Compiler Construction Using Scheme

Erik Hilsdale   J. Michael Ashley
R. Kent Dybvig   Daniel P. Friedman

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{*ehilsdal,jashley,dyb,dfried*}*@cs.indiana.edu*

Abstract

This paper describes a course in compiler design that focuses on the Scheme implementation of a Scheme compiler that generates native assembly code for a real architecture. The course is suitable for advanced undergraduate and beginning graduate students. It is intended both to provide a general knowledge about compiler design and implementation and to serve as a springboard to more advanced courses. Although this paper concentrates on the implementation of a compiler, an outline for an advanced topics course that builds upon the compiler is also presented.

1 Introduction

A good course in compiler construction is hard to design. The main problem is time. Many courses assume C or some similarly low-level language as both the source and implementation language. This assumption leads in one of two directions. Either a rich source language is defined and the compiler is not completed, or the source and target languages are drastically simplified in order to finish the compiler.

Neither solution is particularly satisfying. If the compiler is not completed, the course cannot be considered a success: some topics are left untaught, and the students are left unsatisfied. If the compiler is completed with an oversimplified source language, the compiler is unrealistic on theoretical grounds since the semantics of the language are weak, and if the compiler generates code for a simplified target language, the compiler is unrealistic on practical grounds since the emitted code does not run on real hardware.

An alternative approach is to abandon the assumption that a low-level language be used and switch to a high-level language. Switching to a high-level language as the implementation language has the benefit that the compiler takes less time to implement and debug. Furthermore, using a simple high-level language as the source confers the benefits of a small language without a loss of semantic power. The combination makes it possible to generate code for a real architecture and to complete the compiler within the bounds of a one-semester course.

Scheme is a good choice for both a high-level implementation and source language. It is an extremely expressive language, and the core language is very small.

Title	Compilers I
Goal	To provide a general knowledge of compiler design and implementation and to serve as a springboard to more advanced courses.
Students	Advanced undergraduates and beginning graduate students in Computer Science.
Duration	One fifteen-week semester with two 75-minute lectures per week.
Grading	Five projects, one midterm exam, and one final exam.

Figure 1: Course information

This paper presents a one-semester course in which a Scheme compiler is constructed using Scheme as the implementation language (see Figure 1). While the paper focuses on the compiler constructed during the course, an advanced course in language implementation is outlined that uses the constructed compiler as a testbed.

The paper is organized as follows. Section 2 describes the compiler. Section 3 discusses issues affecting the design of the compiler and the course. Section 4 outlines an advanced topics course that uses the compiler. Section 5 gives our conclusions.

2 The Compiler

The compiler accepts a subset of legal Scheme programs as defined in the Revised^4 Report [7], a subset strong enough to compile itself.

- the language is syntactically restricted so that the only numbers accepted are integers in a bounded range,
- all **lambda** expressions have a fixed arity, *i.e.*, no rest arguments.
- programs cannot have free variables other than references to primitives in operator position,
- symbols cannot be interned at runtime,
- first-class continuations and I/O are not supported,
- derived syntax is not directly supported,
- garbage-collection is not provided, and
- the runtime library is minimal.

These omissions are not detrimental. A primitive can be treated as a value through an inverse-eta transformation [5, page 63] by putting it in a lambda expression that accepts arguments that are in turn passed to the primitive. Derived syntax is not supported directly, but the compiler can macro expand its input as a first step because the compiler is itself written in Scheme and the host programming environment makes a macro expander available. First-class continuations, I/O, and the ability to intern symbols dynamically are important (and are covered in lectures), but they are not pedagogically essential.

The compiler is described below, back to front. The run-time execution model is described first. The representation of the environment and control fixes the target of the compiler and motivates the structure of the compiler's intermediate language. The code generator generates its assembly code from the intermediate language, and the front end translates core Scheme programs to intermediate programs.

2.1 The Run-time Model

The run-time execution model is given in Figure 2. Control is stack-based, with the *fp* register pointing to the base of the current frame. A frame consists of a return address, the arguments to the active procedure, and temporary values. The *cp* register points to the closure of the active procedure, and the closure holds the values of the procedure's free variables. The *ap* register points to the next free location in the heap. An accumulator register *ac0* and three temporary registers *t0*, *t1*, and *t2* are used for intermediate values.

