# A Simple Scheme Machine

Ben Lipton

May 4, 2012

**Abstract**

This project is a processor design based on the Scheme programming language. The processor accepts input in a linked-list binary format conceptually similar to the list data structure used extensively in Scheme. It recursively descends into this data structure, evaluating subprograms until it can compute the value of the program as a whole. Also presented is a compiler from Scheme to this binary format, written in Scheme (and, to some extent, able to run on the processor!). The resulting hardware is quite simple, and is capable of running a wide subset of Scheme software with the only problem being its tendency to run out of memory due to lack of a garbage collector.

## 1 Introduction

This project introduces a nontraditional processor design for execution of programs written in the Scheme programming language. Scheme is a variant of Lisp, a language that has been used extensively in the artificial intelligence community, and is also the core language in which the popular Emacs text editor is built [5]. The hardware of this processor is closely based on the design described in [8], and is implemented in the Verilog hardware description language. In addition to the processor implementation, this project implements a compiler, written in Scheme, which converts Scheme programs into the linked-list-oriented format understood by the processor. This compiler allows complex programs to be run on the processor without manual translation, and also, by virtue of being written in Scheme, provides an interesting test case to verify the correct operation of the processor.

There would seem to be little market for a processor that operates only on Scheme's unusual data model, when standard processors are more than capable of executing (compiled) Scheme code and many other languages besides. However, this processor has some design features, like its organizational simplicity and its unusual use of linked data structures, that make it an interesting point in the design space to explore. The goal of this project was to verify the feasibility of creating such a processor, to extend it with features that permit execution of "useful" programs, and to evaluate its performance and complexity. It is not expected that its performance will rival that of commercial processors, especially since a fair comparison will be difficult to establish due to the Scheme processor's unconventional organization.

In addition to the stated goals, it is important to evaluate the external environmental and cultural impact of any project. Though the technology sector as a whole has a significant environmental impact, the effect of the introduction of a new processor technology is generally negligible, and the unusual nature of this processor makes a large rate of adoption unlikely. So the main impact, if any, is the potential for giving engineers new ideas about computer organization that may lead to other innovations in the future. Even though the main idea behind this design comes from a paper published in 1979, with the growing need for low-power processors, a new and very simple architecture design might turn out to be quite useful.

This report begins by discussing the inspriation of the project, two previous hardware designs based around Lisp-like languages. Section 3 contains a brief introduction to important concepts in Scheme. Section 4 describes the design of the hardware, and Section 5 describes the implementation of the compiler that supports it. Finally, Section 6 outlines the procedures that are used for testing the processor, and provides some of the test results.

## 2 Related Work

In the late 70s and the 80s, Lisp was a very popular language for artificial intelligence research, but because of its peculiar demands, its performance on mainframes of the day was not very good. From this arose the Lisp Machine, a single-user computer with a processor designed specifically for Lisp code. One such processor was CADR [6], developed at MIT. CADR is stack based rather than having a large register file, but its method of program execution differs very little from other processors. Lisp code would be compiled to a series of instructions, which could then execute on hardware optimized for the types of access patterns common in Lisp programs.

A more unusual approach to executing Lisp code is that of SIMPLE [8]. This approach uses Lisp, or at least the linked data structures of a Lisp program, as a machine language, executing programs directly without changing their structure into a list of sequential instructions. The machine described uses five registers to keep track of the current state of evaluation, and a state machine to update the registers. It begins by attempting to evaluate the top level of the program's structure (the outermost set of parentheses). As it encounters function calls it recursively computes the arguments to the function calls, then applies the function to them. Like CADR, this processor maintains a stack, which it uses to keep track of the previous calls in its recursion. This register machine accesses memory through a "storage manager," which allows memory to be manipulated in the form of lists by implementing the Lisp `car`, `cdr`, and `cons` functions.

The hardware design of the current project is greatly inspired by the machine described in [8]. The recursive evaluation algorithm, the uses of the registers, and the idea of an interface module that abstracts memory accesses into Lisp functions are all used as described. However, the implementation in the current project is entirely new, and is done as a simple Verilog state machine, for easier development and modification than the bus-based register machine whose details, including its full-custom silicon layout, are described in the paper. Furthermore, additional features have been added to the language supported by the new implmenentation, such as arithmetic via an ALU and the ability to retrieve the binary type of an object within a program. These features allow the current processor to support a larger subset of the target language, and to do so more faithfully to the language specification.

## 3 Background

This section introduces some features of Scheme that are important to the functioning of this processor.

### 3.1 Syntax

One of the most noticeable features of Scheme is its syntax, which consists of many nested pairs of parentheses. Each pair of parentheses represents a procedure call, or "application," where the first object inside the parentheses is the procedure being called, and the other objects are the arguments being passed to the procedure. Often, as shown in Figure 1, an argument or even the procedure will consist of another pair of parentheses, which is a subprogram that needs to be evaluated before the outer procedure call can take place. This nested structure means that it makes the most sense to evaluate the program recursively, at each step evaluating the arguments to the function first, then applying the function to those values.

