JANUARY 1989

WRL Research Report 89/1

SCHEME->C a Portable Scheme-to-C Compiler

Joel F. Bartlett

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	WRL-Techreports@decwrl.dec.com
CSnet:	WRL-Techreports@decwrl.dec.com
UUCP:	decwrl!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

SCHEME->C
a Portable
Scheme-to-C Compiler

Joel F. Bartlett

January, 1989

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

One way to make compilers portable is to have them compile to an intermediate language which is implemented on multiple architectures. By using C as the intermediate language and compiling the LISP dialect Scheme to it, it might be possible to achieve the following benefits. First, since C is the lingua franca of workstations the resulting system should be very portable. Second, it should allow Scheme programs to interact with those written in other languages. Finally, it should simplify the compiler as it need not repeat the optimization capability available in the C compiler.

However, there might be some unacceptable costs associated with this. First, there might not be a clean translation from Scheme to C, so that the implementation is not quite Scheme. Second, the two-stage translation might result in inefficient code. Finally, the generated code might be so stylized that it is neither portable nor compatible with other programming languages.

To investigate these issues, such a compiler and run-time system were constructed at Digital Equipment Corporation's Western Research Laboratory. Experience with the system shows that there is a translation of Scheme to C, that has good performance, and is portable.

1. Introduction

Compiling one language to another, rather than to a specific machine's instruction set, is a very old idea. The benefits of compiling high-level languages to an intermediate language, UNCOL[1], and then compiling that language for a specific architecture were first documented in 1958 [17]. Compiling to an UNCOL has at least the following advantages to a compiler writer. First, by emitting UNCOL rather than machine opcodes, a compiler is extremely portable. In theory, it can be ported to any machine which supports the UNCOL. Second, the UNCOL provides a common meeting point for all languages and thus facilitates the construction of programs written in multiple languages. Finally, the UNCOL compiler can simplify the individual language compilers by providing language-independent optimization.

Unfortunately, while many efforts since then have used intermediate languages [3], it is clear that they have not become widely accepted. Instead, standardization efforts have focussed on proprietary instruction set architectures, e.g. the Digitial VAX, and on industry wide standards for higher-level languages, e.g. FORTRAN-77.

However, if one is willing to restrict one's view to workstations running UNIX [10] or its derivatives, one could consider C [9] as a possible UNCOL. All such systems have a C compiler, and as C was the first high-level language for UNIX, most other languages can call C procedures and use its data structures. Also, since the operating system and many of the applications are written in C, the C compiler provides some level of machine-specific optimization. Finally, since workstation performance is passing that of traditional mainframes, the overhead imposed by using two compilers instead of one can be tolerated.

Since C appears to be a satisfactory UNCOL, our attention can now turn to the problems of compiling Scheme to it. In the sections which follow, it is assumed that the reader has some familiarity with C. The Scheme language is introduced and its differences from C are exposed in section 2. Section 3 provides an overview of the compiler and shows how Scheme can be efficiently mapped to C. Following this, section 4 compares the performance of this Scheme system to a native LISP implementation, and section 5 discusses the portability of the implementation. From this, section 6 concludes that the Scheme-to-C compiler has met the desired levels of performance, portability, and simplicity, without serious damage to Scheme's semantics.

---

[1] UNiversial Computer Oriented Language

2. Scheme By Example

Scheme is a dialect of Lisp which strives for simplicity without sacrificing power. It has a small number of rules for forming expressions which have clearly defined semantics and few restrictions on how they are composed.

When examining the examples which introduce the language, the reader should not concentrate on the obvious differences in syntax between Scheme and C, but on the differences in semantics. Specific attention is paid to those items which do not cleanly translate to C. Readers interested in going beyond this informal introduction to Scheme are encouraged to consult the standard [13] which was used to form this section, or the text [1].

2.1. Identifiers, Variables, and Binding

Scheme's identifiers are similar to those used in most languages. For example, `A1` is a legal identifier in Scheme and C. However, Scheme also allows identifiers of the form `<=` or `vector->list` and considers upper and lower case forms of a letter to be identical. Thus, `Vector->List` is the same identifier as `vECtor->lISt`.

One of the uses of identifiers is for variables, which name locations where values can be stored. A variable that does this is said to be *bound* to the location. The set of all bindings in effect at some point in the program is known as the *environment*. Every Scheme system has a *top level environment* which starts out with a number of variables bound to useful primitive values, most of which are primitive procedures that manipulate data. New variable bindings can be added to the environment by expressions of the following form:

`(define x 23)`	binds X to a location containing the value 23
`(define plus +)`	binds PLUS to a location containing the value of +, a predefined function to add two numbers

Variables do not have any type information associated with them. Instead, the type is associated with the object which is the value of the variable.

2.2. Primitive Expressions

A Scheme expression is a construct that returns a value. An expression consisting of a variable evaluates to the value in the location bound to the variable.

```
(define x 23)          Creates an initial binding for X
x       ==>     23     this means:  X "evaluates to" 23
```

Constants such as strings or numbers evaluate "to themselves":

```
"This is a string"      ==>     "This is a string"
23                      ==>     23
3.14159                 ==>     3.14159
#t                      ==>     #t          the boolean for true
#f                      ==>     #f          the boolean for false
```

Other expressions may be used as constants by "quoting" them:

```
(quote a)               ==>     a
'a                      ==>     a
```

A procedure call is written by simply enclosing in parentheses the expressions for the procedure to be called and the arguments to be passed to it. These expressions are evaluated in any order, then the arguments are passed to the procedure, and finally the procedure returns a value.

```
(+ 3 4)                 ==>     7
(+ (abs -3) 4)          ==>     7
(= 1 1)                 ==>     #t
(< 2 1)                 ==>     #f
```

Procedures are created by the evaluation of a `lambda` expression. The environment in effect when the lambda expression is evaluated is retained as part of the procedure. When the procedure is later called with some actual arguments, these are added to the saved environment, creating an extended environment. The expressions in the body of the lambda expression are evaluated sequentially in the extended environment, and the result of the last expression in the body is returned as the result of the procedure. A few examples are:

```
(lambda (x) (+ x 1))            ==>     #*PROCEDURE*        a procedure

(define inc
        (lambda (x) (+ x 1)))   ==>     INC                 defines a procedure which adds 1
                                                            to its argument

(define (inc x) (+ x 1))        ==>     INC                 an equivalent way of defining inc,
                                                            where the lambda is implied

(inc 23)                        ==>     24                  a procedure call to the previously
                                                            defined procedure

((lambda (x) (+ x 1)) 3)        ==>     4                   a procedure call, where the first
                                                            argument is a lambda expression
```

Unlike many earlier LISP's which used dynamically scoped variables, Scheme's variables are lexically scoped. As a result, the variable bindings in force at any point in the program can be determined at compile time.

