An Incremental Approach to Compiler Construction

Abdulaziz Ghuloum
Department of Computer Science, Indiana University, Bloomington, IN 47408
aghuloum@cs.indiana.edu

Abstract

Compilers are perceived to be magical artifacts, carefully crafted by the wizards, and unfathomable by the mere mortals. Books on compilers are better described as wizard-talk: written by and for a clique of all-knowing practitioners. Real-life compilers are too complex to serve as an educational tool. And the gap between real-life compilers and the educational toy compilers is too wide. The novice compiler writer stands puzzled facing an impenetrable barrier, "better write an interpreter instead."

The goal of this paper is to break that barrier. We show that building a compiler can be as easy as building an interpreter. The compiler we construct accepts a large subset of the Scheme programming language and produces assembly code for the Intel-x86 architecture, the dominant architecture of personal computing. The development of the compiler is broken into many small incremental steps. Every step yields a fully working compiler for a progressively expanding subset of Scheme. Every compiler step produces real assembly code that can be assembled then executed directly by the hardware. We assume that the reader is familiar with the basic computer architecture: its components and execution model. Detailed knowledge of the Intel-x86 architecture is not required.

The development of the compiler is described in detail in an extended tutorial. Supporting material for the tutorial such as an automated testing facility coupled with a comprehensive test suite are provided with the tutorial. It is our hope that current and future implementors of Scheme find in this paper the motivation for developing high-performance compilers and the means for achieving that goal.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; K.3.2 [*Computer and Information Science Education*]: Computer science education

Keywords Scheme, Compilers

1. Introduction

Compilers have traditionally been regarded as complex pieces of software. The perception of complexity stems mainly from traditional methods of teaching compilers as well as the lack of available examples of small and functional compilers for real languages.

Compiler books are polarized into two extremes. Some of them focus on "educational" toy compilers while the others focus on "industrial-strength" optimizing compilers. The toy compilers are too simplistic and do not prepare the novice compiler writer to construct a useful compiler. The source language of these compilers often lacks depth and the target machine is often fictitious. Niklaus Wirth states that "to keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice"[20]. On the other extreme, advanced books focus mainly on optimization techniques, and thus target people who are already well-versed in the topic. There is no gradual progress into the field of compiler writing.

The usual approach to introducing compilers is by describing the structure and organization of a finalized and polished compiler. The sequencing of the material as presented in these books mirrors the passes of the compilers. Many of the issues that a compiler writer has to be aware of are solved beforehand and only the final solution is presented. The reader is not engaged in the process of developing the compiler.

In these books, the sequential presentation of compiler implementation leads to loss of focus on the big picture. Too much focus is placed on the individual passes of the compiler; thus the reader is not actively aware of the relevance of a single pass to the other passes and where it fits in the whole picture. Andrew Appel states that "a student who implements all the phases described in Part I of the book will have a working compiler"[2]. Part I of Appel's book concludes with a 6-page chapter on "Putting it all together" after presenting 11 chapters on the different passes of Tiger.

Moreover, practical topics such as code generation for a real machine, interfacing to the operating system or to other languages, heap allocation and garbage collection, and the issues surrounding dynamic languages are either omitted completely or placed in an appendix. Muchnick states that "most of the compiler material in this book is devoted to languages that are well suited for compilation: languages that have static, compile-time type systems, that do not allow the user to incrementally change the code, and that typically make much heavier use of stack storage than heap storage"[13].

2. Preliminary Issues

To develop a compiler, there are a few decisions to be made. The source language, the implementation language, and the target architecture must be selected. The development time frame must be set. The development methodology and the final goal must be decided. For the purpose of our tutorial, we made the decisions presented below.

2.1 Our Target Audience

We do not assume that the reader knows anything about assembly language beyond the knowledge of the computer organization, memory, and data structures. The reader is assumed to have very limited or no experience in writing compilers. Some experience with writing simple interpreters is helpful, but not required.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

We assume that the reader has basic knowledge of `C` and the `C` standard library (e.g. `malloc`, `printf`, etc.). Although our compiler will produce assembly code, some functionality is easier to implement in `C`; implementing it directly as assembly routines distracts the reader from the more important tasks.

2.2 The Source Language

In our tutorial, we choose a subset of Scheme as the source programming language. The simple and uniform syntax of Scheme obviates the need for a lengthy discussion of scanners and parsers. The execution model of Scheme, with strict call-by-value evaluation, simplifies the implementation. Moreover, all of the Scheme primitives in the subset can be implemented in short sequences of assembly instructions. Although not all of Scheme is implemented in the first compiler, all the major compiler-related issues are tackled. The implementation is a middle-ground between a full Scheme compiler and a toy compiler.

In choosing a specific source language, we gain the advantage that the presentation is more concrete and eliminates the burden of making the connection from the abstract concepts to the actual language.

2.3 The Implementation Language

Scheme is chosen as the implementation language of the compiler. Scheme's data structures are simple and most Scheme programmers are familiar with the basic tasks such as constructing and processing lists and trees. The ability to manipulate Scheme programs as Scheme data structures greatly simplifies the first steps of constructing a compiler, since the issue of reading the input program is solved. Implementing a lexical-scanner and a parser are pushed to the end of the tutorial.

Choosing Scheme as the implementation language also eliminates the need for sophisticated and specialized tools. These tools add a considerable overhead to the initial learning process and distracts the reader from acquiring the essential concepts.

2.4 Choosing The Target Architecture

We choose the Intel-x86 architecture as our target platform. The x86 architecture is the dominant architecture on personal computers and thus is widely available.

Talking about compilers that are detached from a particular architecture puts the burden on the reader to make the connection from the abstract ideas to the concrete machine. Novice compiler writers are unlikely to be able to derive the connection on their own. Additionally, the compiler we develop is small enough to be easily portable to other architectures, and the majority of the compiler passes are platform independent.

2.5 Development Time Frame

The development of the compiler must proceed in small steps where every step can be implemented and tested in one sitting. Features that require many sittings to complete are broken down into smaller steps. The result of completing every step is a fully working compiler. The compiler writer, therefore, achieves progress in every step in the development. This is in contrast with the traditional development strategies that advocate developing the compiler as a series of passes only the last of which gives the sense of accomplishment. With our approach of incremental development, where every step results in a fully working compiler for some subset of Scheme, the risk of not "completing" the compiler is minimized. This approach is useful for people learning about compilers on their own, where the amount of time they can dedicate constantly changes. It is also useful in time-limited settings such as an academic semester.