The procedure call convention for non-tail calls is as follows. The caller first saves the closure pointer at the top of its frame. The callee's frame is then built by pushing a return address and then evaluating each argument and pushing its value. The operator is evaluated last, and its value is placed in the *cp* register. Finally, the frame pointer is incremented to point to the base of the callee's frame and control is transferred by a jump indirect through the closure pointer. On return, the callee places the return value in the accumulator *ac0* and jumps to the return address at the base of its frame. The caller restores the frame pointer to its old position and reloads the *cp* register with its old value.

The calling convention is simpler for tail calls. The arguments are evaluated and pushed, and the operator is then evaluated and stored in the *cp* register. The arguments are moved downwards to overwrite arguments of the caller's frame, and control is transferred to the callee. The frame pointer does not move.

Values are represented using 64-bit tagged pointers with the low three bits used for tag information [23]. Four of the nine data-types, booleans, characters, fixnums, and the empty list, are immediate data-types and are encoded directly in the pointer. Vectors, pairs, closures, strings, and symbols are allocated in the heap. Since the low three bits are used for the tag, allocation must proceed on eight-byte boundaries. A heap allocated object is tagged by subtracting eight from the pointer to the object and then adding the tag. Fields of the object can be referenced efficiently using a displacement operand. A type check is also efficient, requiring at worst a mask, compare, and branch.

2.2 Code Generation

The code generator produces code for the run-time model from the intermediate language of Figure 3. The language is similar to core Scheme despite several syntactic differences. The principal

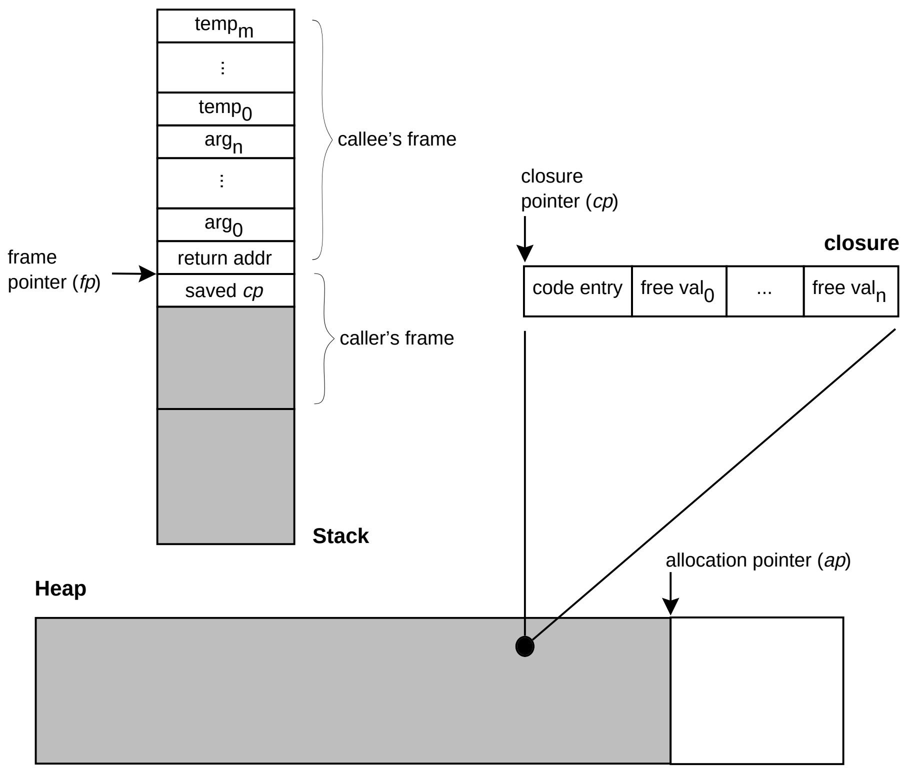

Figure 2: The run-time model is stack based, and a display closure is used to access variables free in the active procedure. Heap allocation is performed by incrementing a dedicated allocation pointer.

$$\begin{array}{rcl}
E & = & i \mid R \mid (\textbf{begin}\ E\ E) \mid (\textbf{if}\ E\ E\ E) \mid (E\ E\ \ldots) \mid \\
 & & (P\ E\ \ldots) \mid (\textbf{closure}\ (v\ \ldots)\ (R\ \ldots)\ E) \mid \\
 & & (\textbf{let}\ ((v\ E)\ \ldots)\ E) \\
R & = & (\textbf{free}\ n\ v) \mid (\textbf{bound}\ n\ v) \mid (\textbf{local}\ v) \\
P & \in & \mathit{Primitives} \\
i & \in & \mathit{Immediates} \\
v & \in & \mathit{Variables} \\
n & \in & \mathbf{N}
\end{array}$$

Figure 3: The intermediate language

difference is that variable references are subsumed by the **free**, **bound**, and **local** forms.

