \to \sigma'',\ \langle s_2, \sigma''\rangle \to \sigma'}{\langle s_1; s_2, \sigma\rangle \to \sigma'} \tag{23}$$

$$\frac{\langle s, \sigma\rangle \to \sigma'}{\langle \{s\}, \sigma\rangle \to \sigma'} \tag{24}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{true},\ \langle s_1, \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \to \sigma'} \tag{25}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{false},\ \langle s_2, \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma\rangle \to \sigma'} \tag{26}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{false}}{\langle \texttt{while}\ b\ s, \sigma\rangle \to \sigma} \tag{27}$$

$$\frac{\langle b, \sigma\rangle \to \mathit{true},\ \langle s, \sigma\rangle \to \sigma'',\ \langle \texttt{while}\ b\ s, \sigma''\rangle \to \sigma'}{\langle \texttt{while}\ b\ s, \sigma\rangle \to \sigma'} \tag{28}$$

SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

$$\frac{\langle s, \emptyset\rangle \to \sigma,\ \langle a, \sigma\rangle \to 1}{\langle s; a\rangle \to i} \tag{29}$$

Equivalence of Equational Semantics and SOS

So far we have discussed two different ways to give semantics to a programming language. One that we used several times by now in defining languages or tools was called *equational semantics*, and the other that was defined in terms of a transition relation that was called *structural operational semantics*.

Both semantics are very intuitive and assert more or less the same

thing, namely how the different language constructs are to be interpreted. There are however, slight technical differences between the two, especially due to the deterministic nature of our simple imperative language.

An important question when one has more than one semantics is whether the semantics are *equivalent* or not. Proving formally that two or more semantics are equivalent gives us a higher confidence that they are both correct. Each semantics may reflect a viewpoint on the meaning of a programming language, so proving semantics equivalent tells us that the different viewpoints at least agree.

Note that one can do little in practice to ensure that a specification, in particular a semantics of a programming language, is correct, because the specification itself is intended to state the correctness: an implementation is correct if and only if it satisfies the specification. Therefore, the best one can do is to define the semantics in a clean and intuitive language/formalism, which

allows one to confidently claim, without proof, that that semantics is indeed what one means.

There are techniques that check whether a specification is *consistent*, that is, if it at least admits a model (or an implementation). While such techniques increase a bit one's confidence in the correctness of a specification, in general they do not help much in practice, and in particular in programming languages, because there is no guarantee that the existing model is indeed the desired one.

Proving semantics equivalent is like a double-check of the intended meaning, especially when the two semantics are devised by different people. The following theorem states the equivalence of the equational semantics and SOS.

Theorem. *The equational semantics and the structural operational semantics of the simple imperative language are equivalent, that is,*

$$\langle p \rangle \to i \text{ if and only if } \texttt{eval}(p) = i$$

for any program p.

Homework Exercise 1 *Prove the theorem above.*
Hint. *Prove it by structural induction over the language constructs. First prove three lemmas,*

$$\langle a, \sigma \rangle \to i \text{ if and only if } \texttt{eval}(a, \sigma) = i$$

$$\langle b, \sigma \rangle \to t \text{ if and only if } \texttt{eval}(b, \sigma) = t$$

$$\langle s, \sigma \rangle \to \sigma' \text{ if and only if } \texttt{state}(s, \sigma) = \sigma'$$

stating the equivalence of the semantics for arithmetic expressions, boolean expressions, and statements, respectively. When proving the first, e.g., define a predicate $P(a)$ stating the equivalence above for any σ and any i, and then show that $P(a_1)$ and $P(a_2)$ implies

$P(a_1 + a_2)$, etc. Make sure that you use equational reasoning whenever you are in the equational semantics world and that you use SOS reasoning whenever you are in the SOS world. For example, it is incorrect *to "infer" that $\langle s_1; s_2, \sigma \rangle = \langle s_2, \langle s_1, \sigma \rangle \rangle$. What you can infer in SOS is a transition $\langle s_1; s_2, \sigma \rangle \to \sigma'$, but only after inferring the transitions $\langle s_1, \sigma \rangle \to \sigma''$ and $\langle s_2, \sigma'' \rangle \to \sigma'$ for some state σ'', respectively.*

Proving Equivalence of Programs

Without a formal semantics of a programming languages, one cannot prove programs equivalent because "equivalence of programs" is a meaningless concept. However, one can easily define it rigorously in any semantics. For example, p_1 and p_2 are *equivalent*, written $p_1 \equiv p_2$, if and only if:

- SOS: there is some i such that $\langle p_1 \rangle \to i$ and $\langle p_2 \rangle \to i$;
- Equational semantics: $\texttt{eval}(p_1) = \texttt{eval}(p_2)$.

One can define similar equivalence notions for arithmetic expressions, boolean expression and statements. For example, two statements s_1 and s_2 are equivalent, written $s_1 \equiv s_2$, if and only if:

- SOS: for any σ there is a σ' s.t. $\langle s_1, \sigma \rangle \to \sigma'$ and $\langle s_2, \sigma \rangle \to \sigma'$;
- Equational semantics: for any σ, $\texttt{state}(s_1, \sigma) = \texttt{state}(s_2, \sigma)$.

Note that two programs, arithmetic expressions, boolean expressions or statements are equivalent in the SOS if and only if they are equivalent in the equational semantics. Why?

We can now formally prove equivalence properties like

$$\texttt{while}\ b\ s \ \equiv\ \texttt{if}\ b\ \texttt{then}\ s; \texttt{while}\ b\ s\ \texttt{else}\ \texttt{skip}$$

for any boolean expression b and any statement s. It is easier to prove it using the equational semantics. Let σ be some state. Then, according to the equational semantics of `while`,

$$\texttt{state}(\texttt{while}\ b\ s, \sigma) =$$
$$\texttt{if}\ \texttt{eval}(b, \sigma)\ \texttt{then}\ \texttt{state}(\texttt{while}\ b\ s, \texttt{state}(s, \sigma))\ \texttt{else}\ \sigma\ \texttt{fi}$$

and according to the equational semantics of conditionals,

$$\texttt{state}(\texttt{if}\ b\ \texttt{then}\ s; \texttt{while}\ b\ s\ \texttt{else}\ \texttt{skip}, \sigma) =$$
$$\texttt{if}\ \texttt{eval}(b, \sigma)\ \texttt{then}\ \texttt{state}(s; \texttt{while}\ b\ s, \sigma)\ \texttt{else}\ \texttt{state}(\texttt{skip}, \sigma)\ \texttt{fi},$$

where `if_then_else_fi` is an operation that is built-in Maude, but which can be defined equationally as

```
eq if  true then S:Stmt else S':Stmt fi =  S:Stmt .
eq if false then S:Stmt else S':Stmt fi = S':Stmt .
```

By the equational semantics of sequential composition and of `skip`,

$$\begin{aligned}&\texttt{state}(\texttt{while}\ b\ s, \sigma) = \\ &\quad \texttt{state}(\texttt{if}\ b\ \texttt{then}\ s; \texttt{while}\ b\ s\ \texttt{else}\ \texttt{skip}, \sigma),\end{aligned}$$

which shows the desired equivalence of statements.

Homework Exercise 2 (Extra credit: 2 points) *If x, y, and z are three names, then prove that*

$$\begin{aligned}&\texttt{while}\ (x\ \texttt{equals}\ y)\ \{z = a;\ s\};\ z = a\ \equiv \\ &\quad z = a;\ \texttt{while}\ (x\ \texttt{equals}\ y)\ \{s;\ z = a\}\end{aligned}$$

for any arithmetic expression a and for any statement s.

```
*** ************************** ***
*** Simple calculator language ***
*** ************************** ***

--- ------ ---
--- SYNTAX ---
--- ------ ---

fmod SYNTAX is protecting QID .
  protecting INT .  protecting BOOL .
  sorts Name AExp BExp Stmt Pgm .
  subsort Qid < Name .
  subsorts Int Name < AExp .
  subsort Bool < BExp .

  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .

  op _+_ : AExp AExp -> AExp [ditto] .
  op _-_ : AExp AExp -> AExp [ditto] .
  op _*_ : AExp AExp -> AExp [ditto] .
  op _/_ : AExp AExp -> AExp [prec 31] .

  op _<=_ : AExp AExp -> BExp [ditto] .
  op _>=_ : AExp AExp -> BExp [ditto] .
  op _equals_ : AExp AExp -> BExp .
  op _and_ : BExp BExp -> BExp [ditto] .
  op _or_ : BExp BExp -> BExp [ditto] .
  op not_ : BExp -> BExp [ditto] .

  op skip : -> Stmt .
  op _=_ : Name AExp -> Stmt [prec 40] .
  op _;_ : Stmt Stmt -> Stmt [assoc] .
  op {_} : Stmt -> Stmt .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .

  op _;_ : Stmt AExp -> Pgm .
endfm


--- --------- ---
--- SEMANTICS ---
```

```
--- --------- ---

fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
  vars X : Index .  vars I I' : Int .  var S : State .
  eq ([X,I] S)[X] = I .
  eq S[X] = 0 [owise] .
  eq ([X,I'] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm

fmod SEMANTICS is protecting SYNTAX .
  protecting STATE .
  subsort Name < Index .
  op eval : AExp State -> Int .
  op eval : BExp State -> Bool .
  op eval : Pgm -> Int .
  op state : Stmt State -> State .
  var X : Name .  var S : State .  vars A A' : AExp .  var I : Int .
  vars St St' : Stmt .  vars B B' : BExp .

  eq eval(X, S) = S[X] .
  eq eval(I, S) = I .
  eq eval(A + A', S) = eval(A, S) + eval(A', S) .
  eq eval(A - A', S) = eval(A, S) - eval(A', S) .
  eq eval(A * A', S) = eval(A, S) * eval(A', S) .
  eq eval(A / A', S) = eval(A, S) quo eval(A', S) .

  eq eval(true, S) = true .  eq eval(false, S) = false .
  eq eval(A <= A', S) = eval(A, S) <= eval(A', S) .
  eq eval(A >= A', S) = eval(A, S) >= eval(A', S) .
  eq eval(A equals A', S) = eval(A, S) == eval(A', S) .
  eq eval(B and B', S) = eval(B, S) and eval(B', S) .
  eq eval(B or B', S) = eval(B, S) or eval(B', S) .
  eq eval(not B, S) = not eval(B, S) .

  eq state(skip, S:State) = S:State .
  eq state(X = A, S) = S[X <- eval(A,S)] .
```

```
  eq state(St ; St', S) = state(St', state(St, S)) .
  eq state({St'}, S) = state(St', S) .
  eq state(if B then St else St', S) =
     if eval(B, S) then state(St, S) else state(St', S) fi .
  eq state(while B St, S) =
     if eval(B, S) then state(while B St, state(St, S)) else S fi .

  eq eval(St' ; A) = eval(A, state(St', empty)) .
endfm

red eval( skip ; 3 + y) .
red eval( x = 1 ; y = x ; y) .
red eval(
     x = 1 ;
     y = (1 + x) * 2 ;
     z = x * 2 + x * y + y * 2 ;
     x + y + z
) .
red eval(
     x = 17 ;
     y = 1000 ;
     p = 1 ;
     i = y ;
     while (not(i equals 0)) {
       p = p * x ;
       i = i - 1
     } ;
     p
) .
red eval(
     x = 0 ;
     y = 1 ;
     n = 1000 ;
     i = 0 ;
     while (not(i equals n)) {
       y = y + x ;
       x = y - x ;
       i = i + 1
     } ;
     y
) .
red eval(
     n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
```

```
    c = 0 ;
    while (not (n equals 1)) {
      c = c + 1 ;
      if 2 * (n / 2) equals n
      then n = n / 2
      else n = 3 * n + 1
    } ;
    c
) .
```

```
*** ************************* ***
*** Simple calculator language ***
*** ************************* ***

--- ------ ---
--- SYNTAX ---
--- ------ ---

fmod SYNTAX is protecting QID .
  protecting INT .  protecting BOOL .
  sorts Name AExp BExp Stmt Pgm .
  subsort Qid < Name .
  subsorts Int Name < AExp .
  subsort Bool < BExp .

  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .

  op _+_ : AExp AExp -> AExp [ditto] .
  op _-_ : AExp AExp -> AExp [ditto] .
  op _*_ : AExp AExp -> AExp [ditto] .
  op _/_ : AExp AExp -> AExp [prec 31] .

  op _<=_ : AExp AExp -> BExp [ditto] .
  op _>=_ : AExp AExp -> BExp [ditto] .
  op _equals_ : AExp AExp -> BExp .
  op _and_ : BExp BExp -> BExp [ditto] .
  op _or_ : BExp BExp -> BExp [ditto] .
  op not_ : BExp -> BExp [ditto] .

  op skip : -> Stmt .
  op _=_ : Name AExp -> Stmt [prec 40] .
  op _;_ : Stmt Stmt -> Stmt [assoc] .
  op {_} : Stmt -> Stmt .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .

  op _;_ : Stmt AExp -> Pgm .
endfm


--- --------- ---
--- SEMANTICS ---
--- --------- ---

fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
  vars X : Index .  vars I I' : Int .  var S : State .
  eq ([X,I] S)[X] = I .
  eq S[X] = 0 [owise] .
  eq ([X,I'] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm

fmod SEMANTICS is protecting SYNTAX .
  protecting STATE .
  subsort Name < Index .
  op eval : AExp State -> Int .
  op eval : BExp State -> Bool .
  op eval : Pgm -> Int .
  op state : Stmt State -> State .
  var X : Name .  var S : State .  vars A A' : AExp .  var I : Int .
  vars St St' : Stmt .  vars B B' : BExp .

  eq eval(X, S) = S[X] .
  eq eval(I, S) = I .
  eq eval(A + A', S) = eval(A, S) + eval(A', S) .
  eq eval(A - A', S) = eval(A, S) - eval(A', S) .
  eq eval(A * A', S) = eval(A, S) * eval(A', S) .
  eq eval(A / A', S) = eval(A, S) quo eval(A', S) .

  eq eval(true, S) = true .  eq eval(false, S) = false .
  eq eval(A <= A', S) = eval(A, S) <= eval(A', S) .
  eq eval(A >= A', S) = eval(A, S) >= eval(A', S) .
  eq eval(A equals A', S) = eval(A, S) == eval(A', S) .
  eq eval(B and B', S) = eval(B, S) and eval(B', S) .
  eq eval(B or B', S) = eval(B, S) or eval(B', S) .
  eq eval(not B, S) = not eval(B, S) .

  eq state(skip, S:State) = S:State .
  eq state(X = A, S) = S[X <- eval(A,S)] .
  eq state(St ; St', S) = state(St', state(St, S)) .
  eq state({St'}, S) = state(St', S) .
  eq state(if B then St else St', S) =
     if eval(B, S) then state(St, S) else state(St', S) fi .
  eq state(while B St, S) =
     if eval(B, S) then state(while B St, state(St, S)) else S fi .

  eq eval(St' ; A) = eval(A, state(St', empty)) .
endfm

red eval( skip ; 3 + y) .
red eval( x = 1 ; y = x ; y) .
red eval(
      x = 1 ;
      y = (1 + x) * 2 ;
      z = x * 2 + x * y + y * 2 ;
      x + y + z
) .
red eval(
      x = 17 ;
      y = 1000 ;
      p = 1 ;
      i = y ;
      while (not(i equals 0)) {
        p = p * x ;
        i = i - 1
      } ;
      p
) .
red eval(
      x = 0 ;
      y = 1 ;
      n = 1000 ;
      i = 0 ;
      while (not(i equals n)) {
        y = y + x ;
        x = y - x ;
        i = i + 1
      } ;
      y
) .
red eval(
      n = 178378342647823759743985734809582309829798347583490698374986734 9 ;
      c = 0 ;
      while (not (n equals 1)) {
        c = c + 1 ;
        if 2 * (n / 2) equals n
        then n = n / 2
        else n = 3 * n + 1
      } ;
      c
) .
```

CS322 - Programming Language Design

Lecture 23: Denotational Semantics (Part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Denotational semantics, also known as *fix-point semantics*, associates to each programming language construct a well-defined and understood *mathematical object*, typically a function. The mathematical object denotes the behavior of the corresponding language construct, so equivalence of programs is immediately translated into equivalence of mathematical objects. The later equivalence can be then shown using the entire arsenal of mathematics.

We next present a denotational semantics for our simple imperative language and show that it is equivalent to the other two semantics discussed so far, namely the equational semantics and SOS. The denotational semantics of arithmetic and boolean expressions is rather straightforward. The unexpectedly difficult part is to properly give semantics to loops, because it involves non-trivial mathematics regarding fix-points of special operators.

Denotational Semantics of Arithmetic Expressions

An arithmetic expression can be seen as a function taking a state to an integer value, the value of the expression in that state. However, note that not any arithmetic expression is well defined in any state, because of division by zero. Thus, we can define an operator

$$[\![\_]\!] : \texttt{AExp} \to (\Sigma \rightharpoonup \mathbb{Z})$$

taking arithmetic expressions to partial functions from states to integer numbers. As in operational semantics, states are regarded as maps from names to values. In what follows, we will write $[\![a]\!]\sigma$ instead of $[\![a]\!](\sigma)$.

The denotation of names x and integers i is defined as expected:

$$[\![x]\!]\sigma = \sigma(x) \tag{1}$$

$$[\![i]\!]\sigma = i \tag{2}$$

The denotations of addition, subtraction and multiplication are:

$$[\![a_1 + a_2]\!]\sigma = [\![a_1]\!]\sigma + [\![a_2]\!]\sigma \tag{3}$$

$$[\![a_1 - a_2]\!]\sigma = [\![a_1]\!]\sigma - [\![a_2]\!]\sigma \tag{4}$$

$$[\![a_1 * a_2]\!]\sigma = [\![a_1]\!]\sigma * [\![a_2]\!]\sigma \tag{5}$$

For division, we have to consider the situation when the denominator is zero:

$$[\![a_1/a_2]\!]\sigma = \begin{cases} \text{undefined} & \text{if } [\![a_2]\!]\sigma = 0 \\ \text{integer quotient } [\![a_1]\!]\sigma \text{ by } [\![a_2]\!]\sigma & \text{if } [\![a_2]\!]\sigma \neq 0 \end{cases} \tag{6}$$

We implicitly assume that the denotation of an expression in a state is undefined whenever any of its subexpressions is undefined.

Denotational Semantics of Boolean Expressions

Partial functions from states to boolean values are associated to boolean expressions via the denotation operator:

$$[\![\_]\!] : \texttt{BExp} \to (\Sigma \rightharpoonup \{\texttt{true}, \texttt{false}\})$$

The denoting functions are partial because the boolean expression can involve arithmetic expressions which may be undefined. This operator can be defined as follows:

$$[\![\texttt{true}]\!]\sigma = \texttt{true} \tag{7}$$

$$[\![\texttt{false}]\!]\sigma = \texttt{false} \tag{8}$$

$$[\![a_1 \leq a_2]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![a_1]\!]\sigma \leq [\![a_2]\!]\sigma \\ \texttt{false} & \text{otherwise} \end{cases} \tag{9}$$

$$[\![a_1 \geq a_2]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![a_1]\!]\sigma \geq [\![a_2]\!]\sigma \\ \texttt{false} & \text{otherwise} \end{cases} \tag{10}$$

$$[\![a_1 \texttt{ equals } a_2]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![a_1]\!]\sigma = [\![a_2]\!]\sigma \\ \texttt{false} & \text{otherwise} \end{cases} \tag{11}$$

$$[\![b_1 \texttt{ and } b_2]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![b_1]\!]\sigma \text{ and } [\![b_2]\!]\sigma \text{ are } \texttt{true} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{12}$$

$$[\![b_1 \texttt{ or } b_2]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![b_1]\!]\sigma \text{ or } [\![b_2]\!]\sigma \text{ is } \texttt{true} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{13}$$

$$[\![\texttt{not } b]\!]\sigma = \begin{cases} \texttt{true} & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \texttt{false} & \text{if } [\![b]\!]\sigma = \texttt{true} \end{cases} \tag{14}$$

Denotational Semantics of Statements

In addition to partiality due to division by zero in expressions that statements may involve, partiality in the denotation of statements may also occur for another important reason: *loops may not terminate*. For example, the statement `while (x > y) skip` will not terminate in those states in which the value that x denotes is larger than that of y. Mathematically, we say that the function from states to states that this loop statement denotes is undefined in those states in which the loop statement does not terminate. The denotation of statements is therefore an operator

$$[\![\_]\!] : \texttt{Stmt} \to (\Sigma \rightharpoonup \Sigma).$$

We next define the denotational semantics of all statements except loops. Loops will be discussed separately later, because their denotational semantics is less trivial.

The following definitions are natural:

$$[\![\texttt{skip}]\!] = 1_\Sigma \text{ (the identity function on states)} \tag{15}$$

$$[\![x = a]\!]\sigma = \begin{cases} \sigma[x \leftarrow [\![a]\!]\sigma] & \text{if } [\![a]\!]\sigma \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{16}$$

$$[\![s_1; s_2]\!] = [\![s_2]\!] \circ [\![s_1]\!] \text{ (the composition of the two functions)} \tag{17}$$

$$[\![\{s\}]\!] = [\![s]\!] \tag{18}$$

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \text{undef} & \text{if } [\![b]\!]\sigma \text{ undefined} \end{cases} \tag{19}$$

Examples

What is the denotation of $x = 1;\ y = 2;\ x = 3$, i.e., the function

$$[\![x = 1;\ y = 2;\ x = 3]\!] \ ?$$

By the denotation of composition, we get
$[\![x = 3]\!] \circ [\![y = 2]\!] \circ [\![x = 1]\!]$. Applying it on a state σ, one gets
$([\![x = 3]\!] \circ [\![y = 2]\!] \circ [\![x = 1]\!])\sigma$ equals $((\sigma[x \leftarrow 1])[y \leftarrow 2])[x \leftarrow 3]$; let us denote the later state by σ'. By the definition of function update, one can easily see that σ' can be defined as

$$\sigma'(z) = \begin{cases} 3 & \text{if } z = x \\ 2 & \text{if } z = y \\ \sigma(z) & \text{otherwise,} \end{cases}$$

which is nothing but $\sigma[x \leftarrow 3][y \leftarrow 2]$.

