Compiling Scheme using Abstract State Machines

Joel Priestley
Language, Logic, and Information (SLI)
Department of Linguistics
University of Oslo
j.j.priestley@ilf-stud.uio.no

25th April 2003

Contents

List of Figures

Preface

This project is an investigation into the use of *Abstract State Machines* (*ASMs*) in the process of compilation. The aim is to use the expressive power of ASM to capture the semantics of Scheme procedures according to a particular model, and to take advantage of ASM's simple form in the process of compilation. The programming language *Scheme* is the domain or source language for the investigation. The semantic model is *The Environment Model for Evaluation* described by Abelson and Sussman in *Structure and Interpretation of Computer Programs*. The ASMs will be represented in *Abstract state machine Language* (AsmL), a Microsoft proprietary programming language. Using Microsoft Visual Studio, the AsmL code can be compiled, via *C++*, to x86 binaries. The main focus of the project will be on the process of parsing Scheme code, and generating the corresponding AsmL code. Little attention will be payed to the subsequent step of compiling AsmL to C++. The documentation for this project is divided into six chapters. The first two chapters introduce what are effectively the domain and the range of the project, Scheme and ASM, respectively. The third chapter provides an overview of the process, how the different components fit together, while the fourth and fifth chapters give a more detailed description of these components. The last chapter covers briefly the final stage; generating binaries from the generated AsmL code. It is a assumed that the reader of this project has some experience with the programming languages Scheme and Java.

Acknowledgments

I would like to take this opportunity to thank my supervisor Herman Ruge Jervell, for his patience and encouragement.
I would also like to thank my children Hanne and Sam, for putting up with me for the duration of this project, as indeed they always do.

Chapter 1

Scheme

1.1 A Little Scheme

This chapter is intended to give a very brief description of various central aspects of the programming language Scheme. A limited working knowledge of this programming language is assumed. Since the focus of the work is on the semantics of Scheme with regard to the environment model of evaluation, only aspects of the language important for this project will be covered in this chapter. The features looked at here are the *environment* and *procedures*. This will draw attention to topics such as scope, frames, first class objects and closures.

1.2 Lexical Scope

Scheme is a lexically (statically) scoped programming language; i.e. the binding of a variable is lexically apparent, it can be determined by reading the source code. This means that the scope of a variable can be fixed at compile time to a region in the source code where the variable is declared. Generally speaking, a variable declaration presides over the expression it heads. Variable declarations in scheme are made using the `lambda` key word, so basically a variable declaration presides over the text within the parentheses enclosing the lambda expression. However, Scheme is block-structured and allows nesting. A lambda expression can contain a lambda expression, as in the following:

```
(define x
  (lambda (x)
    (map (lambda (x)(+ x 1)) x)))
```

or, equivalently

```
(define (x x)
  (map (lambda (x)(+ x 1)) x))
```

such that

```
>(x '(1 2 3))
>(2 3 4)
```

Here, in the expression `(+ x 1)`, the `x` refers to the innermost `x` (third instance). This property is known as *lexical binding*. It implies that a declaration will not preside over an inner region of code if the inner region has another declaration using the same name. In such cases, inner declarations are said to *shadow* outer declarations, or create *holes* in their scope. Other lexically scoped languages include *Pascal* and *Common Lisp*. Lexical binding can be achieved using the concept of the environment.

1.3 The Environment

An environment is an association of symbols to values. That is, in an environment each of a finite set of symbols maps to some value. This could be by means of a *symbol table*. Mathematically speaking, the environment is a function, a set of ordered pairs, Scheme symbols and values, whose domain is all Scheme identifiers, and whose range is all Scheme values. This could be formalized as follows:

$$F : D \rightarrow R, D = \{s_1, \ldots, s_n\} R = \{v_1, \ldots, v_n\}$$

1.4 Environment Frames

Environments are implemented by means of *frames*. It is within the frame that the symbol-value bindings described above are found. In addition to a set of bindings, a frame has a pointer to its parent frame. Environments are simply linked lists of frames. The frames are linked by the parent pointers, and terminate with the frame called *global environment*, also known as the *null environment* as its parent pointer points to *null*.

1.5 Environment Structure - Visibility

Environments are structured as trees or, more precisely as *directed acircular graphs* (DAGs). DAGs consist of nodes connected by edges. Edges are directional, they can only be traversed in one direction. In a DAG, circles are not allowed, i.e. for any node A, if a node B can be reached from A, then A cannot be reached from B. In terms of the environment, this means that a symbol is visible in a frame if it is bound within the frame itself, or if it is visible in the parent frame. Figure 1.1 shows an environment consisting of 11 frames, E0 to EA. Each frame contains a symbol. From any point in the environment, a symbol is visible if it is in the frame or in a frame accessible

from that point. So in frame *E7*, the symbols *a*, *b*, *e* and *h* are visible, and in frame *EA*, the symbols *a*, *c*, *g* and *k* are visible.

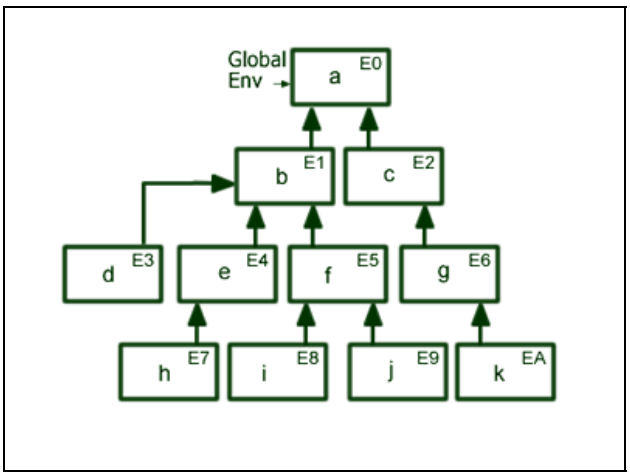

Figure 1.1: An Environment Structure

1.6 Extending The Environment

To add new bindings to an existing environment, that will be visible globally, top-level definitions must be used. Environments can also be extended with new frames, containing new bindings. Existing bindings will be visible in the new frame if they are visible in the parent frame. It is permitted to have multiple occurrences of any identifier at different points in the environment, and for new bindings to shadow existing bindings. This means that some mechanism must be in place to determine which variable occurrence is being referenced. This is discussed in section 1.13. As mentioned under the section on lexical scope, the syntactic form `lambda` is used for variable declarations, it follows then that it is by procedure application (evaluation of lambda expressions) that environments are extended. Procedure applications cause the binding of arguments with variables in an extended environment.

1.7 Procedures, Functions and Side Effects

The terms *function* and *procedure* do not always appear discrete. The situation is further complicated with terms such as *routine* and *method*. Procedures are usually understood as being sequences of instructions that can access and alter local and global variables, as well as (optionally) returning some value. The term function refers to procedures called explicitly for their return values. For simplicity only the term *procedure* is used in this

text. Scheme being a functional language, it is taken for granted that all procedure return values. However, the syntactic form `set!` allows for the assignment of variables, breaking with the strict functional paradigm, allowing for programs such as the following:

```
(define apple 10)

(define (reduce-apple)
  (set! apple (- apple 1)))

(define (bite-apple)
  (if (< apple 1)
      #f
      (begin
        (reduce-apple)
        'bite)))
```

Here the procedure `reduce-apple` is called purely for its side-effect, i.e. decrementing `apple`. The return value of `set!` is not specified by *Revised(5) Report on the Algorithmic Language Scheme* (*r5rs*). The details of this are left up to the implementer. This does not mean that it has no return value, as can been seen;

```
> (if (set! a 3) 'yes 'no)
yes
>
```

This break with the puritan functional programming paradigm is also apparent with the syntactic form `begin`, which is used for sequencing side effects, typical in imperative languages such a C. `begin` returns the value of the last expression evaluated, as below.

```
> (begin (define zz 3) ((lambda(x)(set! zz (+ zz x)))3) zz)
6
>
```

This sequencing of expressions is also allowed, without the explicit use of `begin` in conjunction with the keywords `lambda` and `cond`.

1.8 Procedure Objects - Closures

In functional programming we have the concept of *closures*. A closure is a data structure that holds an unevaluated expression together with the environment in which that expression is to be evaluated. In addition, a closure contains a list of identifiers. All identifiers used in the expression

that are not included in the list must be bound within the environment. The identifiers in the list however are bound on application. Procedures in Scheme are closures. The expression, together with the list of identifiers is referred to as a first-class object, or procedure object. Procedure objects have pointers to their environments, they can be created and returned as the result of evaluating expressions. Such objects can also be passed as input to other procedures, before themselves being applied to input to produce values. This is an important device in Scheme, and other languages such as Common Lisp and ML. It is the basis of *lazy evaluation* and central to the way programming techniques such as *recursion*, *message passing* and *stream-processing* are achieved in Scheme, as well as the modular, object-based strategy similar to the object-oriented approach we know from C++ and Java. In Abelson's & Sussman's *Structure and Interpretation of Computer Programs*, (*SICP*) procedure objects are described as pairs, created by evaluating a lambda expression. The pair consists of a pointer to the environment in which the lambda expression was evaluated, and a text; the body of the lambda-expression. More on this in section 1.12.

1.9 Evaluating Expressions

Expressions can be divided into two main types; *primitive* expressions and *derived* expression. The focus here will be on primitive expressions. These are:

- Variable references
- Literal expressions such as `5` and `#t`.
- Application expressions such as `(* 3 3)` and `(car '(a b))`.
- Procedures, such as `+` and `car`.
- Conditionals, such as `(if #f + *)`
- Assignments, using `define` or `set!`

Evaluation is as follows. Variable references evaluate to the value stored in the frame where the variable is bound. Literal Expressions evaluate to themselves. With application expressions, the operator and the operands are all evaluated separately. Evaluating the operator returns a procedure object. The closing frame of the procedure object is duly extended, with the values of the operands. This new frame is known as the *invocation environment*. The expressions in the body of the procedure object are subsequently evaluated, sequentially, within the invocation environment. The following are simple examples of Scheme interaction.

```
>(+ (+ 7 3) (- 3 4))
9
>
```

The conditional expression evaluates the first operand, here `car`. All objects in Scheme, bar `#f`, are treated as true, including the void value `#<void>`.

```
>(if (if #f #f) #t #f)
#t
>
```

1.10 The Read Eval Print Loop

Scheme is usually interpreted, rather than compiled. An interpreter is an interactive environment, very suited to rapid development and testing of code. The interpreter runs a Read-Evaluate-Print loop, in which Scheme code is read and parsed into an expression structure, then evaluated in the context of the global environment, before the resulting value is printed in some human friendly form. The Eval stage could look something like this:

```
(define (eval obj env)
  (cond((number? obj)obj)
       ((symbol? obj)(get env obj))
       ((conditional? obj)
        (eval-conditional obj env))
       ((assignment? obj)
        (eval-assignment obj env))
       ...))

(define (eval-conditional obj env)
  (if (eval (cadr obj) env)
      (eval (caddr obj) env)
      (eval (cadddr obj) env)))

(define (eval-assignment obj env)
  (add-binding env (second obj)
       (eval (third obj) env)))
```

The main procedure `eval` determines the type of object passed. Numbers evaluate to themselves, while symbols are used as identifiers for obtaining values from the environment `env`, using the `get` procedure. The `eval-conditional` procedure first evaluates the second object in the list to determine whether the third or fourth object should be evaluated and returned. Remember that the first element in the list of a conditional is simply a syntactic marker, namely `if`.

1.11 OOP in Scheme

Briefly, *object-oriented programming* (*OOP*) aims to structure programs in accordance with the perceived entities of the real or abstract world, simulating objective or conceptual reality. The objects of the program will have variables representing attributes associated with their prototypes. So instances of object types (object types are known as classes) will possess state, be dynamic. Also associated with classes are actions, often known as methods. The object-based approach described in *SICP* utilizes procedure objects and environments to achieve a variant of OOP. Using `define`, procedure objects and environments can be bound to identifiers. The environments can be used for storing the state variables of the objects, and the procedure objects can describe the methods.

1.12 The Environment Model of Evaluation

As explained, procedures in Scheme are first class objects. This property is nicely captured in *SICP*. The environment consists of a top-level frame, with a run-time library, and a dynamic list of bindings. When a lambda expression is evaluated within a frame, a procedure object is created. This object is described as a pair;

- A text, as given by the body of the lambda expression evaluated together with its parameter list.
- A pointer to the frame in which the procedure object was created.

The object can be bound to an identifier in the frame in which it was created by the `define` macro. When a procedure object is applied to arguments, a new frame is created in which the procedure's parameters are bound to the arguments. The body (the text part of the procedure object) is then evaluated in the context of this new frame. This frame has as its enclosing environment the frame pointed to by the procedure object. This model can be well illustrated with a simple example. The procedure `countdown` under returns a simple lambda expression.

```
(define countdown
  (lambda(n)
    (lambda()
      (if(<= n 0)
         'ping
         (begin
           (set! n (- n 1))
           #f)))))
```

Evaluating this at top level (in the global environment) causes the symbol `countdown` to be associated with a new procedure object, as shown in figure 1.2. The environment pointer points of course to the global environment. A call to `countdown` with the argument `2` will result in the creation of a new

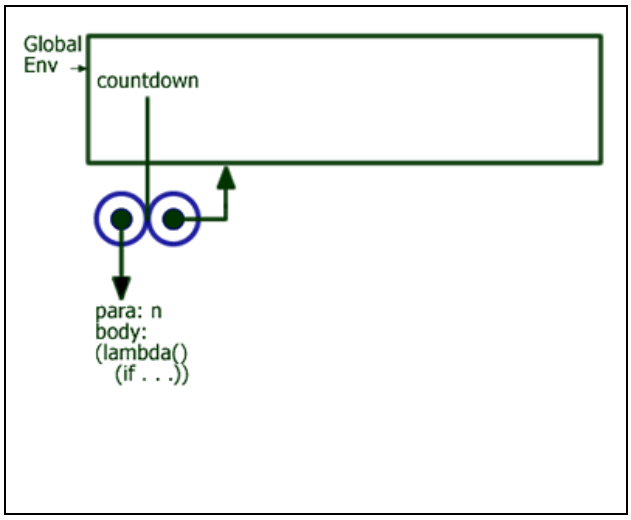

Figure 1.2: Global binding of procedure object to Symbol.

frame where `n` is bound to `2`, seen in figure 1.3. This call simply returns a new procedure object.