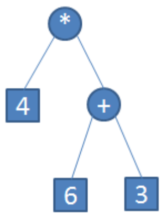
### 3.2 Lists

The recursive, highly nested nature of Scheme's syntax also applies to the favored data structure in the language. Most data in Scheme programs is stored in structures called lists. These are linked lists, consisting of nodes with two fields, named *car* and *cdr*. The nodes themselves are referred to as pairs, and are created using the `cons` procedure. In a standard list, the car of each pair points to an object in the list (which may be of any type) and the cdr of the pair points to the rest of the list, which may be either a pair, to continue the list, or a special "empty list" object, written `'()`.

Significantly, it is also possible to represent Scheme programs in list format. Any Scheme program can be understood to represent a linked list, some of whose items are lists themselves, and so on, where each pair of parentheses represents a list being stored within a list. It is on this linked-list representation of the program that the processor operates,

```
#!r6rs
(import (rnrs))
(* 4 (+ 3 6))
```

(a) An example program in the R6RS [7] Scheme dialect



(b) The tree structure represented by this program

Figure 1: Tree structure of a Scheme program

walking through the list to evaluate all the arguments one by one, and stepping downward into sublists when the value of one of these is needed.

## 4 Hardware Design

The funcitonality of the processor is divided into several components, as shown in Figure 2. The main operation of the processor is orchestrated by the controller, which keeps track of the current point in the evaluation of a Scheme expression, and determines what operation needs to be performed next. The RAM interface provides high-level access methods for RAM, allowing the controller to request the car or cdr of a pair, or to automatically allocate a new pair by specifying both parts. Additionally, the processor supports arithmetic and logic computations using an ALU module. Section 4.1 describes the way Scheme expressions are represented in bits in the memory of the processor. Sections 4.2 and 4.4 discuss the implementation details of these modules.

### 4.1 Memory Representation

The data representation used by the processor is the typed-pointer representation described in [8]. All objects in the processor's data model are represented by 17-bit words, which contain a 3-bit type field, followed by a 14-bit data field. Most data types (including symbols, lists, procedures, and procedure applications) are represented by pairs; in these cases the type field holds the type of the object, and the data field holds a memory address that can be passed to the RAM interface to retrieve the car or cdr of the pair. There are

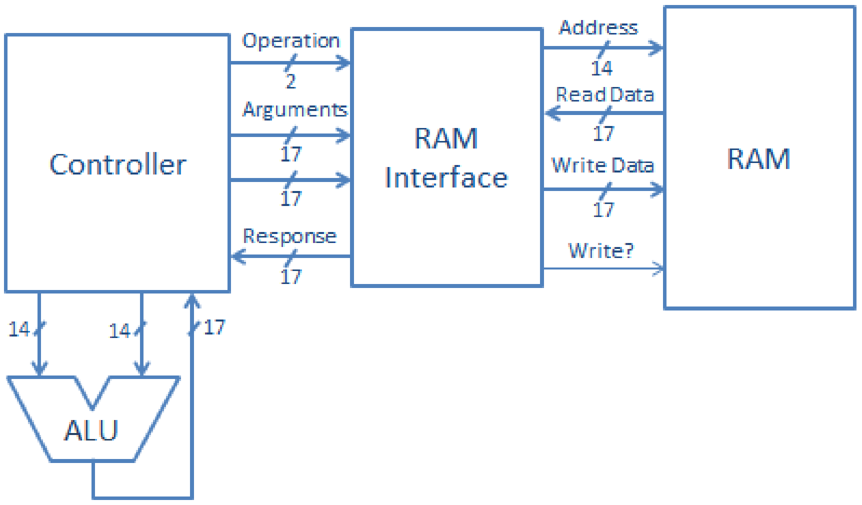


Figure 2: High-level processor organization

some exceptions to this, such as numbers, whose data field contains the number itself, "special objects" like booleans and the empty list, and variable references, which contain two different numerical fields representing the location of the variable relative to the expression currently being evaluated, as well as a bit describing the variable type.

### 4.2 RAM Interface

The RAM interface module is a Mealy state machine that translates the memory option requested by the controller into the appropriate signals to send to RAM, and formats the result from memory for the controller to use. The state diagram for this module is shown in Figure 3.