Conditional execution is provided by the following form. If the value of the first expression is true, then the second expression is evaluated and its value is returned. Otherwise, the third expression is evaluated and its value is returned. For example:

```
(if (> 3 2) 'yes 'no)           ==>     yes         1st expression is true

(if (= 1 2) 'yes (+ 2 1))       ==>      3          1st expression is false
```

The final primitive expression allows the value of a variable to be changed:

```
(define x 10)           ==>      X      binds X to a location containing 10
(+ x 1)                 ==>     11
(set! x 23)             ==>     23      changes X's value to 23
x                       ==>     23
```

While this set of primitive expressions is sufficient to define the semantics of Scheme, it is more convenient to program using more powerful forms which are dervied from them.

2.3. Derived Expressions

Since there are no limitations on how the primitive expressions can be combined, higher-level expressions can be constructed which allow Scheme procedures to be written with a block structure similar to that of C. A set of expressions may be evaluated sequentially by the `begin` expression:

```
(begin (f1 1)
       (f2 2)
       (f3 3)
       'done)                    ==>     DONE
```

Since expressions in the body of a lambda expression are evaluated sequentially, the previous expression is defined as the evaluation of a function with no arguments, whose body is composed of the sequence of expressions:

```
((lambda ()
         (f1 1)
         (f2 2)
         (f3 3)
         'done))                 ==>     DONE
```

A more powerful form is the `and` expression that evaluates its arguments from left to right until one evaluates to a false value. To test that an integer `y` is between one and nine and even, one could write:

```
(and (< y 9) (> y 1) (zero? (remainder y 2)))
```

which can be expressed using the primitive expressions as:

```
(if (< y 9)
    (if (> y 1) (zero? (remainder y 2)) #f)
    #f)
```

While the latter has the same meaning as the former, it's clear the former is easier to write and understand. This completes the introduction to Scheme expressions and our attention will now turn to some of their properties.

2.4. Procedures Are More Than Code

At first glance, lambda seems to be some form of syntactic sugar to declare a procedure. When used to declare a simple top level procedure that computes the absolute value of a number:

```
(define (abs x) (if (> x 0) x (- 0 x)))
```

it seems equivalent to a C procedure declaration. It is only when one takes advantage of the fact that a procedure is composed of an environment as well as code, that one sees its power. The procedure `make-counter` takes an inital value for the counter and returns a procedure which counts:

```
(define (make-counter n)        ==>   MAKE-COUNTER
        (lambda ()
                (set! n (+ n 1))
                (- n 1)))
```
a procedure which returns a procedure

```
(set! c1 (make-counter 100))    ==>   #*PROCEDURE*
```
counter with n bound to a location containing 100

Each time the counter procedure is evaluated, it returns the next number:

```
(c1)                            ==> 100
(c1)                            ==> 101
```

Note that `make-counter` can make many counters, and while they will share the code, they will have their own environments:

```
(set! c2 (make-counter 200))    ==> #*PROCEDURE*
```
counter with n bound to a location containing 200

```
(c2)                            ==> 200
(c2)                            ==> 201
(c1)                            ==> 102
(c2)                            ==> 202
```

Objects with more complex actions and a larger local state can be constructed in a straightforward manner.

2.5. Objects Are Created at Any Time and Last Forever

The procedures representing counters shown in the previous section can be created at any time. Objects are never explicitly released as long as the program has at least one reference to them. The storage allocator may use *garbage collection* to recover storage which is no longer referenced. By constrast, in C most objects exist for the entire duration of the program, or are created on entry to a procedure, and deleted on exit from it. Objects that must have indefinite extent are explicitly allocated and returned by calls to C library routines.

2.6. Continuations

Scheme provides a very general control mechanism based upon *continuations*. A continuation represents the entire future of the computation, i.e. what to do next. In languages which have labels and goto's, the goto statement's action can be described as: ignore the implicit continuation (which is the next sequential statement), and pass control to a continuation at the label.

The Scheme procedure `call-with-current-continuation` packages up the current continuation as an *escape procedure*. The escape procedure is a Scheme procedure of one argument that, if it is later passed a value, will give the value to the continuation that was in effect at the time that escape procedure was created. A simple example is the following which shows how invoking the escape procedure can result in an "upexit" from the sequential execution of expressions in the body of a procedure.

```
(call-with-current-continuation         ==> 1
    (lambda (exit)
            (exit 1)
            (/ 1 0)))
```

Here, the division is never executed, as the procedure `exit` causes an immediate return to the caller of `call-with-current-continuation` with the value `1`. A trickier example which hints at the power available is the following, where an escape procedure is used to return to the "inside" of a previous computation. When the procedure `count`:

```
(define repeat #t)

(define (count n)
        (if (= n 0)
            (begin (display "*")
                   (call-with-current-continuation
                         (lambda (exit) (set! repeat exit))))
            (begin (display n)
                   (count (- n 1))
                   (display n))))
```

is called with 3 as its argument, it prints `321*123` and sets `repeat` to an escape procedure which represents the future of the computation just after the `*` was printed. Every time `repeat` is called, it will return to that point in the computation and print the digits `123`.

This ability to shift control is significantly more powerful than the upexit capability provided by UNIX's *setjmp* and *longjmp* [10], or Common LISP's *catch* and *throw* [16]. As a result, a large variety of control structures can be fabricated, as is shown in [7].

2.7. Iteration is a Special Case of Function Call

Using the concept of continuations (and ignoring the mechanism used for argument passing and value return), the procedure call and return mechanism of C can be informally described as follows. A procedure call consists of two steps: first create a continuation which represents the point to return to following the call and push it on a stack; then transfer control to the called procedure. Following its execution, the called procedure returns by transfering control to a continuation which it pops from the stack.