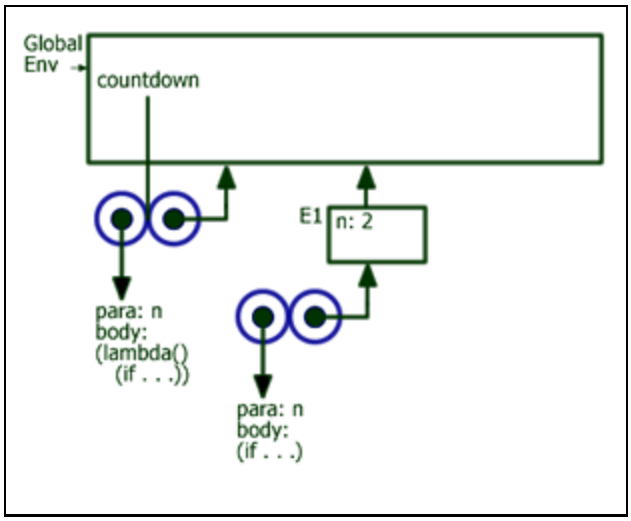

Figure 1.3: New procedure object.

```
> (countdown 2)
```

```
#<procedure>
>
```

Using `define`, we can bind this procedure, in the global environment to say `C1`, as follows:

```
> (define C1 (countdown 2))
>
```

This results in a new frame, `E1`, and a new procedure object, `C1`, pointing to `E1` (figure 1.4). A call to `C1` will add another new frame to the environment,

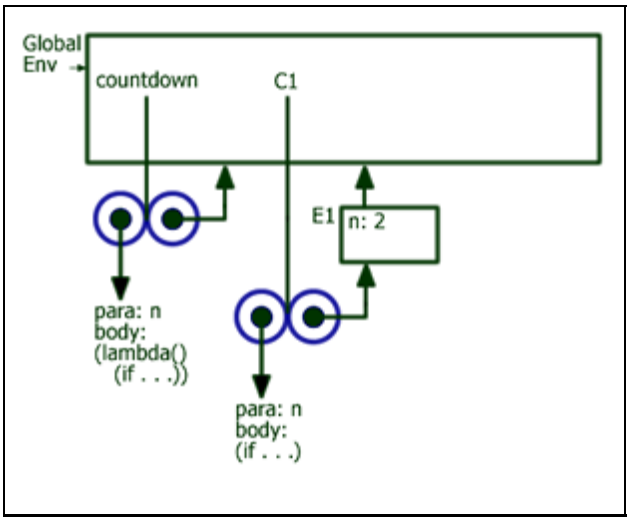

Figure 1.4: Binding the new procedure object.

referred to here as `E2` (figure 1.5). `E2` has `E1` as its enclosing frame, as this is the frame pointed to by the `C1`'s environment pointer. The body of the `C1`, namely the `if` expression, is then executed within the context of `E2`. `E2` contains no binding for `n`, and so the enclosing frame is checked. In `E1` `n` is bound to 2, the if-test fails, and the syntactic form `set!` is applied to reduce `n` by 1, and `false` is returned. After the procedure call has completed, the new frame, `E2`, is lost, as there are no pointers to it.

```
> (C1)
#f
```

1.13 The Environment's Structure - Locating Bindings

On evaluation, when any identifier is referenced, the Environment Model provides an easy method for identifying the correct binding. It is simply a

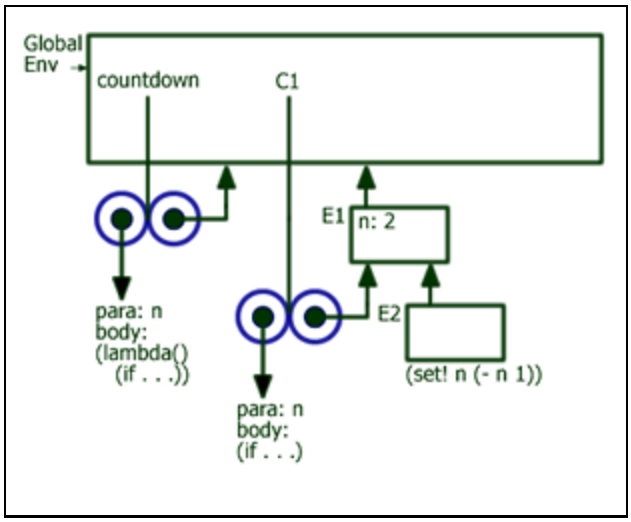

Figure 1.5: A call to C1.

question of proximity. The frame created during procedure application is checked first. There after the search continues on upwards, following the enclosing-environment pointers of each frame checked. This is the case for all symbols referenced in the procedure object's body. Since the environment is structured as a *DAG*, there is no risk of referencing a binding made out of scope; sibling frames are inaccessible.

1.14 A Subset of Scheme

As this project is an investigation into the representation of the semantics of the environment model of evaluation, only an appropriate subset of Scheme will be dealt with; such a subset as is needed for the bank example `make-withdraw` from *SICP*, along with a few other simple examples. The list of Scheme keywords used here is as follows:

```
define
lambda
set!
if
```

While the *list* is the data structure most associated with Scheme, it will not be dealt with here. The reason is that implementing the list structure would require a fair bit of extra work, and is not necessary for capturing the underlying semantics of the environment model. For this purpose numbers and numeric operators will suffice.

To simplify the reading of expressions via the *DOS prompt*, the use of non alpha-numeric symbols has been avoided. This means that the primitive operators have be given alpha-numeric equivalents, as in table 1.1

Scheme operator	**alpha-numeric equivalent**
`+`	add
`-`	take
`*`	times
`/`	div
`>=`	gte
`<`	lt
`=`	eq

Table 1.1: Scheme operators and their alpha-numeric equivalents

Chapter 2

The Abstract State Machine

2.1 Turing Machines

Yuri Gurevich first introduced Abstract State Machines (ASM) under the name *Evolving Algebras*. ASMs bear a strong resemblance to Alan Turing's universal computing machines, or Turing machines. A Turing machine consist of a state variable, an infinite tape on which any of a vocabulary of symbols can be written, and a read/write head that can navigate the tape (figure 2.1). The machine performs calculations defined by a set of *guards*

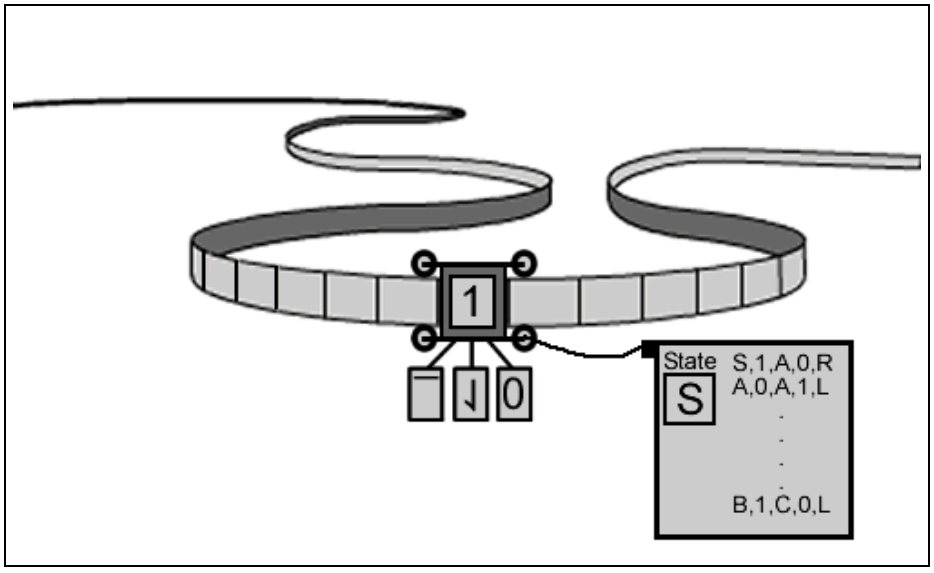

Figure 2.1: Turing Machine

and *updates*, or rules. The guard has two parameters; what state the machine must be in, and what the read-head must be reading on the tape, in order for the subsequent update to apply. The update has three parameters;

the next value of the state variable, what should be written on the tape, and in which direction the read/write head should be moved. Rules have then a very simple five-tuple form. Present state, input from tape, resulting state, output to tape, and direction to move head, or [S,R,N,W,D]. A simple unary adder could be written as follows:

```
1,1,1,#,R          5,1,5,#,L
1,0,2,#,R          5,0,6,#,L
2,1,3,#,R          6,1,6,#,L
2,0,7,#,L          6,0,1,1,R
3,1,3,#,R          7,0,8,#,L
3,0,4,#,L          8,1,8,#,L
4,1,5,0,L          8,0,9,#,#
```

A rule such as 3,0,4,#,L is read:

```
if state is 3 & read 0 then goto state 4 & write nothing &
                                              move head left.
```

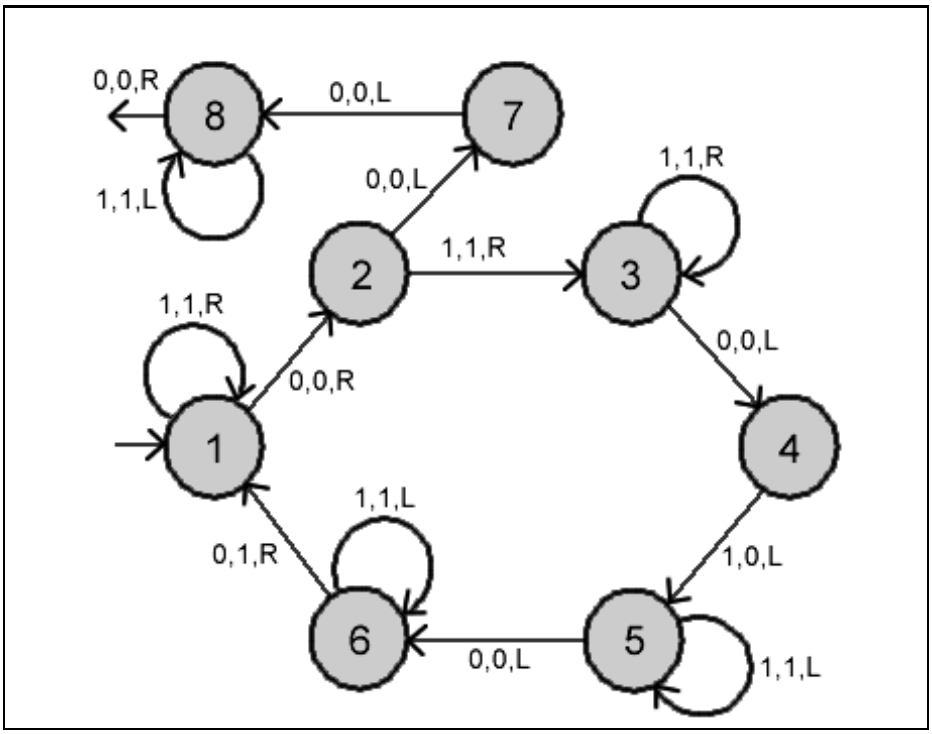

Figure 2.2: Turing Unary Adder

This adder can be visually represented as in figure 2.2, where the states are represented as nodes and the transitions as directed edges. Turing's thesis claims that every computable function is computable by some Turing

machine. As a medium for discussing the concept of computability Turing machines are very useful. However, they are not well suited as a tool for analyzing, verifying or modeling algorithms. The limitations of Turing machines are due to the medium for calculation, the tape.

2.2 Algorithms and Semantics

The problem with Turing machines is their poor operational semantics. That is they are some how not able to catch the essence of algorithms. But the notion of algorithms is not a very formal one. Algorithms seem to be understood as recipes that, if followed correctly, lead to the solution to given problems. However, this definition is rather vague, and Turing's thesis implies that any algorithm can be simulated by a Turing machine. So what is meant by poor operational semantics? Take the problem of computing expressions given in *reverse Polish notation* (RPN), for example `4 4 * 12 12 * +`, which is the equivalent of `12 * 12 + 4 * 4`. This is typically done using a *stack*. The stack is a simple tool for the temporary storage of data which ensures that the relative order of stored elements is maintained, following the *first in, last out* principle. Associated with the stack are the operations *pop*, *push* and *top*. The push operation is for placing new data elements on the top of the stack, pop is for removing the uppermost element, and top is for looking at the element at the top of the stack. Assuming the input is read as a list, one element at a time, then the standard algorithm is as follows:

- read (remove) an element from list
- if the element is a number, push it onto the stack
- if the element is an operator, pop two elements from stack, apply the operator, and push the result onto the stack

If this algorithm is repeated until the list is empty, the result will be the only element remaining on the stack. How would such structures and operations be represented using a Turing machine? Obviously they can be, but equally obviously there will be a tremendous amount of coding involved in implementation. Also, at the level of abstraction of the algorithm it is not necessary to take into consideration what kind of data that will be stored on the stack. In a Turing machine implementation this must be taken into account. Presumably operational semantics were not Turing's aim. Good operational semantics implies being able to model any algorithm at the natural level of abstraction, being able to use the vocabulary of the algorithm. Turing machines have a fixed level of abstraction, a predefined vocabulary, and are not suitable for modeling algorithms in general.

2.3 ASM

The ASM is a more general automata, not confined to just reading and writing symbols on an essentially two-dimensional storage medium. It can have any number of state variables. The ASM thesis claims that any algorithm can be described by an ASM, at the algorithms own level of abstraction. That is, each step of an algorithm maps to a discrete transition of the corresponding ASM, and vice versa. The ASM formalism uses standard mathematical notation to define machines by means of rules and variable updates. The syntax is extremely simple making ASMs very readable. In the run of an ASM, the rules are repeatedly applied, simultaneously, until such a point is reached when all possible consequent states are identical to the present state; in other words, when no rule application results in change of state. The following example is taken from Egon Börger's ASM tutorial (http://www.di.unipi.it/ boerger/). It demonstrates the fidelity of an ASM representation of a well-known algorithm, namely Conway's Game of Life. The game takes place on a matrix of $n \times m$ cells. Cells can be in one of two states; on or off (dead or alive). There are just three simple rules to Game of Life:

- Cells with exactly 3 living neighbours will turn "on" in the next round.
- Cells with exactly 2 living neighbours will stay as they are in the next round.
- Cells with any other number of living neighbours will be "off" the next round.

This translates to the following ASM. Note that the second rule is implicit, and does not need expressing in the ASM. `neighbours(cell)` is a call to an external function.

```
turnOff(cell, n) =
  if status(cell) = On and (n < 2 or n > 3) then
    status(cell) := Off

turnOn(cell, n) =
  if status(cell) = Off and n = 3 then
    status(cell) := On

gameOfLife =
  forall cell in Matrix do
    turnOff(cell, liveNeighbours(cell))
    turnOn(cell, liveNeighbours(cell))
```

The first two functions, `turnOff` and `turnOn` take the arguments `cell` and `n`. `n` is the amount of living neighbours the given cell has. The unary

function `status` gives the mortal status of the given cell. Repeated calls to `gameOfLife` increment the game, one step at a time. Each step consists of applying both `turnOff` and `turnOn` to each cell on the board simultaneously. It is important to remember that the calls to `turnOff` and `turnOn` occur simultaneously. If this happened in sequence, all cells would be turned on.

2.4 Lock-Step Execution

As shown, a major drawback with Turing machines is the excessive amount of steps involved in executing a calculation. This is a result of the very limited medium used, namely the tape. The tape only accommodates one character per square, and it is only possible traverse the tape one square at a time. This means that to represent most data some coding will be necessary. The more complex or abstract the data, the more elaborate the coding. This coding of data means that it will not be possible to apply the original operations or steps of the algorithm directly on the data. The operations will need to be broken down to the abstraction level of the Turing machine. The various steps of the algorithm will require varying numbers of steps in the corresponding Turing machine, often a great many more, as seen above in the unary adder. The point of ASM however, is that any algorithm can be represented at the algorithm's own level of abstraction, in what is called *lock-step* simulation, i.e. the ASM uses one step to simulate one step of the algorithm. This means that ASM provides a representation of any algorithm that will execute at the same speed as the algorithm itself. Where as Turing machines force us to act at a very low, fixed level of abstraction, ASMs are written using the same vocabulary as the algorithm.

2.5 AsmL

AsmL is an ASM programming language provided by Microsoft. It defines a large array of data types and structures and a library of operators and procedures, typical of other larger programming languages. Although ASM is a formalism developed and used by logicians, it is not difficult to understand when treated as a programming language. AsmL is in many respects just another programming language, most notably differing in the way statements are executed in parallel. It is object oriented and the structuring of the Scheme ASM module relies extensively on this feature.

2.6 The Semantics of Scheme Procedures

As has been stated, the desire here is to capture the underlying semantics of Scheme procedures at their natural level of abstraction. Natural level of abstraction is here understood as meaning an interpretation of a procedure in

terms of the model in use, i.e. the environment model. This is an important clarification. When writing Scheme code, a programmer is not likely to be thinking in terms of environment frames, or first class objects. Particularly when writing recursive procedures, the tendency is to ignore the details of execution. These details will however be fully visible in the ASM. Recursive procedures will loose their recursive look, and details of execution should be clear. The reason for this is simply that this project is an exercise in compilation, where the source language is of a higher level than the target language. The AsmL code should have a low level, instruction set look. Of course, AsmL is not a low level language, like *C* or *Assembler*, it allows recursion, is object oriented, etc. But care will be taken to avoid hiding execution details in the resulting code.