Similarly, we can show that

$$[\![(\texttt{if } y > z \texttt{ then } x = 1 \texttt{ else } x = 2); x = 3]\!] = \lambda\sigma.\sigma[x \leftarrow 3],$$

where by abuse of notation we let $\lambda\sigma.\sigma[x \leftarrow 3]$ denote the function taking states σ to states $\sigma[x \leftarrow 3]$. Indeed,
$[\![(\texttt{if } y > z \texttt{ then } x = 1 \texttt{ else } x = 2); x = 3]\!]\sigma = \sigma'[x \leftarrow 3]$, where

$$\sigma' = \begin{cases} \sigma[x \leftarrow 1] & \text{if } [\![y > z]\!]\sigma = \texttt{true} \\ \sigma[x \leftarrow 2] & \text{otherwise,} \end{cases}$$

which can be easily shown equal to $\sigma[x \leftarrow 3]$.

Denotational Semantics of Programs

Before we discuss the denotational semantics of loop statements, we can discuss the denotation of programs, which is much easier. A program consists of a statement followed by an arithmetic expression, with the meaning that the statement is evaluated in the initial state and then the expression is evaluated in the obtained state. Thus, we can define the denotation of programs simply as

$$[\![\_]\!] : \texttt{Pgm} \to \mathbb{Z}$$

$$[\![s; a]\!] = [\![a]\!]([\![s]\!]\phi)$$

where ϕ is the initial state, which, in our current definition of the language, assigns zero to any name. By removing the equation "`eq S[X] = 0 [owise]`", one can define the initial state as a partial function from names to integer numbers.

Equivalence of Expressions, Statements, Programs

Within denotational semantics, it is quite straightforward to define equivalence of expressions, statements or programs. For example, two statements s_1 and s_2 are equivalent if and only if they denote the same mathematical objects, that is, if and only if $[\![s_1]\!] = [\![s_2]\!]$.

One can thus very easily show the equivalence of statements

$$(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2); s \equiv \texttt{if}\ b\ \texttt{then}\ s_1; s\ \texttt{else}\ s_2; s$$

for any statements s_1, s_2 and s_3, and for any boolean expression b. Indeed, the first denotes the function

$$\lambda\sigma.[\![s]\!] \left(\begin{cases} [\![s_1]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \text{undefined} & \text{if } [\![b]\!]\sigma \text{ undefined} \end{cases} \right)$$

while the second denotes the function

$$\lambda\sigma.\begin{cases} [\![s]\!]([\![s_1]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ [\![s]\!]([\![s_2]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \text{undefined} & \text{if } [\![b]\!]\sigma \text{ undefined} \end{cases}$$

It is clear now that the two are equal as mathematical objects. Prove it rigorously!

The major point of denotational semantics that you have to remain with after this class is the following:

> Denotational semantics is an abstract technique to assign meaning to programs, in which programming language constructs are mapped into mathematical objects. The intermediate states are ignored in denotational semantics.

Denotational Semantics of Loops

The only definition left is the denotational semantics of loops, or in other words the partial functions

$$[\![\texttt{while}(b)\ s]\!] : \Sigma \rightharpoonup \Sigma$$

where $[\![\texttt{while}(b)\ s]\!](\sigma) = \sigma'$ if and only if the while loop correctly terminates in state σ' when executed in state σ. Such a σ' may not always exist for two reasons: (1) because the denotation of b or s in some appropriate state encounters a division by zero, and (2) because s (which may contain nested loops) or the while loop itself does not terminate. If we let $\mathcal{W}$ denote the partial function

$[\![\texttt{while}(b)\ s]\!]$, then the most natural definition of $\mathcal{W}$ is:

$$\mathcal{W}(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \mathcal{W}([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \text{undefined} & \text{if } [\![b]\!]\sigma \text{ undefined} \end{cases}$$

Mathematically speaking, this is a problematic definition for several reasons:

- $\mathcal{W}$ is defined in terms of itself;
- it is not clear that such a $\mathcal{W}$ exists;
- in case it exists, it is not clear that such a $\mathcal{W}$ is unique.

We next develop the mathematical machinery needed to rigorously define and reason about partial functions like the $\mathcal{W}$ above.

Partial Functions as Information Bearers

One very convenient interpretation of partial functions that significantly eases the understanding of the subsequent mathematics is as *information* or *knowledge bearers*. More precisely, a partial function $\mathcal{I} : \Sigma \rightharpoonup \Sigma$ can be thought of as carrying knowledge about some states in Σ, namely exactly those on which it is defined. For such a state σ, the knowledge that it carries is $\mathcal{I}(\sigma)$. If $\mathcal{I}$ is not defined on a state $\sigma \in \Sigma$ then we can think of it as "$\mathcal{I}$ *does not have any information about* σ".

Recall that a *partial order* on a set, say D, is a binary relation, say $\sqsubseteq$, which is

- *reflexive*, i.e., $x \sqsubseteq x$,
- *transitive*, i.e., $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$, and
- *anti-symmetric*, i.e., $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.

A partial order relation occurs naturally on partial functions. If $\mathcal{I}, \mathcal{J} : \Sigma \rightharpoonup \Sigma$ are partial functions, then we say that $\mathcal{I}$ *is less informative then or as informative as* $\mathcal{J}$, written $\mathcal{I} \preceq \mathcal{J}$, if and only if for any $\sigma \in \Sigma$, it is either the case that $\mathcal{I}(\sigma)$ is not defined, or both $\mathcal{I}(\sigma)$ and $\mathcal{J}(\sigma)$ are defined and $\mathcal{I}(\sigma) = \mathcal{J}(\sigma)$. If $\mathcal{I} \preceq \mathcal{J}$ then we may also say that $\mathcal{J}$ *refines* $\mathcal{I}$ or that $\mathcal{J}$ *extends* $\mathcal{I}$.

Intuition for the Denotation of Loops

One can, and should, think of each possible iteration of a while loop as an opportunity to refine the knowledge about its denotation. Before the boolean expression b of the while loop $\texttt{while}(b)\ s$ is evaluated the first time, the knowledge that one has about $\mathcal{W}$ is $\mathcal{W}_0 := \bot : \Sigma \rightharpoonup \Sigma$, which denotes the function which is not defined in any state. Therefore, $\mathcal{W}_0$ corresponds to no information.

Now suppose that we evaluate the boolean expression b in some state σ and that it is false. Then the denotation of the while loop should return σ, which suggests that we can refine our knowledge about $\mathcal{W}$ from $\mathcal{W}_0$ to the partial function $\mathcal{W}_1 : \Sigma \rightharpoonup \Sigma$, which is an identity on all those states $\sigma \in \Sigma$ for which $[\![b]\!]\sigma = \texttt{false}$.

So far we have not considered any state in which the loop needs to evaluate its body. Suppose now that for some state σ, it is the case that $[\![b]\!]\sigma = \texttt{true}$, $[\![s]\!]\sigma = \sigma'$, and $[\![b]\!]\sigma' = \texttt{false}$, that is, that the while loop terminates in one iteration. Then we can extend $\mathcal{W}_1$ to a partial function $\mathcal{W}_2 : \Sigma \rightharpoonup \Sigma$, which, in addition to being an identity on those states on which $\mathcal{W}_1$ is defined, takes each σ as above to $\mathcal{W}_2(\sigma) = \sigma'$. Therefore, $\mathcal{W}_1 \preceq \mathcal{W}_2$.

By iterating this process, one can define for any natural number k a partial function $\mathcal{W}_k : \Sigma \rightharpoonup \Sigma$, which is defined on all those states on which the while loop terminates in *at most* k evaluations of its boolean condition (i.e., $k-1$ executions of its body). An

immediate property of the partial functions $\mathcal{W}_0,\ \mathcal{W}_1,\ \mathcal{W}_2,\ \ldots,\ \mathcal{W}_k$ is that they increasingly refine each other, that is, $\mathcal{W}_0 \preceq \mathcal{W}_1 \preceq \mathcal{W}_2 \preceq \cdots \preceq \mathcal{W}_k$.

Informally, the partial functions $\mathcal{W}_k$ approximate $\mathcal{W}$ as k increases; more precisely, for any $\sigma \in \Sigma$, if $\mathcal{W}(\sigma) = \sigma'$, that is, if the while loop terminates and σ' is the resulting state, then one can show that there is some k such that $\mathcal{W}_k(\sigma) = \sigma'$. Moreover, it follows easily that $\mathcal{W}_n(\sigma) = \sigma'$ for any $n \geq k$.

But the main question still remains unanswered: how can we define the denotation $\mathcal{W} :\ \Sigma \rightharpoonup \Sigma$ of while loops? According to the intuitions above, $\mathcal{W}$ should be "some sort of limit" of the (infinite) sequence of partial functions $\mathcal{W}_0 \preceq \mathcal{W}_1 \preceq \mathcal{W}_2 \preceq \ldots \preceq \mathcal{W}_k \preceq \ldots$, but the notion of limit of partial functions partially ordered by the "more informative" partial ordering does not seem to be immediate.

In the next lecture we develop a general theory of fix-points, which

will then be elegantly applied to our special case and thus properly define the denotation $\mathcal{W} :\ \Sigma \rightharpoonup \Sigma$ of while loops. Before that, let us first define a total function $\mathcal{F} :\ (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ taking partial functions to partial functions as follows:

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \sigma & \text{if } [\![b]\!]\sigma = \texttt{false} \\ \alpha([\![s]\!]\sigma) & \text{if } [\![b]\!]\sigma = \texttt{true} \\ \text{undefined} & \text{if } [\![b]\!]\sigma \text{ undefined} \end{cases}$$

Notice that the informal partial functions $\mathcal{W}_k$ above can be now rigorously defined as $\mathcal{F}^k(\bot)$, where $\mathcal{F}^k$ stays for k compositions of $\mathcal{F}$ and $\mathcal{F}^0 = \bot$ by convention. Indeed, one can show by induction on k the following property, where $[\![s]\!]^i$ stays for i compositions of $[\![s]\!] :\ \Sigma \rightharpoonup \Sigma$ and $[\![s]\!]^0$ is by convention the identity (total) function

on Σ:

$$\mathcal{F}^k(\bot)(\sigma) = \begin{cases} [\![s]\!]^i\sigma & \text{if there is } 0 \leq i < k \text{ s.t. } ([\![b]\!] \circ [\![s]\!]^i)\sigma = \texttt{false} \\ & \text{and } ([\![b]\!] \circ [\![s]\!]^j)\sigma = \texttt{true} \text{ for all } 0 \leq j < i \\ \text{undef} & \text{otherwise} \end{cases}$$

Note that the function $\mathcal{F}^k(\bot)$ is well defined, in the sense that if an i as above exists then it is unique.

CS322 - Programming Language Design

Lecture 24: Denotational Semantics (Part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

In what follows we develop a theoretical framework which will allow us to prove an important fix-point theorem stating the existence of least fix-points of certain operators. Applied to our total function

$$\mathcal{F} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$$

this theorem will give us the denotation of while loops.

Posets and (Least) Upper Bounds

Let $(D, \sqsubseteq)$ be a partial order, that is, a set D together with a binary relation $\sqsubseteq$ on it which is reflexive, transitive and anti-symmetric. Partial orders are also called *posets*. Given a set of elements $X \subseteq D$, an element $p \in D$ is called an *upper bound (ub)* of X if and only if $x \sqsubseteq p$ for any $x \in X$. Furthermore, $p \in D$ is called a *least upper bound (lub)* of X if and only if p is an upper bound and for any other upper bound q of X it is the case that $p \sqsubseteq q$.

Note that upper bounds and least upper bounds may not always exist. For example, if $D = X = \{x, y\}$ and $\sqsubseteq$ is the identity relation, then X has no upper bounds. Least upper bounds may not exist even though upper bounds exist. For example, if $D = \{a, b, c, d, e\}$ and $\sqsubseteq$ is defined by $a \sqsubseteq c$, $a \sqsubseteq d$, $b \sqsubseteq c$, $b \sqsubseteq d$, $c \sqsubseteq e$, $d \sqsubseteq e$, then any subset X of D admits upper bounds, but the set $X = \{a, b\}$ does not have a least upper bound.

Due to the anti-symmetry property, least upper bounds are unique when they exist. For that reason, we let $\sqcup X$ denote the lub of X.

Chains

Given a poset $(D, \sqsubseteq)$, a *chain* in D is an infinite sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ of elements in D, also written using set notation as $\{d_n \mid n \in \mathbb{N}\}$. Such a chain is called *stationary* when

there is some $n \in \mathbb{N}$ such that $d_m = d_{m+1}$ for all $m \geq n$.

Recall that the infinite sequence of partial functions $\Sigma \rightharpoonup \Sigma$

$$\bot \preceq \mathcal{F}(\bot) \preceq \mathcal{F}^2(\bot) \preceq \cdots \preceq \mathcal{F}^n(\bot) \preceq \cdots$$

incrementally approximates the desired denotation of a while loop. Thus, we would like to define the denotation of the while loop as $\sqcup\{\mathcal{F}^n(\bot) \mid n \in \mathbb{N}\}$, in case it exists. In the simple case when this sequence regarded as a chain in $(\Sigma \rightharpoonup \Sigma, \preceq)$ is stationary, with the intuition that the while loop terminates in any state in some fixed number of iterations which does not depend on the state, then the denotation of the while loop is the partial function in which the chain stabilizes, that is, its lub. Unfortunately, in most of the practical situations this chain is not stationary. For example, the simple loop `while (k > 0) k = k - 1` terminates in any state, but there is no bound on the number of iterations. Consequently, there is no n such that $\mathcal{F}^n(\bot) = \mathcal{F}^{n+1}(\bot)$. Indeed, the later has

strictly more information than the former: $\mathcal{F}^{n+1}$ is defined on all those states σ with $\sigma(k) = n + 1$, while $\mathcal{F}^n$ is not.

Complete Partial Orders (cpos)

A poset $(D, \sqsubseteq)$ is called a *complete partial order (cpo)* if and only if any of its chains has a lub. $(D, \sqsubseteq)$ is said to *have bottom* if and only if it has a minimal element. Such element is typically denoted by $\bot$, and the poset with bottom $\bot$ is written $(D, \sqsubseteq, \bot)$. If $\{d_n \mid n \in \mathbb{N}\}$ is a chain in $(D, \sqsubseteq)$, then we also let $\bigsqcup_{n\in\mathbb{N}} d_n$ or even $\sqcup d_n$ denote its lub $\sqcup\{d_n \mid n \in \mathbb{N}\}$.

Examples

$(\mathcal{P}(S), \subseteq, \emptyset)$ is a cpo with bottom, where $\mathcal{P}(S)$ is the set of subsets of a set S and $\emptyset$ is the empty set.

$(\mathbb{N}, \leq)$, the set of natural numbers ordered by "less than or equal to", has bottom 0 but is not complete: the sequence $0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots$ has no upper bound in $\mathbb{N}$.

$(\mathbb{N} \cup \{\infty\}, \leq, 0)$, the set of natural numbers plus infinity, where infinity is larger than any number, is a cpo with bottom 0. It is a cpo because any chain is either stationary, in which case its lub is obvious, or is unbounded by any natural number, in which case ∞ is its lub.

$(\mathbb{N}, \geq)$ is a cpo but has no bottom.

$(\mathbb{Z}, \leq)$ is not a cpo and has no bottom.

$(S, =)$, a flat set S where the only partial ordering is the identity, is

a cpo. It has bottom if and only if S has only one element.

Most importantly, $(\Sigma \rightharpoonup \Sigma, \preceq, \perp)$, the set of partial functions $\Sigma \rightharpoonup \Sigma$ ordered by the informativeness relation $\preceq$ is a cpo with bottom $\perp : \Sigma \rightharpoonup \Sigma$, the function which is undefined in each state. You will have to prove this as part of your homework.

Monotone and Continuous Functions

If $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are two posets and $f : D \to D'$ is a function, then f is called *monotone* if and only if $f(x) \sqsubseteq' f(y)$ for any $x, y \in D$ with $x \sqsubseteq y$. If f is monotone, then we simply write $f : (D, \sqsubseteq) \to (D', \sqsubseteq')$.

Monotone functions *preserve chains*, that is, $\{f(d_n) \mid n \in \mathbb{N}\}$ is a chain in $(D', \sqsubseteq')$ whenever $\{d_n \mid n \in \mathbb{N}\}$ is a chain in $(D, \sqsubseteq)$. Moreover, if $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are cpos then for any chain

$\{d_n \mid n \in \mathbb{N}\}$ in $(D, \sqsubseteq)$, we have

$$\bigsqcup_{n \in \mathbb{N}} f(d_n) \sqsubseteq' f(\bigsqcup_{n \in \mathbb{N}} d_n)$$

Indeed, since f is monotone and since $d_n \sqsubseteq \sqcup d_n$ for each $n \in \mathbb{N}$, it follows that $f(d_n) \sqsubseteq' f(\sqcup d_n)$ for each $n \in \mathbb{N}$. Therefore, $f(\sqcup d_n)$ is an upper bound for the chain $\{f(d_n) \mid n \in \mathbb{N}\}$. The rest follows because $\sqcup f(d_n)$ is the lub of $\{f(d_n) \mid n \in \mathbb{N}\}$.

Note that $\sqcup f(d_n) \sqsupseteq' f(\sqcup d_n)$ does not hold in general. Let, for example, $(D, \sqsubseteq)$ be $(\mathbb{N} \cup \{\infty\}, \leq)$, $(D', \sqsubseteq')$ be $(\{0, \infty\}, 0 \leq \infty)$, and f be the monotone function taking any natural number to 0 and ∞ to ∞. For the chain $\{n \mid n \in \mathbb{N}\}$, note that $\sqcup n = \infty$, so $f(\sqcup n) = \infty$. On the other hand, the chain $\{f(n) \mid n \in \mathbb{N}\}$ is stationary in 0, so $\sqcup f(n) = 0$.

One can think of a lub of a chain as a "limit" of that chain. Inspired by the analogous notion of continuous function in

mathematical analysis, which is characterized by the property of preserving limits, we say that a monotone function $f : (D, \sqsubseteq) \to (D', \sqsubseteq')$ is *continuous* if and only if $\sqcup f(d_n) \sqsupseteq' f(\sqcup d_n)$, which is equivalent to $\sqcup f(d_n) = f(\sqcup d_n)$, for any chain $\{d_n \mid n \in \mathbb{N}\}$ in $(D, \sqsubseteq)$.

The key property of the domain of partial functions, allowing us to use the general theory of cpos with continuous functions and especially the next fix-point theorem, is stated below as a homework exercise.

Homework Exercise 1 *$(\Sigma \rightharpoonup \Sigma, \preceq, \perp)$ is a cpo with bottom. Moreover, the functions $\mathcal{F} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ naturally associated to while loops as shown in the previous lecture are continuous.*

The Fix-Point Theorem

Any monotone function $f : (D, \sqsubseteq, \perp) \to (D, \sqsubseteq, \perp)$ defined on a cpo with bottom to itself admits an implicit and important chain, namely $\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \cdot \sqsubseteq f^n(\perp) \sqsubseteq \cdots$, where f^n denotes n compositions of f with itself. The next is a key result in denotational semantics.

Theorem. *Let $(D, \sqsubseteq, \perp)$ be a cpo with bottom, let $f : (D, \sqsubseteq, \perp) \to (D, \sqsubseteq, \perp)$ be a continuous function, and let $fix(f)$ be the lub of the chain $\{f^n(\perp) \mid n \in \mathbb{N}\}$. Then $fix(f)$ is the least fix-point of f.*

Proof. We first show that $\mathit{fix}(f)$ is a fix-point of f:

$$\begin{aligned} f(\mathit{fix}(f)) &= f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) \\ &= \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) \\ &= \bigsqcup_{n\in\mathbb{N}} f^n(\bot) \\ &= \mathit{fix}(f). \end{aligned}$$

Next we show that $\mathit{fix}(f)$ is the least fix-point of f. Let d be another fix-point of f, that is, $f(d) = d$. We can show by induction that $f^n(\bot) \sqsubseteq d$ for any $n \in \mathbb{N}$: first note that $f^0(\bot) = \bot\bot \sqsubseteq d$; assume $f^n(\bot) \sqsubseteq d$ for some $n \in \mathbb{N}$; since f is monotone, it follows that $f(f^n(\bot)) \sqsubseteq f(d) = d$, that is, $f^{n+1}(\bot) \sqsubseteq d$. Thus d is an upper bound of the chain $\{f^n(\bot) \mid n \in \mathbb{N}\}$, so $\mathit{fix}(f) \sqsubseteq d$.

Corollary. *Functions $\mathcal{F} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ associated to while loops as shown in the previous lecture admit fix-points.*

Proof. It follows by the fix-point theorem, thanks to the

properties proved in your homework exercise above.