When the last thing that a procedure does is to call another procedure, that call is called a *tail-call*. For example, in the following C code that tests whether a non-negative integer is even or odd:

```
int  even( i )
        int  i;
{
        return( if  (i == 0)  1  else  odd( i-1 ) );
}

int  odd( i )
        int  i;
{
        return( if  (i == 0) 0  else  even( i-1 ) );
}
```

the call to `odd` inside `even` and the call to `even` inside `odd` are tail-calls. Even though C always pushes a continuation on the stack on a procedure call, it is not necessary for tail-calls as there is no need to return to the calling procedure. That is, tail-calls can be thought of as branches.

Scheme on the other hand requires that the implementation be *properly tail-recursive*. A Scheme system must identify tail-calls and not save another continuation when such a call is made. This allows the Scheme version of the previous program to be written as:

```
(define (even i) (if (= i 0) #t (odd (- i 1))))

(define (odd i) (if (= i 0) #f (even (- i 1))))
```

which executes with a constant amount of stack space, irrespective of the value of `i`. This is because the calls to `odd` and `even` inside `even` and `odd` are recognized as tail-calls and no continuation is saved when the calls are made.

By recognizing how procedure calls are used, Scheme is able to avoid having an interation primitive. When a programmer wishes to iterate, they often use the derived expression `do`. With it, the programer writes an iterative form which is then expanded by the Scheme implementation into an expression involving recursion. When this expression is executed, those calls which are tail-calls are recognized as such and processed efficiently. That is, the program can be expressed in an iterative manner, and the generated code can be iterative, but the programming language does not directly support iteration.

2.8. A LISP Without Lists?

In spite of the fact that *pairs*[2] and *lists* are as prevalent in Scheme programs as arrays and integers are in C programs, the discussion to this point has ignored them. The actual data types supported by Scheme and the operations performed upon them are but a minor part of the implementation of Scheme. In fact, the only time that lists are used in the primitive expressions is as part of the mechanism which allows procedures which accept a variable number of arguments.

With this overview of the semantics of Scheme, we can now turn our attention to the problems involved in translating Scheme to C.

3. Compiling Scheme to C

As can be seen from the introduction to Scheme, there are major differences from C in the semantics of procedure call and the lifetimes of objects. Smaller though significant differences exist in the semantics of procedures and procedure argument passing. Finally, Scheme introduces a new concept with its escape procedures which capture continuations.

Before examining the solutions to these problems, a short overview of the compilation process is in order. The design of this process owes much to Steele's RABBIT compiler [15] done at MIT and the ORBIT compiler [11] done at Yale.

---

[2] A *pair* is a type of Scheme object with two fields, the *car* and the *cdr*. Pairs are allocated by calls to the procedure *cons* which takes the values to place in the two fields as its arguments. Each of these fields may in turn contain any Scheme object. A *list* is either a special object called the *empty list*, or a pair whose cdr field contains a list.

Starting with an initial source program:

```
(define (sample x y z) (if (or (and x y) z) 0 1))
```

Scheme->C converts all expressions to their equivalent primitive expressions:

```
(define
  (sample x y z)
  (if
    ((lambda (x thunk) (if x x (thunk)))
      ((lambda (x thunk) (if x (thunk) x)) x (lambda () Y))
      (lambda () Z))
    0
    1))
```

Following this all variables are uniquely named, a process known as alpha-conversion [15], and each primitive form is replaced by an equivalent internal form:

```
($DEFINE S1000
     ($LAMBDA L1001
          ($IF
            (<APPLY> () L1005
             (X1006
               <-
               (<APPLY> () L1008
                (X1009 <- X1002)
                (T1010 <- ($LAMBDA L1011 Y1003))
                ($IF X1009 ($CALL () T1010) X1009)))
             (T1007 <- ($LAMBDA L1012 Z1004))
             ($IF X1006 X1006 ($CALL () T1007)))
            C1014
            C1013)))
```

Lambda expressions have their arguments replaced by a unique name. The arguments and other information associated with the expression are stored in a side table indexed by the unique name. All function calls are modified to have an additional initial argument which is a flag identifying tail-calls. The flag is set on tail-calls to the id of the lambda expression that is exited when the call is made. A call where the function is a lambda expression is displayed differently so that the lambda bindings can be observed[3]. Thus, the expression:

```
(<APPLY> () L1008
 (X1009 <- X1002)
 (T1010 <- ($LAMBDA L1011 Y1003))
 ($IF X1009 ($CALL () T1010) X1009)))
```

is actually represented internally as:

```
($CALL ()
       ($LAMBDA L1008
                ($IF X1009 ($CALL () T1010) X1009))
       X1002
       ($LAMBDA L1011 Y1003))
```

[3]Those familiar with the ORBIT compiler will note that this compiler does not do CPS conversion.

Following conversion to the internal form, the program is optimized using rewrite rules similar to those used in the ORBIT compiler. After the application of these transformations, the sample expression becomes:

```
($DEFINE S1000
     ($LAMBDA L1001
          (<APPLY> L1001 L1020
           (Y1022 <- ($LAMBDA L1024 ($IF Z1004 C1014 C1013)))
           ($IF X1002
                ($IF Y1003 C1014 ($CALL L1001 Y1022))
                ($CALL L1001 Y1022)))))
```

At this point in the translation process, one can start to consider emitting code. However, before that can be done, a method for calling procedures must be picked.

3.1. Procedure Call: a Dilemma

One of the strengths of a good C compiler is that it provides significant processor independent and dependent optimization. While many move frequently used local variables to registers[4], compilers on RISC processors such as the MIPS R2000 [8] make a significant effort to optimize procedure calls. Some procedure arguments are passed in the registers and calls to leaf procedures[5] are optimized. By using C's procedure call and return mechanisms, one is able to take advantage of these optimizations. In addition, by retaining C's procedure call and return mechanism, call-outs to and call-backs from other languages can be done in a straightforward manner.

However, there is a drawback to this approach: C is not properly tail-recursive, so all procedure calls result in a stack marker being pushed on the runtime stack. It would appear that the compiler writer is faced with a dilemma: use C's mechanisms and cease to be properly tail-recursive, or avoid them and lose significant amounts of optimization.