The **free** and **bound** forms each include an index indicating the offset from the *cp* or *fp* register at which the variable's value can be found, while the **local** form includes only the name of a variable. Offsets from the *fp* for **local** are determined by the code generator. The **closure** form is like **lambda**, but the locations of the free variables are made explicit. Constants are restricted to immediate values.

As an example, the following Scheme program

```
(let ((f (lambda (x)
           (let ((y (+ x 1)))
             (lambda (z)
               (cons y (cons z (quote (1 2)))))))))
  ((f 4) 5))
```

is written as follows in the intermediate language.

```
(let ((t (cons 1 (cons 2 '()))))
  (let ((f (closure (x) ((local t))
             (let ((y (+ (bound 0 x) 1)))
               (closure (z) ((local y) (free 0 t))
                 (cons (free 0 y)
                       (cons (bound 0 z)
                             (free 1 t)))))))))
    (((local f) 4) 5)))
```

Assignment is not part of the intermediate language, since variable assignment cannot be directly supported using the chosen run-time model. Closure formation copies the values of the free variables into the closure data structure. Therefore, a variable's value can occupy more than one location. For example, in the above program the value of *y* occupies a **local** location in a stack frame and a **free** location in a closure. Allowing multiple locations precludes direct assignment to those locations. Because variable assignment cannot be directly supported, variable assignments in the source language are converted to structure mutations by a source-to-source transformation discussed in Section 2.3.2.

Assembly code can be generated from intermediate programs in one pass. Code is generated bottom-up with the invariant that the result of evaluating a subexpression is left in the accumulator *ac0*. Arguments to primitives are stored in temporary locations on the stack, and code for primitives is generated inline. The code generated for primitives is unsafe, *i.e.*, no type checking is performed.

Offsets for **free** and **bound** references are provided. Computing frame offsets for **local** references requires a lexical environment to be passed downwards. The environment maps local variable names to frame offsets. The environment is necessary since temporary locations used for primitive and procedure applications can be interspersed with local bindings.

After code generation, the resulting assembly code is assembled using the system assembler `as` and linked against a C and assembly code stub using `ld`. The C stub obtains memory from the operating system for the stack and heap. The assembly code stub initializes the registers and places a return address back to C at the base of the stack. Upon return to C, the value left in the accumulator *ac0* is printed by a simple C-coded printer.

2.3 Compiling to Intermediate Code

The front-end of the compiler consists conceptually of three parts: a scanner and parser, a sequence of source-to-source translations, and a transformation that assigns locations to variables.

2.3.1 Scanning and Parsing

The scanner is specified as a Deterministic Finite Automaton (DFA). Most Scheme tokens are recognized. The exception is that the only legal numbers are exact integers without base prefixes, *i.e.*, the nonterminal `<number>` is redefined to be `<sign><digit>`$^+$. The scanner also incorporates a symbol table by forgoing the host system's implementation of *string–>symbol* and uses a private implementation instead.

The parser uses a recursive descent algorithm. It takes a sequence of Scheme tokens and returns a Scheme datum.

2.3.2 Code Transformation

Three source-to-source transformations are performed on the forms the parser produces. The first transformation invokes the host system macro expander to expand the input program and then regularizes the expanded program. The second transformation eliminates **set!** forms. The third transformation eliminates complex quoted data.

Regularization. The first transformation is a one-pass traversal over the input form. The pass

- checks for syntax errors and flags unbound variables, *i.e.*, variables that are neither **lambda**-bound nor the name of an inlined primitive,
- ensures that primitives are used only in operator position,
- wraps unquoted constants in **quote** expressions,
- transforms the bodies of multiple-bodied **lambda** expressions into **begin** expressions, and
- makes **begin** expressions take two sub-expressions.

$$
\begin{array}{rcl}
E & = & (\textbf{quote}\ \mathsf{c}) \mid v \mid (\textbf{set!}\ v\ E) \mid (\textbf{begin}\ E\ E) \mid \\
  &   & (\textbf{if}\ E\ E\ E) \mid (E\ E\ \ldots) \mid (P\ E\ \ldots) \mid \\
  &   & (\textbf{lambda}\ (v\ \ldots)\ E) \\
c & \in & \textit{Constants} \\
v & \in & \textit{Variables} \\
P & \in & \textit{Primitives}
\end{array}
$$

Figure 4: Grammar for regularized Scheme

The output of this pass is a Scheme program in the language defined by the grammar given in Figure 4.