Denotational Semantics of Loops

We are now ready to define the denotational semantics of loops.

> If $\mathcal{F} : (\Sigma \rightharpoonup \Sigma) \to (\Sigma \rightharpoonup \Sigma)$ is the function associated to the statement $\texttt{while}(b)\ s$ as shown at the end of the previous lecture, then we define $[\![\texttt{while}(b)\ s]\!]$ to be $\mathit{fix}(\mathcal{F})$.

Therefore, we gave the denotational semantics of the entire simple imperative language. A natural question now is whether this new semantics is equivalent to the other semantics. Being able to prove so increases our confidence in the correctness of our language definitions, in particular in the correctness of our interpreter that we got as a consequence of the fact that Maude is executable.

Homework Exercise 2 *Write up the proof of the following*

important theorem.

Theorem. *The equational semantics, the SOS and the denotational semantics of our language are all equivalent.*

Proof sketch. The equivalence of the two has been already shown in previous lectures. We therefore only need to show that the equational and the denotational semantics are equivalent. We have to show that for any program `P` and any integer number `I`, `eval(P) = I` is provable in the equational specification if and only if $[\![\texttt{P}]\!] = \texttt{I}$ in the denotational semantics. We need three auxiliary lemmas, which can be proved by structural induction:

(L1) `eval(A,`σ`) = I` if and only if $[\![\texttt{A}\sigma]\!] = \texttt{I}$ for any arithmetic expression `A`, any state σ and any integer `I`;

(L2) `eval(B,`σ`) =` *bool-value* if and only if $[\![\texttt{A}\sigma]\!] = \mathit{bool-value}$ for any boolean expression `A`, any state σ and any boolean value *bool-value*, i.e., `true` or `false`;

(L3) `state(S,`σ`) =` σ' if and only if $[\![\texttt{S}]\!]\sigma = \sigma'$ for any statement `S` and any states σ, σ'. The hard case is the loop. If $\mathcal{W}$ is the denotation of a while loop `while(`b`)` s, i.e., a fix point, you may need to first show the following property: $\mathcal{W}(\sigma) = \sigma'$ if and only if there is some $n \in \mathbb{N}$ such that $[\![s]\!]^n\sigma = \sigma'$, $([\![b]\!] \circ [\![s]\!]^n)\sigma = \texttt{false}$ and $([\![b]\!] \circ [\![s]\!]^k)\sigma = \texttt{true}$ for all $k < n$.

Exercise 1 *Read Chapters 1 to 5 in Winskel.*

Exercise 2 *Using denotational semantics, show that*

$$\texttt{while}\ b\ s \equiv \texttt{if}\ b\ \texttt{then}\ s;\texttt{while}\ b\ s\ \texttt{else skip}$$

CS322 - Programming Language Design

Lecture 25: Axiomatic Semantics (Hoare Logic)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Using equational, SOS or denotational semantics we have been able to formally prove equivalence of programs. Moreover, since we were also able to prove equivalence of semantics, we have a high confidence that each of the semantics is a correct definition of the language under consideration.

However, so far we have investigated no method that allows us to prove programs correct. We have not even formally defined what it means for a program to be correct. In this lecture we very informally discuss a standard technique to define and prove correctness of programs, called *Hoare logic*, which is quite successful in the context of non-concurrent programs. Starting with the next lecture we will discuss useful techniques for proving correctness of concurrent programs as well.

Suppose that one wants to show that the program

```
s = 0 ; n = 1 ;
while (n <= p) (
  s = s + n ;
  n = n + 1
)
```

indeed calculates the sum of the first p natural numbers, that is, that $s = p * (p + 1)/2$ at the end of its execution. A first observation is that this program calculates the correct sum only if run in a state in which p denotes a positive number (technically speaking, it is also correct for $p = -1$, but we ignore this unnecessary generality). Thus, in order to state the correctness of this program, we would like to say the following:

> If executed in a state in which $p \geq 0$ then it is the case that $s = p * (p + 1)/2$ in the state obtained at the end of the execution of this program.

As will be discussed later, the statement above can be formally written as

```
{p >= 0}
  s = 0 ; n = 1 ;
  while (n <= p) (
    s = s + n ;
    n = n + 1
  )
{s == p * (p + 1) / 2}
```

where `{p >= 0}` and `{s == p * (p + 1) / 2}` are *assertions*, the first called *pre-condition* and the second called *post-condition*. In this lecture, to avoid syntactic conflicts, we will enclose blocks by normal parentheses rather than curly brackets.

State Assertions

The assertions above are nothing but boolean expressions, using the syntax that we have already defined for our simple imperative language. However, in general one may need to state more complex state properties, such as, for example, $(\exists k)x = 2^k$, saying that x is a power of 2.

Our goal in this lecture is to understand the major underlying concepts of Hoare logic and the basic intuitions for what is called *theorem proving* in software analysis/verification, so we will *not* formalize rigorously the syntax and semantics of assertions.

Intuitively, besides the usual boolean expression operators, we also allow *quantifiers* with the syntax $(\forall\langle IntVar\rangle)\ \langle Assertion\rangle$ and $(\exists\langle IntVar\rangle)\ \langle Assertion\rangle$. There is an intuitive notion of *validity* of assertions, saying that an assertion is valid if and only if it holds in any possible state. For example, $(\forall i)(x > i \vee x = i \vee x < i)$ is valid.

The famous *Gödel's incompleteness theorem* tells us that there is no way to find an algorithm able to prove any valid assertion, which tells us that there is no way to prove *any* property one may want to prove about a program.

Indeed, if one had an algorithm to prove any property about a program, then, given any assertion `A`, one may use that algorithm to prove that {} `skip` {`A`}, which is equivalent to proving `A`. Thus, an algorithm able to prove any property about programs would immediately translate into an algorithm to prove any valid assertion, which contradicts Gödel's incompleteness theorem.

Partial Correctness Assertions

Triples $\{\texttt{A}\}\texttt{S}\{\texttt{B}\}$, where $\texttt{A}$ and $\texttt{B}$ are assertions and $\texttt{S}$ is a statement, are called *partial correctness assertions*. They are called "partial" because they allow statements to be undefined. Formally, $\{\texttt{A}\}\texttt{S}\{\texttt{B}\}$ is *valid* iff for any state σ satisfying the assertion $\texttt{A}$, if the statement $\texttt{S}$ is defined in state σ and ends up in σ' then s' satisfies $\texttt{B}$. In particular, the partial correctness assertion $\{\texttt{true}\}$ `while(true) skip` $\{\texttt{false}\}$ is valid because the while loop will never terminate.

We let $\sigma \models \texttt{A}$ denote the fact that state σ satisfies $\texttt{A}$. Then $\{\texttt{A}\}\texttt{S}\{\texttt{B}\}$ is valid, written $\models \{\texttt{A}\}\texttt{S}\{\texttt{B}\}$, iff

$$(\forall \sigma \in \Sigma)\ (\sigma \models \texttt{A} \text{ and } [\![\texttt{S}]\!]\sigma \text{ defined} \Rightarrow [\![\texttt{S}]\!]\sigma \models \texttt{B}).$$

To simplify writing, by convention we assume that "undefined" satisfies any assertion, so the above transforms into:

$$(\forall \sigma \in \Sigma)\ (\sigma \models \texttt{A} \Rightarrow [\![\texttt{S}]\!]\sigma \models \texttt{B}).$$

We can now formally state the correctness of a statement as a partial correctness assertion, where the pre-condition gives the conditions under which the statement is executed and the post-condition gives the state properties at the end of the execution of the statement, in case that is reached. There are situations in which one may want to work with so-called *total correctness assertions*, often written `[A]S[B]`, which require the statement $\texttt{S}$ to also be defined. We only consider partial correctness assertions in this lecture.

Hoare Logic

Hoare logic is defined as a set of inference rules for deriving valid correctness assertions. Like in the case of SOS, one can start with basic facts and then derive complex properties.

For the `skip` statement, the following is natural:

$$\frac{\cdot}{\{A\}\ \texttt{skip}\ \{A\}}$$

For an assignment $x = a$, a post-condition `B` holds if and only if `B` in which each occurrence of x is replaced by a holds as a pre-condition:

$$\frac{\cdot}{\{B[x \leftarrow a]\}\ \texttt{x = a}\ \{B\}}$$

We do not get into the technicalities underlying the formalization of assertions, but it is worth noticing that $B[x \leftarrow a]$ replaces only the *free occurrences* of x in B by the arithmetic expression a, and that bound variables in B *may need to be renamed* in order to avoid capturing names that occur free in a.

For sequential composition, one needs to "discover" an intermediate assertion `C`:

$$\frac{\{A\}\ \texttt{s}_1\ \{C\}, \quad \{C\}\ \texttt{s}_2\ \{B\}}{\{A\}\ \texttt{s}_1\texttt{;}\ \texttt{s}_2\ \{B\}}$$

In the case of conditionals, one has to treat the cases in which the condition is true and false, respectively:

$$\frac{\{A \wedge b\}\ \texttt{s}_1\ \{B\}, \quad \{A \wedge \neg b\}\ \texttt{s}_2\ \{B\}}{\{A\}\ \texttt{if b then s}_1\ \texttt{else s}_2\ \{B\}}$$

Loops are the hardest to prove in practice, because one has to *discover an invariant assertion*, which sometimes is a highly non-trivial task. By analogy to mathematics, one can think of invariant discovery as "lemma discovery":

$$\frac{\{A \wedge b\}\ \texttt{s}\ \{A\}}{\{A\}\ \texttt{while(b)}\ \texttt{s}\ \{A \wedge \neg b\}}$$

A is called a *loop invariant*, because the body of the loop preserves it whenever it executes, that is, the condition holds. The invariant will then become false at the end of the loop.

Like in mathematics, where it is sometimes easier to prove a more general result then to prove directly the more particular one, we are often able to more easily prove more general partial correctness assertions. The following allows us to use a more general fact in order to derive a less general one:

$$\frac{\models A \Rightarrow A', \quad \{A'\}\ \texttt{s}\ \{B'\}, \quad \models B \Rightarrow B'}{\{A\}\ s\ \{B\}}$$

We will stay informal with respect to the formal language for

assertions, and also implicitly with respect to what validity means for assertions. Intuitively, A is valid, written $\models A$, if and only if it holds in any state.

We write $\vdash \{A\}\ s\ \{B\}$ whenever $\{\texttt{A}\}\ \texttt{s}\ \{\texttt{B}\}$ is *derivable* using the rules above. One can show the following important *soundness* result for *Hoare logic*:

Theorem. $\vdash \{A\}\ s\ \{B\}$ *implies* $\models \{A\}\ s\ \{B\}$.

The proof can be done by showing that each inference rule is sound.

Example

The following example shows the correctness of the program discussed at the beginning of this lecture, adding the first p natural numbers. The proof is quite detailed and, except the proofs of validity of state assertions, it explicitly shows all the derivation steps. We start by splitting the proof task into two subtasks:

$$\frac{PCA_1, \quad PCA_2}{\texttt{\{p>=0\} s=0; n=1; while(n<=p) (s=s+n; n=n+1) \{s==p*(p+1)/2\}}}$$

where PCA_1 is a partial correctness assertion derived as follows:

$$\frac{PCA_3, \quad PCA_4}{\texttt{\{p>=0\} s=0; n=1 \{p>=0 } \wedge \texttt{ s==0 } \wedge \texttt{ n==1\}}}$$

where PCA_3 is derived as follows:

$$\frac{\models \texttt{p>=0} \Rightarrow \texttt{(p>=0} \wedge \texttt{ 0==0)}, \quad \frac{\cdot}{\texttt{\{p>=0} \wedge \texttt{ 0==0\} s=0 \{p>=0} \wedge \texttt{ s==0\}}}}{\texttt{\{p>=0\} s=0 \{p>=0} \wedge \texttt{ s==0\}}}$$

and PCA_4 as follows:

$$\frac{\models \texttt{(p>=0} \wedge \texttt{ s==0)} \Rightarrow \texttt{(p>=0} \wedge \texttt{ s==0} \wedge \texttt{ 1==1)}, \quad \left(\frac{\cdot}{\texttt{\{p>=0} \wedge \texttt{ s==0} \wedge \texttt{ 1 == 1\} n=1 \{p>=0} \wedge \texttt{ s==0} \wedge \texttt{ n==1\}}}\right)}{\texttt{\{p>=0} \wedge \texttt{ s==0\} n=1 \{p>=0} \wedge \texttt{ s==0} \wedge \texttt{ n==1\}}}$$

In order to show the partial correctness assertion PCA_2, that is, $\texttt{\{p>=0} \wedge \texttt{ s==0} \wedge \texttt{ n==1\} while(n<=p)(s=s+n;n=n+1) \{s==p*(p+1)/2\}}$, we pick the invariant `A` in the rule associated to while loops to be `n<=p+1` $\wedge$ `s==n*(n-1)/2`. We first have to show that `A` is indeed an invariant, that is, that $\texttt{\{A} \wedge \texttt{ n<=p\} s=s+n;n=n+1 \{A\}}$:

$$\frac{PCA_5, \ \texttt{\{n+1<=p+1} \wedge \texttt{ s==(n+1)*((n+1)-1)/2\} n=n+1 \{n<=p+1} \wedge \texttt{ s==n*(n-1)/2\}}}{\texttt{\{n<=p+1} \wedge \texttt{ s==n*(n-1)/2} \wedge \texttt{ n<=p\} s=s+n;n=n+1 \{n<=p+1} \wedge \texttt{ s==n*(n-1)/2\}}}$$

where PCA_5 can be derived as follows:

$$\frac{\models \texttt{(n<=p+1} \wedge \texttt{ s==n*(n-1)/2} \wedge \texttt{ n<=p)} \Rightarrow \texttt{B}, \quad PCA_6}{\texttt{\{n<=p+1} \wedge \texttt{ s==n*(n-1)/2} \wedge \texttt{ n<=p\} s=s+n \{n+1<=p+1} \wedge \texttt{ s==(n+1)*((n+1)-1)/2\}}}$$

where `B` is the assertion `n+1<=p+1` $\wedge$ `s+n==(n+1)*((n+1)-1)/2`,

and PCA_6 is the partial condition assertion:

$$\frac{\cdot}{\texttt{\{B\} s=s+n \{n+1<=p+1 } \wedge \texttt{ s==(n+1)*((n+1)-1)/2\}}}$$

Therefore, the assertion `A` above is indeed an invariant, so by the Hoare rule of the while loop we can derive

$$\frac{\texttt{\{A } \wedge \texttt{ n<=p\} s=s+n;n=n+1 \{A\}}}{\texttt{\{A\} while(n<=p) (s=s+n;n=n+1) \{A } \wedge \neg\texttt{(n<=p)\}}}$$

PCA_2 now can be derived by noting that

$$\models \texttt{(p>=0 } \wedge \texttt{ s==0 } \wedge \texttt{ n==1)} \Rightarrow \texttt{(n<=p+1 } \wedge \texttt{ s==n*(n-1)/2)}$$

and that

$$\models \texttt{(n<=p+1 } \wedge \texttt{ s==n*(n-1)/2 } \wedge \neg\texttt{(n<=p))} \Rightarrow \texttt{s==p*(p+1)/2}$$

The hard part in proofs in Hoare logic is to find the appropriate invariants. In the example above, one would not be able to do the proof if one considered the invariant `A` to be just `s==n*(n-1)/2` instead of `n<=p+1 ∧ s==n*(n-1)/2`. The extra condition `n<=p+1`

may seem vacuous, but notice that it is very important to prove the partial correctness assertion for the while loop.

Exercise 1 *Prove the following partial correctness assertion using the Hoare logic inference rules:*

```
{x == n and n >= 0 and y == 1}
  while(x > 0) (
    y = x * y ;
    x = x - 1)
{y = n!}
```

where $n! = 1 * 2 * \cdots * n$ *is the factorial of* n*. As usual, do all the state assertion validity computations by hand and only show the applications of the Hoare inference rules.*

Homework Exercise 1 *Prove the following partial correctness assertion using the Hoare logic inference rules:*

```
{x == m and y == n and n >= 1 and z == 1}
  while (y >= 1) (
    while (even?(y)) (
      x = x * x ;
      y = y / 2) ;
    z = z * x ;
    y = y - 1)
{z = m^n}
```

where `m^n` *is the power of* `m` *to the* `n` *and* `even?(y)` *tests whether* `y` *is an event number. As usual, do all the state assertion validity computations by hand and only show the applications of the Hoare inference rules.*

CS322 - Programming Language Design

Lecture 26: Defining a Multithreaded Language (Part 1)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

In the remaining part of the class, we discuss concurrency-related aspects of programming languages. More precisely, we will define an extension of our functional language in which an arbitrary number of threads can execute concurrently, communicating with each other via shared variables.

One can define multithreaded languages in many different ways. Our preferred method, which we will pursue the next lecture, is to extend a *continuation-based* definition of a single-threaded version of the language, in such a way that several continuations live together in the state, one continuation per thread.

Before that, we informally discuss *rewriting logic*, a logic for concurrency which is supported by Maude. Like we did with equational logic, instead of discussing the underlying theory of rewriting logic, we prefer to just introduce it informally via examples.

A Simple Vending Machine

Vending machines are (simple) concurrent and/or nondeterministic systems by their nature, because one can perform several actions at the same time in a random order. For example, one can insert coins of different values and request items at the same time, or first insert a large number of coins and then request all the desired items, etc.

We next assume an over-simplified vending machine which accepts dollars (`$`) and quarters (`q`), and which sells coffee (`c`) and tea (`t`), the first for a dollar and the second for three quarters. To keep the problem simple, assume that our vending machine requires its user to first insert the needed amount of money and only after that to request items. To keep it non-trivial, assume that our vending machine has a bug: one is not allowed to buy tea by simply providing three quarters, but only by providing a dollar for which one gets a tea and a quarter back. To compensate for this bug, the

vending machine allows one to change four quarters into a dollar. The following is a *rewriting logic* specification of such a simple and buggy vending machine:

```
mod VENDING-MACHINE is
  sorts Coin Item State .
  subsorts Coin Item < State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .
  ops $ q : -> Coin .
  ops c t : -> Item .
  op init : -> State .
  eq init = $ $ q q .

  rl $ => c .
  rl $ => t q .
  rl q q q q => $ .
endm
```

We assume that the state of the vending machine reflects the result of an interaction with a hypothetical user. Thus, a state can contain dollars and quarters, but also coffee and tea, with the meaning that the former are still available for use, while the later were already delivered to the user. We assume that `init` is an initial state, which can be seen as the state in which the user is ready to order his/her desired items (one could add special rules for adding money to the machine, but we prefer to keep it simple).

Rewriting rules are introduced with the keyword `rl`. Unlike equations whose meaning is that the two terms are interpreted to equal values in any admissible model/implementation, the meaning of rewriting rules is that of *transformations* or *transitions*. While equations can be used either from left-to-right or from right-to-left in equational reasoning, as we did for example when we proved the equivalence of semantics, rewriting rules can be used only from left-to-right. Therefore, the actions that they denote are

irreversible. Indeed, note that the first two rules say that one can change a dollar for a coffee or for a tea and a quarter back, but that the reverse operations are not allowed. The third rule states that four quarters can be changed to a dollar (so can be used to buy coffee or tea), but that the reverse process is not allowed.

The Maude command `reduce`, or its abbreviation `red`, can only reduce terms by applying equations in a left-to-right strategy. It cannot enable rewriting rules. There is a special command, `rewrite` with its abbreviation `rew`, which enables both the rewriting rules and the equations. For example,

```
rew init .
***> can be: q q q c t
```

Operationally, one can and should think of `rew` as working exactly like `red`, where the equations and the rules are treated the same way, from left to right. Semantically, there is an important difference between the two: `red` reduces terms to equal terms,

while `rew` transforms terms into terms to which they can correctly transit. Thus, one can intuitively think of reductions as atomic, or instantaneous, and of rewrites as taking place in a certain amount of time necessary to perform the transition.

Good equational specification practice suggests that one should device *canonical* specifications, that is, specifications which, if regarded as term rewriting systems, terminate and return a unique normal form on any term. Thus, applying reductions with `red` is equivalent to evaluating some well-defined function. This is one of the reasons for which the equational specifications were also called *functional*, and were introduced in Maude using the keywords `fmod ... endfm`.

Rewriting logic specifications are typically *not* canonical, reflecting the intuition that systems can evolve in different consistent ways. For example, a dollar can evolve into a coffee or into a tea and a quarter back. Rewriting logic specifications typically do *not*

terminate, reflecting the intuition that many systems are reactive, in the sense that they are continuously ready to answer to external stimuli. If we had rules "`rl S => S $`" and "`rl S => S q`" for adding new dollars and quarters, then our vending machine specification would have not terminated either.

The `rew` command takes an optional argument telling it how many rewrite rules to apply:

```
rew [1] init .
***> can be: $ q q c
```

The current term is internally stored by Maude, and so one can continue its rewriting by specifying a new number of steps together with the command `cont`:

```
cont 1 .
***> can be: q q q c t
cont 1 .
***> stuck at: q q q c t
```

One important property to verify in a concurrent system is whether certain states are *reachable*, that is, whether there is any sequence of rewrite rules (mixed with equational reductions) that leads to those states. One natural and obvious to answer question in the case of our vending machine is whether one can buy 2 cups of tea. Maude provides a very useful command, `search`, allowing one to systematically explore the state space for states matching a given pattern. In our case, one can type:

```
search init =>+ t t S:State .
```

where `=>+` stays for "one or more steps", and one gets the following output listing all four states matching the pattern "two cups of tea":

```
Solution 1 (state 5)
states: 6  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> q q q q
```



```
Solution 2 (state 6)
states: 7  rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> $

Solution 3 (state 7)
states: 8  rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> c

Solution 4 (state 8)
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> q t

No more solutions.
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
```

Each solution refers to a specific state in the state space and returns the substitution under which the given pattern matches the corresponding state.