2.6 Development Methodology

We advocate the following iterative development methodology:

1. Choose a small subset of the source language that we can compile directly to assembly.
2. Write as many test cases as necessary to cover the chosen subset of the language.
3. Write a compiler that accepts an expression (in the chosen subset of the source language) and outputs the equivalent sequence of assembly instructions.
4. Ensure that the compiler is functional, i.e. it passes all the tests that are written beforehand.
5. Refactor the compiler, if necessary, making sure that none of the tests are broken due to incorrect refactoring.
6. Enlarge the subset of the language in a very small step and repeat the cycle by writing more tests and extending the compiler to meet the newly-added requirements.

A fully working compiler for the given subset of the language is available at every step in the development cycle starting from the first day of development. The test cases are written to help ensure that the implementation meets the specifications and to guard against bugs that may be introduced during the refactoring steps. Knowledge of compilation techniques as well as the target machine is built incrementally. The initial overhead of learning the assembly instructions of the target machine is eliminated—instructions are introduced only when they are needed. The compiler starts small and is well focused on translating the source language to assembly, and every incremental step reinforces that focus.

2.7 Testing Infrastructure

The interface to the compiler is defined by one Scheme procedure, `compile-program`, that takes as input an s-expression representing a Scheme program. The output assembly is emitted using an `emit` form that routes the output of the compiler to an assembly file.

Defining the compiler as a Scheme procedure allows us to develop and debug the compiler interactively by inspecting the output assembly code. It also allows us to utilize an automated testing facility. There are two core components of the testing infrastructure: the test-cases and the test-driver.

The test cases are made of sample programs and their expected output. For example, the test cases for the primitive `+` may be defined as follows:

```
(test-section "Simple Addition")
(test-case '(+ 10 15) "25")
(test-case '(+ -10 15) "5")
...
```

The test-driver iterates over the test cases performing the following actions: (1) The input expression is passed to `compile-program` to produce assembly code. (2) The assembly code and a minimal run-time system (to support printing) are assembled and linked to form an executable. (3) The executable is run and the output is compared to the expected output string. An error is signaled if any of the previous steps fails.

2.8 The End Goal

For the purpose of this paper, we define the end goal to be writing a compiler powerful enough to compile an interactive evaluator. Building such a compiler forces us to solve many interesting problems.

A large subset of Scheme's core forms (`lambda`, `quote`, `set!`, etc) and extended forms (`cond`, `case`, `letrec`, internal `define` etc.) must be supported by the compiler. Although most of these forms are not essential, their presence allows us to write our programs in a more natural way. In implementing the extended forms, we show how a large number of syntactic forms can be added without changing the core language that the compiler supports.

A large collection of primitives (`cons`, `car`, `vector?`, etc.) and library procedures (`map`, `apply`, `list->vector`, etc.) need to be implemented. Some of these library procedures can be implemented directly, while others require some added support from the compiler. For example, some of the primitives cannot be implemented without supporting variable-arity procedures, and others require the presence of `apply`. Implementing a writer and a reader requires adding a way to communicate with an external run-time system.

3. Writing a Compiler in 24 Small Steps

Now that we described the development methodology, we turn our attention to the actual steps taken in constructing a compiler. This section is a brief description of 24 incremental stages: the first is a small language composed only of small integers, and the last covers most of the requirements of R^{5}RS. A more detailed presentation of these stages is in the accompanying extended tutorial.

3.1 Integers

The simplest language that we can compile and test is composed of the fixed-size integers, or fixnums. Let's write a small compiler that takes a fixnum as input and produces a program in assembly that returns that fixnum. Since we don't know yet how to do that, we ask for some help from another compiler that does know: gcc. Let's write a small `C` function that returns an integer:

```
int scheme_entry(){
  return 42;
}
```

Let's compile it using `gcc -O3 --omit-frame-pointer -S test.c` and see the output. The most relevant lines of the output file are the following:

```
1.          .text
2.          .p2align 4,,15
3.  .globl scheme_entry
4.          .type   scheme_entry, @function
5.  scheme_entry:
6.          movl    $42, %eax
7.          ret
```

Line 1 starts a text segment, where code is located. Line 2 aligns the beginning of the procedure at 4-byte boundaries (not important at this point). Line 3 informs the assembler that the `scheme_entry` label is global so that it becomes visible to the linker. Line 4 says that `scheme_entry` is a function. Line 5 denotes the start of the `scheme_entry` procedure. Line 6 sets the value of the `%eax` register to 42. Line 7 returns control to the caller, which expects the received value to be in the `%eax` register.

Generating this file from Scheme is straightforward. Our compiler takes an integer as input and prints the given assembly with the input substituted in for the value to be returned.

```
(define (compile-program x)
  (emit "movl $~a, %eax" x)
  (emit "ret"))
```

To test our implementation, we write a small `C` run-time system that calls our `scheme_entry` and prints the value it returns:

```
/* a simple driver for scheme_entry */
#include <stdio.h>
int main(int argc, char** argv){
  printf("%d\n", scheme_entry());
  return 0;
}
```

3.2 Immediate Constants

Values in Scheme are not limited to the fixnum integers. Booleans, characters, and the empty list form a collection of immediate values. Immediate values are those that can be stored directly in a machine word and therefore do not require additional storage. The types of the immediate objects in Scheme are disjoint, consequently, the implementation cannot use fixnums to denote booleans or characters. The types must also be available at run time to allow the driver to print the values appropriately and to allow us to provide the type predicates (discussed in the next step).

One way of encoding the type information is by dedicating some of the lower bits of the machine word for type information and using the rest of the machine word for storing the value. Every type of value is defined by a mask and a tag. The mask defines which bits of the integer are used for the type information and the tag defines the value of these bits.

For fixnums, the lower two bits ($mask = 11_b$) must be 0 ($tag = 00_b$). This leaves 30 bits to hold the value of a fixnum. Characters are tagged with 8 bits ($tag = 00001111_b$) leaving 24 bits for the value (7 of which are actually used to encode the ASCII characters). Booleans are given a 7-bit tag ($tag = 0011111_b$), and 1-bit value. The empty list is given the value 00101111_b.