### 4.3 Arithmetic-Logic Unit

The ALU is capable of basic arithmetic and logic operations on 14-bit inputs. The module is not very complex; oprations are simply implemented using behavioral Verilog. The ALU does have one distinctive characteristic: the output includes a type field, so that logical operations like `or` and `==` are labeled as booleans ("special objects"), while arithmetic operations like `+` and `-` are marked as numbers. This allows the processor to correctly handle programs like

```
(if 1 'yes 'no)
```

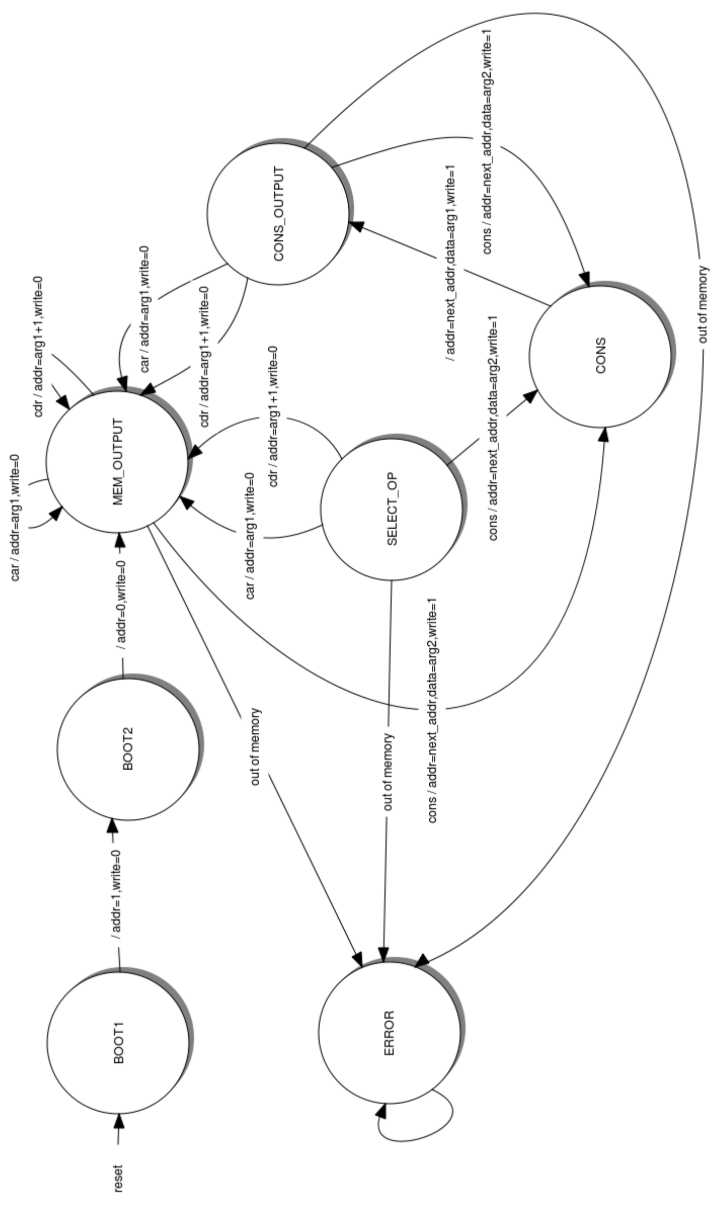

reset
BOOT1
/ addr=1,write=0
BOOT2
/ addr=0,write=0
MEM_OUTPUT
car / addr=arg1,write=0
cdr / addr=arg1+1,write=0
car / addr=arg1,write=0
cdr / addr=arg1+1,write=0
CONS_OUTPUT
car / addr=arg1,write=0
cdr / addr=arg1+1,write=0
SELECT_OP
cons / addr=next_addr,data=arg2,write=1
CONS
/ addr=next_addr,data=arg1,write=1
cons / addr=next_addr,data=arg2,write=1
out of memory
out of memory
out of memory
cons / addr=next_addr,data=arg2,write=1
ERROR

Figure 3: State diagram for RAM interface. The state machine begins in the BOOT1 state, which reads the length of the program from address 1 and stores it so that it can check for running out of RAM. Then it reads the starting object from address 0 and returns it to the controller. In addition to the outputs shown, which are directed towards RAM, the RAM interface also produces a response for the controller, which is the latest value read from RAM in the MEM_OUTPUT state, and the address of the most recently allocated pair in the CONS_OUTPUT state.

which should produce `'yes` even though the data field of the number 1 happens to match that of the object used to represent “false.”

## 4.4 Main Controller

The controller contains the core logic of the processor. It maintans all of the processor state apart from that stored in RAM, and coordinates the processor’s traversal of the program’s tree structure, determining what to evaluate next and what the environment of variable bindings should be for that evaluation.

The controller is implemented as a Mealy state machine augmented with five registers that can be read or written at any transition. These registers hold state about the data involved in the current computation, so that the states of the Mealy machine only need to handle control tasks. The functions of the registers are the following:

**EXP** Pointer to the list head of the current expression to be evaluated.

**ENV** Pointer to the head of the environment data structure, which is a list of scopes, each of which is a list of variable values in the order the variables are bound.

**ARGS** Pointer to a list of values that will be passed as arguments to a function. These values are built up one by one, and then added to the environment in which that function executes.