Sometimes one is interested in only a limited number of solutions, usually one, especially in situations in which the rewriting specification does not terminate. The number of desired solutions can be given as an optional argument:

```
search [1] init =>+ t t S:State .
```

outputs:

```
Solution 1 (state 5)
states: 6  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> q q q q
```

There are situations in which one is only interested in solutions which are terminal states, that is, states which cannot be rewritten further. This can be specified using the special arrow "`=>!`":

```
search init =>! t t S:State .
```

outputs the following two non-rewritable solutions:

```
Solution 1 (state 7)
```



```
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> c

Solution 2 (state 8)
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> q t

No more solutions.
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
```

You may have noticed that it is even possible to get three cups of tea and a quarter back. Indeed,

```
search init =>! t t t S:State .
```

outputs the expected result:

```
Solution 1 (state 8)
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
S:State --> q
```

```
No more solutions.
states: 9  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
```

Knowing that there is a solution may not be sufficient in all applications. There may be situations in which one wants to know the sequence of transitions which led to the solution. Knowing the state number in the search graph associated to the satisfying solution, 8 in the case above, one can ask Maude to output the path from the original state to the desired state using a command like the one below:

```
show path 8 .
```

which outputs:

```
state 0, State: $ $ q q
===[ rl $ => q t . ]===>
state 2, State: $ q q q t
===[ rl $ => q t . ]===>
```



```
state 5, State: q q q q t t
===[ rl q q q q => $ . ]===>
state 6, State: $ t t
===[ rl $ => q t . ]===>
state 8, State: q t t t
```

One can even ask for the entire search graph, as follows:

```
show search graph .
```

The search graph generated by the last search command is displayed, which may be only a small part of the entire state space. In our case we obtain the output:

```
state 0, State: $ $ q q
arc 0 ===> state 1 (rl $ => c [label coffee] .)
arc 1 ===> state 2 (rl $ => q t [label tea] .)

state 1, State: $ q q c
arc 0 ===> state 3 (rl $ => c [label coffee] .)
```

```
arc 1 ===> state 4 (rl $ => q t [label tea] .)

state 2, State: $ q q q t
arc 0 ===> state 4 (rl $ => c [label coffee] .)
arc 1 ===> state 5 (rl $ => q t [label tea] .)

state 3, State: q q c c

state 4, State: q q q c t

state 5, State: q q q q t t
arc 0 ===> state 6 (rl q q q q => $ [label change] .)

state 6, State: $ t t
arc 0 ===> state 7 (rl $ => c [label coffee] .)
arc 1 ===> state 8 (rl $ => q t [label tea] .)

state 7, State: c t t
```



```
state 8, State: q t t t
```

This simple example practically showed all Maude's features and commands that we will need to use in order to define our multithreaded functional programming language and to analyze multithreaded programs in that language.

Next lecture we will modify/extend the continuation-based definition of the sequential version of our language shown in the file `k-eq.maude`. We will discuss this definition in more detail in the next lecture.

Exercise 1 *Read the language definition in the file* `k-eq.maude`.

```
--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _and_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  ops car cdr : Exp -> Exp .
  op cons : Exp Exp -> Exp .
  op emptyList : -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
```

```
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PL-SYNTAX is
  extending AEXP-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
```

```
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
  op l : Nat Nat -> LocationList .
  vars N # : Nat .
  eq l(N,0) = noLoc .
  eq l(N,#) = l(N), l(N + 1, # - 1) .
endfm

fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
```

```
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[noLoc <- noVal] = M .
  eq ([L,V] M)[L,Ll <- V',Vl] = ([L,V'] M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op _,_ : State State -> State [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
endfm

fmod GENERIC-EXP-K-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op [_@_] ->_ : ExpList Env Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
```

```
  vars E E' : Exp .  var El : ExpList .
  var V : Value .  var Vl : ValueList .
  var L : Location .  var Ll : LocationList .
  var X : Name .  var Xl : NameList .
  var S : State .  var I : Int .  var K : Continuation .
  var M : Store .  vars Env Env' : Env .

  op int : Int -> Value .
  eq k([I @ Env] -> K) = k(int(I) -> K) .
  eq k([X @ ([X,L] Env)] -> K), m([L,V] M) = k(V -> K), m([L,V] M) .

  op [_@_|_] -> _ : ExpList Env ValueList Continuation -> Continuation .
---  eq k(exp() -> K) = k(val(noVal) -> K) .
  eq k([(E,E',El) @ Env] -> K) =
     k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
  eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
  eq k(V -> [(E,El) @ Env | Vl] -> K) =
     k([E @ Env] -> [El @ Env | Vl,V] -> K) .

  op _->_ : LocationList Continuation -> Continuation .
  eq k(Vl -> Ll -> K), m(M) = k(K), m(M[Ll <- Vl]) .

  op length_ : NameList -> Nat .
  eq length() = 0 .
  eq length(X,Xl) = 1 + length(Xl) .
endfm

fmod AEXP-K-SEMANTICS is protecting AEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  vars E E' : Exp .  vars I I' : Int .
  var K : Continuation .  var Env : Env .
  eq k([(E + E') @ Env] -> K) = k([(E,E') @ Env] -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
  eq k([(E - E') @ Env] -> K) = k([(E,E') @ Env] -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k([(E * E') @ Env] -> K) = k([(E,E') @ Env] -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k([(E / E') @ Env] -> K) = k([(E,E') @ Env] -> / -> K) .
```

```
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
endfm

fmod BEXP-K-SEMANTICS is protecting BEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op not -> _ : Continuation -> Continuation .
  op and -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .  var Env : Env .
  eq k([(E == E') @ Env] -> K) = k([(E,E') @ Env] -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k([(E >= E') @ Env] -> K) = k([(E,E') @ Env] -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k([(E <= E') @ Env] -> K) = k([(E,E') @ Env] -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k([(not E) @ Env] -> K) = k([E @ Env] -> not -> K) .
  eq k(bool(B) -> not -> K) = k(bool(not B) -> K) .
  eq k([(E and E') @ Env] -> K) = k([(E,E') @ Env] -> and -> K) .
  eq k((bool(B),bool(B')) -> and -> K) = k(bool(B and B') -> K) .
endfm

fmod LIST-K-SEMANTICS is protecting LIST-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op list -> _ : Continuation -> Continuation .
  op car -> _ : Continuation -> Continuation .
  op cdr -> _ : Continuation -> Continuation .
  op cons -> _ : Continuation -> Continuation .
  op null? -> _ : Continuation -> Continuation .
  var E E' : Exp .  var El : ExpList .  var K : Continuation .
  var V : Value .  var Vl : ValueList .  var Env : Env .
  eq k([list(El) @ Env] -> K) = k([El @ Env] -> list -> K) .
  eq k(Vl -> list -> K) = k([Vl] -> K) .
  eq k([car(E) @ Env] -> K) = k([E @ Env] -> car -> K) .
  eq k([V,Vl] -> car -> K) = k(V -> K) .
  eq k([cdr(E) @ Env]  -> K) = k([E @ Env] -> cdr -> K) .
  eq k([V,Vl] -> cdr -> K) = k([Vl] -> K) .
  eq k([emptyList @ Env] -> K) = k([noVal] -> K) .
  eq k([cons(E,E') @ Env] -> K) = k([(E,E') @ Env] -> cons -> K) .
  eq k((V,[Vl]) -> cons -> K) = k([V,Vl] -> K) .
```

```
  eq k([null?(E) @ Env] -> K) = k([E @ Env] -> null? -> K) .
  eq k([Vl] -> null? -> K) = k(bool(noVal == Vl) -> K) .
endfm

fmod IF-K-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [if(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E E' : Exp .  var B : Bool .
  var K : Continuation .  var Env : Env .
  eq k([(if BE then E else E') @ Env] -> K) =
     k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K) =
     k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endfm

fmod BINDING-K-SEMANTICS is protecting BINDING-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  sort Aux .
  op a : Nat NameList ExpList BindingList -> Aux .
  var N : Nat .  var Xl : NameList .  var El : ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq a(N, Xl ,El, (X = E, Bl)) = a(N + 1, (Xl,X), (El,E), Bl) .
endfm

fmod LET-K-SEMANTICS is protecting LET-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op let : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq let Bl in E = let(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([let(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K), n(N + #) .
endfm

fmod PROC-K-SEMANTICS is protecting PROC-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op function : Nat NameList Exp -> Exp .
  op fn -> _ : Continuation -> Continuation .
  op closure : Nat NameList Exp Env -> Value .
  var Xl : NameList .  var El : ExpList .  var F E : Exp .
  var K : Continuation .  var Env : Env .  var M : Store .
  vars N # : Nat .  var Vl : ValueList .
```

```
  eq proc(Xl) E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E) @ Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K), n(N), m(M) =
     k([E @ Env[Xl <- l(N,#)]] -> K), n(N + #), m(M[l(N,#) <- Vl]) .
endfm

fmod LETREC-K-SEMANTICS is protecting LETREC-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op letrec : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq letrec Bl in E = letrec(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([letrec(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env[Xl <- l(N,#)]] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K),
       n(N + #) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var K : Continuation .  var L : Location .
  eq k([(set X = E) @ ([X,L] Env)] -> K) =
     k([E @ ([X,L] Env)] -> L -> int(1) -> K) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op ignore -> _ : Continuation -> Continuation .
  var E : Exp .  var El; : ExpList; .  var Env : Env .
  var K : Continuation .  var V : Value .
  eq k([{E} @ Env] -> K) = k([E @ Env] -> K) .
  eq k([{E ; El;} @ Env] -> K) = k([E @ Env] -> ignore
        -> ([{El;} @ Env] -> K)) .
  eq k(V -> ignore -> K) = k(K) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [while(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E : Exp .  var Vl : ValueList .
  var K : Continuation .  var Env :  Env .
```

```
  eq k([(while BE E) @ Env] -> K) =
     k([BE @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(true)) -> [while(BE,E) @ Env] -> K) =
     k([(E,BE) @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(false)) -> [while(BE,E) @ Env] -> K) = k(int(1) -> K) .
endfm

fmod PL-K-SEMANTICS is protecting PL-SYNTAX .
  extending AEXP-K-SEMANTICS .
  extending BEXP-K-SEMANTICS .
  extending LIST-K-SEMANTICS .
  extending IF-K-SEMANTICS .
  extending LET-K-SEMANTICS .
  extending PROC-K-SEMANTICS .
  extending LETREC-K-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .

  op eval : Exp -> [Value] .
  op [_] : State -> [Value] .
  var E : Exp .  var V : Value .  var S : State .
  eq eval(E) = [k([E @ noEnv] -> stop), n(0), m(noStore)] .
  eq [k(V -> stop), S] = V .
endfm

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
```

```
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) (x * (y - z))
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
```

```
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
```

```
    in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
          if n == 0
          then 1
          else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
```

```
                if x == 0
                then 0
                else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
             if n == 0
             then 1
             else n * f(n - 1)
  in f(5)
) .
***> should be 120

red eval(
  letrec 'times4 = proc(x)
                   if x == 0
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

red eval(
  letrec 'even = proc(x)
                 if x == 0
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if x == 0
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
```

```
    in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if y == 0 then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
             let d = set c = c + 1
```

```
            in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
            let d = set x = x + 1
            in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
            let t = x
            in let d = set x = y
               in let d = set y = t
                  in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
          if a == 0 then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       'even = proc() if x == 0
                  then 1
                  else let d = set x = x - 1
                       in 'odd(),
       'odd  = proc() if x == 0
                  then 0
```

```
                else let d = set x = x - 1
                     in 'even()
    in let d = set x = 7
       in 'odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         'even = proc() if x == 0 then 1
                  else let d = set x = x - 1
                       in 'odd(),
         'odd = proc() if x == 0 then 0
                  else let d = set x = x - 1
                       in 'even()
  in 'odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                  if x == 0
                  then 0
```

```
                else 4 + 'times4(x - 1) ;
     'times4(3)
    }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
     f = proc(a, b)
         {
          set a = a + b ;
          set b = a - b ;
          set a = a - b
         }
  in {
      f(x,y) ;
      x
    }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
         f(set y = y + 3) ;
         y
        }
) .
***> should be 8

red eval(
  let y = 5,
     f = proc(x) x + x,
     g = proc(x) set x = x + 3
  in {
      f(g(y));
      y
    }
) .
***> should be 5

red eval(
```

```
  let n = 178378342647, c = 0
  in { while not (n == 1) {
         set c = c + 1 ;
         if 2 * (n / 2) == n
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if x == 0
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      'odd = proc(x, o, e)
               if x == 0 then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if x == 0 then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
```

```
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                y = 2,
                a = 3,
                z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if n == 0
               then m
               else f(n - 1, m * n)
  in f(100, 1)
) .
***> works for 50000: a number of 213237 digits in about 40 seconds

red eval(
  letrec f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
  in f(10000)
) .
***> works for 70000: a number of 308760 digits in about 60 seconds

red eval(
  letrec a = proc(l,r)
               if null?(l) then r
               else cons(car(l), a(cdr(l), r)),
         h = proc(l,i,r,n)
               if n == 1 then list(list(l,r))
               else a(a(h(l, r, i, n - 1), list(list(l,r))),
                      h(i, l, r, n - 1))
  in h(1,2,3,3)
) .

red eval(
```

```
  letrec f = proc(n,l)
             if null?(l) then -1
             else if n == car(l)
                 then 1
                 else let p = f(n, cdr(l))
                      in if p == -1 then -1
                         else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .
```

```
--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _and_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  ops car cdr : Exp -> Exp .
  op cons : Exp Exp -> Exp .
  op emptyList : -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm
```

```
fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod PL-SYNTAX is
  extending AEXP-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
endfm


-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
  op l : Nat Nat -> LocationList .
  vars N # : Nat .
  eq l(N,0) = noLoc .
  eq l(N,#) = l(N), l(N + 1, # - 1) .
endfm

fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[noLoc <- noVal] = M .
  eq ([L,V] M)[L,Ll <- V',Vl] = ([L,V'] M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op _,_ : State State -> State [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
endfm

fmod GENERIC-EXP-K-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op [_@_] ->_ : ExpList Env Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .

  vars E E' : Exp .  var El : ExpList .
  var V : Value .  var Vl : ValueList .
  var L : Location .  var Ll : LocationList .
  var X : Name .  var Xl : NameList .
  var S : State .  var I : Int .  var K : Continuation .
  var M : Store .  vars Env Env' : Env .

  op int : Int -> Value .
  eq k([I @ Env] -> K) = k(int(I) -> K) .
  eq k([X @ ([X,L] Env)] -> K), m([L,V] M) = k(V -> K), m([L,V] M) .

  op [_@_|_] -> _ : ExpList Env ValueList Continuation -> Continuation .
---  eq k(exp() -> K) = k(val(noVal) -> K) .
  eq k([(E,E',El) @ Env] -> K) =
     k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
  eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
  eq k(V -> [(E,El) @ Env | Vl] -> K) =
     k([E @ Env] -> [El @ Env | Vl,V] -> K) .

  op _->_ : LocationList Continuation -> Continuation .
  eq k(Vl -> Ll -> K), m(M) = k(K), m(M[Ll <- Vl]) .

  op length_ : NameList -> Nat .
  eq length() = 0 .
  eq length(X,Xl) = 1 + length(Xl) .
endfm
```

```
fmod AEXP-K-SEMANTICS is protecting AEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  vars E E' : Exp .  vars I I' : Int .
  var K : Continuation .  var Env : Env .
  eq k([(E + E') @ Env] -> K) = k([(E,E') @ Env] -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
  eq k([(E - E') @ Env] -> K) = k([(E,E') @ Env] -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k([(E * E') @ Env] -> K) = k([(E,E') @ Env] -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k([(E / E') @ Env] -> K) = k([(E,E') @ Env] -> / -> K) .
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
endfm

fmod BEXP-K-SEMANTICS is protecting BEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op not -> _ : Continuation -> Continuation .
  op and -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .  var Env : Env .
  eq k([(E == E') @ Env] -> K) = k([(E,E') @ Env] -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k([(E >= E') @ Env] -> K) = k([(E,E') @ Env] -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k([(E <= E') @ Env] -> K) = k([(E,E') @ Env] -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k([(not E) @ Env] -> K) = k([E @ Env] -> not -> K) .
  eq k(bool(B) -> not -> K) = k(bool(not B) -> K) .
  eq k([(E and E') @ Env] -> K) = k([(E,E') @ Env] -> and -> K) .
  eq k((bool(B),bool(B')) -> and -> K) = k(bool(B and B') -> K) .
endfm

fmod LIST-K-SEMANTICS is protecting LIST-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op list -> _ : Continuation -> Continuation .
  op car -> _ : Continuation -> Continuation .
  op cdr -> _ : Continuation -> Continuation .
  op cons -> _ : Continuation -> Continuation .
  op null? -> _ : Continuation -> Continuation .
  var E E' : Exp .  var El : ExpList .  var K : Continuation .
  var V : Value .  var Vl : ValueList .  var Env : Env .
  eq k([list(El) @ Env] -> K) = k([El @ Env] -> list -> K) .
  eq k(Vl -> list -> K) = k([Vl] -> K) .
  eq k([car(E) @ Env] -> K) = k([E @ Env] -> car -> K) .
  eq k([V,Vl] -> car -> K) = k(V -> K) .
  eq k([cdr(E) @ Env]  -> K) = k([E @ Env] -> cdr -> K) .
  eq k([V,Vl] -> cdr -> K) = k([Vl] -> K) .
  eq k([emptyList @ Env] -> K) = k([noVal] -> K) .
  eq k([cons(E,E') @ Env] -> K) = k([(E,E') @ Env] -> cons -> K) .
  eq k((V,[Vl]) -> cons -> K) = k([V,Vl] -> K) .
  eq k([null?(E) @ Env] -> K) = k([E @ Env] -> null? -> K) .
  eq k([Vl] -> null? -> K) = k(bool(noVal == Vl) -> K) .
endfm

fmod IF-K-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [if(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
```

```
  vars BE E E' : Exp .  var B : Bool .
  var K : Continuation .  var Env : Env .
  eq k([(if BE then E else E') @ Env] -> K) =
     k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K) =
     k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endfm

fmod BINDING-K-SEMANTICS is protecting BINDING-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  sort Aux .
  op a : Nat NameList ExpList BindingList -> Aux .
  var N : Nat .  var Xl : NameList .  var El : ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq a(N, Xl ,El, (X = E, Bl)) = a(N + 1, (Xl,X), (El,E), Bl) .
endfm

fmod LET-K-SEMANTICS is protecting LET-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op let : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq let Bl in E = let(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([let(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K), n(N + #) .
endfm

fmod PROC-K-SEMANTICS is protecting PROC-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op function : Nat NameList Exp -> Exp .
  op fn -> _ : Continuation -> Continuation .
  op closure : Nat NameList Exp Env -> Value .
  var Xl : NameList .  var El : ExpList .  var F E : Exp .
  var K : Continuation .  var Env : Env .  var M : Store .
  vars N # : Nat .  var Vl : ValueList .
  eq proc(Xl) E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E) @ Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K), n(N), m(M) =
     k([E @ Env[Xl <- l(N,#)]] -> K), n(N + #), m(M[l(N,#) <- Vl]) .
endfm

fmod LETREC-K-SEMANTICS is protecting LETREC-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op letrec : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq letrec Bl in E = letrec(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([letrec(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env[Xl <- l(N,#)]] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K),
       n(N + #) .
endfm

fmod VAR-ASSIGNMENT-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var K : Continuation .  var L : Location .
  eq k([(set X = E) @ ([X,L] Env)] -> K) =
     k([E @ ([X,L] Env)] -> L -> int(1) -> K) .
endfm

fmod BLOCK-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op ignore -> _ : Continuation -> Continuation .
  var E : Exp .  var El; : ExpList; .  var Env : Env .
  var K : Continuation .  var V : Value .
  eq k([{E} @ Env] -> K) = k([E @ Env] -> K) .
  eq k([{E ; El;} @ Env] -> K) = k([E @ Env] -> ignore
        -> ([{El;} @ Env] -> K)) .
  eq k(V -> ignore -> K) = k(K) .
endfm

fmod LOOP-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [while(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E : Exp .  var Vl : ValueList .
  var K : Continuation .  var Env :  Env .
  eq k([(while BE E) @ Env] -> K) =
     k([BE @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(true)) -> [while(BE,E) @ Env] -> K) =
     k([(E,BE) @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(false)) -> [while(BE,E) @ Env] -> K) = k(int(1) -> K) .
endfm

fmod PL-K-SEMANTICS is protecting PL-SYNTAX .
  extending AEXP-K-SEMANTICS .
  extending BEXP-K-SEMANTICS .
  extending LIST-K-SEMANTICS .
  extending IF-K-SEMANTICS .
  extending LET-K-SEMANTICS .
  extending PROC-K-SEMANTICS .
  extending LETREC-K-SEMANTICS .
  extending VAR-ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending LOOP-SEMANTICS .

  op eval : Exp -> [Value] .
  op [_] : State -> [Value] .
  var E : Exp .  var V : Value .  var S : State .
  eq eval(E) = [k([E @ noEnv] -> stop), n(0), m(noStore)] .
  eq [k(V -> stop), S] = V .
endfm

red eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

red eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

red eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

red eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

red eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

red eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

red eval(
  proc(x, y, z) (x * (y - z))
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

red eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

red eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

red eval(
   (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

red eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

red eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

red eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

red eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

red eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

red eval(
  let f = proc(n)
            if n == 0
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

red eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

red eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

red eval(
  let 'makemult = proc('maker, x)
                    if x == 0
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

red eval(
  letrec f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
```

```
  in f(5)
) .
***> should be 120

red eval(
  letrec ’times4 = proc(x)
                     if x == 0
                     then 0
                     else 4 + ’times4(x - 1)
  in ’times4(3)
) .
***> should be 12

red eval(
  letrec ’even = proc(x)
                   if x == 0
                   then 1
                   else ’odd(x - 1),
         ’odd = proc(x)
                   if x == 0
                   then 0
                   else ’even(x - 1)
  in ’odd(17)
) .
***> should be 1

red eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

red eval(
  let x = 10
  in letrec f = proc(y) if y == 0 then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

red eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

red eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

red eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

red eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

red eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

red eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if a == 0 then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

red eval(
  let x = 0
  in letrec
       ’even = proc() if x == 0
                      then 1
                      else let d = set x = x - 1
                           in ’odd(),
       ’odd  = proc() if x == 0
                      then 0
                      else let d = set x = x - 1
                           in ’even()
     in let d = set x = 7
        in ’odd()
) .
***>  should be 1

red eval(
  letrec x = 18,
         ’even = proc() if x == 0 then 1
                        else let d = set x = x - 1
                             in ’odd(),
          ’odd = proc() if x == 0 then 0
                        else let d = set x = x - 1
                             in ’even()
  in ’odd()
) .
***>  should be 0

red eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

red eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

red eval(
  let ’times4 = 0
  in {
       set ’times4 = proc(x)
                       if x == 0
                       then 0
                       else 4 + ’times4(x - 1) ;
       ’times4(3)
     }
) .
***> should be 12

red eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

red eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

red eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

red eval(
  let n = 178378342647, c = 0
  in { while not (n == 1) {
         set c = c + 1 ;
         if 2 * (n / 2) == n
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

red eval(
  let f = proc(x, g)
            if x == 0
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

red eval(
  let x = 17,
      ’odd = proc(x, o, e)
                 if x == 0 then 0
                 else e(x - 1, o, e),
      ’even = proc(x, o, e)
                 if x == 0 then 1
                 else o(x - 1, o, e)
  in ’odd(x, ’odd, ’even)
) .
***> should be 1

red eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

red eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

red eval( letrec f = proc(x) z + x + 5,
                  y = 2,
                  a = 3,
                  z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

red eval(
  letrec f = proc(n,m)
               if n == 0
               then m
               else f(n - 1, m * n)
  in f(100, 1)
```

```
) .
***> works for 50000: a number of 213237 digits in about 40 seconds

red eval(
  letrec f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
  in f(10000)
) .
***> works for 70000: a number of 308760 digits in about 60 seconds

red eval(
  letrec a = proc(l,r)
               if null?(l) then r
               else cons(car(l), a(cdr(l), r)),
         h = proc(l,i,r,n)
               if n == 1 then list(list(l,r))
               else a(a(h(l, r, i, n - 1), list(list(l,r))),
                      h(i, l, r, n - 1))
  in h(1,2,3,3)
) .

red eval(
  letrec f = proc(n,l)
               if null?(l) then -1
               else if n == car(l)
                    then 1
                    else let p = f(n, cdr(l))
                         in if p == -1 then -1
                            else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .
```