2.7 The Environment Model with AsmL

Some decisions must be made regarding how Scheme code shall be represented using AsmL. Particularly regarding the data structures and administration. As established, the model to be used here is the Environment Model of Evaluation from *SICP*. The intention is to capture faithfully the semantics of any procedure written in Scheme in the terms of this model. There are two main components used in the model; the environment, and the procedure object. Before dealing with these two components however, a quick look at the representation of *types* is called for.

2.8 Types

Scheme offers a range of different data types, amongst others numbers, strings, lists and importantly here, procedure objects. Because of type constraints on pointers in AsmL, and indeed most languages, a generic Scheme object or *form* will be useful. This suggests the use of *classes* and *inheritance*; a general Scheme class can spawn the necessary types. This will allow the use of generic pointers that can refer to all the different types used in Scheme, such as procedures and numeric values.

```
class ScmObj
```

This class will never be realized directly, and so this definition suffices.

2.9 The Procedure Object

One of the main components of the Environment Model is the procedure object. The procedure object consists of a frame pointer, a list of parameter labels or identifiers, and a procedure body. As mentioned in section 2.8,

procedure objects are just one of various different Scheme objects used in Scheme, and they shall extend the generic Scheme object class `ScmObj`. The following is the class definition for the procedure object.

```
abstract class ProcObj extends ScmObj
 abstract Ep as Env
 abstract getPara() as Seq of String
 abstract Body(e as Env) as ScmObj
```

The `abstract` tag preceding the `ProcObj` declaration indicates that this class cannot be implemented directly, such an object would make no sense. `abstract` is really to indicate that a new *type* is being created. Other classes will be declared that implement `ProcObj`, the actual procedures, both user-defined and standard. The `abstract` keyword simply tells the compiler that any class extending `ProcObj` must override the methods and fields marked as abstract. This gives assurance that any subclasses of `ProcObj` can be treated in the desired way. The actual procedure objects will *extend* this class using the `extend` keyword.

```
(define Foo
  (lambda (n m)
    ...))
```

The Scheme procedure `Foo` would translate to something like the following:

```
class Foo extends ProcObj
 Para = ["n", "m"]

 getPara() as Seq of String
  return Para

 Body(e as Env) as ScmObj
  step
   ...
```

This is just a definition of a particular procedure object. Actual procedure objects are, of course created using the `new` keyword.

```
new Foo(Global)
```

This will create a new `Foo` procedure object in the top-level environment.

2.10 The Environment

The other main component of the model is the environment itself. The environment consists of frames. Each frame has a set of symbol-value bindings, and a pointer to its enclosing environment. The values bound can be

any of various types, as mentioned in section 2.8. AsmL provides a data type found in other programming languages, called the *map*. Maps, as the name suggests, associate identifiers and values, which is exactly what the environment function does. The following is the definition of the class `Env`. It includes a pointer to an enclosing environment, that will be `undef` for the global environment, as well as a `Map` from `String` to `ScmObj` (Scheme objects), for holding definitions. The methods `Define` and `Value` are for adding and accessing definitions, respectively.

```
class Env
  Ep as Env
  var Bindings as Map of String to ScmObj = {|->}

  Define(sym as String, val as ScmObj)
    Bindings(sym) := val

  Value(Sym as String) as ScmObj
    if(Boundp(Sym)) then return Bindings(Sym)
    elseif(Ep <> undef) then return Ep.Value(Sym)
    else return undef

  Boundp(Sym as String) as Boolean
    if (Bindings(Sym) <> undef) return true
    else return false

  shared ExtEnv(Para as Seq of String,
                Args as Seq of ScmObj, env as Env) as Env
   var l as Integer = length(Para)
   var i as Integer = 0
   e = new Env(env)
   step while i < l
    e.Define(Para(i), Args(i))
    i := i + 1
   step
    return e
```

The method `ExtendEnv` is a so called *static* or *class* method, that is, a method that due to its domain is associated with a class, but that is not associated with each object of that class. In AsmL jargon, such methods are called *global methods*, as opposed to *instance-level methods*. They are defined using the `shared` tag. `ExtEnv` creates and returns a new environment frame. It takes as its arguments a sequence of character strings and a sequence of corresponding Scheme objects, of equal lengths. These are the symbol-value pairs to be bound within the newly created environment. Additionally the

method is passed a pointer to the environment that is to be the enclosing environment.

2.11 Numeric Values

While Scheme provides a variety of data types, only operations on numeric values are dealt with here, specifically integers. The `ScmObj` class `NumVal` encapsulates integers, enabling them to be passed and returned between procedure objects, as well as bound in environments.

```
class NumVal extends ScmObj
 var Val as Integer

 getVal() as Integer
  return Val
```

This class consists simply of an integer holder for the value, as well as an accessor.

2.12 Standard Procedures

Built into Scheme is a small set of primitive procedures such as arithmetic operators and list procedures. There are also built-in library procedures implemented using these primitive procedures. All such procedures are bound to identifiers in the top-level environment. Some such procedures will need to be available in the ASM. For practical reasons, only a small subset of the operations defined in *r5rs* will be implemented here; those needed for the chosen subset of Scheme covered, essentials such as the arithmetic operations. The way in which standard procedures are treated is not described in the environment model, the same applies for compiled procedures. Two approaches seem viable here. The first being that identifiers such as `+` and `car` are still associated with procedure objects, with parameter lists and environment pointers. These objects should not however, have bodies in the same sense as procedure objects hitherto described. They should only be references to directly executable AsmL procedures, and not to procedure objects. Such standard procedures should be available in the Scheme interpreter ASM, as procedure objects defined in the top-level environment. The other approach is to have standard procedures *hard coded* directly into the user defined procedure objects in which the are used, as AsmL procedure calls, avoiding further reference to procedure objects; an *in-line* variant. This requires that the status of a procedure in the Scheme code as a standard procedure must be asserted at compile-time. This could be achieved using a list of identifiers, reserved names. The first approach seems however preferable as all procedure applications in the Scheme code can be treated

alike. The following is an example of such a standard procedure. In most respects it is just like a user-defined procedure. The difference is seen in its return statement; a reference to an AsmL procedure. Usually all procedure calls would be made using the `Eval` procedure.

```
class Addition extends ProcObj
 Para = ["n", "m"]

 getPara() as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal
  var n as NumVal
  step
   m := e.Value("m") as NumVal
   n := e.Value("n") as NumVal
  step
   return Plus(m, n)
```

The AsmL procedure `Plus` is as follows:

```
Plus(n as NumVal, m as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  return new NumVal(x + y)
```

Another interesting standard procedure is `SetBang`. The procedure object definition is as follows:

```
class SetBang extends ProcObj
 Para = ["sym", "val"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var val = e.Value("val") as NumVal
  var sym = e.Value("sym") as Symbol
  var found as Boolean
  step
   found := SetB(e, sym.id(), val)
  step
   if(found) return val
   else return new Error("set!: cannot set undefined
                          identifier: "+sym.id())
```

This simply makes a call to the AsmL procedure `SetB`, before returning the new value assigned. The procedure `SetB` works its way recursively upwards through the environment frames, starting with the one passed by `SetBang`, until it finds an occurrence of the target symbol. On doing so, it then changes the bound value. If no occurrence of the target symbol is located, `SetB` does returns false. If false is returned, the SetBang `Body` method returns another `ScmObj` type; `Error` (see section 2.15).

```
SetB(e as Env, sym as String, val as ScmObj) as Boolean
 v as NumVal = val as NumVal
 step
  if(e.Boundp(sym))
   e.Bindings(sym) := val
   return true
  elseif(e.Ep <> undef)SetB(e.Ep, sym, val)
  else return false
```

2.13 Define

So far, only the creation of procedure objects has been discussed. For such objects to be of any use they must be bound somewhere. This is just what the `Env` method `Define` does. Adding a procedure object of type `Foo` to an environment `Global` would be done as follows.

```
Global.Define("Foo", new Foo(Global))
```

This binds an object of the `Foo` class within the environment `Global`, giving the object `Global` as its enclosing environment. The method `Define` updates the local map `Bindings` with this string-object pair.

2.14 Administration

The model for evaluation clarifies issues of scope, assignment, etc. The process of applying procedures remains to be addressed. Procedure application triggers the extension of the environment, the binding of identifiers to values, and the evaluation of procedure bodies. Some agent responsible for these actions is needed. The agent will also require a limited bookkeeping capability. On procedure application, the agent must obtain the correct procedure object from the environment, determine the objects environment, and create a new frame within this environment. The appropriate bindings must then be added to this new frame. Next, the agent must somehow effect the evaluation of the procedure body in the context of the extended environment and return the resulting value for printing. The requirements are fairly limited; an environment pointer for the new frame, and a procedure

object pointer. However, in many cases, such as recursive procedures, the evaluation is dependent upon the value of successive evaluations.

2.15 Evaluation

In Scheme, evaluation of an expression is triggered by parentheses. The enclosing of a symbol or expression within parentheses, along with any other symbols, expressions or literals, forces evaluation. In the ASM evaluation will be achieved by the explicit use of a procedure `Eval`. Essentially, the `Eval` procedure is as follows.

```
Eval(proc as String,
     args as Seq of ScmObj, env as Env) as ScmObj
 var pro as ProcObj = env.Value(proc) as ProcObj
 step
  Frame := ExtEnv(pro.getPara(), args, APPROPRIATE ENVIRONMENT)
 step
  return pro.Body(Frame)
```

This AsmL procedure has a character string `proc`, a sequence of Scheme objects `args` and an environment `env` as its formal parameters. The name under which the required procedure object is bound is given by the character string. The correct procedure object is subsequently located from the environment referenced or its enclosing environment. The arguments, or actual parameters for the procedure application are given in the sequence `args`. The next step is to bind the arguments to the parameters specified by the procedure object, within the new environment frame created using `ExtEnv`. This frame is given the procedure object's environment `pro.Ep` as its enclosing environment, except in the case of the procedure object being `set!`. The final step of `Eval` is to return the procedure object's `Body` method, in its appropriate context, i.e. with the newly created environment frame as its argument. The reason that this works is that the AsmL implementation of any procedure object either only contains standard procedures, implemented directly as AsmL procedures, or it utilizes `Eval` itself. So if `Eval` is viewed as an agent, then many evaluations, or runs will involve multiple agents. In this way, all runs are ultimately broken down into a series of standard procedures, implemented using AsmL procedures. For robustness and usability, the `Eval` procedure must be more elaborate. Various checks need to be in place, and suitable alternatives provided in case of invalid or illegal procedure application.

```
Eval(proc as String, args as Seq of ScmObj,
                        env as Env) as ScmObj
 var Obj as ScmObj = env.Value(proc)
 var Args as Seq of ScmObj = []
```

```
var Frame as Env
var pro as ProcObj
var Actu as Integer
var Form as Integer
var i as Integer = 0
step
 if(Obj <> undef)
  step
   if(Obj is ProcObj)
    step
     pro := Obj as ProcObj
    step
     if(pro is SetBang)
      i := 1
      Args := Args+[args(0)]
     Actu := length(args)
     Form := length(pro.getPara())
    step
     if(Actu = Form)
      step while i < Actu
       if(args(i) is Symbol)
        step
         var sym as Symbol = args(i) as Symbol
        step
         Args := Args+[env.Value(sym.id()) as ScmObj]
       else
        Args := Args+[args(i)]
       i := i + 1
      step
       if(pro is SetBang)
        Frame := ExtEnv(pro.getPara(), Args, env)
       else Frame := ExtEnv(pro.getPara(), Args, pro.Ep)
      step
       return pro.Body(Frame)
     else
      return new Error(wrong number of arguments)
   else
    return new Error(invalid procedure)
 else
  return new Error(undefined identifier)
```

This version of `Eval` does not take for granted the existence of the procedure named, nor that the correct number of arguments has been past. In case of error or disparity, `Eval` returns another `ScmObj` heir, namely an `Error`

object.

```
class Error extends ScmObj
 var msg as String

 getMsg() as String
  return msg
```

In the case of a valid procedure application, `Eval` passes the newly created environment frame to the `Body` method of the identified procedure object, i.e. evaluates the procedure in the context of the extended environment. It is important to notice that on creating the new environment frame, the identified procedure object's environment pointer is passed to `ExtEnv`. In this way the new frame gets the procedure object's environment as its enclosing environment. This is not however the case when the procedure being applied is `set!`, as discussed in section 2.16.

2.16 Assignment

Assigning new values to existing identifiers in Scheme is achieved using `set!` (pronounced *set bang*). As already mentioned, `set!` breaks with the pure functional approach to programming. It also breaks with normal rules of evaluation. This is due to `set!`'s inconsistency with functional programming. Usually, procedure application is a case of extending an environment with a frame in which evaluation can take place, and returning the resulting value. What happens within the frame is of no concern or consequence to the outside world, so to speak. Also, and importantly, the enclosing environment of the frame is the environment indicated by the procedure object being utilized. For standard procedures this means the global environment. From this point in the environment *tree*, only global identifiers will be visible. This means that identifiers referred to in the procedure application are not necessarily visible in the context of the evaluation. For `set!` to be of any use it must be able to access other relevant frames, or nodes of the environment tree, where the target identifiers reside. `set!` is only ever applied for its *side effects*. This is presumably the reason the *r5rs* does not specify its return value. Figure 2.3 shows the application of `set!` using normal rules of application. It is easy to see that no occurrences of the relevant symbol are visible. Indeed, it could be the case that another occurrence of the named symbol could appear in the global environment, causing an unintended reassignment. Figure 2.4 shows how the assignment must be applied in order to succeed as intended.

Another point to note here is the way in which variables are treated in the bodies of procedure objects. It could be tempting to store values obtained from environments locally. This would cut down on variable reference, and might be considered more efficient. However, this has its hazards, as can be shown with a little example.

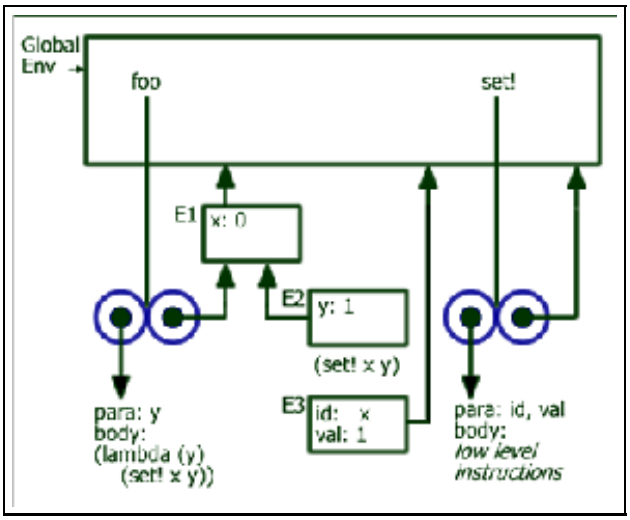

Figure 2.3: Bad Set!