**STACK** The controller recursively evaluates the parts of the input expression. When it must evaulate a subexpression that is passed as an argument of a larger expression, it pushes the current state onto the list pointed to by this register so that it can return once the value of the subexpression is determined.

**VAL** When the controller has finished evaluating an expression, whether the whole input or a subexpression, this register holds the value. Also used as a scratch register for many operations.

The operation of the controller is outlined in Figure 4, and its state machine is shown in detail in Figure 5. The first step is to boot the processor by reading the initial state out of the first two words of memory. This leaves the processor in the EVAL state, which dispatches on the type of the object in EXP to decide how to evaluate it. So-called self-evaluating data types (symbols, lists, numbers, special objects) are returned as-is by being copied to the VAL register. The other types of objects are evaluated as described in the following sections.

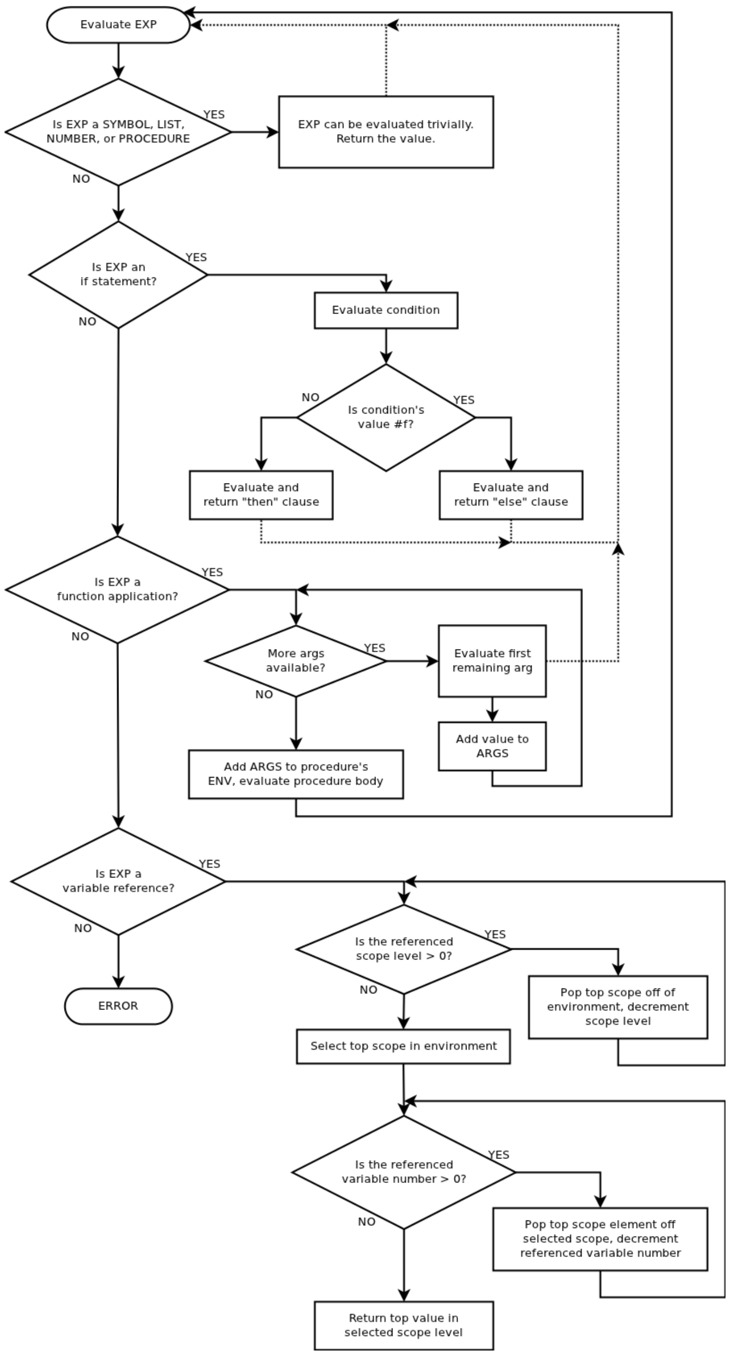

Evaluate EXP
Is EXP a SYMBOL, LIST, NUMBER, or PROCEDURE
YES
NO
EXP can be evaluated trivially. Return the value.
Is EXP an if statement?
YES
NO
Evaluate condition
Is condition's value #f?
NO
YES
Evaluate and return "then" clause
Evaluate and return "else" clause
Is EXP a function application?
YES
NO
More args available?
YES
NO
Evaluate first remaining arg
Add value to ARGS
Add ARGS to procedure's ENV, evaluate procedure body
Is EXP a variable reference?
YES
NO
ERROR
Is the referenced scope level > 0?
YES
NO
Pop top scope off of environment, decrement scope level
Select top scope in environment
Is the referenced variable number > 0?
YES
NO
Pop top scope element off selected scope, decrement referenced variable number
Return top value in selected scope level