CS322 - Programming Language Design

Lecture 27: Defining a Multithreaded Language (Part 2)

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We now define a non-trivial multithreaded programming language. Our approach is to enrich a previous continuation-based definition of a functional language with threads. By defining a multithreaded/concurrent language in Maude, we have two major benefits:

1. On the one hand, like before, we will get an *interpreter for free*. A thread scheduler for the defined programming language will be implicitly obtained as a consequence of Maude's internal rewriting rule application scheduler. In this version of our language definition we have no control on the thread scheduler.
2. On the other hand, and perhaps even more importantly, we get a *multithreaded program analysis tool* for free, which uses Maude's `search` command to find whether the program can reach certain good or bad states.

Syntax for Standard Functional Features

Most of the syntax below is almost identical to the syntax of our previous languages. We list it below just for the sake of completeness:

```
fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
```



```
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _and_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
```

```
  op list_ : ExpList -> Exp .
  ops car cdr : Exp -> Exp .
  op cons : Exp Exp -> Exp .
  op emptyList : -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm
```



```
fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm
```

```
fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm
```


Syntax for Concurrency

We next introduce the syntax of the concurrency-related features. The next module introduces the language construct `spawn`, which takes an expression and return another expression. Its intended meaning, which is rigorously defined later, is to *create a new thread* that executes the argument expression in the current environment. The new thread executes concurrently with the other thread(s), and they all have access to the same store and can obviously *share data*.

```
fmod SPAWN-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op spawn_ : Exp -> Exp [prec 50] .
endfm
```

The following is an example expression that uses multiple threads of execution:

```
let a = 0, b = 0, c = 0
in let x = spawn letrec l = proc()
                              if not(b == 0) then set c = b + 1
                              else l()
                 in l(),
       y = spawn letrec l = proc()
                              if not(a == 0) then set b = a + 1
                              else l()
                  in l(),
       z = spawn set a = 1
   in letrec l = proc() if not(c == 0) then c else l()
      in l()
***> should be 3
```


Since threads can share data, there is an inherent potential for *data-races* (we will see examples later). To avoid data-races, synchronization mechanisms are needed to be added to the language. There are different ways to add synchronization to a language. We will adopt one of the simplest: locks which can be acquired and released by threads. We will use the following syntax:

```
fmod SYNCHRONIZATION-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op lock : Exp -> Exp .
  op acquire_ : Exp -> Exp .
  op release_ : Exp -> Exp .
endfm
```

Thus a lock is just like any other expression in the language, so it can be passed to and returned from functions, etc.

Putting All the Syntax Together

```
fmod PL-SYNTAX is
  extending AEXP-SYNTAX .  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .  extending IF-SYNTAX .
  extending LET-SYNTAX .   extending LETREC-SYNTAX .
  extending PROC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending SPAWN-SYNTAX .
  extending SYNCHRONIZATION-SYNTAX .
endfm
```

We will next define the entire multithreaded language except the synchronization part, which is only discussed informally and left as a homework exercise (the last one in the class) for you.

Defining the State Infrastructure

As usual, we need locations, environments, values and stores:

```
fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
  op l : Nat Nat -> LocationList .
  vars N # : Nat .
  eq l(N,0) = noLoc .
  eq l(N,#) = l(N), l(N + 1, # - 1) .
endfm
```

```
fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm
```



```
fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm
```

```
fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[noLoc <- noVal] = M .
  eq ([L,V] M)[L,Ll <- V',Vl] = ([L,V'] M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M [L,V'])[Ll <- Vl] [owise] .
endfm
```


Additionally, for reasons that will become self-explanatory later, our design decision is to define a *continuation-based semantics*, so also need continuations as part of the state infrastructure:

```
fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm
```

For the time being we keep the continuations minimal, but new continuation items will be added as we add new features to the language.

We can now define the state as a set of attributes. Three attributes are important, namely continuations, the store and the next free location, which can be easily identified (or matched) in the state by being wrapped with the symbols `k`, `m` (from "memory") and `n`:

```
fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op _,_ : State State -> State [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
endfm
```


Continuation-based Semantics

The `search` command exhaustively tries all *rewriting rules* in a breadth-first strategy, until a state matching the desired pattern is found. Equations are *not* considered state transitions, so they are applied in the background without increasing the state space explored by `search`. This tells us that the more rewriting rules, the more complex the search analysis is. A major design decision in the subsequent definition is:

> *Declare as few rewriting rules as possible*. In principle, start with a purely equational specification and then conservatively analyze each equation and make it a rewriting rule if and only if it corresponds to an operational step in the execution of a program which can interact with the external world, so it can influence or be influenced by the execution of the other threads.

We will see that, for our simple language (without synchronization, which you are supposed to define as a HW exercise), there are only *two rewriting rules* needed, one for read of and the other for write into memory. The intuition for this is relatively straightforward: these are the only means by which the threads can interact with each other. Everything else is "internal cooking", that is, has nothing to do with the external world but only with the internal machinery and conventions on how to evaluate expressions.

Think, for example, of an expression `3 + 5` encountered within the execution of a thread. This expression can obviously be "internally" evaluated to `8` without "telling" anything the outside world. Suppose now that the expression is `(3 + 5) * (x + 2)`. Like before, `3 + 5` can be evaluated internally, but now an interaction with the outside world is needed in order to request the value of `x`. That's because other threads may write the variable `x`, which may happen before or after the current thread reads it. So

we have to give the `search` algorithm a chance to try both possibilities. Once `x` is read, then the remaining part of the computation can be again performed internally and atomically.

Before we start presenting the continuation based semantics, let us first recall how an expression will be later on evaluated. The code below is part of the last module putting everything together:

```
op eval : Exp -> [Value] .
op [_] : State -> [Value] .
var E : Exp .  var V : Value .  var S : State .
eq eval(E) = [k([E @ noEnv] -> stop), n(0), m(noStore)] .
eq [k(V -> stop), S] = V .
```

So the original expression is evaluated by adding a continuation to the state containing that expression at its top. The continuation-based semantics defined next will evaluate that expression to a value replacing the expression in the continuation. The bracket operation will return that value.

Each new thread will add a new continuation to the state. All the equations and rules below are generic, that is, can be applied on any of the continuations in the state. They involve, via AC matching, a *minimum amount of state knowledge*, so one can modularly add new features to this language, that may store new items in the state, without modifying the existing definitions.

Semantics of Generic Expressions

Following the modularity principles, each language construct comes together with whatever state infrastructure it needs. Generic expressions, containing names, integers and lists of expressions, therefore add a new type of value, integer numbers, and a new continuation item placing an expression list on top of an existing continuation together with its environment. Thus, unlike in some of our previous continuation-based definitions, expressions "to be

evaluated" are placed together with their environments in the continuations. This is because of the multithreaded nature of our new language: each thread executes in its own environment.

Exercise 1 *Modify the definition below such that each thread maintains its own environment. Each expression evaluated by a thread can modify that environment to its needs, but it has to recover it after its evaluation. This would significantly reduce the size of the continuations, but it would increase the amount of computation (the recovery of the environment).*

We define a rewriting logic specification (`mod ... endm`) as below. We start by adding the state infrastructure:

```
mod GENERIC-EXP-K-SEMANTICS is protecting ... (see file)
  op [_@_] ->_ : ExpList Env Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op int : Int -> Value .
```

If the expression to be evaluated in a continuation is an integer, then we just replace it by its value. Note that this operation is atomic, or internal to the thread, so we use an equation:

```
***  variable declarations; see the file
  eq k([I @ Env] -> K) = k(int(I) -> K) .
```

The next is the *first rewriting rule* and it refers to reading the value associated to a name:

```
rl k([X @ ([X,L] Env)] -> K), m([L,V] M) => k(V -> K), m([L,V] M) .
```

The above *must be a rule* because it refers to how a particular thread interacts with the outside world. It may be the case that another thread is just about to write `X`, and if it does it, the current thread can have a totally different behavior. So depending upon which of the read or write operations takes place first, the program can have different behaviors. since we want all these behaviors to be explored by `search`, we must keep the above a rewriting rule.

The next operation and equations provide the machinery needed for evaluating a list of expressions. How these expressions are evaluated has nothing to do with the external world, so all the below can safely be equations:

```
op [_@_|_] -> _ : ExpList Env ValueList Continuation -> Continuation .
eq k([(E,E',El) @ Env] -> K) =
   k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
eq k(V -> [(E,El) @ Env | Vl] -> K) =
   k([E @ Env] -> [El @ Env | Vl,V] -> K) .
```

There are situations when memory can be safely accessed without informing the outside world, in the sense that a potential reordering of such a memory access with actions in other threads does not affect the behavior of the program. These situations are the following:

- When a function is invoked (we only have call by value) and it writes its actual parameters. No other thread has access to the newly declared locations, so this memory writing operation can be considered atomic;
- When a `let` or `letrec` writes the values of its binding expressions (after they are evaluated) to the locations allocated to its binding names, for the same reason as above.

For such atomic memory block-writes we use the following equation:

```
op _->_ : LocationList Continuation -> Continuation .
eq k(Vl -> Ll -> K), m(M) = k(K), m(M[Ll <- Vl]) .
```


Semantics of Arithmetic, Boolean, List Expressions

How the subexpressions involved in an arithmetic, boolean or list operation are evaluated and the calculation of the final result can again safely be regarded as internal operations of each thread:

```
mod AEXP-K-SEMANTICS is protecting ... (see file)
  op + -> _ : Continuation -> Continuation .
  ... (see file for other operations and variables)
  eq k([(E + E') @ Env] -> K) = k([(E,E') @ Env] -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
endm
```

See the file `k-rl.maude` for the remaining equations defining the continuation-based semantics of boolean expressions and lists.

Semantics of Conditionals

It may seem that the semantics of conditionals need to be given via rewriting rules, because they involve a "choice" between their left and right branch after the condition has been evaluated. Fortunately, this is not the case. Consider, for example, the conditional "`if 3 == 2 + 4 then x + y else 2 * 5`"; its evaluation obviously has nothing to do with the other threads running concurrently, so it can be done atomically, using equations:

```
mod IF-K-SEMANTICS is protecting ...
  op [if(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
... variable declarations
  eq k([(if BE then E else E') @ Env] -> K) =
     k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K) =
     k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endm
```


Semantics of `Let`, `Letrec` and Functions

Since we aim at high efficiency besides clean definition, in this language semantics we prefer to *pre-process* each program to provide auxiliary information that allows us to devise a more efficient semantics. For example, we need to allocate new locations whenever a function is invoked or a let/letrec expression is evaluated. It is therefore useful to know apriori how many locations are needed to be allocated and also what are the names they need to be bound to. While these can be easily calculated at "runtime", one is better off preprocessing the syntax and augment the syntax tree/term of the program correspondingly. Check the file `k-rl.maude` to see how programs are pre-processed. We next assume them directly pre-processed.

Both `let` and `letrec`, despite their inherent memory write access, can in fact be treated atomically. That's because no other thread can have access to the locations allocated for their binding expressions:

```
mod LET-K-SEMANTICS is protecting ... (see file)
... (see file for additional declarations)
  var N # : Nat .
  eq k([let(#,Xl,El,E) @ Env] -> K), n(N) = n(N + #),
     k([El @ Env] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K) .
endm

mod LETREC-K-SEMANTICS is protecting ... (see file)
... (see file for additional declarations)
  eq k([letrec(#,Xl,El,E) @ Env] -> K), n(N) = n(N + #),
     k([El @ Env[Xl <- l(N,#)]] -> l(N,#) ->
           [E @ Env[Xl <- l(N,#)]] -> K) .
endm
```


There is nothing special about functions. They are first evaluated as a list of expressions and then the corresponding closure application is applied. All these can be done with equations. However, note that during the list evaluation process, if the function is referred by name, then a memory read will be performed for the corresponding closure, and this will be done using the rewriting rule for memory reads.

```
mod PROC-K-SEMANTICS is protecting ... (see file)
  op fn -> _ : Continuation -> Continuation .
  op closure : Nat NameList Exp Env -> Value .
... (see file for variable declarations)
  eq proc(Xl) E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E) @ Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K), n(N), m(M) =
     k([E @ Env[Xl <- l(N,#)]] -> K), n(N + #), m(M[l(N,#) <- Vl]) .
endm
```

Semantics of Variable Assignment

The meaning of variable assignment needs to be given by a *rewriting rule* because it involves a memory write which may lead to a different behavior of other threads reading that same location. A new continuation constructor is defined, `_=>_`, to distinguish this special memory update from the atomic block update that was used for functions and bindings:

```
mod VAR-ASSIGNMENT-K-SEMANTICS is extending ... (see file)
  op _=>_ : Location Continuation -> Continuation .
  eq k([(set X = E) @ ([X,L] Env)] -> K) =
     k([E @ ([X,L] Env)] -> L => int(1) -> K) .
  rl k(V -> L => K), m(M) => k(K), m(M[L <- V]) .
endm
```

Blocks and loops have no special treatment, so their semantics is equation based and identical to their single-threaded version.

Semantics of Spawn

In our continuation semantics, starting a new thread corresponds to nothing but *adding a new continuation* to the state. However, such a continuation has a different meaning than the continuation of the initial execution thread, because once its expression is evaluated the *thread needs to be eliminated*. Thus, threads are only used for their side-effects. The expression `spawn(E)` returns immediately with value 1, so it also used for its side-effect:

```
mod SPAWN-K-SEMANTICS is extending SPAWN-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op die : -> Continuation .
  var V : Value .  var E : Exp .  var Env : Env .  var K : Continuation .
  eq k(V -> die) = empty .
  eq k([spawn(E) @ Env] -> K) = k(int(1) -> K), k([E @ Env] -> die) .
endm
```

Examples

Let us first note the power of continuation-based language definitions. Recall that the standard factorial program was able to evaluate for `n = 7000` but not for `n = 8000` in the standard SOS-like equational semantics of the deterministic version of our language:

```
letrec f = proc(n)
             if n == 0
             then 1
             else n * f(n - 1)
in f(7000)
```

For larger values Maude generated a segmentation fault error. That was because Maude had to maintain the control context for its conditional equations, which, due to the recursive nature of the factorial program, grew quite fast.

However, with the new continuation-based semantics, Maude is required to store *no control context* at all, so we are now able to calculate the factorial of numbers as big as the memory of your computer can hold. On my computer it was able to compute `70,000!`, a number of `308760` in about 1 minute. On smaller numbers, the continuation based semantics needs fewer rewrites and it is also faster than the standard semantics.

```
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                     if a == 1 then c
                     else {set c = c + 1 ; f()}
        in f()
```

The above can return any number, depending upon how long the write of the spawn thread is delayed. The following finds an execution in which it returns 10:

```
search [1] eval(
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                     if a == 1 then c
                     else {set c = c + 1 ; f()}
        in f()
) =>! int(10) .
```


What values can the following program return?

```
  let x = 0
  in let u = spawn set x = x + 1,
         v = spawn set x = x + 1
     in x
```

Using `search`, we can see that it can return any of the values `0`, `1`, and `2`. Why? How can we make sure that the two spawn threads finished their computation when we evaluate `x` in the body of the `let`?

One possibility is to make some dummy assignments at the end of the threads' computation and then check whether they took place or not in a loop in the body of `let`:

```
let a = 1, b = 1, x = 0
in let u = spawn {set x = x + 1 ; set a = 0},
       v = spawn {set x = x + 1 ; set b = 0}
   in letrec l = proc()
                   if (a == 0) and (b == 0) then x
                   else l()
      in l()
```

Using `search`, one can see now that the program above cannot evaluate to `0` anymore, but that it can still evaluate to `1`. Why? The reason is following: both threads may read their `x` before any of them writes it, and then they can both write it! This "unexpected" error is called a *data-race* and it one of the main source for erroneous executions of concurrent programs.

Programs can have quite unexpected behaviors in the context of data-races. For example, it is conjectured by J. Moore of UT Texas that the following program can return any possible natural value!

```
search [1] eval(
  letrec c = 1,
         f = proc() {set c = c + c ; f()}
  in let a = spawn f(), b = spawn f()
     in c
) =>! int(500) .
```

For 500, `search` can prove the result in about 20 seconds. For 100 in about 1 second.

Synchronization

Nevertheless, data-races are highly undesirable in programs. Synchronization between threads is the usual way to avoid data-races.