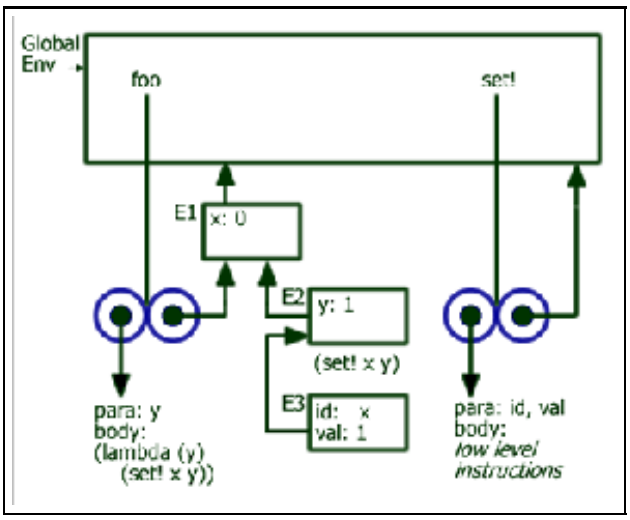

Figure 2.4: Set!

```
(define lazy
  (lambda (x)
    (lambda (y)
     (if (> x y)
      (set! y x))
    y)))

class lazy extends ProcObj
 Para = ["x"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  return new lazy_lambda(e)

class lazy_lambda extends ProcObj
 Para = ["y"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var x = e.Value("x") as NumVal
  var y = e.Value("y") as NumVal
  if(x > y)
   Eval("set!", [new Symbol("y") as ScmObj, e.Value("x")], e)
  step
   return y
```

This is an incorrect representation of the semantics of the above Scheme code. The reason is that the local variable `y` used in the return value is not a pointer to the value stored in the environment `e` under the symbol `y`. It is merely a copy, made prior to the if-test, and there is no guarantee that its value is up to date. If it was not for `set!` then this would be fine, as identifier-value pairs would remain constant.

2.17 Embedded Lambda Expressions

As established, a procedure object in Scheme is created by the use of the `lambda` key word. It is often useful to have lambda expressions return new lambda expressions, as in the *SICP* bank example. This can be achieved in the ASM by simply having the `Body` methods of such procedure objects return new objects. The example under shows how this can be done, using the

suffix *lambda*, to generate a class name built on the name of the enveloping class.

```
(define foo
  (lambda(x)
    (lambda(y)
      (set! y x)
      y)))
```

The Scheme procedure `foo` translates into the following AsmL code.

```
class foo extends ProcObj
 Para = ["x"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  return new foo_lambda(e)

class foo_lambda extends ProcObj
 Para = ["y"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  Eval("set!", [new Symbol("y") as ScmObj, e.Value("x")], e)
  step
   return Eval(new Symbol("y"), e)
```

Chapter 3

Compilation

3.1 Overview

Roughly speaking, the process of compiling source code can be divided in two; parsing and generating. These two steps will be dealt with in detail in separate chapters, but a general overview of the process will be helpful. Figure 3.1 shows the data flow, from Scheme source code to AsmL source code. The discs represent Java objects, while the boxes represent classes of static Java methods.

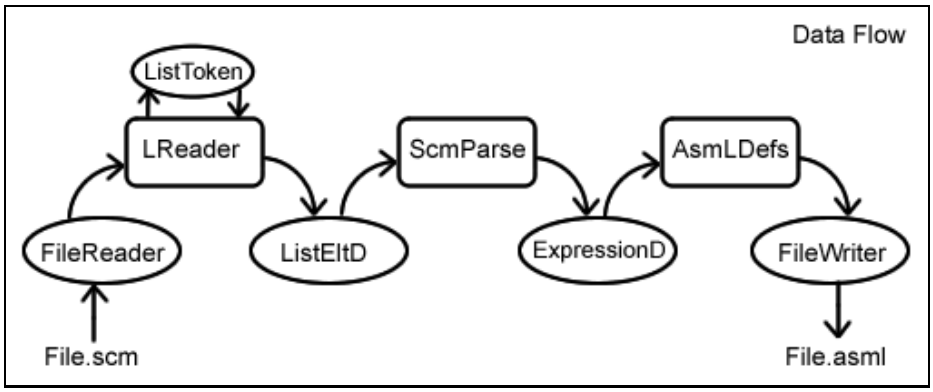

Figure 3.1: The data flow

3.2 Scheme Syntax

Though the process of compilation was described as a two step operation, it is clear from the diagram that there are three stages. The extra stage is a pre-parse needed for structuring the code in accordance with Scheme syntax. Scheme syntax is fortunately very simple; all code is written as parenthe-

sized lists. This is particularly useful as the list is also the predominant data structure of Scheme. It means that Scheme has a well defined set of operators and procedures for processing lists. From this point of view it would have been far simpler to implement a Scheme to AsmL compiler in Scheme. However, for reasons of no importance to this project, Java was chosen.

3.3 Reading Source Code

Using Java, parsing directly from the text string would be difficult. The operations defined on strings are limited, and we would not be able to perform the parse at an appropriate level of abstraction. That is, the vital list operations of *first* and *rest* cannot be performed on strings, obviously. The Scheme `read` function parses strings into list structures, which include symbols and numbers. Scheme represents lists using *pairs*. A *pair* in Scheme consists of a *car* field, and a *cdr* field. Pairs are created using the procedure `cons`, and the fields are accessed using the procedures `car` and `cdr` respectively. Both car and cdr fields can contain any data structure defined in Scheme. For simplification however, only pairs, symbols, numbers and a representation of the empty list, will be considered here. While parsing the list representation of the Scheme code will be dealt with in chapter 4, parsing the text stream to a list structure will be looked at here.

3.4 Lists

A list can be defined recursively as either the empty list, or a pair whose cdr is a list. In BNF this could be stated as follows:

```
<list>::=[]
      ::=[_,<list>]
```

The set of lists as defined in *r5rs* is the smallest set X such that

- The empty list is in X.
- If list is in X, then any pair whose cdr field contains list is also in X.

Lists are constructed by assembling chains of pairs, as in figure 3.2. The first stage parse is to derive this list structure. A list structure can be viewed as a tree. Pair objects will form the nodes of the tree, while the atomic list elements comprise the leaves, or terminal nodes.

3.5 Parsing Text

The *LReader* class (appendix B.3), so named to distinguish it from the Java's own *Reader* class, is responsible for parsing the source with regards to a list

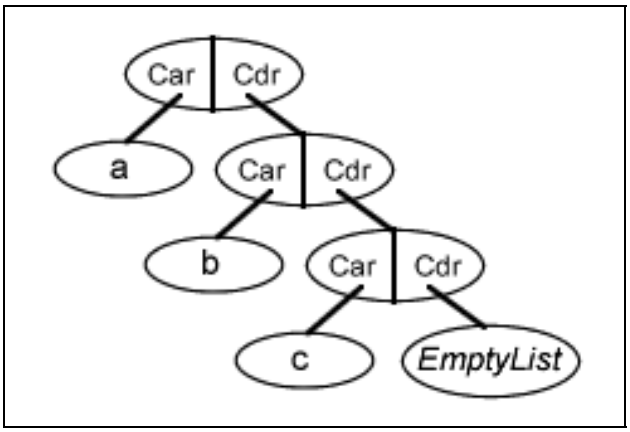

Figure 3.2: The list '(a b c)

structure. The list is represented as a tree, using derivatives of the *ListEltD* class (appendix B.1.1); that is *Cons* (B.1.2), *Literal* (B.1.3) or *Symbol* (B.1.4). List structures are formed using these classes as in figure 3.3. As

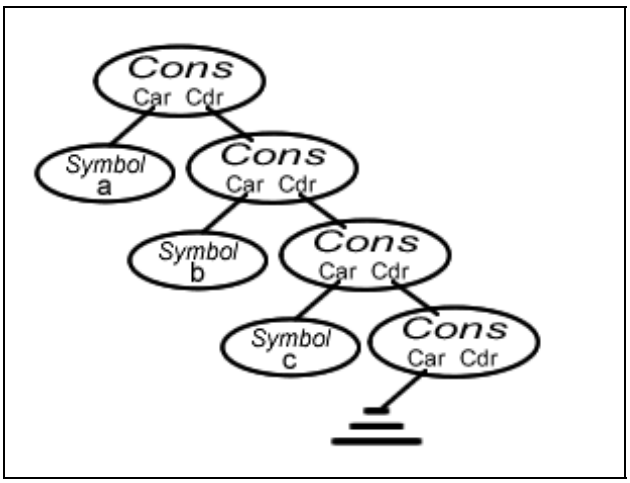

Figure 3.3: The list '(a b c) using the ListEltD class

can be seen, the inner nodes are built from `Cons` objects, while the leaves are all either `Symbols` of `Literals`, the exception being the empty list. The empty list is represented using a `Cons` whose `Car` field is set to `null`. This is distinct from the list whose first element is the empty list. A good case could be made for writing a special derivative of `ListEltD` particularly for this purpose, and while the present solution suffices, this would appear to be a nicer approach. This is however an opinion formed with hindsight. LReader's `read` method takes a Java `Reader` as its argument, and returns

a `ListEltD`. Most of the work is done in the `LReader.ReadList` method. The actual elements of the text file are read and interpreted using the LReader's `tokenString` method, which creates and returns objects of the `ListToken` class (appendix B.2). `ListToken` acts as a buffer between the `LReader` and the Java `StreamTokenizer`. This is helpful in that the tokens, or list elements read can be strings, characters or numbers. The `ListToken` class has methods and fields that reveal what kind of element it is. The `ListToken` objects represent symbols, numbers, left-parenthesis or right-parenthesis. While reading a list (using the `LReader.ReadList` method), a left-parenthesis causes the creation of a new `Cons` with its `Car` and `Cdr` fields filled by successive calls to `ReadList`. A right-parenthesis causes an empty list `Cons`. A `ListToken` with a number prompts the creation of a `Literal`, while a `ListToken` with a string creates a `Symbol`. The process of parsing the input file to a list structure is shown in figure 3.4.

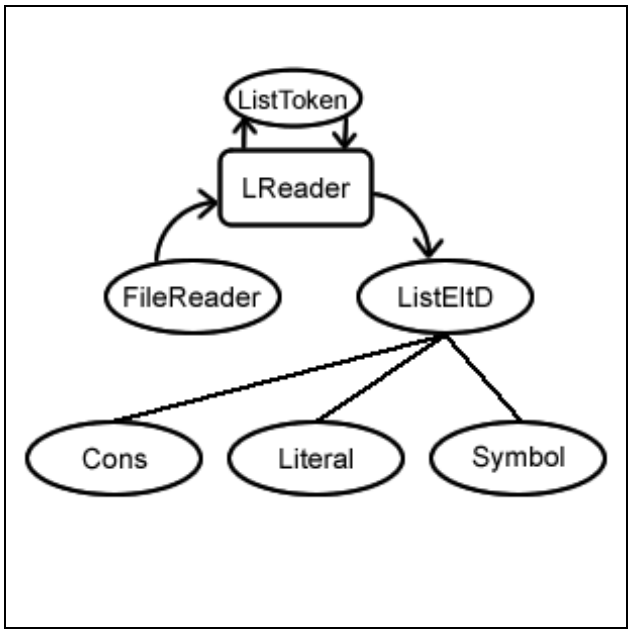

Figure 3.4: The list reader

3.6 Parsing Lists

Once the Scheme code is represented as a ListEltD object, it needs to be parsed with regards to the syntax of Scheme's keywords and special forms. The details of this process are described chapter 4. The Java class structure is quite similar to the structure just described in section 3.5. The class responsible for the parse is `ScmParse` (appendix C.3), which consists of

various static methods responsible for parsing the various expression types. `ParseExpression` is the method responsible for identifying the type of expression found. The `ScmParse` class inherits a library of methods from the superclass `JScheme` (C.2), for processing lists and identifying Scheme keywords.

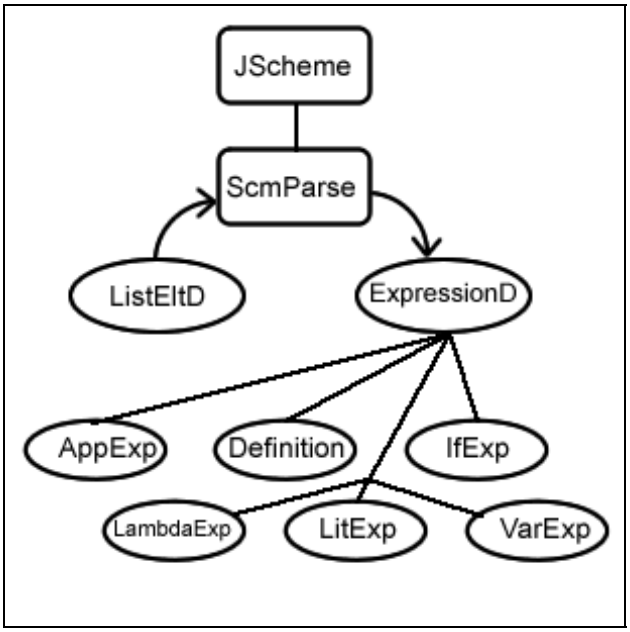

Figure 3.5: The ListEltD parser

3.7 Generating AsmL

The final step is generating an output file (figure 3.6), containing the AsmL source code. The `AsmLDefs` (appendix D) class contains an array of static methods for assembling the AsmL code representation of the Scheme procedure parsed. The code generated is returned by the `addGlobalDef` method as a string.

3.8 Coordination

The various pieces of the compiler need to be tied together, and the data flow coordinated. The Java class `Schasm` (Scheme Asm, appendix E) is responsible for this (figure 3.7). `Schasm` has four pointers; a `FileReader` FR, a `ListEltD` L, an `ExpressionD` E and a `FileWriter` FW. Each is initiated to point to a newly created object of the relevant type, and the `compile` method carries out the whole process.

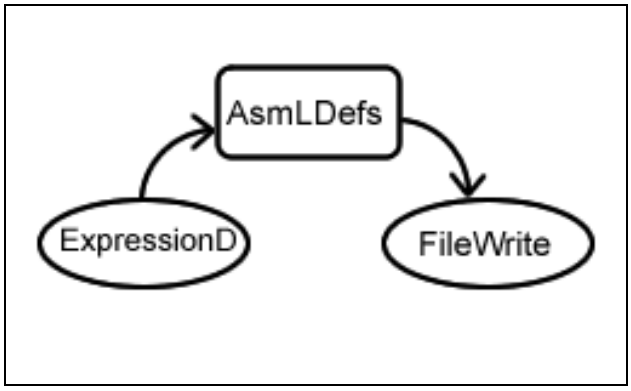

Figure 3.6: The AsmL definition generator

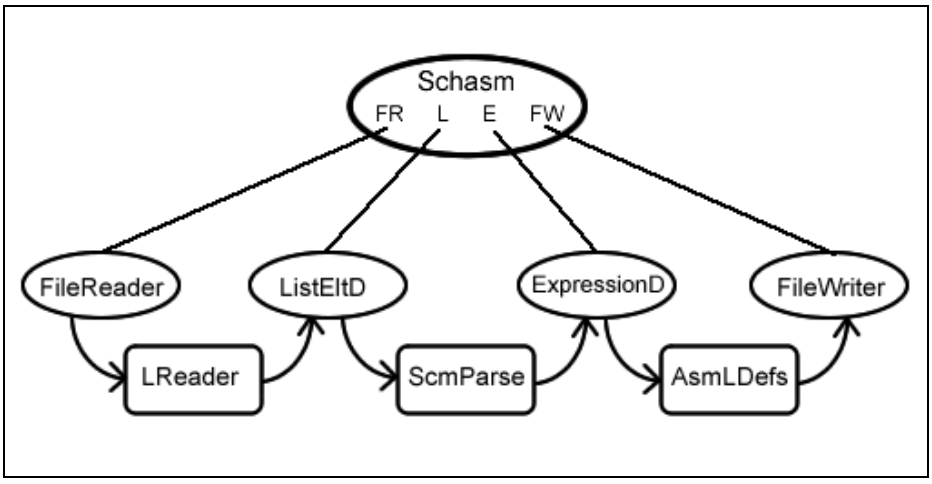

Figure 3.7: Schasm

Chapter 4

Parsing Scheme Expressions

4.1 Syntax

First step in the process of generating an ASM from Scheme source code is generating an abstract syntax tree representation of the Scheme expressions described by the code. The parse is done in accordance with a grammar, written in *BNF*. To build such a grammar, the components or forms of Scheme expressions must be identified. Scheme uses a parenthesized-list Polish notation to describe programs and other data. A typical scheme expression might look like the following,

```
(lambda (grutts hoot)
  (moggify grutts (rompus (dwindle grutts hoot))))
```

revealing three types of expressions, i.e. *variable expressions*, such as `grutts` and `hoot`, *lambda expressions*, and *application expressions*, such as `moggify`, `rompus` and `dwindle`. This could be encompassed by the following grammar, taken from *Essentials*:

```
<expression> ::= <identifier>
                 [var-exp(id)]

             ::= (lambda(<identifier>)<expression>)
                 [lambda-exp(id body)]

             ::= (<expression><expression>)
                 [app-exp(rator rand)]
```