Figure 4: Recursive execution algorithm used in processor

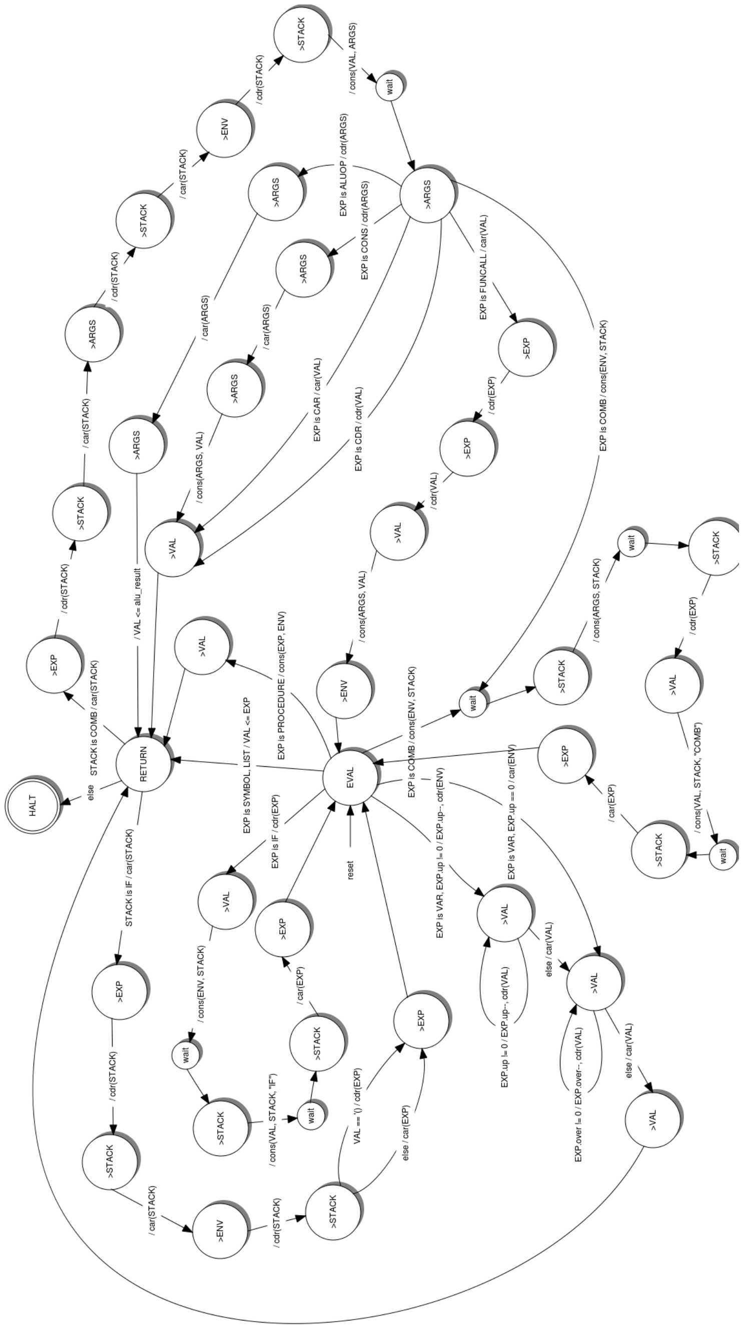


Figure 5: Controller state diagram. Execution begins in EVAL state. Many transitions depend on the type or value of some object, shown in the text before the "/" character. Most transitions also specify a memory operation, shown after the "/" character. The results of memory operations are generally stored into a register, which is written in the target state of the transition, after the ">" character.

### 4.4.1 Conditionals

Conditional expressions are evaluated fairly simply. First, the condition must be evaluated. The processor achieves this by pushing its state onto the stack, including a marker that shows that it was evaluating a conditional. The condition expression is loaded into the EXP register, and then the processor transitions into the EVAL state to evaluate the expression for however long that takes. Eventually it will come up with an answer, put it in the VAL register, and go into the RETURN state, where it will pop the stack, realize that it should be finishing up a conditional, and continue by evaluating and returning one of the following expressions depending on the value in VAL. The second one is returned if the condition evaluates to `#f`, Scheme's "false" value; otherwise, the first one is returned.

### 4.4.2 Procedures and Procedure Applications

Procedures, or lambda expressions, are evaluated by storing a copy of the current value of the ENV register alongside the code for the procedure. This creates a *closure*, a procedure that remembers the environment in which it was defined. So, if a procedure refers to a variable that is not one of its parameters, whenever that procedure is called it will use that same variable, regardless of what variables may be defined in the scope in which it is called.