Assignment Elimination. Scheme's variable assignment form, **set!**, is eliminated from the language in the second transformation. A variable *x* that is the subject of a **set!** expression is shadowed by another variable *x* that is bound to a vector whose single element is the value of the first *x*. A reference to *x* then becomes a vector reference, and an assignment to *x* becomes a vector update. As an example, the following program

```
(lambda (x y)
  (begin
    (set! x (quote 3))
    (+ x y)))
```

is transformed into

```
(lambda (x y)
  ((lambda (x)
     (begin
       (vector-set! x (quote 0) (quote 3))
       (+ (vector-ref x (quote 0)) y)))
   (vector x)))
```

Complex Quote Elimination. The third transformation eliminates complex quoted constants from the program. A complex quoted constant is a quoted symbol, vector, string, or list. An instance of such a constant is replaced by a reference to a fresh variable, and the variable is bound in a lexical scope surrounding the whole program. Symbols are treated specially since multiple occurrences of the same symbol must be commonized. Since symbols can occur in other complex quoted constants, symbols are created and bound in a lexical scope outside the scope in which other quoted data is created. For example,

```
(vector (quote (3 ab)) (quote ab))
```

becomes

$$\begin{array}{rcl}
E & = & i \mid v \mid (\textbf{begin}\ E\ E) \mid (\textbf{if}\ E\ E\ E) \mid \\
  &   & (E\ E \ldots) \mid (P\ E \ldots) \mid \\
  &   & (\textbf{lambda}\ (v \ldots)\ E) \\
i & \in & \textit{Immediates} \\
v & \in & \textit{Variables} \\
P & \in & \textit{Primitives}
\end{array}$$

Figure 5: Grammar for simplified Scheme

```
((lambda (t0)
   ((lambda (t1)
      (vector t1 t0))
    (cons 3 (cons t0 '()))))
 (make-symbol (string #\a #\b)))
```

The output of this pass is a Scheme program in the language defined by the grammar given in Figure 5.

2.3.3 Variable Addressing

The last transformation before code generation assigns locations to variables and transforms the simplified Scheme program into the intermediate language. One analysis pass determines the free variables of each **lambda** expression. A second pass rewrites the program. Each bound and free variable reference is converted into a **bound** and **free** form respectively, and **lambda** expressions are converted into **closure** expressions. Also, expressions of the form

```
((lambda (v0 ... vn) E)
 E0 ... En)
```

are rewritten as **let** expressions, and **let**-bound variables are converted into **local** forms. The output is in the language defined by the grammar in Figure 3.

3 The Course

Each student implements the compiler in one semester. The implementation is divided into five projects, most requiring close to three weeks of class-time (see Figure 6).

For the first project the students implement the scanner and parser for Scheme syntax described in Section 2.3.1. They specify a DFA for the scanner and use one of two macro packages, **declare-table** or **state-case**, to convert the DFA into a runnable scanner. The **declare-table** and **state-case** macros are supplied so that students can spend time learning enough theory to write a DFA rather than spending all their time on the scanner's implementation details. Extra time is devoted to this fairly simple project to allow those without prior Scheme experience to become more familiar with the language.

Because parsing Scheme data is a trivial exercise, a second project is assigned to write a parser for an Algol-like language with complex precedence and associativity rules for its prefix, suffix, and

Project	Weeks
Scanning and Parsing Scheme	3
Parsing Algol-like Syntax	3
Code Transformation	3
Allocation of Variable Locations	2
Code Generation	4
	15

Figure 6: Projects

```
   ⟨expression⟩ = proc ( ⟨formals⟩ ) ⟨expression⟩
                | let ⟨variable⟩ ⟨bindings⟩ in ⟨expression⟩
                | let ⟨bindings⟩ in ⟨expression⟩
                | if ⟨expression⟩ then ⟨expression⟩ else ⟨expression⟩
                | if ⟨expression⟩ then ⟨expression⟩
                | ⟨variable⟩ ! ⟨expression⟩
                | ⟨expression⟩ ? ⟨expression⟩
                | ⟨expression⟩ . ⟨expression⟩
                | ? ⟨expression⟩
                | ⟨expression⟩ hd
                | ⟨expression⟩ tl
                | ⟨expression⟩ ( ⟨actuals⟩ )
                | ( ⟨expression list⟩ )
                | ⟨variable⟩
                | ⟨literal⟩
      ⟨formals⟩ = ϵ | ⟨variable list⟩
⟨variable list⟩ = ⟨variable⟩ | ⟨variable⟩ , ⟨variable list⟩
     ⟨bindings⟩ = ϵ | ⟨binding list⟩
 ⟨binding list⟩ = ⟨binding⟩ | ⟨binding⟩ , ⟨binding list⟩
      ⟨binding⟩ = ⟨variable⟩ <- ⟨expression⟩
      ⟨actuals⟩ = ϵ | ⟨expression⟩
⟨expression list⟩ = ⟨expression⟩ | ⟨expression⟩ , ⟨expression list⟩
      ⟨literal⟩ = ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩
```