4.2 Abstract Syntax Tree

A parse of the above Scheme expression in section 4.1 should produce an abstract syntax tree that could be visually represented as in figure 4.1 An

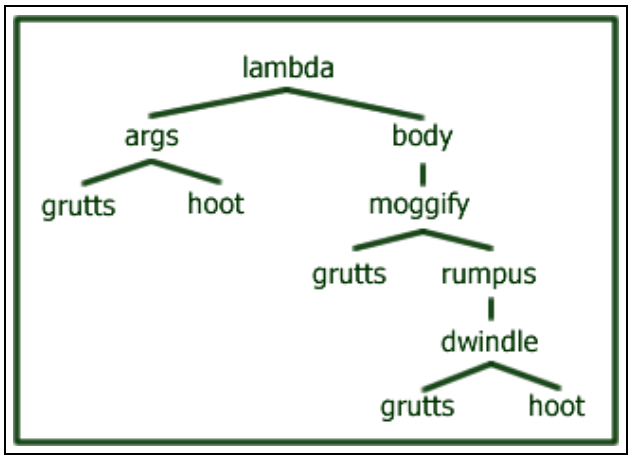

Figure 4.1: Abstract Syntax Tree.

abstract Java class can be used to represent the Scheme expression, and extensions of this class can then implement the specific manifestations of expression, i.e. variable, lambda etc. Thus the grammar above will yield the following Java class-hierarchy:

```
abstract class Expression

class VarExp extends Expression
class LambdaExp extends Expression
class AppExp extends Expression

Expression
        |
        |-->VarExp
        |
        |-->LambdaExp
        |
        |-->AppExp
```

The classes `LambdaExp` and `AppExp` will both need pointers to nested expressions; `id`, `body`, `rator` and `rand`. The pointers will facilitate the construction of the abstract syntax tree, where instances of the expression classes (objects) will form the nodes and leaves of the tree, while the afore-mentioned pointers will comprise the edges. It is here the use of an abstract class declaration is necessary; on creating an expression object, the type of a nested expression is still not known, so nested expression pointers must be of most general type.

4.3 Abstraction

An issue that has to be addressed is that of abstraction. It is desirable that the class and method structure of the Java code should resemble the data structure abstractions of Scheme closely. This will place constraints on the way the Java code is written. An example of this is seen with the `Cons` class. The `Cons` class has fields `car` and `cdr`. The class is given default access status, and the fields have private access, in accordance with a cautious approach to programming. This means that the class must be equipped with methods for accessing the values stored in these fields from external classes. Given such accessors, it would be tempting to use them as implementations of the Scheme functions `car` and `cdr`. But this would break with the abstraction. The following is a an example of an interaction with the Scheme interpreter:

```
> (car 'grutts)
car: expects argument of type <pair>; given grutts
```

`car` is a primitive Scheme standard procedure, and not a procedure associated with a pair, that is, not a method of `Cons`. This implies implementing `car`, `cdr` and various other procedures externally to any `ListElt` class derivatives. The class `JScheme` (appendix C.2) is just such a collection of static methods that operate upon `ListElt` objects. The importance of abstraction is also seen when traversing a list, due to the need for type-casting in Java. That is, any attempt to access the `car` field of an argument `(ListElt)datum`, known to be of type `Cons`, requires type-casting, i.e.

```
if(pair(datum))
 recur(datum.car())
```

would need to be rewritten to

```
if(pair(datum)){
 Cons l = (Cons)datum;
 recur(l.car());
}
```

This may not seem too long-winded, but it would cloud any resemblance to Scheme the parse method might have. Writing the `car` and `cdr` methods externally to the `Cons` class, with the signature `ListElt car(ListElt)` will maintain the desired level of abstraction, enabling the following style:

```
if(pair(datum))
 recur(car(datum))
```

This is particularly desirable when writing longer car-cdr chains, such as

```
foo(car(cdr(cdr(datum))))
```

4.4 Read Failure

Operating at this level of abstraction, some device for dealing with error should be provided. The Scheme code input could be incorrectly formed. Various `JScheme` methods, such as `car` and `cdr`, will throw `ExpectsPair` exceptions if sent `Literals` or `Symbols`. Likewise, exceptions will be thrown on encountering malformed *if* expression and *application* expressions.

4.5 Kleene Star

The original grammar needs to be extended, by the introduction of *Kleene star*. This is to accommodate multiple arguments to lambda expressions and multiple operands to application expressions. The extended grammar is as follows:

```
<expression> ::= <number>
                 [lit-exp(datum)]

             ::= <identifier>
                 [var-exp(id)]

             ::= (lambda({<identifier>}*)<expression>)
                 [lambda-exp(ids body)]

             ::= (<expression>{<expression>}*)
                 [app-exp(rator rands)]
```

The introduction of Kleene star to the grammar forces a restructuring of the classes `LambdaExp` and `AppExp`. Prior to this elaboration, the two classes needed only simple `Expression` pointers to their nested expressions; i.e. `ARG` and `RAND`, respectively. These must subsequently be changed to `Expression` array pointers; `ARGS` and `RANDS` (see appendices C.1.2 and C.1.5). The creation of objects of type `LambdaExp` and `AppExp` will be facilitated using the `ParseList` method in `ScmParse` (appendix C.3). This method traverses iteratively lists of expressions, such as those found in lambda or application expressions. Regarding the use of arrays; they seem preferable to vectors for two reasons. Firstly, they do not incur the same amount of overhead as vectors, though this is maybe not worth worrying about. Secondly, vectors entail type-casting; they can contain any Java object, and do not need to be homogeneous. Thus accessing the contents of a vector demands type-casting. The disadvantage of using arrays here is that they are static data types. Whereas vectors grow to accommodate more data, arrays demand reallocation. The solution in `ParseList` is to use vectors to hold elements while traversing the list, and then convert the vector to an `Expression` array, before returning. A further change to the grammar here is the addition

of literal expressions, to accommodate numbers. However, this needs further attention as literal expressions will need also to encompass strings and quoted symbols. One solution would be to let the `LitExp` class spawn subclasses for numbers, strings and quoted symbols. For now however, literals shall be used for numbers.

4.6 Conditionals

Again the grammar must be extended, to describe another important concept of any programming language; the conditional. Only the *if* syntactic form will be covered here. The Scheme `cond` form is a macro, that expands to an elaborate `if` construction at compile time. The grammar now looks as follows:

```
<expression> ::= <number>
                 [lit-exp(datum)]

             ::= <identifier>
                 [var-exp(id)]

             ::= (if <expression> <expression> <expression>)
                 [if-exp (test-exp true-exp false-exp)]

             ::= (if <expression> <expression>)
                 [if-exp (test-exp true-exp)]

             ::= (lambda({<identifier>}*)<expression>)
                 [lambda-exp(ids body)]

             ::= (<expression>{<expression>}*)
                 [app-exp(rator rands)]
```

The `if` expression `IfExp` has been added with the help of two new rules in the grammar. The first for the conditional with an alternative should the test expression evaluate to `false`, the second without this alternative expression. It is worth noting here that in Scheme, an `if` expression whose test expression evaluates to `false`, and that does not have an alternative expression, returns the Scheme value `#<void>`. The details of this point are Scheme implementation dependent. In *MsScheme*, `#<void>` evaluates to `true`. The void value could possibly be represented by the Java `null` value, both being distinct from false. In the Scheme code, `if`, like `lambda`, is the name of a syntactic form. It tells the interpreter or compiler to read the rest of the list that it heads, in a particular way. On reading `if` here, the parse module will use the method `ParseIf`. `ParseIf` will determine the length

of the list. Three elements indicates the use of the first new rule, two the second. A different number will trigger an exception.

4.7 Definitions

A final class must be added to the heirs of `Expression`. There is as yet no mechanism for dealing with the binding of symbols to expression using the `define` macro. For this the class `Definition` shall be introduced (appendix C.1.3). `Definition` needs two pointers; an `Expression` pointer for the expression that is bound and a `String` pointer for the name under which the expression shall be bound.

4.8 The Parse

Using the grammar described, and the assortment of classes defined, the parse is relatively simply. It uses `ParseExpression`, a recursive method.

```
Expression ParseExpression(ListElt datum)throws Exception{
    if(symbol(datum))
       return new VarExp((Symbol)datum);
    if(literal(datum))
       return new LitExp((Literal)datum);
    if(pair(datum)){
      if(define(car(datum)))
         return
           (new Definition
     (new VarExp
               ((Symbol)car(cdr(datum))),
                ParseExpression(car(cdr(cdr(datum))))));
      if(lambda(car(datum)))
         return
           (new LambdaExp
             (ParseList(car(cdr(datum))),
              ParseList(cdr(cdr(datum)))));
      if(ifop(car(datum)))
         return ParseIf(datum);
      else
        return
          new AppExp(ParseExpression(car(datum)),
                     ParseList(cdr(datum)));
    }
    return null;
}
```

On encountering conditionals the auxiliary method `ParseIf` is called, as the syntax of conditionals does not conform to the syntax of simple expressions.

```
IfExp ParseIf(ListElt datum)throws Exception{
    int args = length(datum); //args including if-keyword.
    if(args<3 || args>4)
        throw (new BadSyntax("if", W.write(datum)));
    ListElt test, con, alt;
    test = car(cdr(datum));
    con = car(cdr(cdr(datum)));
    if(length(cdr(datum))==3)//ie alternative.
        alt = car(cdr(cdr(cdr(datum))));
    else alt = null;
    return new IfExp(ParseExpression(test),
                     ParseExpression(con),
                     ParseExpression(alt));
}
```

Another exception to the syntax of simple expressions is encountered in lambda expressions, both in their argument lists, and their bodies. The `ParseList` method will be used in both cases.

```
Expression[] ParseList(ListElt datum)throws Exception{
    Vector tmp = new Vector(5);
    ListElt p = datum;
    while(!nullp(p)){
        tmp.addElement(ParseExpression(car(p)));
        p = cdr(p);
    }
    return VecToArr(tmp);
}
```

Chapter 5

Generating AsmL

5.1 The Generator Module

The generator module defined in the file *AsmLDef.java* (appendix D) provides AsmL definitions that can be used together with the AsmL Scheme interpreter (appendix A), and its standard and library procedures. The generator module produces a textual, AsmL representation of a given Scheme expression structure, according to a fairly simple template. Its output can be divided in two. Firstly a `Define` statement for adding an associated pair to the global environment. Secondly, a new class definition for any lambda expressions.

5.2 Two Definitions

The two components of the generator's output correspond to the symbol-object binding in the global environment, and the creation of a procedure object, in accordance with the environment model for evaluation. The `AsmLDefs` class consists of, along with a number of static methods, two static character string variables; `GlobalDefs` and `ProcObjs`. Once execution has completed, `GlobalDefs` should contain a global definition, with the following form: `Global.Define(`*quoted string*`,` *ScmObj*`)`. The appearance of *ScmObj* depends upon the value specified by the original Scheme expression parsed. The scope of this project is here limited to lambda expressions and procedure applications, as the two together cover a suitably large portion of Scheme. The other string variable, `ProcObj`, should given the occurrence of a lambda expression, contain the code for a new class definition, an extension of the `ProcObj` class. This class should represent the semantics of the Scheme procedure, and have the following form:

```
class foo extends ProcObj
 Para = ["x"]
```

```
getPara()as Seq of String
 return Para

Body(e as Env) as ScmObj
 step
  .
  .
  .
 step
  return ScmObj
```

5.3 Adding Closures

The occurrence of a lambda expression in Scheme indicates the creation of a closure. When the method `addGlobalDef` is passed a definition whose value is a lambda expression, the method `addClosure` is called, with the appropriate expression as its argument, along with the name indicated by the definition's symbol. `addClosure` consists of two *for* loops, and it writes an AsmL class definition. The first loop assembles the parameter list, the second builds the sequence of expressions comprising the body of the procedure. Creating closures in the ASM is simply a case of constructing objects of this class, using the `new` constructor. Augmentation of the environment with a new closure is achieved using `Define`, as follows:

```
Global.Define("some_procedure", new some_procedure(Global))
```

5.4 Building Expressions

AsmL representations of the Scheme expressions are built using the method `BuildExp`. `BuildExp` is indirectly recursive in that it makes calls to methods that in turn call `BuildExp`, reflecting the recursive nature of Scheme expressions. The building of any expression is either referred to one of three methods (`If`, `BuildProcApp` or `addClosure`), or terminates in the case of a symbol or literal. The method returns an AsmL representation of the Scheme expression passed.

5.5 Building Conditionals

Compiling conditionals is facilitated with the `BuildIf` method. `BuildIf` makes two or three calls to `BuildExp`, depending on the presences of an alternative statement for test failure (that is the conditional returning false). The first statement, the conditional, is packed into the context of the `if` keyword, and braces. All test statements will be of type `ScmObj`, and the AsmL

`if` requires a statement of type `boolean`. The AsmL procedure `notFalse` (appendix A.3) solves this mismatch, taking as its argument a `ScmObj` instance and returning a `boolean`. `notFalse` tests for the occurrence of the type `False` (appendix A.2), which is a subclass of the class `ScmObj`. `False` is returned by the various AsmL procedures found in the AsmL Scheme interpreter, such as the equality operators `Gte`, `Lt` and `Eq`.

5.6 Building Procedure Applications

Since Scheme procedure applications are lists, headed by a procedure reference and followed by zero or more expressions, a procedure that traverses lists is needed. The method `BuildProcApp` has as its heart a loop. The input expression is traversed iteratively, and each occurrence of an expression is duly passed to `BuildExp`, and the results concatenated. The subsequent string is packed into a call to the AsmL procedure `Eval`.

5.7 AsmL Syntax

AsmL has very simple syntax. Block depth is determined by indentational depth. The character used for indentation is the *space*, *tab* not being permitted. This means that block depth must be taken into account on building AsmL procedures. The argument `blockDepth` seen in the various methods of `AsmLDefs` is an integer for just this purpose. `blockDepth` dictates the amount of spaces prefixed any line of AsmL. It is duly incremented or zeroed at the appropriate points in the data flow.

5.8 Return Statements

The last expression in any Scheme procedure is always a return statement. If this expression is some kind of branch then there will be multiple return expressions. The `boolean rtrn` is passed from method to method, to determine whether the `return` keyword should be prefixed the AsmL statement being generated.