Procedure applications are the most complex part of the evaluation process. First, the processor follows the recursive procedure outlined in the previous section to evaluate all of the arguments of the procedure call, as well as the procedure itself. As these values are computed, they are pushed to the top of the ARGS register, which is used as a stack. The order of evaluation of arguments is not significant in Scheme, but because the next thing to be evaluated is the body of the procedure, it is most convenient for the last value to be computed to be the procedure itself. This is achieved by reversing the order of the parameters in the memory representation of the procedure application.

Once the closure object is available, the code part of the procedure is placed in EXP to be evaluated. For this evaluation ENV is set to the environment stored in the closure, with the list stored in ARGS appended to it. In other words, the variables available to the executing procedure are the ones that were in scope when it was created, as well as the just computed argument values. When the procedure's code finishes being evaluated, the result will be passed up to the expression containing this procedure application.

### 4.4.3 Variable References

Scheme has a property called lexical scoping. This means that the mapping between references and variables is determined at compile time. A variable reference always refers to the variable of the same name that is defined closest to the reference. Other variables of the same name defined in higher scopes are "shadowed" by this definition. Since this association exists at compile time, it is possible to replace variable names, which are time consuming to compare, with numerical indices of how many scopes “up” to go to find the variable’s definition, and which place “over” in that scope that particular variable is. Figure 6 demonstrates this. In the processor, variable references are defined in this fashion, and they are evaluated by iteratively walking “up” and then “over” in the data structure pointed to by ENV in order to find the approprate value.

```
(lambda (x y)
    ; Variables visible:
    ;   x => 0 up, 1 over
    ;   y => 0 up, 2 over
    ...
    (lambda (x z)
        ; Variables visible:
        ;   x => 0 up, 1 over (shadows other x)
        ;   y => 1 up, 2 over
        ;   z => 0 up, 2 over
        ...)...)
```

Figure 6: Lexical scoping

## 5 Compiler Design

The compiler has a fairly standard organization, outlined in Figure 7. Its functionality is divided into a preprocessor, which expands a few of the standard macros of the Scheme language; a front end, which converts Scheme expressions into a high-level intermediate representation with additional information added; a back end, which produces binary representations of all the objects defined in the program; and a linker, which replaces all references to defined objects and symbols with the memory addresses of these objects. The result is an array of binary values, which are loaded directly into the memory of the simulated processor to be evaluated. The following sections describe the operation of each of these elements of the compiler.

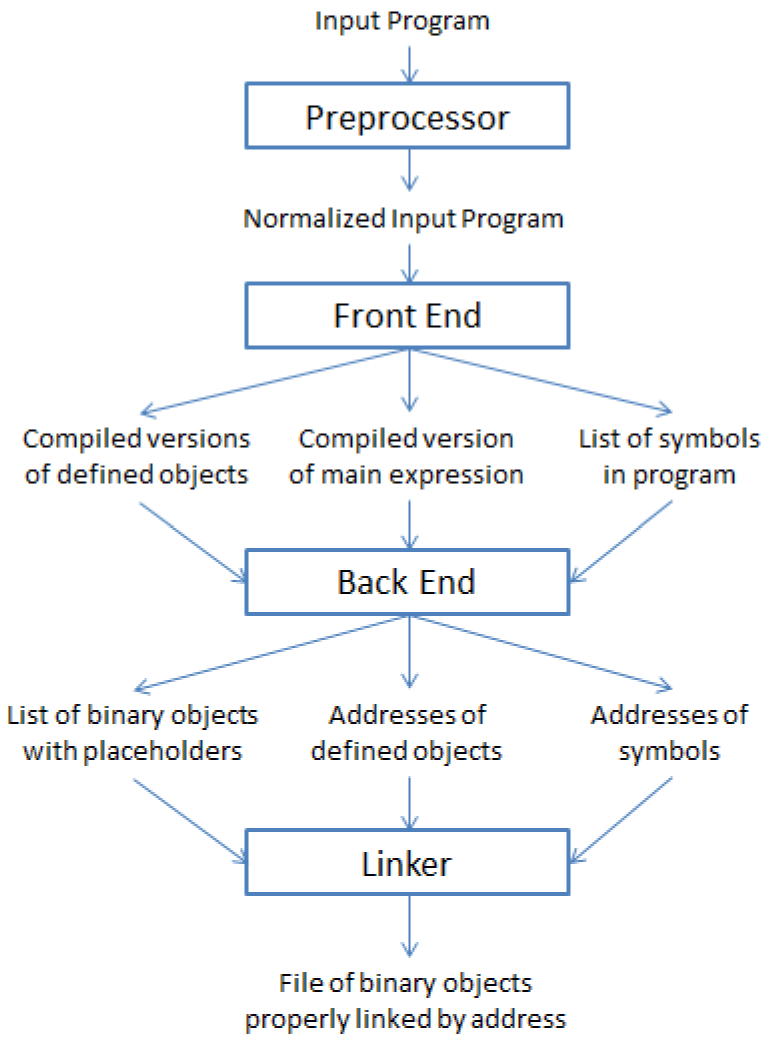

Input Program
Preprocessor
Normalized Input Program
Front End
Compiled versions of defined objects
Compiled version of main expression
List of symbols in program
Back End
List of binary objects with placeholders
Addresses of defined objects
Addresses of symbols
Linker
File of binary objects properly linked by address