After some thought, it was decided to use C's procedure call mechanism and preserve as much of the tail-recursive semantics as possible through compile-time analysis.

3.2. Procedure Call Analysis

Following the program transformation phase, the compiler analyzes how each procedure is called and uses that information to decide how code for it should be emitted. In the previous example, the lambda expression `L1001` is the procedure `sample`. Since it is defined at the top level of the program, it is translated into a C procedure. The next lambda expression, `L1020`, is only called once, so it can be generated in-line where it is called. The final lambda expression, `L1024`, is called twice. Each call is a tail-call which is to exit from the lambda expression `L1001` following the execution of `L1020`. These conditions allow `L1024` to be generated in-line at the end of the code for `L1001`, and calls to it be branches. Thus the C code for the sample program becomes:

---

[4]Typically the compilers will ignore `register` declarations and rely on their own program analysis.

[5]A *leaf procedure* is a procedure which does not call any other procedure

```
TSCP  test_sample( x1002, y1003, z1004 )
        TSCP  x1002, y1003, z1004;
{                                                  |  code for L1020
        if  ( FALSE( x1002 ) )  goto  L1032;       |     tail-call to L1024
        if  ( FALSE( y1003 ) )  goto  L1032;       |     tail-call to L1024
        return( _TSCP( 0 ) );                      |
L1032:
        if  ( TRUE( z1004 ) )  goto  L1033;        |  code for L1024
        return( _TSCP( 4 ) );                      |
L1033:                                             |
        return( _TSCP( 0 ) );                      |
}
```

which is similar to the code that one would write in C if one expanded the `and` and `or` operators by hand.[6]

The optimization mechanisms used here are not specific to `and` and `or`, as they are done only using the primitive expression types. For example, a do loop which prints a sequence of numbers can be expressed as:

```
(define (print-1-to-limit limit)
    (do ((i 1 (+ i 1)))
        ((> i limit) 'done)
        (display i)))

(print-1tolimit 8)      ==>     12345678DONE
```

Following expansion to primitive expression types, it results in the following expression:

```
(define print-1-to-limit
    (lambda (limit)
      ((lambda (doloop)
         (set!
           doloop
           (lambda (i)
             (if
               (> i limit)
               'DONE
               (lambda () (display i) (doloop (+ i 1))))))
         (doloop 1))
        0)))
```

The loop is turned into a function which is called once with the initial conditions. Inside the loop function, the next iteration is expressed as a tail-call to itself. After the optimizing transforms have been applied, the internal form is:

[6] `TSCP` is the type definition for a "tagged Scheme to C pointer", which is the type representing all Scheme objects. `_TSCP` is a define which casts an expression to the type `TSCP`. `FALSE` is a define which is true if the expression is either the Scheme boolean `#f` or the *empty list*. `TRUE` is a define which is true if the expression is neither `#f` nor the *empty list*.

```
($DEFINE P1000
     ($LAMBDA L1001
          (<APPLY> L1001 L1003
           (D1004 <- C1022)
           ($SET
             D1004
             ($LAMBDA L1005                    <-- loop function
                  ($IF
                    ($CALL () *greater-than* I1006 L1002)
                    C1020
                    (<APPLY> L1005 L1013
                     ($CALL () D632 I1006)
                     ($CALL                    <-- internal tail-call
                       L1005
                       D1004
                       ($CALL () *plus* I1006 C1021))))))
           ($CALL L1001 D1004 C1021))))      <-- initial call
```

where the functions `*greater-than*` and `*plus*` are expanded in-line. The loop procedure `L1005` can be generated in-line because the variable that it is assigned to, `D1004`, is never passed as a procedure argument, and all but one of the calls to `D1004` are tail-calls which are internal to it. The resulting code is iterative as one would expect:

```
TSCP  test_print_2d1_2dto_2dlimit( l1002 )
        TSCP  l1002;
{
        TSCP  X1;

        X1 = _TSCP( 4 );
L1032:
        if  ( GT( INT( X1 ), INT( l1002 ) ) )  goto  L1033;
        scrt6_display( X1, EMPTYLIST );
        X1 = _TSCP( PLUS( INT( X1 ), INT( _TSCP( 4 ) ) ) );
        goto L1032;
L1033:
        return( c1020 );
}
```

While the analysis shown in the previous examples is powerful enough to compile finite state machines correctly, it does have some limitations. Any procedure which must be compiled as a C procedure can only be tail-called by itself. This includes procedures defined at the top level, and any procedure that is called from multiple places where at least one of the calls is not a tail-call. As a result, the previous tail-recursive example involving two top level procedures is not properly compiled:

```
(define (even i) (if (= i 0) #t (odd (- i 1))))

(define (odd i) (if (= i 0) #f (even (- i 1))))
```

However, correct code can be obtained if the user is willing to put a "wrapper" procedure around a set of tail-recursive procedures. By restating the problem as follows, correct iterative code is generated.

```
(define (even? x)
    (define (even x) (if (= x 0) #t (odd (- x 1))))
    (define (odd x) (if (= x 0) #f (even (- x 1))))
    (even x))
```

There are certain to be those in the Scheme community who are offended by the fact that the tail-recursive properties of the language have been compromised. However, the compiler correctly handles tail-recursion in the cases where there is a big win: loops and conditional expressions. With minor changes to the program, such things as finite state machines can be correctly compiled. The inability to tail-call all procedures has been traded for high performance and compatibilty with other tools and languages.

3.3. Procedure Call: Loose Ends

Scheme allows procedures to be defined to accept optional arguments. When the called procedure is entered, these arguments are available in a newly constructed list. In this implementation of Scheme, the calling procedure is required to construct the argument list. While this adds code to each call, it assures that procedures will always be called with the correct fixed number of arguments. An example of a procedure which accepts one required argument and additional optional arguments is:

```
(define (sum x . y)
    (for-each
        (lambda (z)
                (set! x (+ x z)))
        y))
```

which returns the sum of its arguments. Calls to it are compiled as:

```
test_sum( _TSCP( 4 ), EMPTYLIST );                  (sum 1)

X1 = CONS( _TSCP( 16 ), EMPTYLIST );
X1 = CONS( _TSCP( 12 ), X1 );
test_sum( _TSCP( 4 ), CONS( _TSCP( 8 ), X1 ) );    (sum 1 2 3 4)
```

A second consideration in calling procedures is that some procedures must be executed in the environment where they were created. For example on each call to a counter procedure, `c1`:

```
(define (make-counter n)
        (lambda ()
                (set! (+ n 1))
                (- n 1)))

(set! c1 (make-counter 100))
```

the environment where `n` is bound must be supplied. This is done by compiling such procedures with a pointer to the environment as an additional argument. It is the caller's responsibility to supply this argument.