5.9 Variable Reference

As explained Scheme is lexically scoped (see section 1.2). In the case of nested declarations shadowing outer declarations, the resulting AsmL code must obviously be unambiguous. Fortunately, this is never a problem, since the occurrence of a nested lambda expression will lead to the generation of a new, separate class in the AsmL code. Any variable references are always made in the context of the appropriate environment.

```
(define daft
  (lambda (n)
    (lambda (n)(take n 1))))
```

The above Scheme code will translate into the following AsmL code.

```
class daft extends ProcObj
 Para = ["n"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  step
   return new daft_lambda(e)

class daft_lambda extends ProcObj
 Para = ["n"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  step
   return Eval("take", [new Symbol("n") as ScmObj,
               new NumVal(1) as ScmObj], e)
```

Chapter 6

Running the ASM

6.1 The Final Stage

The final stage of the transition from Scheme source code to machine code can be achieved using Microsoft's Visual Studio. It would of course have been both interesting and challenging to have undertaken this step as part of this dissertation, perhaps by writing an ASM to Java compiler, and using a Java compiler to create Java byte code. This would have had the additional benefit of producing platform independent code. However, when such well-developed tools exist already, the payoff of such an undertaking seemed relatively small. Also, the scale of this project would have been significantly larger than now, and probably have exceeded the appropriate size for such a dissertation. This said, it should also be pointed out that a considerable amount of work was involved in learning a sufficient amount about AsmL to be able to code the interpreter module, so that this part of the project did not come without a cost.

6.2 Generating C++ and x86 binaries

To use the output of the Scheme-AsmL compiler, the generated AsmL code must be compiled, using the AsmL plug-in with Microsoft Visual Studio, together with the *Environ.AsmL* file. This is achieved using the crude cut-and-paste method. The generated AsmL code can simply be pasted into the *Environ.asml* file, and compiled in Visual Studio. Given a lambda expression, the generator will write a class definition. The keyword `define` will lead to the code necessary for adding an instance of this class to the global environment. The following example shows a simple Scheme procedure definition, and the corresponding AsmL code.

```
(define loop
 (lambda()
  (if #t (loop))))
```

```
Global.Define("loop", new loop(Global))

class loop extends ProcObj
 Para = []

 Body(e as Env) as ScmObj
  step
   if(notFalse(new Symbol("#t")))
    return Eval("loop", [], e)
```

6.3 Running the Environ interpreter

The following example is to show that the Environ module functions correctly with respect to the environment model. It is a simplified version of the bank example from *sicp*.

```
(define account
  (lambda(bal)
    (lambda(amnt)
      (if (gte bal amnt)
          (set! bal (take bal amnt)))
      bal)))
```

It leads to the following AsmL code.

```
     class account_lambda extends ProcObj
      Para = ["amnt"]

      Body(e as Env) as ScmObj
       step
        if(notFalse(Eval("gte",
                         [new Symbol("bal") as ScmObj,
                          new Symbol("amnt") as ScmObj],
                         e)))
          var wot = Eval("set!",
                         [new Symbol("bal") as ScmObj,
                          Eval("take",
                               [new Symbol("bal") as ScmObj,
                                new Symbol("amnt") as ScmObj],
                               e) as ScmObj],
                          e)
       step
        return Eval(new Symbol("bal"), e)
```

```
class account extends ProcObj
 Para = ["bal"]

 Body(e as Env) as ScmObj
   return new Account_lambda(e)
```

Staying with the bank analogy, a new account object can be added to the environment, as in the next example.

```
(define bill (account 300))
```

Which yields the following:

```
Global.Define("bill",
              Eval("account",
                   [new NumVal(300) as ScmObj], Global)
```

The following is an example of an interaction with PLT DrScheme interpreter, having made the above Scheme definitions, using the appropriate ">=" and "-" operators.

```
> (account 300)
#<procedure>
> (bill 100)
200
> (bill 100)
100
> (bill 100)
0
> (bill 100)
0
>
```

Finally, an example run of the Environ module, compiled together with the generated AsmL code shown above.

```
>[account, 300]
Account_lambda_010D1640
>[Bill, 100]
200
>[Bill, 100]
100
>[Bill, 100]
0
>[Bill, 100]
0
>
```

Appendix A

Environ

A.1 The Environment

```
class Env
 Ep as Env
 var Bindings as Map of String to ScmObj = {|->}

 Define(sym as String, val as ScmObj)
  Bindings(sym) := val

 Value(Sym as String) as ScmObj
  if(Boundp(Sym)) then return Bindings(Sym)
  elseif(Ep <> undef) then return Ep.Value(Sym)
  else return undef

 Boundp(Sym as String) as Boolean
  if (Bindings(Sym) <> undef) return true
  else return false

 shared ExtEnv(Para as Seq of String,
               Args as Seq of ScmObj,
               env as Env) as Env
  var l as Integer = length(Para)
  var i as Integer = 0
  e = new Env(env)
  step while i < l
   e.Define(Para(i), Args(i))
   i := i + 1
  step
   return e
```

A.2 The Scheme Object

```
class ScmObj

class False extends ScmObj

class Error extends ScmObj
 var msg as String

 getMsg() as String
  return msg

class Symbol extends ScmObj
 var ID as String

 id() as String
  return ID

class NumVal extends ScmObj
 var Val as Integer

 getVal() as Integer
  return Val

class ProcApp extends ScmObj
 Proc as String
 Args as Seq of ScmObj

 getProc() as String
  return Proc

 getArgs() as Seq of ScmObj
  return Args

abstract class ProcObj extends ScmObj
 abstract Ep as Env
 abstract getPara() as Seq of String
 abstract Body(e as Env) as ScmObj
```

A.3 Low level Procedures

This is the collection of AsmL procedures that are responsible for carrying out the operations used in the Scheme procedures.

```
notFalse(obj as ScmObj) as Boolean
 if(obj is False) return false
 else return true

SetB(e as Env, sym as String, val as ScmObj)
  step
   v as NumVal = val as NumVal
  step
   if(e.Boundp(sym))e.Bindings(sym) := val
   elseif(e.Ep <> undef)SetB(e.Ep, sym, val)

Plus(n as NumVal, m as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  return new NumVal(x + y)

Minus(n as NumVal, m as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  return new NumVal(x - y)

Divide(n as NumVal, m as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  return new NumVal(x/y)

Multiply(n as NumVal, m as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  return new NumVal(x * y)

Gte(n as NumVal, m as NumVal) as ScmObj
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  if(x >= y) return n
  else return new False()

Lt(n as NumVal, m as NumVal) as ScmObj
 var x as Integer = n.getVal() as Integer
```

```
 var y as Integer = m.getVal() as Integer
 step
  if(x < y) return n
  else return new False()

Eq(n as NumVal, m as NumVal) as ScmObj
 var x as Integer = n.getVal() as Integer
 var y as Integer = m.getVal() as Integer
 step
  if(x = y) return n
  else return new False()

Incr(n as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 step
  return new NumVal(x + 1)

Decr(n as NumVal) as NumVal
 var x as Integer = n.getVal() as Integer
 step
  return new NumVal(x - 1)
```

A.4 The Standard Procedures

This is the selection of first class objects available in the global environment.

```
class Terminate extends ProcObj
 Para = []
 getPara() as Seq of String
  return Para

 Body(e as Env)as ScmObj
  step
   Exit := true
  step
   return new NumVal(0)

class Addition extends ProcObj
 Para = ["n", "m"]

 getPara() as Seq of String
  return Para

 Body(e as Env) as ScmObj
```

```
  var m as NumVal
  var n as NumVal
  step
   m := e.Value("m") as NumVal
   n := e.Value("n") as NumVal
  step
   return Plus(m, n)

class Subtraction extends ProcObj
 Para = ["n", "m"]

 getPara() as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Minus(n, m)

class Division extends ProcObj
 Para = ["n", "m"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Divide(n, m)

class Multiplication extends ProcObj
 Para = ["n", "m"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Multiply(m, n)
```

```
class GTE extends ProcObj
 Para = ["n", "m"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Gte(n, m)

class EQ extends ProcObj
 Para = ["n", "m"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Eq(n, m)

class LT extends ProcObj
 Para = ["n", "m"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var m as NumVal = e.Value("m") as NumVal
  var n as NumVal = e.Value("n") as NumVal
  step
   return Lt(n, m)

class SetBang extends ProcObj
 Para = ["sym", "val"]

 getPara()as Seq of String
  return Para

 Body(e as Env) as ScmObj
  var val = e.Value("val") as NumVal
```

```
  var sym = e.Value("sym") as Symbol
  step
   SetB(e, sym.id(), val)
  step
   return val
```

A.5 The Interpreter

```
Eval(s as Symbol, e as Env) as ScmObj
 if(ErrorSignal = undef)
  return e.Value(s.id())
 else return ErrorSignal as ScmObj
/*-------------------------------------------------*/
Eval(proc as String,
     args as Seq of ScmObj,
     env as Env) as ScmObj
 var error as Error = ErrorSignal
 var RTN as ScmObj
 var Obj as ScmObj = env.Value(proc)
 var Args as Seq of ScmObj = []
 var Frame as Env
 var pro as ProcObj
 var Actu as Integer
 var Form as Integer
 var i as Integer = 0
 step
  if(ErrorSignal = undef)
   if(Obj <> undef)
    step
     if(Obj is ProcObj)
      step
       pro := Obj as ProcObj
      step
/*-----mustn't evaluate the first arg!-----*/
       if(pro is SetBang)
        i := 1
        Args := Args+[args(0)]
       Actu := length(args)
       Form := length(pro.getPara())
      step
       if(Actu = Form)
        step while i < Actu
         if(args(i) is Symbol)
```

```
          step
           var sym as Symbol = args(i) as Symbol
          step
           Args := Args+[env.Value(sym.id()) as ScmObj]
         else
          Args := Args+[args(i)]
         i := i + 1
        step
         if(pro is SetBang)
          Frame := ExtEnv(pro.getPara(), Args, env)
         else
          Frame := ExtEnv(pro.getPara(), Args, pro.Ep)
        step
         RTN := pro.Body(Frame)
        step
         if(RTN is Error) ErrorSignal := RTN as Error
         return RTN
       else return new Error(proc+": expects "+
                             Form+" argument, given "+
                             Actu)
     else return new Error("procedure application:
                           expected procedure,
                           given: "+proc)
   else return new Error("reference to undefined
                         identifier: "+proc)
  else return error as ScmObj
/*-------------------------------------------------*/
newProcApp(Sym as String,
           Args as Seq of NumVal) as ProcApp
 var i as Integer = 0
 var l as Integer = length(Args)
 var SArgs as Seq of ScmObj = []
 var o as ScmObj
 step
  step while i < l
   SArgs := SArgs + [Args(i) as ScmObj]
   i := i + 1
 step
  return new ProcApp(Sym, SArgs)
/*-------------------------------------------------*/
nextExp()as ProcApp
 var CMD as Seq of String
 var i as Integer = 1
 var nargs as Integer = 0
```

```
 var call as String = ""
 var args as Seq of ScmObj = []
 step
  write(">")
  CMD := readlnSeqOfString()
 step
  call := CMD(0)
  nargs := length(CMD)
 step
  step while i < nargs
   args := args+
           [new NumVal(asInteger(CMD(i))) as ScmObj]
   i := i + 1
 step
  return new ProcApp(call, args)
/*-----------------------------------------------

  -----------------------------------------------*/
var Exit as Boolean = false
var ErrorSignal as Error = undef
var Global = new Env(undef)
var PA as ProcApp
var b as NumVal
var v as ScmObj
initLib()
    Global.Define("exit", new Terminate(Global))
    Global.Define("add", new Addition(Global))
    Global.Define("take", new Subtraction(Global))
    Global.Define("times", new Multiplication(Global))
    Global.Define("div", new Division(Global))
    Global.Define("gte", new GTE(Global))
    Global.Define("lt", new LT(Global))
    Global.Define("eq", new EQ(Global))
    Global.Define("set!", new SetBang(Global))
initUsDef()
  step
   Global.Define("rmult", new Times(Global))
   Global.Define("tms", new limes(Global))
   Global.Define("rdivi", new rDivide(Global))
   Global.Define("account", new Account(Global))
  step
   Global.Define("Bill", Eval("account",
                              [new NumVal(300) as ScmObj],
                              Global))
```

```
   Global.Define("Ben", Eval("account",
                             [new NumVal(300) as ScmObj],
                             Global))
   Global.Define("Jill", Eval("account",
                              [new NumVal(300) as ScmObj],
                              Global))

run()
  step
   initLib()
  step
   initUsDef()
  step while not Exit
   step
     PA := nextExp() //Procedure Application
   step
     v := Eval(PA.getProc(), PA.getArgs(), Global)
   step
     if (v is Error)
      step
       e as Error = v as Error
      step
       writeln(e.getMsg())
      step
       ErrorSignal := undef
     else
      if(not(v is NumVal)) writeln(v)
      else
       step
        b := v as NumVal
       step
        writeln(b.getVal())
```

Appendix B

The List Parser

B.1 List Elements

B.1.1 ListEltD

The abstract *List Element* class.

```
abstract class ListEltD{}
```

B.1.2 Cons

The *Cons-pair* class.

```
class Cons extends ListEltD{

    private ListEltD Car;
    private ListEltD Cdr;
        /**
         * Constructs a Cons element.
         * @param car The ListEltD to occupy the Car field.
         * @param cdr The ListEltD to occupy the Cdr field.
         */
    Cons(ListEltD car, ListEltD cdr){
        Car = car;
        Cdr = cdr;
    }
        /**
         * Accessor
         */
    public ListEltD car(){
        return Car;
    }
        /**
```

```
         * Accessor
         */
    public ListEltD cdr(){
        return Cdr;
    }
}
```

B.1.3 Literal

The *literal* class.

```
class Literal extends ListEltD{
    private double v;

        /**
         * Constructs a Literal LstEltD element.
         * @param d The number.
         */
    Literal(double d){
        v = d;
    }
        /**
         * Accessor for v field
         *@return double, v.
         */
    double id(){
        return v;
    }
}
```

B.1.4 Symbol

The *symbol* class.

```
class Symbol extends ListEltD{
    private String id;

        /**
         * Constructs a Symbol LstEltD element.
         * @param s The symbol.
         */
    Symbol(String s){
        id = s;
    }
        /**
```

```
         * Accessor, for id.
         *@return String, id.
         */
    String id(){
        return id;
    }
}
```

B.2 ListToken

The *list token* class.

```
class ListToken{
    private double nval;
    private String sval;
    private boolean isNumber;
    private boolean start;
    private boolean stop;

        /**
         * Constructs a ListToken
         * @param d The number to be encapsuled.
         * The isNumber boolean is set to true.
         */
    ListToken(double d){
        start = false;
        stop = false;
        nval = d;
        isNumber = true;
        sval = "";
    }
        /**
         * Constructs a ListToken
         * @param s The String to be encapsuled.
         */
    ListToken(String s){
        start = false;
        stop = false;
        sval = s;
        isNumber = false;
    }
        /**
         * Constructs a ListToken
         * @param i The char to be encapsuled. A "(" or ")"
```

```
         */
    ListToken(int i){
        sval = "";
        isNumber = false;
        start = false;
        stop = false;
        if(i == '(')
            start=true;
        if(i == ')')
            stop=true;
    }
        /**
         * Accessor
         */
    double nval(){
        return nval;
    }
        /**
         * Accessor
         */
    String sval(){
        return sval;
    }
        /**
         * Predicate, true IFF number.
         */
    boolean isNumber(){
        return isNumber;
    }
        /**
         * Predicate, true IFF list start.
         */
    boolean isStart(){
        return start;
    }
        /**
         * Predicate, true IFF list end.
         */
    boolean isStop(){
        return stop;
    }
}
```