Figure 7: Compiler organization

## 5.1 Preprocessor

The preprocessor makes it possible for the compiler to recognize a larger variety of Scheme syntax while keeping the compiler front end relatively simple. The preprocessor transforms several alternative Scheme syntax forms into the ones that are recognized by the front end. For example, the `let` construct allows intermediate values in a computation to be assigned a name. This construct is preprocessed as follows:

```
(let ((a 12) (b 17)) (expr involving a and b))
```

becomes

```
((lambda (a b) (expr involving a and b)) 12 17)
```

which can be accepted by the front end. This is purely a Scheme-to-Scheme translation, so the results can still be run with a standard Scheme interpreter.

## 5.2 Front End

The role of the compiler's front end is to traverse the input Scheme expression and determine the types of the objects represented, the order the processor expects to see them and the ways they are linked together. It produces output in a format easy to create in Scheme, but containing all the same information as the typed-pointer representation discussed in Section 4.1. The intermediate representation is a Scheme list with each object tagged with a type symbol. In addition to type tagging, the front-end modifies the link structure of the program to match that demanded by the processor. This means, among other things, that function arguments are linked together in the proper (reverse) order, and terminated with a pointer of the proper type. A symbol table is maintained as the traversal occurs; this allows the compiler to annotate variable references with the appropriate relative position in the environment, and to mark references to defined objects as such. These will be replaced with the memory addresses of these objects in the final compiled program.

The front end produces three data structures, which are all consumed by the back end. The first is a mapping of all the names that are defined in the program to compiled versions of the definition value. This includes values defined internal to functions; these names are qualified with the names of their enclosing functions. The second output is the compiled version of the expression that needs to be evaluated in this program. Once all the definitions are in place, this is the expression whose value the processor is expected to return. The last output is a list of symbols used in the program. Since all references to identically-named symbols are supposed to refer to the same object, this list is necessary for the back end to allocate one address for each symbol.

## 5.3 Back End

The compiler's back end translates each expression produced by the front end into the linked, typed binary values that the processor requires. Once again, it traverses the program's tree structure, first walking all the way down to the leaves, which have no dependencies on other objects. Once binary objects for a node's children are generated, the addresses of those generated objects are returned to the function compiling the parent node, so that the parent's binary representation can link correctly to the children. Binary objects are filled into memory sequentially as they are generated, and an object's children must be compiled before the object itself, so after this stage all links in the program point backwards, to smaller memory addresses. After all the expressions in the defined objects and the expression to evaluate have all been converted to binary, the symbols are also inserted into memory as linked lists of ASCII characters. During this process, references to symbols and defined objects are left untouched, as they will be handled by the linker.

This section of the compiler produces three data structures as well. The main output of the back end is the list of binary objects, mostly complete but containing placeholders for defined objects and symbols. In addition, it keeps track of the address of the head of each defined object's linked list. The name–address mappings of defined objects and of symbols are returned by the back end in two separate lists.

### 5.4 Linker

The functionality of the linker is fairly simple compared to the previous sections. The linker iterates through the list of binary objects provided by the compiler back end, and replaces any symbol or defined object placeholders with the addresses of those objects. These addresses are looked up in the data structures returned by the back end. This changes the link structure of the program from a set of trees to a graph in which an arbitrary node can link to one of the nodes that was formerly the root of a tree. After performing this linking step, the head node of the expression to evaluate is brought to the beginning of the list, followed by a word containing the length of the program. These values are used in the processor's boot sequence. Now the list of binary values is ready to be written to a file in a format that can be loaded into the processor's memory by the simulator.

## 6 Testing Methodology

Testing the processor involves several steps. First a Scheme program must be compiled into the typed-pointer representation described in 4.1. A program to perform this translation has been implemented in Scheme; it takes in a Scheme expression and traverses its tree structure, generating words in memory for each pair it encounters, constructing a memory representation that matches the linked structure of the program. This representation is written out to a data file.

Next, the Verilog code for the processor is compiled in Altera Quartus II [4], and then the testbench is run in ModelSim-Altera Edition [3]. The testbench contains a script that reads the values from the previously generated data file into the memory module in the simulator. The simulation is run and the values in memory are written out to a similar data file. The controller also produces an output, called `final_result`, which is a typed pointer to the object returned by the Scheme program. Because the output of the program could be a linked list of any size, it is infeasible to provide the entire object on the output.