We extend our compiler to handle the immediate types appropriately. The code generator must convert the different immediate values to the corresponding machine integer values.

```
(define (compile-program x)
  (define (immediate-rep x)
    (cond
      ((integer? x) (shift x fixnum-shift))
      ...))
  (emit "movl $~a, %eax" (immediate-rep x))
  (emit "ret"))
```

The driver must also be extended to handle the newly-added values. The following code illustrates the concept:

```
#include <stdio.h>
#define fixnum_mask       3
#define fixnum_tag        0
#define fixnum_shift      2
  ...
int main(int argc, char** argv){
  int val = scheme_entry();
  if((val & fixnum_mask) == fixnum_tag){
    printf("%d\n", val >> fixnum_shift);
  } else if(val == empty_list){
    printf("()\n");
  } ...
  return 0;
}
```

3.3 Unary Primitives

We extend the language now to include calls to primitives that accept one argument. We start with the simplest of these primitives: `add1` and `sub1`. To compile an expression in the form `(add1` e`)`, we first emit the code for e. That code would evaluate e placing its value in the `%eax` register. What remains to be done is incrementing

the value of the %eax register by 4 (the shifted value of 1). The machine instruction that performs addition/subtraction is addl/subl.

```
(define (emit-expr x)
  (cond
    ((immediate? x)
     (emit "movl $~a, %eax" (immediate-rep x)))
    ((primcall? x)
     (case (primcall-op x)
       ((add1)
        (emit-expr (primcall-operand1 x))
        (emit "addl $~a, %eax" (immediate-rep 1)))
       ...))
    (else ...)))
```

The primitives integer->char and char->integer can be added next. To convert an integer (assuming it's in the proper range) to a character, the integer (already shifted by 2 bits) is shifted further 6 bits to make up a total of *char-shift*, the result is then tagged with the *char-tag*. Converting a character to a fixnum requires a shift to the right by 6 bits. The choice of tags for the fixnums and characters is important for realizing this concise and potentially fast conversion.

We implement the predicates null?, zero?, and not next. There are many possible ways of implementing each of these predicates. The following sequence works for zero? (assuming the value of the operand is in %eax):

```
1.    cmpl   $0, %eax
2.    movl   $0, %eax
3.    sete   %al
4.    sall   $7, %eax
5.    orl    $63, %eax
```

Line 1 compares the value of %eax to 0. Line 2 zeros the value of %eax. Line 3 sets %al, the low byte of %eax, to 1 if the two compared values were equal, and to 0 otherwise. Lines 4 and 5 construct the appropriate boolean value from the one bit in %eax.

The predicates integer? and boolean? are handled similarly with the exception that the tag of the value must be extracted (using andl) before it is compared to the fixnum/boolean tag.

3.4 Binary Primitives

Calls to binary, and higher-arity, primitives cannot in general be evaluated using a single register since evaluating one subexpression may overwrite the value computed for the other subexpression. To implement binary primitives (such as +, *, char<?, etc.), we use a stack to save intermediate values of computations. For example, generating the code for (+ e_0 e_1) is achieved by (1) emitting the code for e_1, (2) emitting an instruction to save the value of %eax on the stack, (3) emitting the code for e_0, and (4) adding the value of %eax to the value saved on the stack.

The stack is arranged as a contiguous array of memory locations. A pointer to the base of the stack is in the %esp register. The base of the stack, 0(%esp), contains the return-point. The return-point is an address in memory where we return after computing the value and therefore should not be modified. We are free to use the memory locations above the return-point (-4(%esp), -8(%esp), -12(%esp), etc.) to hold our intermediate values.

In order to guarantee never overwriting any value that will be needed after the evaluation of an expression, we arrange the code generator to maintain the value of the stack index. The stack index is a negative number that points to the first stack location that is free. The value of the stack index is initialized to -4 and is decremented by 4 (the word-size, 4 bytes) every time a new value is saved on the stack. The following segment of code illustrates how the primitive + is implemented:

```
(define (emit-primitive-call x si)
  (case (primcall-op x)
    ((add1) ...)
    ((+)
     (emit-expr (primcall-operand2 x) si)
     (emit "movl %eax, ~a(%esp)" si)
     (emit-expr
       (primcall-operand1 x)
       (- si wordsize))
     (emit "addl ~a(%esp), %eax" si))
    ...))
```

The other primitives (-, *, =, <, char=?, etc.) can be easily implemented by what we know so far.

3.5 Local Variables

Now that we have a stack, implementing let and local variables is straightforward. All local variables will be saved on the stack and an environment mapping variables to stack locations is maintained. When the code generator encounters a let-expression, it first evaluates the right-hand-side expressions, one by one, saving the value of each in a specific stack location. Once all the right-hand-sides are evaluated, the environment is extended to associate the new variables with their locations, and code for the body of the let is generated in the new extended environment. When a reference to a variable is encountered, the code generator locates the variable in the environment, and emits a load from that location.

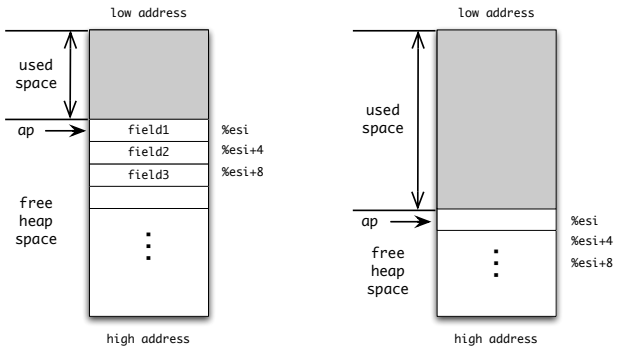
```
(define (emit-expr x si env)
  (cond
    ((immediate? x) ...)
    ((variable? x)
     (emit "movl ~a(%esp), %eax" (lookup x env)))
    ((let? x)
     (emit-let (bindings x) (body x) si env))
    ((primcall? x) ...)
    ...))

(define (emit-let bindings body si env)
  (let f ((b* bindings) (new-env env) (si si))
    (cond
      ((null? b*) (emit-expr body si new-env))
      (else
       (let ((b (car b*)))
         (emit-expr (rhs b) si env)
         (emit "movl %eax, ~a(%esp)" si)
         (f (cdr b*)
            (extend-env (lhs b) si new-env)
            (- si wordsize)))))))
```

3.6 Conditional Expressions

Conditional evaluation is simple at the assembly-level. The simplest implementation of (if test conseq altern) is:

```
(define (emit-if test conseq altern si env)
  (let ((L0 (unique-label)) (L1 (unique-label)))
    (emit-expr test si env)
    (emit-cmpl (immediate-rep #f) eax)
    (emit-je L0)
    (emit-expr conseq si env)
    (emit-jmp L1)
    (emit-label L0)
    (emit-expr altern si env)
    (emit-label L1)))
```

The code above first evaluates the test expression and compares the result to the false value. Control is transferred to the alternate



(A) Before allocation (B) After allocation

Figure 1. Illustration of the heap. The allocation pointer (ap) is is held in the `%esi` register and its value is always aligned on 8-byte boundaries. Individual objects are allocated at address `%esi` and the allocation pointer is bumped to the first boundary after the object.

code if the value of the test was false, otherwise, it falls through to the consequent.

3.7 Heap Allocation

Scheme's pairs, vector, strings, etc. do not fit in one machine word and must be allocated in memory. We allocate all objects from one contiguous area of memory. The heap is preallocated at the start of the program and its size is chosen to be large enough to accommodate our current needs. A pointer to the beginning of the heap is passed to `scheme_entry` to serve as the allocation pointer. We dedicate one register, `%esi`, to hold the allocation pointer. Every time an object is constructed, the value of `%esi` is incremented according to the size of the object.