B.3 LReader

The *list reader* class.

```
public class LReader {
    private static Reader R;
    private static StreamTokenizer strTok;

        /**
         * Reads first element of the input list,
         * decides what to do.
         * @param r, Reader. The in-file.
         * @return ListEltD.
         * Returns a ListEltD representation of the
         * file.. hopefully.
         */
    public static ListEltD read(Reader r){
        R=r;
        strTok = new StreamTokenizer(R);
//quote != string delimiter in scheme
        strTok.wordChars(33, 39);
//allow diverse op-symbols
        strTok.wordChars(42,64);
        ListToken t = getNextToken();
        if(listStart(t))
            return readList();
        if(t.isNumber())
            return new Literal(t.nval());
        return new Symbol(t.sval());
    }

        /**
         * The main motor of the class.
         * Traverses the list recursively.
         * @return Cons, the ListEltD list object.
         */
    private static Cons readList(){
        ListToken t = getNextToken();
        if(listEnd(t))return new Cons(null, null);
        if(listStart(t))
         return new Cons(readList(), readList());
        if(isNumber(t))
            return new Cons(new Literal(t.nval()),
                            readList());
```

```
        return new Cons(new Symbol(t.sval()),
                        readList());
    }
        /**
         * A predicate.
         * Returns true IFF datum holds number.
         * @param datum A ListToken
         * @return boolean
         */
    private static boolean isNumber(ListToken datum){
        return (datum.isNumber());
    }
        /**
         * A predicate.
        * Returns true IFF datum is start of a list.
         * @param datum A ListToken
         * @return boolean
*/
    private static boolean listStart(ListToken datum){
        return (datum.isStart());
    }

        /**
         * A predicate.
* Returns true IFF datum is end of a list.
         * @param datum A ListToken
         * @return boolean
         */
    private static boolean listEnd(ListToken datum){
        return (datum.isStop());
    }

        /**
         * Returns result of call to tokenString.
         * @return ListToken
         * @exception ... needs attention
         */
    private static ListToken getNextToken(){
        int tok;
        try{
            tok = strTok.nextToken();
            if(tok == strTok.TT_EOF)return null;
            return tokenString(strTok, tok);
        }
```

```
        catch(Exception e){
            return null;
        }
    }

        /**
         * Builds and returns a new ListToken
         * @param tokenizer The input Stream
         * @param t The next token
         * @return ListToken
         */
    private static ListToken tokenString(StreamTokenizer
                                         tokenizer, int t){
        if(t=='-')return new ListToken(new String("-"));
        if(t==tokenizer.TT_WORD)
            return new ListToken(tokenizer.sval);
        if(t==tokenizer.TT_NUMBER)
            return new ListToken(tokenizer.nval);
        return new ListToken(t);
    }
}
```

Appendix C

The Scheme Parser

C.1 Expressions

C.1.1 ExpressionD

```
abstract class ExpressionD{
    abstract String getXML();
    abstract String id();
}
```

C.1.2 AppExp

Application Expressions.

```
class AppExp extends ExpressionD{
    private String xmlrep;
    private String ID;
    private ExpressionD RATOR;
    private ExpressionD RANDS[];
    private String vars[];

        /**
         * Constructs a AppExp object.
         * @param e The operator.
         * @param es The operands.
         * Constructor sets RATOR & RANDS
         * to e & es respectively. Also, all
         * free variables are identified
 * and added to $vars$.
         */
    AppExp(ExpressionD e, ExpressionD es[])throws Exception{
        if(e instanceof LitExp)
            throw new ExpectsProcedure("procedure application",
```

```
                                    (LitExp)e);
    Vector v = new Vector(5);
    RATOR = e;
    RANDS = es;
    ID = e.id();
    xmlrep = "<app>\n<rator>\n"+e.getXML()+"</rator>\n";
    for(int i = 0; i < es.length; i++){
        if(es[i] instanceof VarExp)
            if(!v.contains(es[i].id()))//not already found.
                v.addElement(es[i].id());//add to var list.
        if(es[i] instanceof AppExp){
            AppExp a = (AppExp)es[i];
            collectVars(v, a.getVars());
        }
        if(es[i] instanceof IfExp){
            IfExp a = (IfExp)es[i];
            collectVars(v, a.getVars());
        }
        xmlrep +="<rand no="+(i+1)+">\n"
        +es[i].getXML()+"</rand>\n";
    }
    addVars(v);
    xmlrep +="</app>\n";
}
    /**
     * @param v The Vector for collecting free variables.
     * @param vs The String array with all free variables.
     * Adds all free variables to Vector.
     */
void collectVars(Vector v, String[] vs){
    for(int i = 0; i < vs.length; i++)
        if(!v.contains(vs[i]))
            v.addElement(vs[i]);
}
    /**
     * @param v Vector containing
     * all free variables (no duplicates).
     * Initiates vars array.
     */
void addVars(Vector v){
    int s = v.size();
    vars = new String[s];
    for(int i = 0; i<s; i++)
        vars[i] = (String)v.elementAt(i);
```

```
    }
        /**
         * Accessor
         */
    ExpressionD getRator(){
        return RATOR;
    }
        /**
         * Accessor
         */
    ExpressionD[] getRands(){
        return RANDS;
    }
        /**
         * Accessor
         */
    String id(){
        return ID;
    }
        /**
         * Accessor
         */
    String[] getVars(){
        return vars;
    }
        /**
         * Accessor
         */
    String getXML(){
        return xmlrep;
    }
}
```

C.1.3 Definition

Definitions.

```
class Definition extends ExpressionD{
    private String key, xmlrep;
    private ExpressionD val;

        /**
         * Constructs a new Definition object.
```

```
         * @param id The Symbol underwhich
         * the expression will be bound.
         * @param e The expression to be bound.
         */
    Definition(ExpressionD id, ExpressionD e){
        xmlrep = new String("<define>\n");
        xmlrep += "<key>"+id.id()+"</key>\n";
        xmlrep += "<val>\n"+e.getXML()+"</val>\n</define>\n";
        key = id.id();
        val = e;
    }
        /**
         * Accessor
         * @returns String key.
         */
    String id(){
        return key;
    }
        /**
         * Accessor
         * @returns String xmlrep.
         */
    String getXML(){
        return xmlrep;
    }
        /**
         * Accessor
         * @returns String val.
         */
    ExpressionD getVal(){
        return val;
    }
}
```

C.1.4 IfExp

Conditional expressions.

```
class IfExp extends ExpressionD{
    private String xmlrep;
    private String ID;
    private ExpressionD TEST;
    private ExpressionD CON;
    private ExpressionD ALT;
```

```
private String vars[];
     /**
     * Constructs a IfExp object.
     * @param e1 The Test.
     * @param e2 The Consequence.
     * @param e3 The Alternative.
     */
IfExp(ExpressionD e1, ExpressionD e2, ExpressionD e3){
    Vector v = new Vector(5);
    TEST = e1;
    CON = e2;
    ALT = e3;
    xmlrep = "<if>\n<if-test>\n"+
             TEST.getXML()+"</if-test>\n";
    xmlrep +="<if-con>\n"+CON.getXML()+"</if-con>\n";
    if(ALT!=null)xmlrep+="<if-alt>\n"+
                         ALT.getXML()+"</if-alt>\n";
    xmlrep += "</if>\n";
    ID = TEST.id();
    initVars(v, e1);
    initVars(v, e2);
    if(e3 != null)initVars(v, e3);
    addVars(v);
}
     /**
     * Uses collectVars to collect free variables.
     * @param v Temporary holder for free variables.
     * @param e The Expression where the vars are found.
     */
private void initVars(Vector v, ExpressionD e){
    if(e instanceof VarExp)
        if(!v.contains(e.id()))
            v.addElement(e.id());
    if(e instanceof AppExp){
        AppExp a = (AppExp)e;
        collectVars(v, a.getVars());
    }
    if(e instanceof IfExp){
        IfExp a = (IfExp)e;
        collectVars(v, a.getVars());
    }
}
     /**
     * Adds Strings in vs to Vector v. No duplicates.
```

```
     * @param v Vector for collecting Strings.
     * @param vs String array.
     */
 void collectVars(Vector v, String[] vs){
    for(int i = 0; i < vs.length; i++)
        if(!v.contains(vs[i]))
            v.addElement(vs[i]);
 }
    /**
     * Initiates global vars.
     * @param v Vector (of Strings).
     */
 void addVars(Vector v){
    int s = v.size();
    vars = new String[s];
    for(int i = 0; i<s; i++)
        vars[i] = (String)v.elementAt(i);
 }
    /**
     * Accessor
     *@return String, the ID field.
     */
 String id(){
    return ID;
 }
    /**
     * Accessor
     *@return String, the xmlrep field.
     */
 String getXML(){
    return xmlrep;
 }
    /**
     * Accessor
     *@return String, the TEST field.
     */
 ExpressionD getTest(){
    return TEST;
}
    /**
     * Accessor
     *@return ExpressionD, the CON field.
     */
 ExpressionD getCon(){
```

```
        return CON;
    }
        /**
         * Accessor
         *@return ExpressionD, the ALT field.
         */
    ExpressionD getAlt(){
        return ALT;
    }
        /**
         * Accessor
         *@return String[], the vars array field.
         */
    String[] getVars(){
        return vars;
    }
}
```

C.1.5 LambdaExp

Lambda expressions.

```
class LambdaExp extends ExpressionD{
    private String ID, xmlrep;
    private ExpressionD ARGS[];
    private ExpressionD BODY[];
    private String vars[];
         /**
         * Constructs a LambdaExp object.
         * @param es ExpressionD array.
 * The formal parameters.
         * @param es2 Expressiond array.
         * The sequence of expressions.
         */
    LambdaExp(ExpressionD es[],
              ExpressionD[] es2)throws Exception{
        Vector v = new Vector(5);
        xmlrep = new String("<lambda>\n");
        for(int i = 0; i<es.length;i++){
            ExpressionD t = es[i];
            if(!(t instanceof VarExp))
                throw(new
                    ExpectsVarExp("Lambda", es[i]));
```

```
            xmlrep += "<arg no="+(i+1)+">"+
                      es[i].getXML()+"</arg>\n";
        }
        xmlrep += "<body>\n";
        for(int i = 0; i<es2.length; i++){
            xmlrep += "<exp no="+(i+1)+">"+
                      es2[i].getXML()+"</arg>\n";
            if(es2[i] instanceof VarExp)
                if(!v.contains(es2[i].id()))
                    v.addElement(es2[i].id());
            if(es2[i] instanceof AppExp){
                AppExp a = (AppExp)es2[i];
                collectVars(v, a.getVars());
            }
            if(es2[i] instanceof IfExp){
                IfExp a = (IfExp)es2[i];
                collectVars(v, a.getVars());
            }
        }
        addVars(v);
        xmlrep += "</body>\n</lambda>\n";
        ID = "lambda";
        ARGS = es;
        BODY = es2;
    }
        /**
         * Adds Strings in vs to Vector v.
    * No duplicates.
         * @param v Vector for collecting Strings.
         * @param vs String array.
         */
    void collectVars(Vector v, String[] vs){
        for(int i = 0; i < vs.length; i++)
            if(!v.contains(vs[i]))
                v.addElement(vs[i]);
    }
        /**
         * Initiates global vars.
         * @param v Vector of Strings.
         */
    void addVars(Vector v){
        int s = v.size();
        vars = new String[s];
        for(int i = 0; i<s; i++)
```

```
            vars[i] = (String)v.elementAt(i);
    }
        /**
         * Accessor for vars array field.
         *@return String[], vars array field.
         */
    String[] getVars(){
        return vars;
    }
        /**
         * Accessor for id field.
         *@return String, ID.
         */
    String id(){
        return ID;
    }
        /**
         * Accessor for ARGS array field.
         *@return ExpressionD[], ARGS array field.
         */
    ExpressionD[] getArgs(){
        return ARGS;
    }
        /**
         * Accessor for xmlrep field.
         *@return String, xmlrep.
         */
    String getXML(){
        return xmlrep;
    }
        /**
         * Accessor for BODY array field.
         *@return ExpressionD[], BODY array field.
         */
    ExpressionD[] getBody(){
        return BODY;
    }
}
```

C.1.6 LitExp

Literal expressions.

```
class LitExp extends ExpressionD{
```

```
    private int val;
    private String ID, xmlrep;
         /**
         * Constructs a LitExp object.
         * @param l Literal
         */
    LitExp(Literal l){
        val = (int)l.id();
        ID = Integer.toString(val);
        xmlrep = "<LitExp>"+
                 Integer.toString(val)+"</LitExp>\n";
    }
        /*
         * Gives a String representation of
 * the int val.
         *@return String, Integer.toString(val);
         */
    String getVal(){
        return Integer.toString(val);
    }
        /*
         * Gives a String representation of
         * the int val.
         *@return String, Integer.toString(val);
         */
    String id(){
        return Integer.toString(val);
    }
        /*
         * Accessor for xmlrep field.
         *@return String, xmlrep field.
         */
    String getXML(){
        return xmlrep;
    }
}
```

C.1.7 VarExp

Variable expressions.

```
class VarExp extends ExpressionD{
    private String ID, xmlrep;
        /**
```

```
         * Constructs a VarExp object.
         * @param s Symbol
         */
    VarExp(Symbol s){
        ID = s.id();
        xmlrep = "<var>"+s.id()+"</var>\n";
    }
        /*
         * Constructs a VarExp object
         * from a String.
         * @param s, the name.
         */
    VarExp(String s){
        ID = s;
    }
        /*
         *Accessor, get ID.
         *@return String, ID.
         */
    String id(){
        return ID;
    }
        /*
         *Accessor, get xmlrep.
         *@return String, xmlrep.
         */
    String getXML(){
        return xmlrep;
    }
}
```

C.2 JScheme

A minimal Scheme library.

```
public class JScheme{
   /**
    *@param datum A ListEltD
    *@return boolean
    *A predicate. Returns true IFF given a ListEltD
    *representation of Scheme's empty list.
    */
    public static boolean nullp(ListEltD datum){
        if(datum instanceof Symbol ||
```

```
        datum instanceof Literal)
         return false;
      Cons c = (Cons)datum;
      if(c.car()==null)return true;
      return false;
  }

/**
  * @param datum A ListEltD
  * @return ListEltD
  * @exception ExpectsPair
  * If datum is of type Cons,
  * then datum's car field is returned.
  */
  public static ListEltD car(ListEltD datum)
                          throws Exception{
      if(datum instanceof Symbol ||
         datum instanceof Literal)
          throw (new ExpectsPair("car",
                                 (Symbol)datum));
      Cons c = (Cons)datum;
      return c.car();
  }

/**
  * @param datum A ListEltD
  * @return ListEltD
  * @exception ExpectsPair
  * If datum is of type Cons,
  * then datum's cdr field is returned.
  */
  public static ListEltD cdr(ListEltD datum)
                          throws Exception{
      if(datum instanceof Symbol ||
         datum instanceof Literal)
          throw (new ExpectsPair("cdr", (Symbol)datum));
      Cons c = (Cons)datum;
      return c.cdr();
  }
/**
  * @param datum A ListEltD
  * @return boolean
  * A predicate. Returns true IFF datum
  * is an instanceof Symbol
```

```
  */
  public static boolean symbol(ListEltD datum){
      return (datum instanceof Symbol);
  }

/**
  * @param datum A ListEltD
  * @return boolean
  * A predicate. Returns true IFF datum
  * is an instanceof Literal
  */
  public static boolean literal(ListEltD datum){
      return (datum instanceof Literal);
  }

/**
  * @param datum A ListEltD
  * @return boolean
  * A predicate. Returns true IFF datum
  * is an instanceof Cons AND datum's car != null
  */
  public static boolean pair(ListEltD datum){
      if(datum instanceof Cons){
          Cons c = (Cons)datum;
          return (c.car()!=null);
      }
      return false;
  }
/**
  * @param datum A ListEltD
  * @return int
  * @exception ExpectsPair
  * Returns the length of the list passed.
  */
  public static int length(ListEltD datum)
                     throws Exception{
      if(!(datum instanceof Cons))
          throw (new ExpectsPair("length",
                                 (Symbol)datum));
      int i = 0;
      for(;car(datum)!=null;i++)
          datum = cdr(datum);
      return i;
  }
```

```
  /**
    * @param datum A ListEltD
    * @return boolean
    * A predicate. Returns true IFF given a ListEltD
    * containing the word "define".
    */
    public static boolean define(ListEltD datum){
        return compareTo(datum, "define");
    }
  /**
    * @param datum A ListEltD
    * @return boolean
    * A predicate. Returns true IFF given a ListEltD
    * containing the word "lambda".
    */
    public static boolean lambda(ListEltD datum){
        return compareTo(datum, "lambda");
    }
  /**
    * @param datum A ListEltD
    * @return boolean
    * A predicate. Returns true IFF given a ListEltD
    * containing the word "if".
    */
    public static boolean ifop(ListEltD datum){
        return compareTo(datum, "if");
    }
    private static boolean compareTo(ListEltD datum,
                                     String target){
        Symbol sm;
        String s;
        if(symbol(datum)){
            sm = (Symbol)datum;
            s = sm.id();
            return s.compareTo(target) == 0;
        }
        return false;
    }
}
```