Homework Exercise 1 *Add synchronization to our language. More precisely, add semantics for the language constructors* `lock`, `acquire` *and* `release` *with the following meaning:*

- `lock(Exp)` *evaluates* `Exp`*, which is expected to evaluate to an integer, say* i*, and then returns a new type of value wrapping that integer, say* `mutex(`i`)`*. This value acts like any other value in the language, so it can be stored and passed by functions;*
- `acquire Exp` *evaluates* `Exp`*, which is expected to evaluate to a lock value, and then the current thread acquires that lock, if not taken by any other thread, and continues its execution. If the*

lock is already taken by another thread then the current thread cannot continue its execution, but has to wait for the lock to be released;

- `release Exp` *evaluates* `Exp`*, which is expected to evaluate to a lock value, and then the current thread releases that lock in case it has it.*

A correct, synchronized, implementation of the two-thread program each incrementing `x` by `1` is shown below. Notice that locks are like any other expressions in the language. The following program should always return `2`; `search` will not be able to find any execution returning `1`:

```
let a = 1, b = 1, x = 0
in let u = spawn {
             acquire lock(1) ;
             set x = x + 1 ;
             release lock(1) ;
             set a = 0
           },
       v = spawn {
             acquire lock(1) ;
             set x = x + 1 ;
             release lock(1) ;
             set b = 0
           }
   in letrec l = proc()
                   if (a == 0) and (b == 0) then x
                   else l()
      in l()
```

```
--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _and_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  ops car cdr : Exp -> Exp .
  op cons : Exp Exp -> Exp .
  op emptyList : -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
```

```
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod SPAWN-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op spawn_ : Exp -> Exp [prec 50] .
endfm
```

```
fmod SYNCHRONIZATION-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op lock : Exp -> Exp .
  op acquire_ : Exp -> Exp .
  op release_ : Exp -> Exp .
endfm

fmod PL-SYNTAX is
  extending AEXP-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending SPAWN-SYNTAX .
  extending SYNCHRONIZATION-SYNTAX .
endfm



-----------------
--- Semantics ---
-----------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
  op l : Nat Nat -> LocationList .
  vars N # : Nat .
  eq l(N,0) = noLoc .
  eq l(N,#) = l(N), l(N + 1, # - 1) .
endfm

fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
```

```
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[noLoc <- noVal] = M .
  eq ([L,V] M)[L,Ll <- V',Vl] = ([L,V'] M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  sorts StateAttribute State .
```

```
  subsort StateAttribute < State .
  op empty : -> State .
  op _,_ : State State -> State [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
endfm

mod GENERIC-EXP-K-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op [_@_] ->_ : ExpList Env Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .

  vars E E' : Exp .  var El : ExpList .
  var V : Value .  var Vl : ValueList .
  var L : Location .  var Ll : LocationList .
  var X : Name .  var Xl : NameList .
  var S : State .  var I : Int .  var K : Continuation .
  var M : Store .  vars Env Env' : Env .

  op int : Int -> Value .
  eq k([I @ Env] -> K) = k(int(I) -> K) .
*** should be rl
  rl k([X @ ([X,L] Env)] -> K), m([L,V] M) => k(V -> K), m([L,V] M) .

  op [_@_|_] -> _ : ExpList Env ValueList Continuation -> Continuation .
---  eq k(exp() -> K) = k(val(noVal) -> K) .
  eq k([(E,E',El) @ Env] -> K) =
     k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
  eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
  eq k(V -> [(E,El) @ Env | Vl] -> K) =
     k([E @ Env] -> [El @ Env | Vl,V] -> K) .

*** atomic memory block write; useful for let and letrec
  op _->_ : LocationList Continuation -> Continuation .
  eq k(Vl -> Ll -> K), m(M) = k(K), m(M[Ll <- Vl]) .

  op length_ : NameList -> Nat .
  eq length() = 0 .
  eq length(X,Xl) = 1 + length(Xl) .
endm

mod AEXP-K-SEMANTICS is protecting AEXP-SYNTAX .
```

```
  extending GENERIC-EXP-K-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  vars E E' : Exp .  vars I I' : Int .
  var K : Continuation .  var Env : Env .
  eq k([(E + E') @ Env] -> K) = k([(E,E') @ Env] -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
  eq k([(E - E') @ Env] -> K) = k([(E,E') @ Env] -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k([(E * E') @ Env] -> K) = k([(E,E') @ Env] -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k([(E / E') @ Env] -> K) = k([(E,E') @ Env] -> / -> K) .
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
endm

mod BEXP-K-SEMANTICS is protecting BEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op not -> _ : Continuation -> Continuation .
  op and -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .  var Env : Env .
  eq k([(E == E') @ Env] -> K) = k([(E,E') @ Env] -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k([(E >= E') @ Env] -> K) = k([(E,E') @ Env] -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k([(E <= E') @ Env] -> K) = k([(E,E') @ Env] -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k([(not E) @ Env] -> K) = k([E @ Env] -> not -> K) .
  eq k(bool(B) -> not -> K) = k(bool(not B) -> K) .
  eq k([(E and E') @ Env] -> K) = k([(E,E') @ Env] -> and -> K) .
  eq k((bool(B),bool(B')) -> and -> K) = k(bool(B and B') -> K) .
endm

mod LIST-K-SEMANTICS is protecting LIST-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op list -> _ : Continuation -> Continuation .
  op car -> _ : Continuation -> Continuation .
```

```
  op cdr -> _ : Continuation -> Continuation .
  op cons -> _ : Continuation -> Continuation .
  op null? -> _ : Continuation -> Continuation .
  var E E' : Exp .  var El : ExpList .  var K : Continuation .
  var V : Value .  var Vl : ValueList .  var Env : Env .
  eq k([list(El) @ Env] -> K) = k([El @ Env] -> list -> K) .
  eq k(Vl -> list -> K) = k([Vl] -> K) .
  eq k([car(E) @ Env] -> K) = k([E @ Env] -> car -> K) .
  eq k([V,Vl] -> car -> K) = k(V -> K) .
  eq k([cdr(E) @ Env]  -> K) = k([E @ Env] -> cdr -> K) .
  eq k([V,Vl] -> cdr -> K) = k([Vl] -> K) .
  eq k([emptyList @ Env] -> K) = k([noVal] -> K) .
  eq k([cons(E,E') @ Env] -> K) = k([(E,E') @ Env] -> cons -> K) .
  eq k((V,[Vl]) -> cons -> K) = k([V,Vl] -> K) .
  eq k([null?(E) @ Env] -> K) = k([E @ Env] -> null? -> K) .
  eq k([Vl] -> null? -> K) = k(bool(noVal == Vl) -> K) .
endm

mod IF-K-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [if(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E E' : Exp .  var B : Bool .
  var K : Continuation .  var Env : Env .
  eq k([(if BE then E else E') @ Env] -> K) =
     k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K) =
     k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endm

mod BINDING-K-SEMANTICS is protecting BINDING-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  sort Aux .
  op a : Nat NameList ExpList BindingList -> Aux .
  var N : Nat .  var Xl : NameList .  var El : ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq a(N, Xl ,El, (X = E, Bl)) = a(N + 1, (Xl,X), (El,E), Bl) .
endm

mod LET-K-SEMANTICS is protecting LET-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op let : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
```

```
  vars N # : Nat .
 ceq let Bl in E = let(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([let(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K), n(N + #) .
endm

mod PROC-K-SEMANTICS is protecting PROC-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op function : Nat NameList Exp -> Exp .
  op fn -> _ : Continuation -> Continuation .
  op closure : Nat NameList Exp Env -> Value .
  var Xl : NameList .  var El : ExpList .  var F E : Exp .
  var K : Continuation .  var Env : Env .  var M : Store .
  vars N # : Nat .  var Vl : ValueList .
  eq proc(Xl) E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E) @ Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K), n(N), m(M) =
     k([E @ Env[Xl <- l(N,#)]] -> K), n(N + #), m(M[l(N,#) <- Vl]) .
endm

mod LETREC-K-SEMANTICS is protecting LETREC-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op letrec : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq letrec Bl in E = letrec(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([letrec(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env[Xl <- l(N,#)]] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K),
       n(N + #) .
endm

mod VAR-ASSIGNMENT-K-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var K : Continuation .  var L : Location .  var V : Value .
  var M : Store .
  op _=>_ : Location Continuation -> Continuation .
  eq k([(set X = E) @ ([X,L] Env)] -> K) =
     k([E @ ([X,L] Env)] -> L => int(1) -> K) .
*** should be rl
  rl k(V -> L => K), m(M) => k(K), m(M[L <- V]) .
```

```
endm

mod BLOCK-K-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op ignore -> _ : Continuation -> Continuation .
  var E : Exp .  var El; : ExpList; .  var Env : Env .
  var K : Continuation .  var V : Value .
  eq k([{E} @ Env] -> K) = k([E @ Env] -> K) .
  eq k([{E ; El;} @ Env] -> K) = k([E @ Env] -> ignore
        -> ([{El;} @ Env] -> K)) .
  eq k(V -> ignore -> K) = k(K) .
endm

mod LOOP-K-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [while(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E : Exp .  var Vl : ValueList .
  var K : Continuation .  var Env :  Env .
  eq k([(while BE E) @ Env] -> K) =
     k([BE @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(true)) -> [while(BE,E) @ Env] -> K) =
     k([(E,BE) @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(false)) -> [while(BE,E) @ Env] -> K) = k(int(1) -> K) .
endm

mod SPAWN-K-SEMANTICS is extending SPAWN-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op die : -> Continuation .
  var V : Value .  var E : Exp .  var Env : Env .  var K : Continuation .
  eq k(V -> die) = empty .
  eq k([spawn(E) @ Env] -> K) = k(int(1) -> K), k([E @ Env] -> die) .
endm

mod SYNCHRONIZATION-K-SEMANTICS is extending SYNCHRONIZATION-SYNTAX .
*** add your code here
endm

mod PL-K-SEMANTICS is protecting PL-SYNTAX .
  extending AEXP-K-SEMANTICS .
  extending BEXP-K-SEMANTICS .
  extending LIST-K-SEMANTICS .
  extending IF-K-SEMANTICS .
  extending LET-K-SEMANTICS .
```

```
  extending PROC-K-SEMANTICS .
  extending LETREC-K-SEMANTICS .
  extending VAR-ASSIGNMENT-K-SEMANTICS .
  extending BLOCK-K-SEMANTICS .
  extending LOOP-K-SEMANTICS .
  extending SPAWN-K-SEMANTICS .

  op eval : Exp -> [Value] .
  op [_] : State -> [Value] .
  var E : Exp .  var V : Value .  var S : State .
  eq eval(E) = [k([E @ noEnv] -> stop), n(0), m(noStore)] .
  eq [k(V -> stop), S] = V .
endm


rew eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

rew eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

rew eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

rew eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5
```

```
rew eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

rew eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

rew eval(
  proc(x, y, z) (x * (y - z))
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

rew eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

rew eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

rew eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

rew eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

rew eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
```

```
    h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

rew eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

rew eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

rew eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

rew eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

rew eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping
```

```
rew eval(
  let f = proc(n)
          if n == 0
          then 1
          else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

rew eval(
  let f = proc(n) n + n
  in let f = proc(n)
              if n == 0
              then 1
              else n * f(n - 1)
     in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

rew eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---              f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

rew eval(
  let 'makemult = proc('maker, x)
                  if x == 0
                  then 0
                  else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

rew eval(
  letrec f = proc(n)
              if n == 0
```

```
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be 120

rew eval(
  letrec 'times4 = proc(x)
                   if x == 0
                   then 0
                   else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

rew eval(
  letrec 'even = proc(x)
                 if x == 0
                 then 1
                 else 'odd(x - 1),
         'odd = proc(x)
                 if x == 0
                 then 0
                 else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

rew eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined

rew eval(
  let x = 10
  in letrec f = proc(y) if y == 0 then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping
```

```
rew eval(
  let c = 0
  in let f = proc()
             let c = c + 1
             in c
     in f() + f()
) .
***>  should be 2

rew eval(
  let f = let c = 0
          in proc()
             let c = c + 1
             in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

rew eval(
  let c = 0
  in let f = proc()
             let d = set c = c + 1
             in c
     in f() + f()
) .
***>  should be 3

rew eval(
  let f = let c = 0
          in proc()
             let d = set c = c + 1
             in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

rew eval(
  let x = 0
  in let f = proc (x)
             let d = set x = x + 1
             in x
     in f(x) + f(x)
```

```
) .
***> should be 2

rew eval(
  let x = 0, y = 1
  in let f = proc(x, y)
             let t = x
             in let d = set x = y
                in let d = set y = t
                   in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

rew eval(
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
          if a == 0 then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

rew eval(
  let x = 0
  in letrec
       'even = proc() if x == 0
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if x == 0
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

rew eval(
  letrec x = 18,
         'even = proc() if x == 0 then 1
                        else let d = set x = x - 1
```

```
                    in 'odd(),
        'odd = proc() if x == 0 then 0
                  else let d = set x = x - 1
                    in 'even()
  in 'odd()
) .
***>  should be 0

rew eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

rew eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

rew eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                     if x == 0
                     then 0
                     else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

rew eval(
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
```

```
          set b = a - b ;
          set a = a - b
         }
  in {
      f(x,y) ;
      x
    }
) .
***> should be 3

rew eval(
  let f = proc(x) x + x
  in let y = 5
     in {
         f(set y = y + 3) ;
         y
        }
) .
***> should be 8

rew eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
      f(g(y));
      y
     }
) .
***> should be 5

rew eval(
  let n = 178378342647, c = 0
  in { while not (n == 1) {
         set c = c + 1 ;
         if 2 * (n / 2) == n
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185
```

```
---------------

rew eval(
  let f = proc(x, g)
          if x == 0
          then 1
          else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

rew eval(
  let x = 17,
      'odd = proc(x, o, e)
               if x == 0 then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if x == 0 then 1
               else o(x - 1, o, e)
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

rew eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

rew eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

rew eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19
```

```
----------

rew eval(
  letrec f = proc(n,m)
             if n == 0
             then m
             else f(n - 1, m * n)
  in f(100, 1)
) .
***> works for 50000: a number of 213237 digits in about 40 seconds

rew eval(
  letrec f = proc(n)
             if n == 0
             then 1
             else n * f(n - 1)
  in f(7000)
) .
***> works for 70000: a number of 308760 digits in about 60 seconds

rew eval(
  letrec a = proc(l,r)
             if null?(l) then r
             else cons(car(l), a(cdr(l), r)),
         h = proc(l,i,r,n)
             if n == 1 then list(list(l,r))
             else a(a(h(l, r, i, n - 1), list(list(l,r))),
                    h(i, l, r, n - 1))
  in h(1,2,3,3)
) .

rew eval(
  letrec f = proc(n,l)
             if null?(l) then -1
             else if n == car(l)
                  then 1
                  else let p = f(n, cdr(l))
                       in if p == -1 then -1
                          else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .

rew eval(
```

```
  let a = 0, b = 0, c = 0
  in let x = spawn letrec l = proc()
                          if not(b == 0) then set c = b + 1
                          else l()
                in l(),
         y = spawn letrec l = proc()
                          if not(a == 0) then set b = a + 1
                          else l()

                in l(),
         z = spawn set a = 1
     in letrec l = proc() if not(c == 0) then c else l()
        in l()
) .

rew eval(
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                  if a == 1 then c
                  else {set c = c + 1 ; f()}
        in f()
) .

search [1] eval(
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                  if a == 1 then c
                  else {set c = c + 1 ; f()}
        in f()
) =>! int(10) .

rew eval(
  let x = 0
  in let u = spawn set x = x + 1,
         v = spawn set x = x + 1
     in x
) .

search eval(
  let x = 0
  in let u = spawn set x = x + 1,
```

```
        v = spawn set x = x + 1
    in x
) =>+ int(1) .

search eval(
  let x = 0
  in let u = spawn set x = x + 1,
        v = spawn set x = x + 1
    in x
) =>+ int(0) .

rew eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {set x = x + 1 ; set a = 0},
        v = spawn {set x = x + 1 ; set b = 0}
    in letrec l = proc()
                  if (a == 0) and (b == 0) then x
                  else l()
       in l()
) .

search eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {set x = x + 1 ; set a = 0},
        v = spawn {set x = x + 1 ; set b = 0}
    in letrec l = proc()
                  if (a == 0) and (b == 0) then x
                  else l()
       in l()
) =>+ int(1) .

search eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {set x = x + 1 ; set a = 0},
        v = spawn {set x = x + 1 ; set b = 0}
    in letrec l = proc()
                  if (a == 0) and (b == 0) then x
                  else l()
       in l()
) =>+ int(0) .

search [1] eval(
  letrec n = 10, c = 1,
```

```
        f = proc() {set c = c + c ; f()}
  in let a = spawn f(), b = spawn f()
     in (c == n)
) =>! bool(true) .

search [1] eval(
  letrec c = 1,
        f = proc() {set c = c + c ; f()}
  in let a = spawn f(), b = spawn f()
     in c
) =>! int(50) .

eof

*** use the following as a test case for your HW7 problem
-------------------------------------------------------

rew eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {
              acquire lock(1) ;
              set x = x + 1 ;
              release lock(1) ;
              set a = 0
             },
        v = spawn {
              acquire lock(1) ;
              set x = x + 1 ;
              release lock(1) ;
              set b = 0
             }
     in letrec l = proc()
                    if (a == 0) and (b == 0) then x
                    else l()
        in l()
) .
*** should always return 2
*** search should not find any execution that returns 1
```

```
--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op '(') : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm

fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting INT .
  sorts Exp ExpList .
  subsorts Int Name < Exp < ExpList .
  subsort NameList < ExpList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _/_ : Exp Exp -> Exp [prec 31] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  ops _==_ _<=_ _>=_ _and_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod LIST-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op list_ : ExpList -> Exp .
  ops car cdr : Exp -> Exp .
  op cons : Exp Exp -> Exp .
  op emptyList : -> Exp .
  op null? : Exp -> Exp .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op if_then_else_ : Exp Exp Exp -> Exp .
endfm

fmod BINDING-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sorts Binding BindingList .
  subsort Binding < BindingList .
  op none : -> BindingList .
  op _,_ : BindingList BindingList -> BindingList [assoc id: none prec 71] .
  op _=_ : Name Exp -> Binding [prec 70] .
endfm

fmod LET-SYNTAX is extending BINDING-SYNTAX .
  op let_in_ : BindingList Exp -> Exp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op proc__ : NameList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 0] .
endfm

fmod LETREC-SYNTAX is extending BINDING-SYNTAX .
  op letrec_in_ : BindingList Exp -> Exp .
endfm
```

```
fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op set_=_ : Name Exp -> Exp .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is protecting BEXP-SYNTAX .
  op while__ : Exp Exp -> Exp .
endfm

fmod SPAWN-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op spawn_ : Exp -> Exp [prec 50] .
endfm

fmod SYNCHRONIZATION-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op lock : Exp -> Exp .
  op acquire_ : Exp -> Exp .
  op release_ : Exp -> Exp .
endfm

fmod PL-SYNTAX is
  extending AEXP-SYNTAX .
  extending BEXP-SYNTAX .
  extending LIST-SYNTAX .
  extending IF-SYNTAX .
  extending LET-SYNTAX .
  extending PROC-SYNTAX .
  extending LETREC-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending SPAWN-SYNTAX .
  extending SYNCHRONIZATION-SYNTAX .
endfm


------------------
--- Semantics ---
------------------

fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
  op l : Nat Nat -> LocationList .
  vars N # : Nat .
  eq l(N,0) = noLoc .
  eq l(N,#) = l(N), l(N + 1, # - 1) .
endfm

fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm

fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  op [_] : ValueList -> Value .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  var L : Location .  var M : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .
  eq M[noLoc <- noVal] = M .
  eq ([L,V] M)[L,Ll <- V',Vl] = ([L,V'] M)[Ll <- Vl] .
  eq M[L,Ll <- V',Vl] = (M [L,V'])[Ll <- Vl] [owise] .
endfm

fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm

fmod STATE is
  extending ENVIRONMENT .
  extending STORE .
  extending CONTINUATION .
  sorts StateAttribute State .
  subsort StateAttribute < State .
  op empty : -> State .
  op _,_ : State State -> State [assoc comm id: empty] .
  op k : Continuation -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
endfm

mod GENERIC-EXP-K-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op [_@_] ->_ : ExpList Env Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .

  vars E E' : Exp .  var El : ExpList .
  var V : Value .  var Vl : ValueList .
  var L : Location .  var Ll : LocationList .
  var X : Name .  var Xl : NameList .
  var S : State .  var I : Int .  var K : Continuation .
  var M : Store .  vars Env Env' : Env .

  op int : Int -> Value .
  eq k([I @ Env] -> K) = k(int(I) -> K) .
*** should be rl
  rl k([X @ ([X,L] Env)] -> K), m([L,V] M) => k(V -> K), m([L,V] M) .

  op [_@_|_] -> _ : ExpList Env ValueList Continuation -> Continuation .
---  eq k(exp() -> K) = k(val(noVal) -> K) .
```

```
  eq k([(E,E',El) @ Env] -> K) =
     k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
  eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
  eq k(V -> [(E,El) @ Env | Vl] -> K) =
     k([E @ Env] -> [El @ Env | Vl,V] -> K) .

*** atomic memory block write; useful for let and letrec
  op _->_ : LocationList Continuation -> Continuation .
  eq k(Vl -> Ll -> K), m(M) = k(K), m(M[Ll <- Vl]) .

  op length_ : NameList -> Nat .
  eq length() = 0 .
  eq length(X,Xl) = 1 + length(Xl) .
endm

mod AEXP-K-SEMANTICS is protecting AEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  vars E E' : Exp .  vars I I' : Int .
  var K : Continuation .  var Env : Env .
  eq k([(E + E') @ Env] -> K) = k([(E,E') @ Env] -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
  eq k([(E - E') @ Env] -> K) = k([(E,E') @ Env] -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k([(E * E') @ Env] -> K) = k([(E,E') @ Env] -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k([(E / E') @ Env] -> K) = k([(E,E') @ Env] -> / -> K) .
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
endm

mod BEXP-K-SEMANTICS is protecting BEXP-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op not -> _ : Continuation -> Continuation .
  op and -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .  var Env : Env .
  eq k([(E == E') @ Env] -> K) = k([(E,E') @ Env] -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k([(E >= E') @ Env] -> K) = k([(E,E') @ Env] -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k([(E <= E') @ Env] -> K) = k([(E,E') @ Env] -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k([(not E) @ Env] -> K) = k([E @ Env] -> not -> K) .
  eq k(bool(B) -> not -> K) = k(bool(not B) -> K) .
  eq k([(E and E') @ Env] -> K) = k([(E,E') @ Env] -> and -> K) .
  eq k((bool(B),bool(B')) -> and -> K) = k(bool(B and B') -> K) .
endm

mod LIST-K-SEMANTICS is protecting LIST-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op list -> _ : Continuation -> Continuation .
  op car -> _ : Continuation -> Continuation .
  op cdr -> _ : Continuation -> Continuation .
  op cons -> _ : Continuation -> Continuation .
  op null? -> _ : Continuation -> Continuation .
  var E E' : Exp .  var El : ExpList .  var K : Continuation .
  var V : Value .  var Vl : ValueList .  var Env : Env .
  eq k([list(El) @ Env] -> K) = k([El @ Env] -> list -> K) .
  eq k(Vl -> list -> K) = k([Vl] -> K) .
```

```
  eq k([car(E) @ Env] -> K) = k([E @ Env] -> car -> K) .
  eq k([V,Vl] -> car -> K) = k(V -> K) .
  eq k([cdr(E) @ Env] -> K) = k([E @ Env] -> cdr -> K) .
  eq k([V,Vl] -> cdr -> K) = k([Vl] -> K) .
  eq k([emptyList @ Env] -> K) = k([noVal] -> K) .
  eq k([cons(E,E') @ Env] -> K) = k([(E,E') @ Env] -> cons -> K) .
  eq k((V,[Vl]) -> cons -> K) = k([V,Vl] -> K) .
  eq k([null?(E) @ Env] -> K) = k([E @ Env] -> null? -> K) .
  eq k([Vl] -> null? -> K) = k(bool(noVal == Vl) -> K) .
endm

mod IF-K-SEMANTICS is protecting IF-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [if(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E E' : Exp .  var B : Bool .
  var K : Continuation .  var Env : Env .
  eq k([(if BE then E else E') @ Env] -> K) =
     k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K) =
     k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endm

mod BINDING-K-SEMANTICS is protecting BINDING-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  sort Aux .
  op a : Nat NameList ExpList BindingList -> Aux .
  var N : Nat .  var Xl : NameList .  var El : ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq a(N, Xl ,El, (X = E, Bl)) = a(N + 1, (Xl,X), (El,E), Bl) .
endm

mod LET-K-SEMANTICS is protecting LET-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op let : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq let Bl in E = let(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([let(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K), n(N + #) .
endm

mod PROC-K-SEMANTICS is protecting PROC-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op function : Nat NameList Exp -> Exp .
  op fn -> _ : Continuation -> Continuation .
  op closure : Nat NameList Exp Env -> Value .
  var Xl : NameList .  var El : ExpList .  var F E : Exp .