The types of the objects must also be distinguishable from one another. We use a tagging scheme similar to the one used for fixnums, booleans, and characters. Every pointer to a heap-allocated object is tagged by a 3-bit tag (001_b for pairs, 010_b for vectors, 011_b for strings, 101_b for symbols, and 110_b for closures; 000_b, 100_b and 111_b were already used for fixnums and the other immediate objects). For this tagging scheme to work, we need to guarantee that the lowest three bits of every heap-allocated object is 000_b so that the tag and the value of the pointer do not interfere. This is achieved by always allocating objects at double-word (or 8-byte) boundaries.

Let's consider how pairs are implemented first. A pair requires two words of memory to hold its `car` and `cdr` fields. A call to `(cons 10 20)` can be translated to:

```
movl  $40,  0(%esi)        # set the car
movl  $80,  4(%esi)        # set the cdr
movl  %esi, %eax           # eax = esi | 1
orl   $1,   %eax
addl  $8,   %esi           # bump esi
```

The primitives `car` and `cdr` are simple; we only need to remember that the pointer to the pair is its address incremented by 1. Consequently, the `car` and `cdr` fields are located at -1 and 3 from the pointer. For example, the primitive `caddr` translates to:

```
movl  3(%eax),  %eax       # cdr
movl  3(%eax),  %eax       # cddr
movl  -1(%eax), %eax       # caddr
```

Vectors and strings are different from pairs in that they vary in length. This has two implications: (1) we must reserve one extra memory location in the vector/string to hold the length, and (2) after allocating the object, the allocation pointer must be aligned to the next double-word boundary (allocating pairs was fine because their size is a multiple of 8). For example, a call to the primitive `make-vector` translates to:

```
movl  %eax, 0(%esi)        # set the length
movl  %eax, %ebx           # save the length
movl  %esi, %eax           # eax = esi | 2
orl   $2,   %eax
addl  $11,  %ebx           # align size to next
andl  $-8,  %ebx           #    object boundary
addl  %ebx, %esi           # advance alloc ptr
```

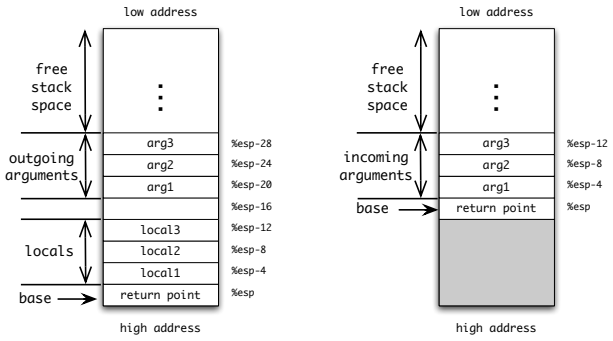
Strings are implemented similarly except that the size of a string is smaller than the size of a vector of the same length. The primitive `string-ref` (and `string-set!`) must also take care of converting a byte value to a character (and vise versa).

3.8 Procedure Calls

The implementation of procedures and procedure calls are perhaps the hardest aspect of constructing our compiler. The reason for its difficulty is that Scheme's `lambda` form performs more than one task and the compiler must tease these tasks apart. First, a `lambda` expression closes over the variables that occur free in its body so we must perform some analysis to determine the set of variables that are referenced, but not defined, in the body of a `lambda`. Second, `lambda` constructs a closure object that can be passed around. Third, the notion of procedure calls and parameter-passing must be introduced at the same point. We'll handle these issues one at a time starting with procedure calls and forgetting all about the other issues surrounding `lambda`.

We extend the language accepted by our code generator to contain top-level labels (each bound to a code expression containing a list of formal parameters and a body expression) and label calls.

```
<Prog>  ::= (labels ((lvar <LExpr>) ...) <Expr>)
<LExpr> ::= (code (var ...) <Expr>)
<Expr>  ::= immediate
          | var
          | (if <Expr> <Expr> <Expr>)
          | (let ((var <Expr>) ...) <Expr>)
          | (primcall prim-name <Expr> ...)
          | (labelcall lvar <Expr> ...)
```



(A) Caller's View (B) Callee's View

Figure 2. The view of the stack from (A) the Caller's side before making the procedure call, and (B) the Callee's side on entry to the procedure.

Code generation for the new forms is as follows:

- For the `labels` form, a new set of unique labels are created and the initial environment is constructed to map each of the `lvars` to its corresponding label.
- For each `code` expression, the label is first emitted, followed by the code of the body. The environment used for the body contains, in addition to the `lvars`, a mapping of each of the formal parameters to the first set of stack locations (-4, -8, etc.). The stack index used for evaluating the body starts above the last index used for the formals.
- For a `(labelcall lvar e ...)`, the arguments are evaluated and their values are saved in consecutive stack locations, skipping one location to be used for the return-point. Once all of the arguments are evaluated, the value of the stack-pointer, `%esp` is incremented to point to one word below the return-point. A `call` to the label associated with the `lvar` is issued. A `call` instruction decrements the value of `%esp` by 4 and saves the address of the next instruction in the appropriate return-point slot. Once the called procedure returns (with a value in `%eax`), the stack pointer is adjusted back to its initial position.

Figure 2 illustrates the view of the stack from the caller and callee perspective.

3.9 Closures

Implementing closures on top of what we have so far should be straightforward. First, we modify the language accepted by our code generator as follows:

- The form `(closure lvar var ...)` is added to the language. This form is responsible for constructing closures. The first cell of a closure contains the label of a procedure, and the remaining cells contain the values of the free variables.
- The `code` form is extended to contain a list of the free variables in addition to the existing formal parameters.
- The `labelcall` is replaced by a `funcall` form that takes an arbitrary expression as a first argument instead of an `lvar`.

The `closure` form is similar to a call to `vector`. The label associated with the `lvar` is stored at `0(%esi)` and the values of the variables are stored in the next locations. The value of `%esi` is tagged to get the value of the closure, and `%esi` is bumped by the required amount.

The `code` form, in addition to associating the formals with the corresponding stack locations, associates each of the free variables with their displacement form the closure pointer `%edi`.

The `funcall` evaluated all the arguments as before but skips not one but two stack locations: one to be used to save the current value of the closure pointer, and one for the return point. After the arguments are evaluated and saved, the operator is evaluated, and its value is moved to `%edi` (whose value must be saved to its stack location). The value of `%esp` is adjusted and an indirect call through the first cell of the closure pointer is issued. Upon return from the call, the value of `%esp` is adjusted back and the value of `%edi` is restored from the location at which it was saved.