C.3 ScmParse

The Scheme parser.

```
public class ScmParse extends JScheme{
        /**
         * @param datum The ListEltD
         * to be parsed.
         * Determines what sort of ListEltD
         * is passed, acts accordingly.
         */
    static ExpressionD ParseExpression(ListEltD datum)
                        throws Exception{
        if(symbol(datum))
            return new VarExp((Symbol)datum);
        if(literal(datum))
            return new LitExp((Literal)datum);
        if(pair(datum)){
            if(define(car(datum)))
             return
              new Definition(new VarExp((Symbol)
                                    car(cdr(datum))),
                        ParseExpression(car(cdr(cdr(datum)))));
            if(lambda(car(datum)))
                return
                 new LambdaExp(ParseList(car(cdr(datum))),
                            ParseList(cdr(cdr(datum))));
            if(ifop(car(datum)))
                return ParseIf(datum);
            else
                return
                    new AppExp(ParseExpression(car(datum)),
                            ParseList(cdr(datum)));
        }
        return null;
    }
        /**
         * @param datum The ListEltD to be parsed.
         * Assembles IfExps, in case of conditionals.
         */
    static IfExp ParseIf(ListEltD datum)throws Exception{
        int args = length(datum); //args including if-keyword.
        if(args<3 || args>4)
            throw (new BadSyntax("if", LWriter.write(datum)));
        ListEltD test, con, alt;
        test = car(cdr(datum));
        con = car(cdr(cdr(datum)));
        if(length(cdr(datum))==3)//ie alternative.
```

```
            alt = car(cdr(cdr(cdr(datum))));
        else alt = null;
        return new IfExp(ParseExpression(test),
                         ParseExpression(con),
                         ParseExpression(alt));
    }
        /**
         * @param datum The ListEltD.
         * Constructs sequences or Expressions.
         * Start off by using a vector, but convert to an array
         * using the VecToArr method under.
         */
    static ExpressionD[] ParseList(ListEltD datum)
                          throws Exception{
        Vector tmp = new Vector(5);
        ListEltD p = datum;
        while(!nullp(p)){
            tmp.addElement(ParseExpression(car(p)));
            p = cdr(p);
        }
        return VecToArr(tmp);
    }
        /**
         * @param v Vector
         * Converts Vector of Expressions to array.
         */
    static ExpressionD[] VecToArr(Vector v){
        ExpressionD a[] = new ExpressionD[v.size()];
        for(int i = 0; i < v.size(); i++)
            a[i]=(ExpressionD)v.elementAt(i);
        return a;
    }
    static String getXML(ExpressionD e){
        return e.getXML();
    }
    static String ExpToString(ExpressionD e){
        return e.id();
    }
}
```

C.4 Exceptions

C.4.1 ParseException

The abstract Parse Exception class.

```
abstract class ParseException extends Exception{
}
```

C.4.2 BadSyntax

```
class BadSyntax extends ParseException{
    private String keyword;
    private String sign;
        /**
         * Constructs a BadSyntax object.
         * @param key The keyword.
         * @param s The code where the error is found.
         */
    BadSyntax(String key, String s){
        keyword = key;
        sign = s;
    }
        /**
         *Constructs and returns error message.
         *@return String
         */
    public String getMessage(){
        return keyword+": bad syntax in: "+sign;
    }
}
```

C.4.3 ExpectsPair

```
class ExpectsPair extends ParseException{
    private String caller;
    private String id;
        /**
         * Constructs a ExpectsPair object.
         * @param s String, the method that expected a Cons.
         * @param l Symbol, the unexpected ListEltD.
         */
    ExpectsPair(String s, Symbol l){
        caller = s;
        id = l.id();
```

```
    }
        /**
         * Constructs a ExpectsPair object.
         * @param s String, the method that expected a Cons.
         * @param l Literal, the unexpected ListEltD.
         */
    ExpectsPair(String s, Literal l){
        caller = s;
        id += l.id();
    }
        /**
         * Error report.
         *@return String, the Error message.
         */
    public String getMessage(){
        return caller+" expects Pair, given "+id;
    }
}
```

C.4.4 ExpectsProcedure

```
class ExpectsProcedure extends ParseException{
    private String caller;
    LitExp le;
        /**
         * Construct ExpectsProcedure exception.
         *@param s String, error message.
         *@param l Literal, the offending object.
         */
    ExpectsProcedure(String s, LitExp l){
        caller = s;
        le = l;
    }
        /**
         *@return String, the error repport.
         */
    public String getMessage(){
        return caller+": expects procedure, given:"+le.id();
    }
}
```

C.4.5 ExpectsSymbol

```
class ExpectsSymbol extends ParseException{
```

```
    private String caller;
    Symbol le;

    ExpectsSymbol(String s, Symbol l){
        caller = s;
        le = l;
    }
    public String getMessage(){
        return caller+" expects Pair,\ngiven Symbol "+le.id();
    }
}
```

C.4.6 ExpectsVarExp

```
class ExpectsVarExp extends ParseException{
    private String caller;
    ExpressionD le;
        /**
         * Constructs ExpectsVarExp exception.
         *@param s String, the expecter.
         *@param l ExpressionD, the expectee.
         */
    ExpectsVarExp(String s, ExpressionD l){
        caller = s;
        le = l;
    }
        /**
         *@return String, the error report.
         */
    public String getMessage(){
        return caller+
               " expects Variable Expression,\ngiven "+le.id();
    }
}
```

Appendix D

The AsmL Generator

D.1 AsmLDefs

The AsmL code generator.

```
class AsmLDefs{
    private static String GlobalDefs = "";
    private static String ProcObjs = "";
    private static final String getPara =
    "\n      getPara()as Seq of String\n       return Para\n";

        /**
         * Returns a textual AsmL representation
         * of a Definition structure.
         * Entry point for building definitions.
         *@param e A Definition
         *@return AsmL String
         */
    static String addGlobalDef(Definition e){
        String name = new String(e.id());
        String s = new String(indent(5)+
                    "Global.Define("+(char)34
                     +name+(char)34)+", ";
        if(Lambdap(e.getVal())){
            s += "new "+name+"(Global))\n";
            addClosure((LambdaExp)e.getVal(), name);
        }
        if(Appp(e.getVal()))
            s += BuildProcApp((AppExp)e.getVal(), false, 6);
        GlobalDefs += s;
        return GlobalDefs+ProcObjs;
    }
```

```
    /**
     * Determines ExpressionD type.
     * Calls appropriate method.
     * Returns a textual AsmL representation
     * of an ExpressionD
     *@param e The ExpressionD
     *@param name String, name under which
     * ExpressionD is bound.
     *@param rtrn boolean, whether expression
     * is return statement.
     *@param blockDepth int, depth of indentation.
     * Indentation is
     * part of AsmL syntax.
     *@return AsmL String
     */
private static String BuildExp(ExpressionD e,
                               String name, boolean
                               rtrn, int blockDepth){
    String r= new String(indent(blockDepth));
    if(rtrn)r+="return ";
    if(Ifp(e))
        return If((IfExp)e, rtrn, blockDepth);
    if(Appp(e))
        return BuildProcApp((AppExp)e, rtrn, blockDepth);
    if(Lambdap(e)){
        addClosure((LambdaExp)e, name+"_lambda");
        return r+"new "+name+"_lambda(e)";
    }
    if(Litp(e))return r+"new NumVal("+e.id()+")";
    else{
        String s="new Symbol("+(char)34+
                  e.id()+(char)34+")";
        if(rtrn)return r+"Eval("+s+", e)";
        return r+s;
    }
}
    /**
     * Concatinates global String ProcObjs
     * with a new lambda
     * espression String.
     *@param e The LambdaExp
     *@param name String, name under which
     * ExpressionD is bound.
     */
```

```
private static void addClosure(LambdaExp e, String name){
    boolean rtrn = false;
    String[] vars = e.getVars();
    String title = new String(
                   "\n     class "+name+
                   " extends ProcObj\n");
    String para = new String("      Para = [");
    String body = new String(
                  "\n      Body(e as Env) as ScmObj\n");
    ExpressionD[] prms = e.getArgs();
    ExpressionD[] bdy = e.getBody();
    for(int j=0; j<prms.length; j++){
        para+=(char)34+prms[j].id()+(char)34;
        if(j<prms.length-1)para+=", ";
    }
    para+="]\n";
    for(int j=0; j<bdy.length; j++){
        body+=new String(indent(7)+"step\n");
        // Flag for return statement!!
        if(j==bdy.length-1)rtrn=true;
        body+=BuildExp(bdy[j], name, rtrn, 8)+"\n";
    }
    ProcObjs+=title+para+body;
}
    /**
     * Builds and returns an AsmL
     * application expression String.
     *@param e The AppExp Application Expression.
     *@param rtrn boolean, whether expression
     * is return statement.
     *@param blockDepth int, depth of indentation.
     * Indentation is
     * part of AsmL syntax.
     *@return AsmL String
     */
private static String BuildProcApp(AppExp e,
                                   boolean rtrn,
                                   int blockDepth){
    String r = indent(blockDepth);
    String app, p=e.id();
    boolean asgn = p.compareTo("set!")==0?true:false;
    ExpressionD es[] = e.getRands();
    int n = es.length;
```

```
    if(rtrn)r+="return ";
    app=new String("Eval("+(char)34+p+(char)34+", [");
    //AsmL requires return vals captured!
    if(asgn)app="var wot = "+app;
    for(int j=0; j<n; j++){
        //surfix -as ScmObj-
        app+=BuildExp(es[j], null, false, 0);
        app+=" as ScmObj";
        if(j<n-1)app+=", ";
    }
    app+="], e)";
    return r+app; //in case of it being return statement.
}
    /**
     * Builds and returns an AsmL conditional
     * expression String.
     *@param e The IfExp Conditional Expression.
     *@param rtrn boolean, whether expression
     * is return statement.
     *@param blockDepth int, depth of indentation.
     * Indentation is
     * part of AsmL syntax.
     *@return AsmL String
     */
private static String If(IfExp e,
                         boolean rtrn,
                         int blockDepth){
    String s = new String(indent(blockDepth)+
                          "if(notFalse(");
    s += BuildExp(e.getTest(), null, false, 0)+"))\n";
    s += BuildExp(e.getCon(), null, rtrn, blockDepth+1)+"";
    if(e.getAlt() != null)
        s+="\n"+indent(blockDepth)+
        "else\n"+
         BuildExp(e.getAlt(), null, rtrn, blockDepth+1);
    return s;
}
    /**
     *A predicate, true IFF e is Defintion
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Definep(ExpressionD e){
    return (e instanceof Definition);
```

```
}
    /**
     *A predicate, true IFF e is LambdaExp
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Lambdap(ExpressionD e){
    return (e instanceof LambdaExp);
}
    /**
     *A predicate, true IFF e is IfExp
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Ifp(ExpressionD e){
    return (e instanceof IfExp);
}
    /**
     *A predicate, true IFF e is AppExp
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Appp(ExpressionD e){
    return (e instanceof AppExp);
}
    /**
     *A predicate, true IFF e is a VarExp
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Varp(ExpressionD e){
    return (e instanceof VarExp);
}
    /**
     *A predicate, true IFF e is LitExp
     *@param e ExpressionD
     *@return boolean
     */
private static boolean Litp(ExpressionD e){
    return (e instanceof LitExp);
}
private static String indent(int n){
    String idnt = "";
    for(int i=0;i<n;i++)idnt+=" ";
```

```
            return idnt;
        }
    }
```

Appendix E

Schasm

E.1 Scheme to AsmL compiler

This class coordinates the different components of the compiler.

```
public class Schasm{
    FileReader FR;
    FileWriter FW;
    ListEltD L;
    ExpressionD E;

        /**
         * Constructs a Schasm
         * @param s The name of the source file.
         * @param t The name of the target file.
         *
         */
    Schasm(String s, String t)throws Exception{
        FR = new FileReader(s);
        FW = new FileWriter(t);
    }
        /**
         * Compiles from FR to FW
         *
         */
    public void compile()throws Exception{
        L = LReader.read(FR);
        try{
            E = ScmParse.ParseExpression(L);
            FW.write(AsmLDefs.addGlobalDef((Definition)E));
        }
        catch(ParseException e){
```

```
            System.out.println(e.getMessage());
        }
        catch(Exception e){
            System.out.println("Exception:"+e+"\n");
        }
    }
        /**
         *Point of entry for whole Shebang.
         */
    public static void main(String prms[]){
        try{
            Schasm S = new Schasm("rec/scm_exp.scm",
                                  "rec/scm_exp.asml");
            S.compile();
            S.FR.close();
            S.FW.close();
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

Appendix F

Java Class Hierarchy

F.1 Class Hierarchy

```
class java.lang.Object
   class AsmLDefs
   class ExpressionD
      class AppExp
      class Definition
      class IfExp
      class LambdaExp
      class LitExp
      class VarExp
   class JScheme
      class ScmParse
   class ListEltD
      class Cons
      class Literal
      class Symbol
   class ListToken
   class LReader
   class LWriter
   class Schasm
   class java.lang.Throwable (implements java.io.Serializable)
      class java.lang.Exception
         class ParseException
         class BadSyntax
         class ExpectsPair
         class ExpectsProcedure
         class ExpectsSymbol
         class ExpectsVarExp
```

References

Various sources have been used in this project. References were needed for three topics; Scheme, parsing, and Abstract State Machines. The source most referred to for a description of Scheme was *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, and Sussman. Chapter three (*Modularity, Objects, and State*), particularly the section titled *The Environment Model of Evaluation*, are central to this project. The model for the semantics of Scheme procedures and their evaluation used here is taken from this source. An on-line version of the book is available at http://mitpress.mit.edu/sicp/. The other major sources used as Scheme references are the *Revised(5) Report on the Algorithmic Language Scheme*, available at http://www.swiss.ai.mit.edu/, as well as the *MIT Scheme User's Manual*, published under the same domain. The approach to parsing, together with the grammar for Scheme are taken from *Essentials of Programming Languages - Second Edition*, by Friedman, Wand, and Haynes. As regards literature on Abstract State Machines, it seems that the web is the main repository. Links to all relevant material are found at http://www.eecs.umich.edu/gasm/. As well as the introduction and the various tutorials found at this site, the documentation, tutorials and examples that accompany Microsoft AsmL have been helpful. As mentioned in the main text however, the documentation available is fairly limited. The AsmL documentation is packaged together with the AsmL down-load located at http://www.research.microsoft.com/foundations/asml/.

- Abelson, Harold, Gerald Jay Sussman and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs*, MIT Press.
- Friedman, Daniel P., Mitchell Wand, Christopher Thomas Haynes. 2001. *Essentials of Programming Languages - Second Edition*, MIT Press.
- *Revised(5) Report on the Algorithmic Language Scheme*, http://www.swiss.ai.mit.edu/

- *MIT Scheme User's Manual*,
  http://www.swiss.ai.mit.edu/
- *Abstract State Machines - A Formal Method for Specification and Verification*,
  http://www.eecs.umich.edu/gasm/
- *Microsoft Research - AsmL*,
  http://www.research.microsoft.com/foundations/asml/