A call to `c1` in the previous example is an example of the most general form of procedure call, where all checking and argument construction must be deferred to runtime. In order to assure that it is correctly done, the following operations must be performed:

- Verify that `c1` is a procedure and that it is being called with the correct number of arguments.
- Form any optional arguments into a list.

- Call the procedure with the procedure's environment as the last argument.

While it would be nice to construct a procedure call with an arbitrary number of arguments with in-line code, C does not allow it. The only mechanism for simulating this is to compute the number of arguments, and use it as an index to a switch statement to select a call with the correct number of arguments. Needless to say, this results in a lot of code, so it must be placed in its own procedure in the runtime library.

One optimization that is made is to assume that the procedure takes no optional arguments. Then, the number of arguments that a procedure **P** is being called with will be known at compile time and a call statement to a procedure variable **X** is constructed. If at runtime **P** is indeed a procedure expecting that fixed number of arguments, then **X** is set to **P**. If this is not true, then **X** is set to a "trampoline" procedure in the runtime library which will handle errors or optional arguments. The environment pointer is always supplied here, as some procedures need it and C permits a procedure to be called with extra arguments which are simply ignored.

Fortunately, there is enough information available at compile time to correctly compile most procedure calls. The compiler uses its knowledge about predefined functions and those defined in the module currently being compiled to generate the correct calling sequence. Other calls emit the previously described general calling sequence. Since it can do many calls in-line, few calls must resort to the most general calling mechanism implemented by the runtime library.

3.4. Storage Allocation

Type information is encoded in objects in a straightforward manner. The implementation requires that pointers be 32-bits long and be byte addresses. Since storage is always allocated on a four-byte word boundary, each object is a 32-bit pointer, with the low-order two bits reserved for a tag. Note that this tagging scheme does not result in the loss of any address bits. Depending upon the value of the tag, the rest of the word is either the object, or a pointer to the object.

Since objects have an indefinite lifetime, they must be allocated from a heap. As the objects are never explicitly freed, some mechanism must be provided for objects which are no longer referenced. A review of the literature on garbage collection [5] suggests that garbage collection might be quite difficult in the C environment. Virtually all collection algorithms start with information found in a root set of pointers which must be in known locations. As a result, the collector must know the conventions for register and stack usage, which must always be followed. Typically this requires that registers which are assumed to contain pointers always be valid, and that pointers never be placed in other registers. Items in the stack must be flagged in such a way that all objects that are pointers can be found, and those that aren't are never mistaken for pointers.

When the Scheme compiler has complete control over the emitted code, conventions can be followed. However, there is no way for a Scheme compiler generating C to enforce any register or stack usage conventions. The C translator is the only agent which decides instruction sequences, how registers are used, or how the stack is allocated. As a result, any algorithm that requires the garbage collector to know exactly where the pointers are cannot be used.

To solve this problem, a new garbage collection algorithm called "mostly-copying" was devised [2]. Instead of requiring that the root set be a known set of cells containing pointers which define all accessible storage, the new algorithm only requires that the root set include pointers or values derived from pointers which define all accessible storage. The cells in this new root set need not all be pointers, nor is it required that there not be cells which "look like pointers", but aren't.

Using this root set, the algorithm divides all accessible objects in the heap into two classes: those which might have a direct reference from the root cells, and those which don't. The former items are left in place, and the latter items are copied into a compact area of memory. In practice, only a very small amount of the heap is left in place, so memory fragmentation is not a problem.

3.5. Runtime Environment

Scheme requires a significantly richer runtime environment than does C. For example, a call to function `a` in:

```
(define (a) (eq? (b) 'b))

(define (b) 'b)
```

must return the value `#t`[7] when both procedures are compiled together, separately compiled, interpreted, or just one is compiled. In order to do this, there can only be one symbol constant `b`.

Another example where complex initialization is required is:

```
(define time (current-time-of-day))
```

where the top level variable `time` is set to the current time of day. If this code is compiled, then a mechanism must be provided which will assure that `time` is initialized at the start of the program. Furthermore, the variable `time` must be available at runtime to a Scheme interpreter. If one enters `time`, then one should get its value. Or, if one enters `(set! time 0)`, one should change the value of `time` seen by interpreted and compiled code.

These examples suggest that C's model of execution where compile-time globals are set to constant values and execution starts at the main procedure is not powerful enough. Instead, an initialization model similar to that used by Modula-2 is used. The Scheme compiler compiles a module at a time, where each module consists of one or more source files. The module name is supplied in a module directive `(module` *module-name*`)`, and `define-external` expressions may be supplied to declare objects defined in other modules. However, unlike Modula-2, there are default external references. All objects defined to be in the initial Scheme environment are predefined. Objects which are not defined in the initial Scheme environment are assumed to be defined in the top level environment at runtime.

---

[7] `eq?` is the finest grain comparator. It returns #t, the boolean "true", when the objects are identical, i.e. the pointers are the same.

The code generated to handle initialization is compatible with that generated by WRL's Modula-2 compiler. Each module which is compiled has an initialization procedure, *module-name*`__init` which is called at the start of execution to initialize the module. This initialization procedure calls the initialization procedures of all other modules needed by that module. At the end of this initialization process, the user's main procedure is executed.