One additional problem needs to be solved. The source language that our compiler accepts has a `lambda` form, and none of the `labels`, `code`, `closure` forms. So, Scheme input must be converted to this form before our code generator can accept it. The conversion is easy to do in two steps:

1. Free-variable analysis is performed. Every `lambda` expression appearing in the source program is annotated with the set of variables that are referenced but not defined in the body of the `lambda`. For example,

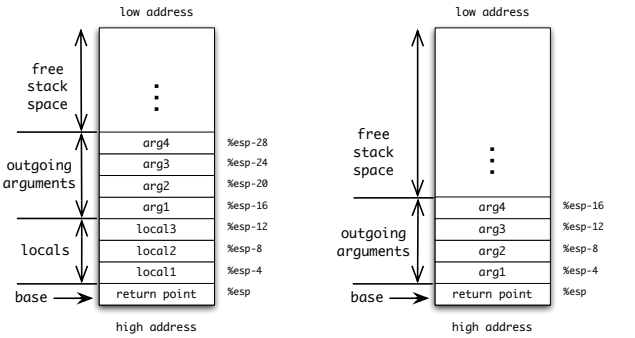


Figure 3. One way of implementing proper tail calls is by collapsing the tail frame. The figures show (A) the evaluation and placement of the arguments on the stack above the local variables, then (B) moving the arguments down to overwrite the current frame immediately before making the tail jump.

```
(let ((x 5))
  (lambda (y) (lambda () (+ x y))))
```

is transformed to:

```
(let ((x 5))
  (lambda (y) (x) (lambda () (x y) (+ x y))))
```

2. The `lambda` forms are transformed into `closure` forms and the codes are collected at the top. The previous example yields:

```
(labels ((f0 (code () (x y) (+ x y)))
         (f1 (code (y) (x) (closure f0 x y))))
  (let ((x 5)) (closure f1 x)))
```

3.10 Proper Tail Calls

The Scheme report requires that implementations be properly tail-recursive. By treating tail-calls properly, we guarantee that an unbounded number of tail calls can be performed in constant space.

So far, our compiler would compile tail-calls as regular calls followed by a return. A proper tail-call, on the other hand, must perform a `jmp` to the target of the call, using the same stack position of the caller itself.

A very simple way of implementing tail-calls is as follows (illustrated in Figure 3):

1. All the arguments are evaluated and saved on the stack in the same way arguments to nontail calls are evaluated.
2. The operator is evaluated and placed in the `%edi` register replacing the current closure pointer.
3. The arguments are copied from their current position of the stack to the positions adjacent to the return-point at the base of the stack.
4. An indirect `jmp`, not `call`, through the address in the closure pointer is issued.

This treatment of tail calls is the simplest way of achieving the objective of the requirement. Other methods for enhancing performance by minimizing the excessive copying are discussed later in Section 4.

3.11 Complex Constants

Scheme's constants are not limited to the immediate objects. Using the `quote` form, lists, vectors, and strings can be turned into constants as well. The formal semantics of Scheme require that quoted constants always evaluate to the same object. The following example must always evaluate to true:

```
(let ((f (lambda () (quote (1 . "H")))))
  (eq? (f) (f)))
```

So, in general, we *cannot* transform a quoted constant into an unquoted series of constructions as the following incorrect transformation demonstrates:

```
(let ((f (lambda () (cons 1 (string #\H)))))
  (eq? (f) (f)))
```

One way of implementing complex constants is by lifting their construction to the top of the program. The example program can be transformed to an equivalent program containing no complex constants as follows:

```
(let ((tmp0 (cons 1 (string #\H))))
  (let ((f (lambda () tmp0)))
    (eq? (f) (f))))
```

Performing this transformation before closure conversion makes the introduced temporaries occur as free variables in the enclosing `lambda`s. This increases the size of many closures, increasing heap consumption and slowing down the compiled programs.

Another approach for implementing complex constants is by introducing global memory locations to hold the values of these constants. Every complex constant is assigned a label, denoting its location. All the complex constants are initialized at the start of the program. Our running example would be transformed to:

```
(labels ((f0 (code () () (constant-ref t1)))
         (t1 (datum)))
  (constant-init t1 (cons 1 (string #\H)))
  (let ((f (closure f0)))
    (eq? (f) (f))))
```

The code generator should now be modified to handle the data labels as well as the two internal forms `constant-ref` and `constant-init`.

3.12 Assignment

Let's examine how our compiler treats variables. At the source level, variables are introduced either by `let` or by `lambda`. By the time we get to code generation, a third kind (free-variables) is there as well. When a `lambda` closes over a reference to a variable, we copied the *value* of the variable into a field in the closure. If more than one closure references the variable, each gets its own copy of the value. If the variable is assignable, then all references and assignments occurring in the code must reference/assign to the same location that holds the value of the the variable. Therefore, every assignable variable must be given one unique location to hold its value.

The way we treat assignment is by making the locations of assignable variables explicit. These locations cannot in general be stack-allocated due to the indefinite extent of Scheme's closures. So, for every assignable variable, we allocate space on the heap (a vector of size 1) to hold its value. An assignment to a variable x is rewritten as an assignment to the memory location holding x (via `vector-set!`) and references to x are rewritten as references to the location of x (via `vector-ref`).

The following example illustrates assignment conversion when applied to a program containing one assignable variable `c`:

```
(let ((f (lambda (c)
           (cons (lambda (v) (set! c v))
                 (lambda () c)))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
=>
(let ((f (lambda (t0)
           (let ((c (vector t0)))
             (cons (lambda (v) (vector-set! c 0 v))
                   (lambda () (vector-ref c 0)))))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
```

3.13 Extending the Syntax

With most of the core forms (`lambda`, `let`, `quote`, `if`, `set!`, constants, variables, procedure calls, and primitive calls) in place, we can turn to extending the syntax of the language. The input to our compiler is preprocessed by a pass, a macro-expander, which performs the following tasks:

- All the variables are renamed to new unique names through α-conversion. This serves two purposes. First, making all variables unique eliminates the ambiguity between variables. This makes the analysis passes required for closure and assignment conversion simpler. Second, there is no fear of confusing the core forms with procedure calls to local variables with the same name (e.g. an occurrence of `(lambda (x) x)` where `lambda` is a lexical variable).
- Additionally, this pass places explicit tags on all internal forms including function calls (`funcall`) and primitive calls (`primcall`).
- Extended forms are simplified to the code forms. The forms `let*`, `letrec`, `letrec*`, `cond`, `case`, `or`, `and`, `when`, `unless`, and internal `define` are rewritten in terms of the core forms.