  var K : Continuation .  var Env : Env .  var M : Store .
  vars N # : Nat .  var Vl : ValueList .
  eq proc(Xl) E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E) @ Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K), n(N), m(M) =
     k([E @ Env[Xl <- l(N,#)]] -> K), n(N + #), m(M[l(N,#) <- Vl]) .
endm

mod LETREC-K-SEMANTICS is protecting LETREC-SYNTAX .
  extending BINDING-K-SEMANTICS .
  op letrec : Nat NameList ExpList Exp -> Exp .
  var Bl : BindingList .  var E : Exp .  var Xl : NameList .
  var El : ExpList .  var K : Continuation .  var Env : Env .
  vars N # : Nat .
 ceq letrec Bl in E = letrec(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([letrec(#,Xl,El,E) @ Env] -> K), n(N) =
     k([El @ Env[Xl <- l(N,#)]] -> l(N,#) -> [E @ Env[Xl <- l(N,#)]] -> K),
       n(N + #) .
endm

mod VAR-ASSIGNMENT-K-SEMANTICS is extending VAR-ASSIGNMENT-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  var X : Name .  var E : Exp .  var Env : Env .
  var K : Continuation .  var L : Location .  var V : Value .
  var M : Store .
  op _=>_ : Location Continuation -> Continuation .
  eq k([(set X = E) @ ([X,L] Env)] -> K) =
     k([E @ ([X,L] Env)] -> L => int(1) -> K) .
*** should be rl
  rl k(V -> L => K), m(M) => k(K), m(M[L <- V]) .
endm

mod BLOCK-K-SEMANTICS is extending BLOCK-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op ignore -> _ : Continuation -> Continuation .
  var E : Exp .  var El; : ExpList; .  var Env : Env .
  var K : Continuation .  var V : Value .
  eq k([{E} @ Env] -> K) = k([E @ Env] -> K) .
  eq k([{E ; El;} @ Env] -> K) = k([E @ Env] -> ignore
       -> ([{El;} @ Env] -> K)) .
  eq k(V -> ignore -> K) = k(K) .
endm

mod LOOP-K-SEMANTICS is extending LOOP-SYNTAX .
  extending BEXP-K-SEMANTICS .
  op [while(_,_) @_] -> _ : Exp Exp Env Continuation -> Continuation .
  vars BE E : Exp .  var Vl : ValueList .
  var K : Continuation .  var Env :  Env .
  eq k([(while BE E) @ Env] -> K) =
     k([BE @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(true)) -> [while(BE,E) @ Env] -> K) =
     k([(E,BE) @ Env] -> [while(BE,E) @ Env] -> K) .
  eq k((Vl,bool(false)) -> [while(BE,E) @ Env] -> K) = k(int(1) -> K) .
endm

mod SPAWN-K-SEMANTICS is extending SPAWN-SYNTAX .
  extending GENERIC-EXP-K-SEMANTICS .
  op die : -> Continuation .
  var V : Value .  var E : Exp .  var Env : Env .  var K : Continuation .
  eq k(V -> die) = empty .
  eq k([spawn(E) @ Env] -> K) = k(int(1) -> K), k([E @ Env] -> die) .
endm

mod SYNCHRONIZATION-K-SEMANTICS is extending SYNCHRONIZATION-SYNTAX .
*** add your code here
endm

mod PL-K-SEMANTICS is protecting PL-SYNTAX .
  extending AEXP-K-SEMANTICS .
  extending BEXP-K-SEMANTICS .
  extending LIST-K-SEMANTICS .
  extending IF-K-SEMANTICS .
  extending LET-K-SEMANTICS .
  extending PROC-K-SEMANTICS .
  extending LETREC-K-SEMANTICS .
  extending VAR-ASSIGNMENT-K-SEMANTICS .
  extending BLOCK-K-SEMANTICS .
  extending LOOP-K-SEMANTICS .
  extending SPAWN-K-SEMANTICS .

  op eval : Exp -> [Value] .
  op [_] : State -> [Value] .
  var E : Exp .  var V : Value .  var S : State .
  eq eval(E) = [k([E @ noEnv] -> stop), n(0), m(noStore)] .
  eq [k(V -> stop), S] = V .
endm


rew eval(
  let x = 5, y = 7
  in x + y
) .
***> should be 12

rew eval(
  let x = 1
  in let x = x + 2
     in x + 1
) .
***> should be 4

rew eval(
  let x = 1
  in let y = x + 2
     in x + 1
) .
***> should be 2

rew eval(
  let x = 1
  in let z = let y = x + 4
             in y
     in z
) .
***> should be 5

rew eval(
  let x = 1
  in let x = let x = x + 4
             in x
     in x
) .
***> should be 5

rew eval(
  let x = 1
  in (x + (let x = 10 in x))
) .
***> should be 11

rew eval(
  proc(x, y, z) (x * (y - z))
) .
***> should be closure((x,y,z), x * (y - z), noEnv)

rew eval(
  (proc(y, z) y + 5 * z) (1,2)
) .
***> should be 11

rew eval(
  let f = proc(y, z) y + 5 * z
  in f(1,2) + f(3,4)
) .
***> should be 34

rew eval(
  (proc(x, y) (x(y))) (proc(z) 2 * z, 3)
) .
***> should be 6

rew eval(
  let x = proc(x) x in x(x)
) .
***> should be closure(x, x, noEnv)

rew eval(
  let f = proc(x, y) x + y,
      g = proc(x, y) x * y,
      h = proc(x, y, a, b) (x(a,b) - y(a,b))
  in h(f, g, 1, 2)
) .
***> should be 1

rew eval(
  let y = 1
  in let f = proc(x) y
     in let y = 2
        in f(0)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

rew eval(
  let y = 1
  in (proc(x, y) (x y)) (proc(x) y, 2)
) .
***> should be 1 under static scoping and 2 under dynamic scoping

rew eval(
  let x = 1
  in let x = 2,
         f = proc (y, z) y + x * z
     in f(1,x)
) .
***> should be 3 under static scoping and 5 under dynamic scoping

rew eval(
  let x = 1
  in let x = 2,
         f = proc(y, z) y + x * z,
         g = proc(u) u + x
     in f(g(3), 4)
) .
***> should be 8 under static scoping and 13 under dynamic scoping

rew eval(
  let a = 3
  in let p = proc(x) x + a, a = 5
     in a * p(2)
) .
***> should be 25 under static scoping and 35 under dynamic scoping

rew eval(
  let f = proc(n)
            if n == 0
            then 1
            else n * f(n - 1)
  in f(5)
) .
***> should be undefined under static scoping and 120 under dynamic scoping

rew eval(
  let f = proc(n) n + n
  in let f = proc(n)
               if n == 0
```

```
              then 1
              else n * f(n - 1)
    in f(5)
) .
***> should be 40 under static scoping and 120 under dynamic scoping

rew eval(
  let a = 0
  in let a = 3, p = proc() a
     in let a = 5,
            f = proc(x) (p())
---            f = proc(a) (p())
        in f(2)
) .
***> should be 0 under static scoping and 5 under dynamic scoping
---***> should be 0 under static scoping and 2 under dynamic scoping

rew eval(
  let 'makemult = proc('maker, x)
                    if x == 0
                    then 0
                    else 4 + 'maker('maker, x - 1)
  in let 'times4 = proc(x) ('makemult('makemult,x))
     in 'times4(3)
) .
***> should be 12

rew eval(
  letrec f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
  in f(5)
) .
***> should be 120

rew eval(
  letrec 'times4 = proc(x)
                     if x == 0
                     then 0
                     else 4 + 'times4(x - 1)
  in 'times4(3)
) .
***> should be 12

rew eval(
  letrec 'even = proc(x)
                   if x == 0
                   then 1
                   else 'odd(x - 1),
         'odd = proc(x)
                   if x == 0
                   then 0
                   else 'even(x - 1)
  in 'odd(17)
) .
***> should be 1

rew eval(
  let x = 1
  in letrec x = 7,
            y = x
     in y
) .
***> should be undefined
```

```
  let x = 0, y = 3, z = 4,
      f = proc(a, b, c)
            if a == 0 then c else b
  in f(x, y / x, z) + x
) .
***> should be undefined

rew eval(
  let x = 0
  in letrec
       'even = proc() if x == 0
                      then 1
                      else let d = set x = x - 1
                           in 'odd(),
       'odd  = proc() if x == 0
                      then 0
                      else let d = set x = x - 1
                           in 'even()
     in let d = set x = 7
        in 'odd()
) .
***>  should be 1

rew eval(
  letrec x = 18,
         'even = proc() if x == 0 then 1
                        else let d = set x = x - 1
                             in 'odd(),
          'odd = proc() if x == 0 then 0
                        else let d = set x = x - 1
                             in 'even()
  in 'odd()
) .
***>  should be 0

rew eval(
  let x = 3, y = 4
  in let d = set x = x + y
     in let d = set y = x - y
        in let d = set x = x - y
           in 2 * x + y
) .
***> should be 11

rew eval(
  let x = 3, y = 4
  in { set x = x + y ;
       set y = x - y ;
       set x = x - y ;
       2 * x * y }
) .
***> should be 24

rew eval(
  let 'times4 = 0
  in {
       set 'times4 = proc(x)
                       if x == 0
                       then 0
                       else 4 + 'times4(x - 1) ;
       'times4(3)
     }
) .
***> should be 12

rew eval(
```

```
rew eval(
  let x = 10
  in letrec f = proc(y) if y == 0 then x else f(y - 1)
     in let x = 20
        in f(5)
) .
***> should be 10 under static scoping and 20 under dynamic scoping

rew eval(
  let c = 0
  in let f = proc()
               let c = c + 1
               in c
     in f() + f()
) .
***>  should be 2

rew eval(
  let f = let c = 0
          in proc()
               let c = c + 1
               in c
  in f() + f()
) .
***>  should be 2 under static scoping and undefined under dynamic scoping

rew eval(
  let c = 0
  in let f = proc()
               let d = set c = c + 1
               in c
     in f() + f()
) .
***>  should be 3

rew eval(
  let f = let c = 0
          in proc()
               let d = set c = c + 1
               in c
  in f() + f()
) .
***>  should be 3 under static scoping and undefined under dynamic scoping

rew eval(
  let x = 0
  in let f = proc (x)
               let d = set x = x + 1
               in x
     in f(x) + f(x)
) .
***> should be 2

rew eval(
  let x = 0, y = 1
  in let f = proc(x, y)
               let t = x
               in let d = set x = y
                  in let d = set y = t
                     in 0
     in let d = f(x,y)
        in x + 2 * y
) .
***> should be 2

rew eval(
```

```
  let x = 3, y = 4,
      f = proc(a, b)
          {
            set a = a + b ;
            set b = a - b ;
            set a = a - b
          }
  in {
       f(x,y) ;
       x
     }
) .
***> should be 3

rew eval(
  let f = proc(x) x + x
  in let y = 5
     in {
          f(set y = y + 3) ;
          y
        }
) .
***> should be 8

rew eval(
  let y = 5,
      f = proc(x) x + x,
      g = proc(x) set x = x + 3
  in {
       f(g(y));
       y
     }
) .
***> should be 5

rew eval(
  let n = 178378342647, c = 0
  in { while not (n == 1) {
         set c = c + 1 ;
         if 2 * (n / 2) == n
         then set n = n / 2
         else set n = 3 * n + 1
       } ;
       c }
) .
***> should be 185

---------------

rew eval(
  let f = proc(x, g)
            if x == 0
            then 1
            else x * g(x - 1, g)
  in f(5, f)
) .
***> should be 120

rew eval(
  let x = 17,
      'odd = proc(x, o, e)
               if x == 0 then 0
               else e(x - 1, o, e),
      'even = proc(x, o, e)
               if x == 0 then 1
               else o(x - 1, o, e)
```

```
  in 'odd(x, 'odd, 'even)
) .
***> should be 1

rew eval(
  let f = proc(x) x
  in f(1,2)
) .
***> should be undefined

rew eval(
  let f = proc(x) (x(x))
  in f(1)
) .
***>  should be undefined

rew eval( letrec f = proc(x) z + x + 5,
                 y = 2,
                 a = 3,
                 z = let y = 5, a = 6 in y + a
          in f(a)
) .
***> should be 19

--------------

rew eval(
  letrec f = proc(n,m)
               if n == 0
               then m
               else f(n - 1, m * n)
  in f(100, 1)
) .
***> works for 50000: a number of 213237 digits in about 40 seconds

rew eval(
  letrec f = proc(n)
               if n == 0
               then 1
               else n * f(n - 1)
  in f(7000)
) .
***> works for 70000: a number of 308760 digits in about 60 seconds

rew eval(
  letrec a = proc(l,r)
               if null?(l) then r
               else cons(car(l), a(cdr(l), r)),
         h = proc(l,i,r,n)
               if n == 1 then list(list(l,r))
               else a(a(h(l, r, i, n - 1), list(list(l,r))),
                      h(i, l, r, n - 1))
  in h(1,2,3,3)
) .

rew eval(
  letrec f = proc(n,l)
               if null?(l) then -1
               else if n == car(l)
                    then 1
                    else let p = f(n, cdr(l))
                         in if p == -1 then -1
                            else p + 1
  in f(5, list(2, 6, 7, 2, 6, 5, 7))
) .
```

```
rew eval(
  let a = 0, b = 0, c = 0
  in let x = spawn letrec l = proc()
                                if not(b == 0) then set c = b + 1
                                else l()
                    in l(),
         y = spawn letrec l = proc()
                                if not(a == 0) then set b = a + 1
                                else l()

                    in l(),
          z = spawn set a = 1
      in letrec l = proc() if not(c == 0) then c else l()
         in l()
) .

rew eval(
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                     if a == 1 then c
                     else {set c = c + 1 ; f()}
        in f()
) .

search [1] eval(
  let a = 0, c = 0
  in let d = spawn set a = 1
     in letrec f = proc()
                     if a == 1 then c
                     else {set c = c + 1 ; f()}
        in f()
) =>! int(10) .

rew eval(
  let x = 0
  in let u = spawn set x = x + 1,
         v = spawn set x = x + 1
     in x
) .

search eval(
  let x = 0
  in let u = spawn set x = x + 1,
         v = spawn set x = x + 1
     in x
) =>+ int(1) .

search eval(
  let x = 0
  in let u = spawn set x = x + 1,
         v = spawn set x = x + 1
     in x
) =>+ int(0) .

rew eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {set x = x + 1 ; set a = 0},
         v = spawn {set x = x + 1 ; set b = 0}
     in letrec l = proc()
                     if (a == 0) and (b == 0) then x
                     else l()
        in l()
) =>+ int(1) .

search eval(
```

```
  let a = 1, b = 1, x = 0
  in let u = spawn {set x = x + 1 ; set a = 0},
         v = spawn {set x = x + 1 ; set b = 0}
     in letrec l = proc()
                     if (a == 0) and (b == 0) then x
                     else l()
        in l()
) =>+ int(0) .

search [1] eval(
  letrec n = 10, c = 1,
         f = proc() {set c = c + c ; f()}
  in let a = spawn f(), b = spawn f()
     in (c == n)
) =>! bool(true) .

search [1] eval(
  letrec c = 1,
         f = proc() {set c = c + c ; f()}
  in let a = spawn f(), b = spawn f()
     in c
) =>! int(50) .

eof

*** use the following as a test case for your HW7 problem
-----------------------------------------------------------

rew eval(
  let a = 1, b = 1, x = 0
  in let u = spawn {
                acquire lock(1) ;
                set x = x + 1 ;
                release lock(1) ;
                set a = 0
              },
          v = spawn {
                acquire lock(1) ;
                set x = x + 1 ;
                release lock(1) ;
                set b = 0
              }
     in letrec l = proc()
                     if (a == 0) and (b == 0) then x
                     else l()
        in l()
) .
*** should always return 2
*** search should not find any execution that returns 1
```

```
--------------
--- Syntax ---
--------------

fmod NAME is protecting QID .
  sorts Name NameList .
  subsort Qid < Name < NameList .
--- the following can be used instead of Qids if desired
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w : -> Name .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: () prec 100] .
endfm


*** to have a more realistic PL, we use rational values
fmod GENERIC-EXP-SYNTAX is protecting NAME .
  protecting RAT .
  sorts Exp ExpList RatList .
  subsorts Rat Name < Exp < ExpList .
  subsort NameList < ExpList .

--- RatList needed only for the input to the program
  subsort Rat < RatList < ExpList .
  op nil : -> RatList .
  op _,_ : RatList RatList -> RatList [ditto] .

 --- noExp needed for statements with empty entries
  op noExp : -> Exp .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  op halt : -> Exp .
endfm

fmod AEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  protecting RAT .
  op _+_ : Exp Exp -> Exp [ditto] .  --- prec 33
  op _-_ : Exp Exp -> Exp [ditto] .  --- prec 33
  op _*_ : Exp Exp -> Exp [ditto] .  --- prec 31
  op _/_ : Exp Exp -> Exp [ditto] .  --- prec 31
  op _%_ : Exp Exp -> Exp [prec 31].
  op _^_ : Exp Exp -> Exp [ditto] .  --- prec 29
  op -_ : Exp -> Exp [ditto] .
endfm
```

```
fmod RELEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op _<_  : Exp Exp -> Exp [prec 37] .
  op _<=_ : Exp Exp -> Exp [prec 37] .
  op _>_  : Exp Exp -> Exp [prec 37] .
  op _>=_ : Exp Exp -> Exp [prec 37] .
  op _==_ : Exp Exp -> Exp [prec 37] .
  op _!=_ : Exp Exp -> Exp [prec 37] .
endfm

fmod BEXP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op !_ : Exp -> Exp [prec 38] .
  op _&&_ : Exp Exp -> Exp [prec 39] .
  op _||_ : Exp Exp -> Exp [prec 40] .
endfm

fmod IF-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op if(_)_else_ : Exp Exp Exp -> Exp .
  op if(_)_ : Exp Exp -> Exp .
  vars E E' : Exp .
  eq if (E) E' = if (E) E' else noExp .
endfm

fmod PROC-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op define_(_)_ : Name NameList Exp -> Exp .
  op define_()_ : Name Exp -> Exp .

  op _(_) : Exp ExpList -> Exp [prec 2] .
  op _() : Exp -> Exp [prec 2] .

  op return(_) : Exp -> Exp [prec 0] .

  --- rewriting in the syntax level
  var X : Name . var E : Exp . var El : ExpList .
  eq define X() E = define X(()) E .
  eq E() = E(()) .
endfm

fmod VAR-DECLARATION-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op var_ : NameList -> Exp [prec 101] .
endfm

fmod VAR-ASSIGNMENT-SYNTAX is extending GENERIC-EXP-SYNTAX .
```

```
  op _=_ : Name Exp -> Exp       [prec 35] .
  op ++_ : Name -> Exp           [prec 27] .
  op --_ : Name -> Exp           [prec 27] .
  op _++ : Name -> Exp           [prec 27] .
  op _-- : Name -> Exp           [prec 27] .

  op _+=_ : Name Exp -> Exp      [prec 35] .
  op _-=_ : Name Exp -> Exp      [prec 35] .
  op _*=_ : Name Exp -> Exp      [prec 35] .
  op _/=_ : Name Exp -> Exp      [prec 35] .
  op _%=_ : Name Exp -> Exp      [prec 35] .
  op _^=_ : Name Exp -> Exp      [prec 35] .
  op _(<)=_ : Name Exp -> Exp    [prec 35] .
  op _(<=)=_ : Name Exp -> Exp   [prec 35] .
  op _(>)=_ : Name Exp -> Exp    [prec 35] .
  op _(>=)=_ : Name Exp -> Exp   [prec 35] .
  op _(==)=_ : Name Exp -> Exp   [prec 35] .
  op _(!=)=_ : Name Exp -> Exp   [prec 35] .
  op _&&=_ : Name Exp -> Exp     [prec 35] .
  op _||=_ : Name Exp -> Exp     [prec 35] .
endfm

fmod BLOCK-SYNTAX is extending GENERIC-EXP-SYNTAX .
  sort ExpList; .
  subsort Exp < ExpList; .
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 102] .
  op {_} : ExpList; -> Exp .
endfm

fmod LOOP-SYNTAX is extending GENERIC-EXP-SYNTAX .
  op while(_)_ : Exp Exp -> Exp .
  op for(_;_;_)_ : Exp Exp Exp Exp -> Exp .
--- every possible combination of for
  op for (_;_;)_ : Exp Exp Exp -> Exp .
  op for (_; ;_)_ : Exp Exp Exp -> Exp .
  op for (;_;_)_ : Exp Exp Exp -> Exp .
  op for (_; ;)_ : Exp Exp -> Exp .
  op for (;_;)_ : Exp Exp -> Exp .
  op for (; ;_)_ : Exp Exp -> Exp .
  op break : -> Exp .
  op continue : -> Exp .

  vars E1 E2 E3 : Exp .
```

```
  eq for(E1 ; E2 ;) E3 = for(E1 ; E2 ; noExp) E3 .
  eq for(E1 ; ; E2) E3 = for(E1 ; noExp ; E2) E3 .
  eq for(; E1 ; E2) E3 = for(noExp ; E1 ; E2) E3 .
  eq for(E1 ; ;) E2 = for(E1 ; noExp ; noExp) E2 .
  eq for(; E1 ;) E2 = for(noExp ; E1 ; noExp) E2 .
  eq for(; ; E1) E2 = for(noExp ; noExp ; E1) E2 .
endfm

fmod IO-SYNTAX is protecting GENERIC-EXP-SYNTAX .
  op read() : -> Exp .
  op print(_) : ExpList -> Exp .
endfm

fmod PL-SYNTAX is
  protecting NAME .
  protecting GENERIC-EXP-SYNTAX .
  extending AEXP-SYNTAX .
  extending RELEXP-SYNTAX .
  extending BEXP-SYNTAX .
  extending IF-SYNTAX .
  extending PROC-SYNTAX .
  extending VAR-DECLARATION-SYNTAX .
  extending VAR-ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending LOOP-SYNTAX .
  extending IO-SYNTAX .
endfm

------------------------------------------------------------
--- ... define your semantics here
--- make sure that you provide an operation
---   op eval : ExpList; RatList -> RatList
--- You can assume that programs and inputs are given between
--- parentheses, e.g.:
---
--- red eval((
---   var i, j ;
---   i = read() ;
---   j = read() ;
---   print(j + i) ;
---   print(j - i) ;
--- ), (1/2, 3/4) ) .
---
```

```
--- The output of the above should be
--- result RatList: 5/4,1/4
-------------------------------------------------------------

**************************
*** Parsing examples ***
**************************

parse
  var i, j, k
 .

parse
  if (a > 3) x
 .

parse
  if (b < 3) z
 .

parse
  if (a > 3) (if (b < 3) z) else y
 .

parse
  i = 0
 .

parse
  i += 2
 .

parse
  for(i = 0 ; i < 10 ; ++ j) {a = i}
 .

parse
  for(i = 0 ; i < 10 ; {++ i ; ++ j}) {a = i}
 .

parse
  define f(x) return(1)
 .
```

```
parse
  define f(x) {return(1)}
 .

parse
  define f(x) {return(f(x - 1))}
 .

parse
  define f(x) {return(f(x - 1) * x)}
 .

parse
  if (x <= 1) return(1)
 .

parse
  if (x <= 1) return(1) ;
  return (2)
 .

parse
  define f(x) {
    if (x <= 1) return(1) ;
    return(f(x - 1) * x)
  }
 .

parse
  var a, i, j ;
  for(i = 0 ; i < 10 ; ++ j) {a = i}
 .

parse
  var i ;
  i += 1 ;
  i = 0
 .


----------------
```

```
parse
  var a, i, j ;
  a = read() ;
  i = read() ;
  j = read() ;
  print(a + i + j)
 .

parse
  var a , i ;
  for(i = 0 ; i < 10 ; ++ i) {a = i} ;
  print(a + i)
 .

parse
  var a , i ;
  for(i = 0 ; (i ++) < 10 ; ) {a = i} ;
  print(a + i)
 .

parse
  var j ;
  j = 1024 < 1 ;  --- precedence of the relational operator is very low
  print(j)
 .

parse
  var j ;
  print(j = 4) ;
  print(j)
 .

parse
  var i , j ;
  i = 0 ;
  i += 1 ;      --- i = 1
  i -= 2 ;      --- i = -1
  i *= 3 ;      --- i = -3
  i /= 10 ;     --- i = -3/10
  i += 23/10 ;  --- i = 2
  i += 3 ;      --- i = 5
  i %= 3 ;      --- i = 2
```

```
  i ^= 10 ;     --- i = 1024
  print(i) ;

  i (<)= 1 ;    --- i = 0
  print(i) ;
  i (<=)= 1 ;   --- i = 1
  print(i) ;
  i (>)= 2 ;    --- i = 0
  print(i) ;
  i (>=)= -1 ;  --- i = 1
  print(i) ;
  i (==)= 0 ;   --- i = 0
  print(i) ;
  i (!=)= 1 ;   --- i = 1
  print(i) ;
  i &&= 0 ;     --- i = 0
  print(i) ;
  i ||= 0 ;     --- i = 0
  print(i) ;
  i ||= 1 ;     --- i = 1
  print(i)
 .

parse
  var a, i ;
  for(i = 0 ; i < 10 ; ++ i) {a = i} ;
  i += 1 ;

  define f (x) {
    if (x <= 1) return(1) ;
    return(f(x - 1) * x)
  } ;

  print(f(i))
 .

*** Basic variable declaration, I/O, halt, and simple operators.
parse
  var x, y, z ;
  x = read() ;
  y = 3 ;
  z = read() + read() ;
  print (x + y * z) ;
```

```
  print (- y / z) ;
  print (- z ^ x) ;
  print ((x - y) / (z + y)) ;
  print ( z % y ) ;
  halt ;
  print (2 + 3)
 .

*** Assignments and operator assignments
parse
  var x, y, z, 'x , 'y, 'four ;
  x = read() ;
  'x = x ;
  y = 3 ;
  'four = 4 ;
  z = read() + read() ;
  x += y * z ;
  print (x) ;
  y = - y ;
  y /= z ;
  z ^= 'x ;
  print (z) ;
  'y = 5 / 3 ;
  'x -= 'y ;
  print ('x) ;
  z %= ('four ) ;
  print (z) ;
  x *= 0 ;
  print (x) ;
  x = 14 / 17 ;
  y = (x ++) ;
  print(x) ;
  print(y) ;
  z = (++ x) ;
  print(x) ;
  print(z) ;
  y = (-- x) ;
  print(y) ;
  print(x) ;
  y += (x --) ;
  print(y) ;
  print(x)
 .
```

```
*** Conditionals
parse
  var 'alpha, 'beta, 'result ;
  'alpha = 19 / 21 ;
  'beta = 13 / 15 ;
  if ('alpha > 'beta) 'result = 2
  else 'result = 3 ;
  print ('result) ;
  if ('alpha <= 'result) 'beta = 5 ;
  print ('beta) ;
  if ('alpha * 'result >= 'beta - 'result) 'beta = 6
  else if ('alpha < 'beta * 'result) 'beta = 4 ;
  if (!('alpha != 'beta)) 'result = 5 ;
  print('result) ;
  print('beta)
 .

*** Scoping

parse
  var x, y ;
  x = 2 ;
  y = 3 ;
  {
    x = 4 ;
    y = 5 ;
    {
      print (x)
    } ;
    print (y)
  } ;
  print(x) ;
  print(y)
 .

parse
  var x,y ;
  x = 2 ;
  y = 3 ;
  {
    var x,y ;
    x = 4 ;
```

```
  y = 5 ;
  {
    print (x)
  } ;
  print (y)
 } ;
 print(x) ;
 print(y)
.

----------

parse
 define 'f1 (a)
 {
  define 'fib(b) {
    if ((b == 1) || (b == 0))
       return (1)
    else return ('fib(b - 1) + 'fib(b - 2))
  } ;
  define 'factorial(c) {
    if (c < 1) return (1)
    else return (c * 'factorial(c - 1))
  } ;
 return ('fib(a) + 'factorial(a))
 } ;

 define 'f2(d)
 {
  define 'f21(e) {
    if ('f1(e) > 1000000 * e)
       return (1)
    else  { return (0) } } ;

  define 'f22(f){
    if ((f >= 1000) && (f <= 1500))
       return(1)
    else return(4)} ;

  return ('f21(d - 1) + 'f22(d * 100))
 } ;

 print('f2(9))
```

```
.

parse
  var x, y ;
  y = 2 ;
  for(x = 0 ; x < 10 ; x ++) {
   y *= 2
  } ;
  print(y)
.

parse
  var x, y ;
  y = 2 ;
  for(x = 0 ; x < 10 ; x ++) {
   if (x  % 2 ) {x ++ ; break} ;
   y *= 2
  } ;
  print(y)
.

parse
  var x, y ;
  y = 2 ;
  for(x = 0 ; x < 10 ; x ++) {
   if (x  % 2 ) {x ++ ; continue} ;
   y *= 2
  } ;
  print (y)
.

parse
  define 'function(a,b) {
    var x, y ;
    x = 4 ;
    y = 6 ;
    {
     var y ;
     y = (x += a) ;
     x += b ;
     if (x != y)
       return (1)
    } ;
```

```
    y = x ;
    return(x + y)
  } ;

  var c, d ;
  c = 4 ;
  d = 5 ;
  print('function(c,d))
 .

parse
  define f(n) {return (n + 1)} ;
  print(f(read()))
 .
parse
  define f(n) {
    var j ;
    j = n + 1 ;
    print(j)
  } ;
  var d ;
  d = f(read()) ;
  --- j is undefined
  print(j)
 .

parse
  define f(n) {return (1)} ;
  var d ;
  d = f(read()) ;
   --- n is undefined
  print(n)
 .

parse
  var j ;
  define f(n) {j = n + 1 ; return(j)} ;
  var d ;
  d = f(read()) ;
  print(j)
 .

parse
```

```
  var j ;
  j = 1 ;
  define f(n) {j += n ; return (j)} ;
  var d ;
  d = f(read()) ;
  print(j)
 .

parse
  var j ;
  j = 11 ;
  define f(n) {j -= n ; return (j)} ;
  var d ;
  d = f(read()) ;
  print(j)
 .

parse
  var j ;
  j = 0 ;
  define f(n) {j = ++ n ; return(j)} ;
  var d ;
  d = f(read()) ;
  print(j)
 .

parse
  var j ;
  j = 0 ;
  define f(n) {j = (n ++) + 1 ; return(j)} ;
  var d ;
  d = f(read()) ;
  print(j)
 .

parse
  var j ;
  j = 0 ;
  define f(n) {j = (n --) + 1 ; return(j)} ;
  f(read()) ;
  print(j)
 .
```

```
parse
  define f(n) {
    if (n == 1) 1
    else n * f(n - 1)
  } ;
  print(f(read()))
 .

parse
  define g(n) {return (8)} ;
  define f(n) {return (g(n))} ;
  print(f(read()))
 .

parse
  define 't() {
    define f(n) {return (g(n))} ;
    define g(n) {
      if (n == 1) 1
      else n * g(n - 1)
    } ;

    print(f(read()))
  } ;
  var d ;
  d = 't()
 .

parse
  var j, z, n ;
  j = 0 ;
  z = 0 ;
  while (j <= 10) {
    j ++ ;
    if (j % 2 == 0) {
      z ++ ;
      continue
    } ;
    if (j == 7) break
  } ;

  --- z should be 3
  --- j should be 7
```

```
  n = -- j + z + 1 ;
  --- n should be 10, j is now 6

  --- this is factorial
  define g(n) {
    if (n == 1) return (1)
    else return(n * g(n - 1))
  } ;
  define f(n) {return (g(n))} ;
  print(f(read()) * n)
.

parse
  var j, z ;
  for ({ j = 0 ; z = 1 } ; j < 10 ; j ++) {
    if ((j % 2) == 0 ) {
      z *= 2
    } ;
    if (j == 8) break
  } ;

  print(z) ;
  print(j) ;
  print(z - j)
.

parse
  var j, k, z ;
  k = 1 ;

  for({j = 0 ; z = 1} ; {var k ; k = 2 ; j < 10} ; j ++) {
    if ((j % 2) == 0) {
      z *= 2
    } ;

    if (j == 8) {
      var k ;
      k = 3 ;
      break
    }
  } ;
```

```
  --- j and z are still defined!

  print(z) ;
  print(j) ;
  print(z - j) ;
  print(k)
 .

parse
  define f(n) {
    define g(n) {
      if (n == 1) return(1)
      else return(n * g(n - 1))
    } ;
    return (g(n))
  } ;

  print (f(read()))
 .

parse
  define 'f1(a) {
    define 'f11(a) {
      var j ;
      if (a == 1) j = 1 else j = 'f2(1) ;
      return (j)
    } ;

    define 'f12() {return (2)} ;

    return ('f11(1) + 'f12())  --- should be 3
  } ;

  define 'f2(a) {
    define 'f21(a) {
      var j ;
      if (a == 1) j = 1 else j = 'f1(0) ;
      return (j)
    } ;

    define 'f22() {return (3)} ;

    return ('f21(a) + 'f22())  --- should be 4
```

```
  } ;

  print ('f1(0) + 'f2(0))  --- should be 9
 .

parse
  var k ;
  k = 0 ;
  define 'f1(a) {
    define 'f11(a) {
      var j ;
      if (a == 1) j = 1 else j = 'f2(1) ;
      return (j + k)
    } ;

    define 'f12() return (2) ;

    return ('f11(1) + 'f12())  --- should be 3
  } ;

  var k ;
  k = 1 ;

  define 'f2(a) {
    define 'f21(a) {
      var j ;
      if (a == 1) j = 1 else j = 'f1(0) ;
        return (j)
    } ;

    define 'f22() {return (3)} ;

    return ('f21(a) + 'f22())  --- should be 4
  } ;

  print ('f1(0) + 'f2(0))  --- should be 9
 .

parse
  define f(n) {return (g(n))} ;
  define g(n) {
    if (n == 1) 1
    else n * g(n - 1)
```

```
  } ;
  print(f(read()))
 .

parse
  define e(x) {
    if (x < 0) {
      m = 1 ;
      x = - x
    } ;

    z = 'scale ;
    'scale = 4 + z + (44/100) * x ;
    while ( x > 1 ) {
      f += 1 ;
      x /= 2
    } ;

    v = 1 + x ;
    a = x ;
    d = 1 ;

    for ( i = 2 ; 1 ; i ++ ) {
      e = (a *= x) / (d *= i) ;
      if (e == 0) {
        if (f > 0) while (f --) v = v * v ;
        'scale = z ;
        if (m) return ( 1 / v )
      } ;
      v += e
    }
  } ;

print(e(10))
 .
```