This environment setup process is best shown by example. The classic "hello, world" program in C is:

```
main()
{
        printf( "hello, world\n" );
}
```

which can be written in Scheme as:

```
(module hello (main main))

(define (main clargs)
    (display "hello, world")
    (newline))
```

which compiles into the C code:

```
/* SCHEME->C */

#include "schemetoc/objects.h"

DEFSTRING( t1004, "hello, world", 12 );
DEFSTATICTSCP2( c1003, t1004 );

static void  init_constants()
{
}

DEFTSCP( hello_main_v );
DEFSTRING( t1005, "MAIN", 4 );
EXTERNTSCPP( scrt6_display );
EXTERNTSCP( scrt6_display_v );
EXTERNTSCPP( scrt6_newline );
EXTERNTSCP( scrt6_newline_v );

TSCP  hello_main( c1002 )
        TSCP  c1002;
{
        scrt6_display( c1003, EMPTYLIST );
        return( scrt6_newline( EMPTYLIST ) );
}

void  hello__init(){}

static void  init_modules( compiler_version )
        char *compiler_version;
{
        scrt6__init();
}
```

```
main( argc, argv )
        int argc;  char *argv[];
{
        static int  init = 0;
        if  (init)  return;
        init = 1;
        INITHEAP( 0, argc, argv, hello_main );
        init_constants();
        init_modules( "(hello SCHEME->C COMPILER 09nov88jfb)" );
        INITIALIZEVAR( U_TX( ADR( t1005 ) ),
                       ADR( hello_main_v ),
                       MAKEPROCEDURE( 1,
                                      0, hello_main, EMPTYLIST ) );
        MAXDISPLAY( 0 );
        hello_main( CLARGUMENTS( argc, argv ) );
        SCHEMEEXIT();
}
```

Here, the main program is not the user's code, but the module initialization code. By examining it, one can see the Scheme initialization process.

- When the the C source is compiled, it includes the file `schemetoc/objects.h` which defines the types and defines underlying the Scheme system.
- The constant `t1004` defines the storage for the string "hello, world" which consists of a header identifying the object as a string of length 12, followed by the text. All references to the string in the program are via the variable `c1003` which contains a tagged pointer to the actual string.
- Turning now to the main procedure, the local static variable `init` assures that the initialization module is executed exactly once.
- The define `INITHEAP` calls a function in the run-time library to initialize the heap.
- The procedure `init_constants` is called to build any constant lists or vectors which are required by this module. Since none are needed here, this procedure is empty.
- The call to `init_modules` causes the initialization procedures for all modules required by this one to be executed. In this case, `scrt6__init` is called because that module contains the procedures `display` and `newline`.
- The user's procedure `main` causes both a C procedure `hello_main` and a global variable `hello_main_v` to be created. The global variable is set to a procedure object which in turn points to the C procedure. The define `INITIALIZEVAR` is called to store the procedure object into the variable `hello_main_v`, and to associate the name `main` with that variable.
- The number of entries used in the lexical display are recorded by the define `MAXDISPLAY`.
- The command line arguments are formed into a list and that is passed to the user's procedure which prints "hello, world" and then returns.
- Finally, the process is terminated by the define `SCHEMEEXIT`.

While some of the initialization is superfluous in this example, most of it is needed most of the time so it's always generated.

3.6. Variable Binding

Variables in Scheme differ from those allocated by C in that they have indefinite extent. However, compile time analysis can discover how they are used so that they may be efficiently bound. The factors that the compiler considers are:

- Is the variable bound in the top level environment?
- Otherwise, for lambda-bound variables:
  - Is the variable closed by a lambda expression?
  - Is the variable accessed by a nested procedure?
  - Is the variable ever changed by `set!`?

3.6.1. Global Variables

Variables that are bound in the top level environment are allocated as C globals. Those which are bound to a procedure also require a C procedure with the code. Thus the Scheme module:

```
(module sample)

(define x 10)

(define (inc x) (+ x 1))
```

will be compiled into the following globals:

```
        .
        .
        .
TSCP  sample_x_v;                       Location for X
TSCP  sample_inc_v;                     Location for INC
TSCP  sample_inc( x1004 )               Code for INC
        TSCP  x1004;
{
        .
        .
        .
}
        .
        .
        .
```

3.6.2. Automatic Variables

The most efficent and most common case in allocating lambda-bound variables is when none of the conditions requiring special handling are true. Here, the scope and extent of the variables are identical to that of C's procedure arguments and locals so they can be used. This allows the variable `x` which is the procedure argument of `inc` in the previous example to be allocated by C.

3.6.3. Display-Allocated Variables

Since C does not allow procedures to be nested lexically, the compiler must provide some mechanism for allowing one C procedure to access another's lexical variables. This is done by allocating such variables to a *display*, implemented by a set of global variables.

In the following contrived example:

```
(define (a x)
        (define (b y) x)
        (b x)
        (b 2))
```

the procedure `b`, nested within `a`, needs to access the variable `x` which is bound on the call `a`.

Since `b` is called twice, and one of the calls is not a tail-call, it must be compiled as a separate C procedure. When `a` is compiled, a slot in the display is allocated to `x`. On entry to `a` the contents of that display entry are saved in a local variable, and the value of `x` is placed in it. Within `a` and `b`, all references to `x` refer to the value in the display slot. When `a` is exited, the display entry is restored to its previous contents.

Code for b

```
TSCP  test_b1004( y1006 )
        TSCP  y1006;
{
        return( DISPLAY( 0 ) );
}
```

Code for a

```
TSCP  test_a( x1002 )
        TSCP  x1002;
{
        TSCP  SD0 = DISPLAY( 0 );
        TSCP  SDVAL;

        DISPLAY( 0 ) = x1002;
        test_b1004( DISPLAY( 0 ) );
        SDVAL = test_b1004( _TSCP( 8 ) );
        DISPLAY( 0 ) = SD0;
        return( SDVAL );
}
```

Variables must also be allocated to the display when a binding is "captured" by a lambda expression. In the example described earlier which makes counters:

```
(define (make-counter n)
        (lambda ()
                (set! n (+ n 1))
                (- n 1)))
```

analysis done by the compiler discovers that the variable `n` is captured by the procedure returned as the result of calling `make-counter`. However, `n` is also modified by `set!` which changes how the display slot is used.

3.6.4. Heap-Allocated Variables

In order to be compatible with the implementation method used for saving the display and continuations, variables allocated on the stack or in the display cannot be changed after their initial binding. So, variables which are changed by `set!` are bound to a pointer to a pair containing the value, rather than to the actual value[8]. While this adds a level of indirection to each variable access, Scheme programmers are encouraged to write in a functional manner so `set!` is not extensively used.