3.14 Symbols, Libraries, and Separate Compilation

All of the primitives that we supported so far were simple enough to be implemented directly in the compiler as a sequence of assembly instructions. This is fine for the simple primitives, such as `pair?` and `vector-ref`, but it will not be practical for implementing more complex primitives such as `length`, `map`, `display`, etc..

Also, we restricted our language to allow primitives to occur only in the operator position: passing the value of the primitive `car` was not allowed because `car` has no value. One way of fixing this is by performing an inverse-η transformation:

$$\texttt{car} \Rightarrow \texttt{(lambda (x) (car x))}.$$

This approach has many disadvantages. First, the resulting assembly code is bloated with too many closures that were not present in the source program. Second, the primitives cannot be defined recursively or defined by using common helpers.

Another approach for making an extended library available is by wrapping the user code with a large `letrec` that defines all the primitive libraries. This approach is discouraged because the intermixing of user-code and library-code hinders our ability to debug our compiler.

A better approach is to define the libraries in separate files, compiling them independently, and linking them directly with the user code. The library primitives are initialized before control enters the user program. Every primitive is given a global location, or a label, to hold its value. We modify our compiler to handle two additional forms: `(primitive-ref` x`)` and `(primitive-set!` x v`)` which are analogous to `constant-ref` and `constant-init` that

we introduced in 3.11. The only difference is that global labels are used to hold the values of the primitives.

The first library file initializes one primitive: `string->symbol`. Our first implementation of `string->symbol` need not be efficient: a simple linked list of symbols suffices. The primitive `string->symbol`, as its name suggests, takes a string as input and returns a symbol. By adding the core primitives `make-symbol`[1] and `symbol-string`, the implementation of `string->symbol` simply traverses the list of symbols looking for one having the same string. A new symbol is constructed if one with the same name does not exist. This new symbol is then added to the list before it is returned.

Once `string->symbol` is implemented, adding symbols to our set of valid complex constants is straightforward by the following transformation:

```
  (labels ((f0 (code () () 'foo)))
    (let ((f (closure f0)))
      (eq? (funcall f) (funcall f))))
=>
  (labels ((f0 (code () () (constant-ref t1)))
           (t1 (datum)))
    (constant-init t1
      (funcall (primitive-ref string->symbol)
               (string #\f #\o #\o)))
    (let ((f (closure f0)))
      (eq? (funcall f) (funcall f))))
```

3.15 Foreign Functions

Our Scheme implementation cannot exist in isolation. It needs a way of interacting with the host operating system in order to perform Input/Output and many other useful operations. We now add a very simple way of calling to foreign `C` procedures.

We add one additional form to our compiler:

```
<Expr> ::= (foreign-call <string> <Expr> ...)
```

The `foreign-call` form takes a string literal as the first argument. The string denotes the name of the `C` procedure that we intend to call. Each of the expressions are evaluated first and their values are passed as arguments to the `C` procedure. The calling convention for `C` differs from the calling convention that we have been using for Scheme in that the arguments are placed below the return point and in reverse order. Figure 4 illustrates the difference.

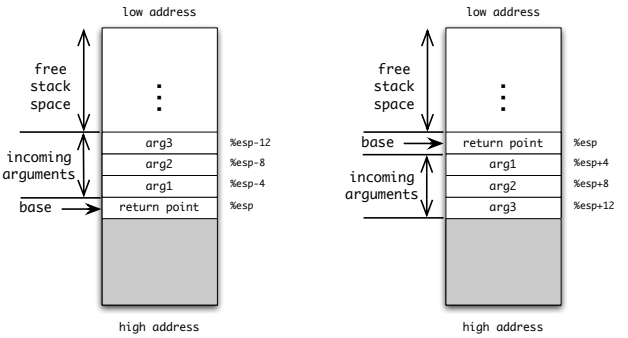
To accommodate the `C` calling conventions, we evaluate the arguments to a `foreign-call` in reverse order, saving the values on the stack, adjusting the value of `%esp`, issuing a `call` to the named procedure, then adjusting the stack pointer back to its initial position. We need not worry about the `C` procedure clobbering the values of the allocation and closure pointer because the Application Binary Interface (ABI) guarantees that the callee would preserve the values of the `%edi`, `%esi`, `%ebp` and `%esp` registers[14].

Since the values we pass to the foreign procedure are tagged, we would write wrapper procedures in our run-time file that take care of converting from Scheme to `C` values and back.

We first implement and test calling the `exit` procedure. Calling `(foreign-call "exit" 0)` should cause our program to exit without performing any output. We also implement a wrapper around `write` as follows:

```
ptr s_write(ptr fd, ptr str, ptr len){
  int bytes = write(unshift(fd),
                    string_data(str),
                    unshift(len));
  return shift(bytes);
}
```

[1] Symbols are similar to pairs in having two fields: a string and a value



(A) Parameter passing to Scheme (B) Parameter passing to `C`

Figure 4. The parameters to Scheme functions are placed on the stack above the return point while the parameters to `C` functions are placed below the return point.

3.16 Error Checking and Safe Primitives

Using our newly acquired ability to `write` and `exit`, we can define a simple `error` procedure that takes two arguments: a symbol (denoting the caller of `error`), and a string (describing the error). The `error` procedure would write an error message to the console, then causes the program to exit.

With `error`, we can secure some parts of our implementation to provide better debugging facilities. Better debugging allows us to progress with implementing the rest of the system more quickly since we won't have to hunt for the causes of segfaults.

There are three main causes of fatal errors:

1. Attempting to call non-procedures.
2. Passing an incorrect number of arguments to a procedure.
3. Calling primitives with invalid arguments. For example: performing `(car 5)` causes an immediate segfault. Worse, performing `vector-set!` with an index that's out of range causes other parts of the system to get corrupted, resulting in hard-to-debug errors.

Calling nonprocedures can be handled by performing a procedure check before making the procedure call. If the operator is not a procedure, control is transferred to an error handler label that sets up a call to a procedure that reports the error and exits.

Passing an incorrect number of arguments to a procedure can be handled by a collaboration from the caller and the callee. The caller, once it performs the procedure check, sets the value of the `%eax` register to be the number of arguments passed. The callee checks that the value of `%eax` is consistent with the number of arguments it expects. Invalid arguments cause a jump to a label that calls a procedure that reports the error and exits.

For primitive calls, we can modify the compiler to insert explicit checks at every primitive call. For example, `car` translates to:

```
   movl %eax, %ebx
   andl $7, %ebx
   cmpl $1, %ebx
   jne L_car_error
   movl -1(%eax), %eax
   ...
L_car_error:
   movl car_err_proc, %edi    # load handler
   movl $0, %eax              # set arg-count
   jmp *-3(%edi)              # call the handler
   ...
```

Another approach is to restrict the compiler to unsafe primitives. Calls to safe primitives are not open-coded by the compiler, instead, a procedure call to the safe primitive is issued. The safe primitives are defined to perform the error checks themselves. Although this strategy is less efficient than open-coding the safe primitives, the implementation is much simpler and less error-prone.