In order to interpret the pointer produced by the processor, another Scheme program is run that parses the output file and allows objects in RAM to be looked up by their typed-pointer address. The value from `final_result` is looked up, and the output value of the program should be printed. If the result is not correct, debugging via ModelSim is necessary. Fortunately, as of now, the controller never overwrites values in memory, so any pointers used during the computation can be looked up in the final memory output file to confirm their accuracy.

Scheme code for testing the processor was derived from two sources, in addition to microbenchmarks that were written to test specific aspects of the system. The first is [1], a well-known Scheme textbook, from which a benchmark relating to Huffman trees was taken. This program runs correctly with only one modification: one of the functions in the benchmark has another function defined inside it, and this is a feature not yet correctly supported by the compiler.

The second benchmark that was used for testing the processor is the compiler itself. In order to be able to run properly, a few modifications needed to be made to the compiler:

- Internal definitions were made external
- Primitives like `car` and `cdr` could not be passed to functions directly, so they were replaced with normal procedures that call these functions
- The process for compiling symbols was made simpler, as symbols in the processor are already stored in a processor-compatible format.

Running on the processor, the compiler was able to successfully compile the following very simple program:

```
17
```

More complex programs were attempted, but the processor ran out of memory due to the lack of garbage collection of discarded stack items. Further testing (with more RAM available) is required to see if the compiler can be run reliably.

## 7 Future Work

Though it is nice to see the evaluation algorithm working so effectively, there are still a number of improvements that would make this a better processor, or a better implementation of Scheme, or both. The most important desirable feature at the moment is garbage collection. Scheme needs garbage collection because each use of `cons` allocates space for a new pair, but there is no programmatic way to erase an object once it is created. So, not only do Scheme programs leak memory without garbage collection, but the processor itself allocates pairs for its stack, to hold its environment, and to collect argument values to pass to a procedure. These are never destroyed when the relevant evaluation finishes, they just sit around taking up space. Garbage collection would significantly increase the size of program that could be run, because at the moment even a program that does nothing has a time limit imposed by the size of the memory.

The processor is missing a number of features of Scheme, as only a subset of the language was implemented. However, even in the subset that exists, the processor has some performance characteristics that are incorrect for a true Scheme implementation. First of all, definitions are implemented as pointers to the code in the definition. However, in order for internal definitions to function correctly (and in order for this to be a correct Scheme implementation, defined functions should be closures, which means they should remember the environment where they were defined. This is not the case in the processor's Scheme implementation, and it is the reason that internal definitions do not function correctly. Additionally, the Scheme specification requires something called *tail call optimization.* This means that when one function calls another as the last step in its evaluation, the second function should replace the first on the stack rather than being stacked above it. This optimization creates a potentially unlimited recursion depth, but the processor does not implement it.

A third project that could improve the performance of the processor would be to add parallelism. If it remains the case that no mutation operations are implemented in the processor, all objects created in memory will continue to be read-only. This means that the arguments of a function could be evaluated in parallel without competing. In fact, any two parts of the program can be evaluated in parallel as long as one is not a parent, grandparent, etc. of the other. There is a potential bottleneck, because the processor currently accesses memory on most clock cycles, but multiple data caches may be able to alleviate this problem.

Lastly, one original goal of the project was to implement the processor in hardware. It was originally planned to do this on an FPGA to make the transition easier, but unfortunately there was not enough time to make this happen. An FPGA implementation would be slightly tricky because of the difficulty of retrieving output data from the RAM inside the FPGA, but having a real hardware implementation would be an important next step for the project.

## 8 Final Words

I'd like to thank Tali Moreshet for all her advice, Guy Steele and Gerald Sussman for their interesting paper, and [2] and [9] for lots of assistance with Verilog coding. All the code for this project can be found online at http://www.benjaminlipton.com/downloads.html.

## References

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996. http://mitpress.mit.edu/sicp/.

[2] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digial Logic with Verilog Design.* McGraw Hill, 2003.

[3] Altera Corporation. ModelSim-Altera Starter Software, May 2012. https://www.altera.com/download/software/modelsim-starter.

[4] Altera Corporation. Quartus II Web Edition Software, May 2012. http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html.

[5] GNU Project - Free Software Foundation (FSF). GNU Emacs, May 2012. http://www.gnu.org/software/emacs/.

[6] Thomas F. Knight, Jr., David A. Moon, Jack Holloway, and Guy L. Steele, Jr. The CADR microprocessor. Technical report, Massachusetts Institute of Technology, 1979.

[7] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, and Jonathan Rees. Revised$^6$ report on the algorithmic language Scheme, 2007. http://www.r6rs.org/.

[8] Guy L. Steele and Gerald J. Sussman. Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode. Technical report, Massachusetts Institute of Technology, 1979.

[9] Deepak Kumar Tala. Verilog tutorial, May 2012. http://www.asic-world.com/verilog/veritut.html.