With this in mind, we can now examine the code for `make-counter`:

Procedure returned by make-counter that does the counting.

```
TSCP  test_l1003( c1019 )
        TSCP  c1019;
{
        TSCP  X2, X1;

        X1 = DISPLAY( 0 );
        DISPLAY( 0 ) = CLOSURE_VAR( c1019, 0 );
        PAIR_CAR( DISPLAY( 0 ) ) =
                        _TSCP( PLUS( INT( PAIR_CAR( DISPLAY( 0 ) ) ),
                                     INT( _TSCP( 4 ) ) ) );
        X2 = _TSCP( DIFFERENCE( INT( PAIR_CAR( DISPLAY( 0 ) ) ),
                                INT( _TSCP( 4 ) ) ) );
        DISPLAY( 0 ) = X1;
        return( X2 );
}
```

Code for make-counter

```
TSCP  test_make_2dcounter( n1002 )
        TSCP  n1002;
{
        TSCP  SD0 = DISPLAY( 0 );
        TSCP  SDVAL;

        DISPLAY( 0 ) = CONS( n1002, EMPTYLIST );
        SDVAL = MAKEPROCEDURE( 0,
                               0,
                               test_l1003,
                               MAKECLOSURE( EMPTYLIST,
                                            1, DISPLAY( 0 ) ) );
        DISPLAY( 0 ) = SD0;
        return( SDVAL );
}
```

Here, `n` is allocated a slot in the display, but since it is changed by `set!` its actual value must be retained in the heap. This binding of `n` is made part of the returned procedure. Each time such a procedure is called, a pointer to the saved environment holding the binding of `n` is supplied as its argument. On entry, the contents of `n`'s entry in the display is saved in a local variable. On exit, the display is restored from the local variable.

[8]ORBIT's solution to this problem is to explicitly add this level of indirection by a process they call "assignment conversion" during the program transformation phase of compilation. The result is similar code, but the mechanism is more obvious, and perhaps preferred, as it's not hidden in the code generator as it is in this compiler.

3.7. Continuations

Various mechanisms have been used to create continuations in Scheme, as is summarized in [4]. Given the fact that we wish to avail ourselves of C's procedure call/return and local variable allocation mechanisms, it makes sense to implement continuations using the *stack* strategy. When a continuation is constructed by `call-with-current-continuation`, the contents of the stack, display, and registers are saved and made a part of the escape procedure. When this procedure is called, it restores the stack, display, and registers, and then returns its argument as its value. Stack copying can be minimized by only copying the portion of the stack which has not already been copied into some other continuation and only restoring the portion of the stack which is not already in place.

While this scheme requires copying, it has two distinct advantages: there is no overhead for the programs which do not use explicit continuations and the mechanism is portable.

3.8. Summary

This completes discussion of the mapping of Scheme's semantics to C. Even though virtually all of Scheme's semantics were correctly handled, the implementation cannot properly handle all tail-calls. While unfortunate, the discussions which follow on performance and portability suggest that this is not a fatal flaw.

4. Performance in C

While compiling Scheme to C may provide a very portable system, one's first inclination is to expect that there is a performance penalty. One test that was run suggests that this is not the case. The Gabriel benchmarks [6] were run on a MicroVax II and Scheme->C's performace was compared against that of a native Lisp implementation, VAX Common Lisp V2.2 as reported in [14].

When the benchmarks were compiled using the standard ULTRIX C compiler, the geometric mean of the run-times showed that they were 5% slower than VAX Common Lisp. Better results were obtained when the benchmarks were compiled with the VAX C compiler: the geometric mean of the run-times in this case was the same as with VAX Common Lisp. The best results were obtained when the benchmarks were compiled with the GNU C compiler. Here, the geometric mean of the times showed that Scheme->C was 8% faster than VAX Common Lisp. The results of the individual benchmarks are shown in Figure 1. For each benchmark, there is a data point for each compiler[9] that shows the speed relative to VAX Common Lisp. Values to the left of 1.0 show performance less than VAX Common Lisp, values to the right show greater performance.

Much of the variance in performance in the individual benchmarks is probably due to differences in the systems. The block compilation model used by Scheme->C and its tail-call analysis probably help performance. On the other hand it loses some performance, in that it uses

[9]Except for BOYER, where the VAX C compiler was unable to compile the benchmark.

the standard procedure call and return instructions, has no floating point optimization, and `cons` is implemented by a library procedure rather than in-line code.

In spite of these differences, one can still conclude that performance need not suffer by compiling to C, and the times obtained when compiling with GNU C indicate that it may be possible to gain. With the advent of powerful C compilers, and RISC machines such as the MIPS R2000 which benefit greatly from extensive optimization, compiling to C as is done here and in Kyoto Common Lisp (KCL) may be preferable to retargetting a common backend when building a portable compiler.

5. Porting Scheme->C

The Scheme->C system makes few requirements on the target system other than it have a C compiler. The tagged pointer scheme requires that processor addresses be 32-bits in length and denote the first byte of the addressed object. It does not care about byte-order in a word. While UNIX systems typically have one address space for code and data, the implementation does not require it. In order to save and restore state for `call-with-current-continuation`, there must be a way to save and restore the process's stack and registers.

The Scheme->C system is broken up into processor dependent and independent modules as follows:

- A Scheme-to-C compiler written in Scheme which emits machine-independent C code.
- A machine-dependent definitions file which is used to compile the C code produced by the compiler to produce a machine-dependent object file.
- Machine-independent portions of the run-time library written in C and Scheme.
- Machine-dependent portions of the run-time library written in C and assembly language.

Organizing the modules in this manner simplifies porting the compiler to a new system as no host system with an existing implementation is needed. Instead, all work can be done on the target system: the machine-dependent parts of the run-time system are changed, and then the entire system is compiled using the system's C compiler. Following the initial implementation of the compiler on the Titan workstation developed by WRL [12], it was ported to the Digital VAX, and the DECstation 3100 which uses a MIPS R2000 processor. Each of these ports took less than one week and required less than 100 lines of machine specific code.