3.17 Variable-arity Procedures

Scheme procedures that accept a variable number of arguments are easy to implement in the architecture we defined so far. Suppose a procedure is defined to accept two or more arguments as in the following example:

```
(let ((f (lambda (a b . c) (vector a b c))))
  (f 1 2 3 4))
```

The call to `f` passes four arguments in the stack locations `%esp-4`, `%esp-8`, `%esp-12`, and `%esp-16` in addition to the number of arguments in `%eax`. Upon entry of `f`, and after performing the argument check, `f` enters a loop that constructs a list of the arguments last to front.

Implementing variable-arity procedures allows us to define many library procedures that accept any number of arguments including `+`, `-`, `*`, `=`, `<`, ..., `char=?`, `char<?`, ..., `string=?`, `string<?`, ..., `list`, `vector`, `string`, and `append`.

Other variations of `lambda` such as `case-lambda`, which allows us to dispatch different parts of the code depending on the number of actual arguments, can be implemented easily and efficiently by a series of comparisons and conditional jumps.

3.18 Apply

The implementation of the `apply` primitive is analogous to the implementation of variable-arity procedures. Procedures accepting variable number of arguments convert the extra arguments passed on the stack to a list. Calling `apply`, on the other hand, splices a list of arguments onto the stack.

When the code generator encounters an `apply` call, it generates the code in the same manner as if it were a regular procedure call. The operands are evaluated and saved in their appropriate stack locations as usual. The operator is evaluated and checked. In case of nontail calls, the current closure pointer is saved and the stack pointer is adjusted. In case of tail calls, the operands are moved to overwrite the current frame. The number of arguments is placed in `%eax` as usual. The only difference is that instead of calling the procedure directly, we `call/jmp` to the `L_apply` label which splices the last argument on the stack before transferring control to the destination procedure.

Implementing `apply` makes it possible to define the library procedures that take a function as well as an arbitrary number of arguments such as `map` and `for-each`.

3.19 Output Ports

The functionality provided by our compiler so far allows us to implement output ports easily in Scheme. We represent output ports by vector containing the following fields:

0. A unique identifier that allows us to distinguish output ports from ordinary vectors.
1. A string denoting the file name associated with the port.
2. A file-descriptor associated with the opened file.
3. A string that serves as an output buffer.
4. An index pointing to the next position in the buffer.
5. The size of the buffer.

The `current-output-port` is initialized at startup and its file descriptor is 1 on Unix systems. The buffers are chosen to be sufficiently large (4096 characters) in order to reduce the number of trips to the operating system. The procedure `write-char` writes to the buffer, increments the index, and if the index of the port reaches its size, the contents of the buffer are flushed using `s_write` (from 3.15) and the index is reset. The procedures `output-port?`, `open-output-file`, `close-output-port`, and `flush-output-port` are also implemented.

3.20 Write and Display

Once `write-char` is implemented, implementing the procedures `write` and `display` becomes straightforward by dispatching on the type of the argument. The two procedures are identical except for their treatment of strings and characters and therefore can be implemented in terms of one common procedure. In order to write the fixnums, the primitive `quotient` must be added to the compiler.

Implementing `write` in Scheme allows us to eliminate the now-redundant writer that we implemented as part of the `C` run-time system.

3.21 Input Ports

The representation of input ports is very similar to output ports. The only difference is that we add one extra field to support "unreading" a character which adds very minor overhead to the primitives `read-char` and `peek-char`, but greatly simplifies the implementation of the tokenizer (next step). The primitives added at this stage are `input-port?`, `open-input-file`, `read-char`, `unread-char`, `peek-char`, and `eof-object?` (by adding a special end-of-file object that is similar to the empty-list).

3.22 Tokenizer

In order to implement the `read` procedure, we first implement `read-token`. The procedure `read-token` takes an input port as an argument and using `read-char`, `peek-char`, and `unread-char`, it returns the next token. Reading a token involves writing an deterministic finite-state automata that mimics the syntax of Scheme. The return value of `read-token` is one of the following:

- A pair `(datum . `x`)` where x is a fixnum, boolean, character, string, or symbol that was encountered next while scanning the port.
- A pair `(macro . `x`)` where x denotes one of Scheme's predefined reader-macros: `quote`, `quasiquote`, `unquote`, or `unquote-splicing`.
- A symbol `left-paren`, `right-paren`, `vec-paren`, or `dot` denoting the corresponding non-datum token encountered.
- The end-of-file object if `read-char` returns the end-of-file object before we find any other tokens.

3.23 Reader

The `read` procedure is built as a recursive-descent parser on top of `read-token`. Because of the simplicity of the syntax (i.e. the only possible output is the eof-object, data, lists, and vectors) the entire implementation, including error checking, should not exceed 40 lines of direct Scheme code.

3.24 Interpreter

We have all the ingredients required for implementing an environment-passing interpreter for core Scheme. Moreover, we can lift the first pass of the compiler and make it the first pass to the interpreter as well. We might want to add some restriction to the language of the interpreter (i.e. disallowing `primitive-set!`) in order to prevent

the user code from interfering with the run-time system. We might also like to add different binding modes that determine whether references to primitive names refer to the actual primitives or to the current top-level bindings and whether assignment to primitive names are allowed or not.

4. Beyond the Basic Compiler

There are several axes along which one can enhance the basic compiler. The two main axes of enhancements are the feature-axis and the performance-axis.

4.1 Enhancing Features

The implementation presented in Section 3 featured many of the essential requirements for Scheme including proper tail calls, variable arity-procedures, and `apply`. It also featured a facility for performing foreign-calls that allows us to easily leverage the capabilities provided by the host operating system and its libraries. With separate compilation, we can implement an extended library of procedures including those required by the R^{5}RS or the various SRFIs. The missing features that can be added directly without changing the architecture of the compiler by much include:

- A full numeric tower can be added. The extended numerical primitives can either be coded directly in Scheme or provided by external libraries such as GNU MP.
- Multiple values are easy to implement efficiently using our stack-based implementation with very little impact on the performance of procedure calls that do not use multiple values [3].
- User-defined macros and a powerful module system can be added simply by compiling and loading the freely-available portable `syntax-case` implementation [7, 18].
- Our compiler does not handle heap overflows. Inserting overflow checks before allocation attempts should be simple and fast by comparing the value of the allocation pointer to an allocation limit pointer (held elsewhere) and jumping to an overflow handler label. A simple copying collector can be implemented first before attempting more ambitious collectors such as the ones used in Chez Scheme or The Glasgow Haskell Compiler [6, 12].
- Similarly, we did not handle stack overflows. A stack-based implementation can perform fast stack overflow checks by comparing the stack pointer to an end of stack pointer (held elsewhere) and then jumping to a stack-overflow handler. The handler can allocate a new stack segment and wire up the two stacks by utilizing an underflow handler. Implementing stack overflow and underflow handlers simplifies implementing efficient continuations capture and reinstatement [9].
- Alternatively, we can transform the input program into continuation passing style prior to performing closure conversion. This transformation eliminates most of the stack overflow checks and simplifies the implementation of `call/cc`. On the downside, more closures would be constructed at run-time causing excessive copying of variables and more frequent garbage collections. Shao et al. show how to optimize the representation of such closures [15].