CS322 - Sample Final Exam

Name

Time: 3 hours
Total: 100 points
You can use any books, computers, tea, coffee, hammers, axes, etc., but *no laptops*!

Problem 1. (10 points)
Evaluate the expressions below under both static and dynamic scoping.

1.
```
let x = 1
in let x = 2,
       f = proc (value y, value z) y + x * z
   in f(1,x)
```

2.
```
let f = let c = 0
        in proc()
             let d = set c = c + 1
             in c
in f() + f()
```

3.
```
let f = proc(value n) n + n
in let f = proc(value n)
             if zero?(n)
             then 1
             else n * f(n - 1)
   in f(5)
```

4.
```
let y = 5,
    f = proc(need x) x + x,
    g = proc(reference x) set x = x + 3
in {
     f(g(y));
     y
   }
```

5.
```
let f = proc(name x) x + x
in let y = 5
   in {
        f(set y = y + 3) ;
        y
      }
```

Problem 2. (10 points)
What values do the following evaluate to? Consider both static and dynamic scoping.

1. ```
   let a = 1, x = proc(a) (a + 1)
   in let a = 10, x = proc(a) (a + 10), y = x(a)
      in y
   ```

2. ```
   let a = 1, x = proc(a) (a + 1)
   in letrec y = x(a), x = proc(a) (a + 10)
      in y
   ```

3. ```
   let a = 1, x = proc(a) (a + 1)
   in letrec y = proc() (x(a)), x = proc() (a + 10), a = 10
      in y
   ```

4. ```
   let a = 1, x = proc(a) (a + 1)
   in letrec y = proc(a) (x()), x = proc() (a + 10), a = 10
      in y(a)
   ```

5. ```
   let a = 1, x = proc(a) (a + 1)
   in letrec y = (proc(a) (x()))(10), x = proc() (a + 10)
      in y
   ```

Problem 3. (10 points)

Briefly explain what are the advantages and the drawbacks of static versus dynamic scoping. Make sure that you refer to both writing programs and implementing or defining the language. Is it possible to have an expression which is undefined under static scoping but is defined under dynamic scoping? How about the vice-versa, that is, an expression which is defined under static scoping but undefined under dynamic scoping?

Problem 4. (10 points)
Comment on the advantages and disadvantages of using continuations in defining programming languages. Consider now defining an imperative language, like Pascal, C or BC. Would you use a continuation-based style for this task or not? Why? What if you add exceptions to your design?
   ```

Problem 5. (10 points)
Suppose that we want to add "for" loops of the form

```
for <Name> := <Exp> to <Exp> do <Exp>
```

to our functional language, in addition to while loops. `<Name> := <Exp>` acts like a local binding, in the sense that the newly defined `<Name>` is not assumed to have been declared apriori, is not seen by the first expression (to which it is bound), and should disappear from the environment after the "for" expression evaluates. More precisely, the first expression sees the environment that the "for" loop sees, while the second and the third expressions see that environment enriched with a new entry for `<Name>`, which may thus shadow a previous declaration with the same name.

1. Give a translation of the "for" language construct above into an expression using the other functional language constructs, including the imperative ones;

2. Give a continuation-based semantics for "for", *without* using the continuation-based definition of while directly. Feel free to work with any of our continuation-based semantics and to reuse everything we defined except the operations and the equations introduced to define the semantics of "while".

Problem 6. (10 points)
Suppose that you want to design a programming language with both exceptions and threads. An interesting design decision that needs to be made is to define the behavior of a thread which is spawned within the try part of a `try ... catch ...` expression and which throws an exception. A related question is what happens with the threads spawned within the try expression when the try expression itself throws an exception: do they all die or they are allowed to continue their submissive existence? Argue briefly for a most reasonable decision for the above and then explain informally how one could rigorously define it in a continuation-based semantics. Note that different valid decisions may be possible here.

Problem 7. (10 points)
Comment on the advantages and the disadvantages of static and dynamic type checking. Give an example of a program which executes correctly but which does not type check statically. Following the pre-typing and the unification-based typing methods discussed in class, type the following two expressions (do not go to the last detail, but enough to convince us that you understand this technique well):

```
(proc(x, y) (x(y)))
  (proc(x) x, proc(y) y)

letrec f = proc(x, g)
            if zero?(x)
            then 1
            else x * g(x - 1, g),
       g = proc() f
in f(5, g())
```

Can one device a static type checker which admits exactly those programs which have a type safe runtime behavior? Explain why.

Problem 8. (10 points)
Transform the following functional program in a continuation passing style equivalent program, by applying the CPS procedure discussed in class:

```
letrec r = proc(n, l)
             if null?(l) then emptylist
             else if n equals car(l)
                  then r(n, cdr(l))
                  else cons(car(l), r(n, cdr(l)))
in r(3, list(3,1,3,2,3,3,3,4,3,5,3,3))
```

Like in the problem above, go into as much detail as needed in order to show that you understand the CPS procedure well.

Hint (there will be no hints in the final exam!). The final answer is

```
(proc(k0)
   letrec r = proc(n, l, k1)
               if null?(l) then k1(emptylist)
               else if n equals car(l)
                    then r(n, cdr(l), k1)
                    else r(n, cdr(l), proc(v0) k1(cons(car(l), v0)))
   in r(3, list(3,1,3,2,3,3,3,4,3,5,3,3), k0)
) (proc(x) x)
```

Problem 9. (10 points)
Prove, using the Hoare rules, the correctness of the partial correctness assertion:

```
{1 <= N}
  P = 0 ;
  C = 1 ;
  while (C <= N) (P = P + M ; C = C + 1)
{P = M * N}
```

Problem 10. (10 points)
Consider a simple imperative language, admitting integer arithmetic ($+$,$*$,$-$) and boolean ($==$,$>$,$\neg$,$\wedge$) expressions, assignment, conditionals, while loops and a special `read()` expression whose value you can assume to be each time a random integer. Now suppose that you are asked to design a program analysis tool which analyzes programs in this language for uninitialized variables, that is, for variables which are used before they are assigned a value. Is it possible to develop such a tool that would work correctly on any input program? If yes, explain why and give a brief description of how you would do it. If no, explain why and give a concrete example on which one would not expect it to work.