In spite of the fact that there is only a small amount of machine dependent code, machine specific optimizations that have a high payoff can still be done. For example, the Titan automatically ignores the low-order two bits of an address when a word in memory is accessed. Thus on a Titan, the type tag need not be masked off before a memory access, and the `car` of a pair is accessed:

```
#define PAIR_CAR( tscp ) ( ((SCP)( tscp ))->pair.car )
```

while on the VAX, the tag must be explicitly deleted so the code is:

```
#define PAIR_CAR( tscp ) ( ((SCP)((char*)tscp-PAIRTAG))->pair.car )
```

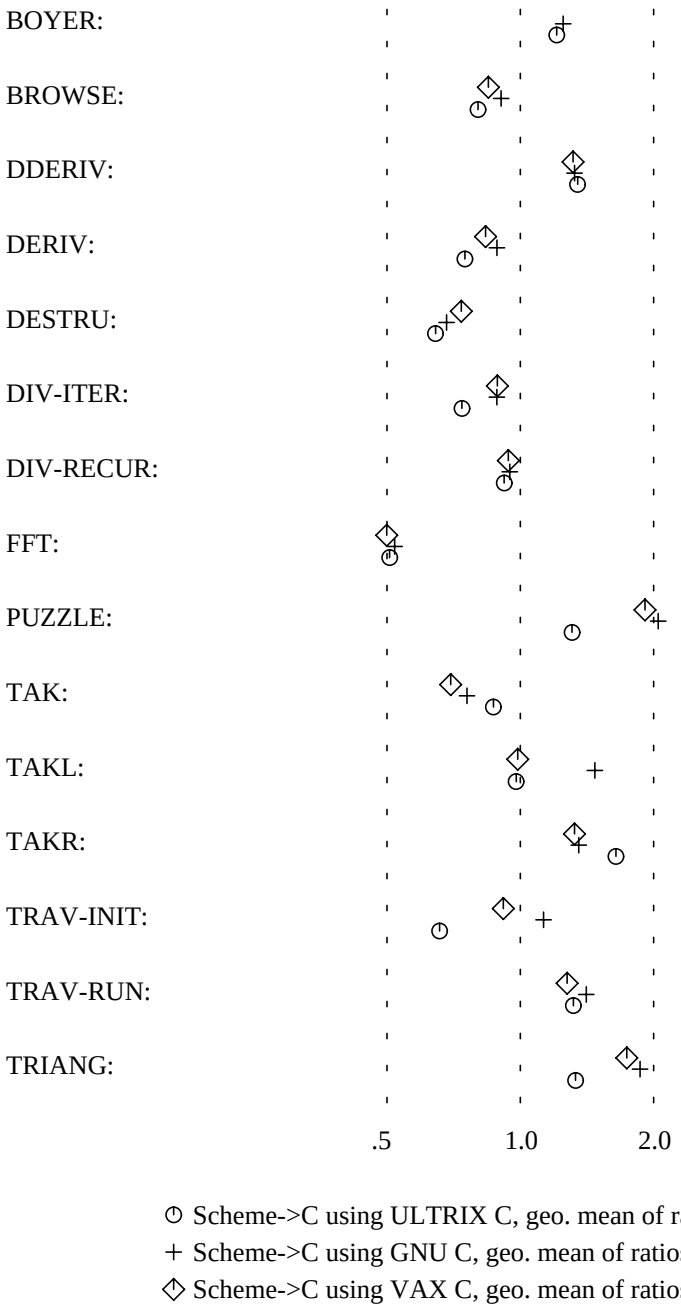

Figure 1: Performance of Scheme->C relative to VAX Common Lisp V2.2

6. Conclusion

A strong theme throughout this mapping of Scheme to C has been to try to use as much of C's facilities as possible. By doing so, the implementation is portable, and a good C compiler can produce high-performance code. C's procedure call/return mechanism was retained for just this reason. While doing so makes the implementation not properly tail-recursive, it does allow C

code produced by the Scheme compiler to be compatible with other C programs or code written in other languages. In a similar vein, the garbage collection algorithm used allows Scheme data structures to be mixed with other languages' data structures.

In spite of the success of this effort, Scheme->C is not for everyone. Its greatest weakness is that it is not a LISP environment. It does not provide any of the tools that an experienced LISP programmer expects. On the other hand, the fact that Scheme->C consists of a compiler and run-time system which can be used with other parts of the computing system is its greatest strength. By combining it with other tools and languages, it is possible to construct stand-alone programs or embedded applications in conventional environments.

7. Acknowledgements

The introduction to Scheme in section 2 used material from the *Revised^3 Report on the Algorithmic Language Scheme*, who's authors have made available to the Scheme community. The Gabriel benchmarks were translated to Scheme by Will Clinger. Bob Alverson, Jon Bertoni, Mary Jo Doherty, Michael Fetterman, John Ousterhout, and David W. Wall carefully read this paper and their insightful comments greatly improved it.

I thank you all.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots*. Technical Report WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, February, 1988. Also published in LISP Pointers, Vol. 1, No. 6.

[3] Frederick C. Chow and Mahadevan Ganapathi. Intermediate Languages in Compiler Construction - A Bibliography. *SIGPLAN Notices* 18(11):21-23, November, 1983.

[4] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation Strategies for Continuations. In *1988 ACM Conference on LISP and Functional Programming*, pages 124-131. August, 1988.

[5] Jacques Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys* 13(3):341-367, September, 1981.

[6] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* The MIT Press, 1985.

[7] Christopher T. Haynes and Daniel P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages* 9(4):582-598, October, 1987.

[8] Gerry Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, Inc., 1987.

[9] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., 1978.

[10] Brian W. Kernighan, Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Inc., 1984.

[11] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219-233. July, 1986.

[12] Michael J. K. Nielsen. *Titan System Manual*. Technical Report WRL Research Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[13] Jonathan Rees, William Clinger (Editors). Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21(12):37-79, December, 1986.

[14] Walter van Roggen. Lisp Implementations. *LISP Pointers* 1(5):37-42, December, 1987.

[15] Guy L. Steele Jr. *RABBIT: a Compiler for Scheme*. Technical Report AI Memo No. 452, Massachusetts Institute of Technology, May, 1978.

[16] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[17] Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, T. Steel. The Problem of Programming Communication with Changing Machines: A Proposed Solution. *Communications of the ACM* 1(8):12-18, August, 1958. Part 2: 1:9 9-15.

Table of Contents

List of Figures