4.2 Enhancing Performance

The implementation of Scheme as presented in Section 3 is simple and straightforward. We avoided almost all optimizations by performing only the essential analysis passes required for assignment and closure conversion. On the other hand, we have chosen a very compact and efficient representation for Scheme data structures. Such choice of representation makes error-checks faster and reduces the memory requirements and cache exhaustion.

Although we did not implement any source-level or backend optimizations, there is no reason why these optimization passes cannot be added in the future. We mention some "easy" steps that can be added to the compiler and are likely to yield high payoff:

- Our current treatment of `letrec` and `letrec*` is extremely inefficient. Better `letrec` treatment as described in [19] would allow us to (1) reduce the amount of heap allocation since most `letrec`-bound variables won't be assignable, (2) reduce the size of many closures by eliminating closures with no free-variables, (3) recognize calls to known procedures which allows us to perform calls to known assembly labels instead of making all calls indirect through the code pointers stored in closures, (4) eliminate the procedure check at calls to statically-known procedure, (5) recognize recursive calls which eliminates re-evaluating the value of the closures, (6) skip the argument-count check when the target of the call is known statically, and (7) `cons`ing up the rest arguments for known calls to procedures that accept a variable number of arguments.
- Our compiler introduces temporary stack locations for all complex operands. For example, `(+` e `4)` can be compiled by evaluating e first and adding 16 to the result. Instead, we transformed it to `(let ((`t_0 e`)) (+` t_0 `4))` which causes unnecessary saving and reloading of the value of e. Direct evaluation is likely to yield better performance unless good register allocation is performed.
- Our treatment of tail-calls can be improved greatly by recognizing cases where the arguments can be evaluated and stored in place. The greedy-shuffling algorithm is a simple strategy that eliminates most of the overhead that we currently introduce for tail-calls[4].
- None of the safe primitives were implemented in the compiler. Open-coding safe primitives reduces the number of procedure calls performed.
- Simple copy propagation of constants and immutable variables as well as constant-folding and strength-reduction would allow us to write simpler code without fear of inefficiencies. For example, with our current compiler, we might be discouraged from giving names to constants because these names would increase the size of any closure that contains a reference to them.

More sophisticated optimizations such as register allocation [5, 4, 16], inlining [17], elimination of run time type checks [10, 21], etc. could be targeted next once the simple optimizations are performed.

5. Conclusion

Compiler construction is not as complex as it is commonly perceived to be. In this paper, we showed that constructing a compiler for a large subset of Scheme that targets a real hardware is simple. The basic compiler is achieved by concentrating on the essential aspects of compilation and freeing the compiler from sophisticated analysis and optimization passes. This helps the novice compiler writers build the intuition for the inner-workings of compilers without being distracted by details. First-hand experience in implementing a basic compiler gives the implementor a better feel for the compiler's shortcomings and thus provide the motivation for enhancing it. Once the basic compiler is mastered, the novice implementor is better equipped for tackling more ambitious tasks.

Acknowledgments

I extend my thanks to Will Byrd, R. Kent Dybvig, and the anonymous reviewers for their insightful comments.

Supporting Material

The extended tutorial and the accompanying test suite are available for download from the author's website:

`http://www.cs.indiana.edu/~aghuloum`

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 1986.

[2] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, UK, 1998.

[3] ASHLEY, J. M., AND DYBVIG, R. K. An efficient implementation of multiple return values in scheme. In *LISP and Functional Programming* (1994), pp. 140–149.

[4] BURGER, R. G., WADDELL, O., AND DYBVIG, R. K. Register allocation using lazy saves, eager restores, and greedy shuffling. In *SIGPLAN Conference on Programming Language Design and Implementation* (1995), pp. 130–138.

[5] CHAITIN, G. J. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (New York, NY, USA, 1982), ACM Press, pp. 98–101.

[6] DYBVIG, R. K., EBY, D., AND BRUGGEMAN, C. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Tech. Rep. 400, Indiana University, 1994.

[7] DYBVIG, R. K., HIEB, R., AND BRUGGEMAN, C. Syntactic abstraction in scheme. *Lisp Symb. Comput. 5*, 4 (1992), 295–326.

[8] DYBVIG, R. K., HIEB, R., AND BUTLER, T. Destination-driven code generation. Tech. Rep. 302, Indiana University Computer Science Department, February 1990.

[9] HIEB, R., DYBVIG, R. K., AND BRUGGERMAN, C. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, NY, June 1990), vol. 25, pp. 66–77.

[10] JAGANNATHAN, S., AND WRIGHT, A. K. Effective flow analysis for avoiding runtime checks. In *2nd International Static Analysis Symposium* (September 1995), no. LNCS 983.

[11] KELSEY, R., CLINGER, W., AND (EDITORS), J. R. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (1998), 26–76.

[12] MARLOW, S., AND JONES, S. P. The new ghc/hugs runtime system.

[13] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[14] SCO. System V Application Binary Interface, Intel386™ Architecture Processor Supplement Fourth Edition, 1997.

[15] SHAO, Z., AND APPEL, A. W. Space-efficient closure representations. In *LISP and Functional Programming* (1994), pp. 150–161.

[16] TRAUB, O., HOLLOWAY, G. H., AND SMITH, M. D. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 142–151.

[17] WADDELL, O., AND DYBVIG, R. K. Fast and effective procedure inlining. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS '97)* (September 1997), vol. 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pp. 35–52.

[18] WADDELL, O., AND DYBVIG, R. K. Extending the scope of syntactic abstraction. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM Press, pp. 203–215.

[19] WADDELL, O., SARKAR, D., AND DYBVIG, R. K. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher Order Symbol. Comput. 18*, 3-4 (2005), 299–326.

[20] WIRTH, N. *Compiler Construction*. Addison Wesley Longman Limited, Essex, England, 1996.

[21] WRIGHT, A. K., AND CARTWRIGHT, R. A practical soft type system for scheme. *Transactions on Programming Languages and Systems* (1997).