Figure 7: Grammar for an Algol-like language

infix operators. The students write an LL(1) grammar for the new syntax and implement a parser that recognizes expressions in the language specified in Figure 7. The parser returns an equivalent Scheme datum that can be fed to the rest of the compiler. Although traditional parsing techniques are taught [1], a more functional approach [21] might be an attractive alternative.

The third and fourth assignments involve implementing the transformations described in Sections 2.3.2 and 2.3.3. In order to ease the handling of such forms as the **let** form of the intermediate language, a macro package, **synlambda**, is made available to the students that extends Scheme with pattern-matching capabilities. Pattern matching over S-expressions does cause some execution overhead that can be avoided by first transforming the input into records. The source-to-source nature of the transformations can be obscured, however, if performed on records.

The last project is the implementation of the code generator described in Section 2.2. Students use the *Alpha Architecture Reference Manual* [20] for information on the general form of the assembly code generated. Object code for a working code generator is also made available for comparison.

The compiler is designed to balance pedagogy and reality. On one hand, the compiler should be straightforward so that it can be fully implemented by an undergraduate in one semester. On the other hand, it should not be a toy compiler; it should generate reasonably efficient code in a native assembly language. The compiler presented in Section 2 is well balanced between these two goals.

If necessary, the compiler could be simplified by macro expanding **begin** forms and not recognizing **let** expressions when assigning locations to variables. The generated code would be much less efficient, however. **begin** expressions can be macro expanded into **let** expressions. For example, (**begin** E_0 E_1) would become (**let** ((t E_0)) E_1), where t does not occur free in E_1. The cost of such an expansion is that a frame location is used unnecessarily to store the unreferenced variable t. If **let** expressions were not recognized, then a **let** expression would be treated as a full procedure call, which clearly has a higher run-time cost than binding the variables locally in the current stack frame.

If time is a premium, *e.g.*, in a ten-week quarter system, The Scheme 48 compiler [17] may be simpler to implement, though it sacrifices our goal of targeting genuine hardware. For another exploration of a simple Scheme compiler see Clinger and Hansen [8]. Additional background reading emphasizing proof of correctness of a Scheme compiler, with an extensive bibliography, can be found in [13].

It is notable that our compiler does not convert its input to continuation-passing style (CPS). Such a transformation simplifies the regularized language and is a transformation employed by some compilers for (mostly) functional languages [2, 18, 22]. While simplifying the regularized language is appealing, converting to CPS would not simplify the compiler used in this course. Pedagogically, converting to CPS requires an extra pass and obscures the correlation between the intermediate program and its source-level counterpart. This hinders debugging.

An alternative to converting to CPS is to A-normalize [12] the source program. An A-normalized program names all complex intermediate expressions and fixes the order of evaluation. It is similar to CPS, but the representation of control is not made explicit at the source level. Like conversion to CPS, A-normalization simplifies the source language and is an appropriate move in some contexts, *e.g.*, for the static analysis of programs [11]. It is inappropriate, however, for a one-semester course in compiler construction. Assigning a temporary location to every intermediate value, *e.g.*, the test part of a conditional, is unnecessary and results in the generation of poor code unless the pass that assigns locations to variables is made more sophisticated. Assigning locations and generating code can be done more simply and more directly when the program is in direct style.

We have chosen to use a stack rather than a heap model. While abandoning a stack discipline and using closures to represent continuations makes *call-with-current-continuation* trivial to implement, procedure calls are more expensive [3] unless some effort is taken to share continuation closures [19]. In addition, because heap allocation of continuation frames obviates many traditional problems of stack management, a compiler based around a heap discipline would not be appropriate for a course on general compiler design and implementation.

4 Advanced Coursework

Topics courses in advanced language implementation have been built on top of the first course. Such courses are intended for graduate students who have completed the first semester course. The students use their compilers from the first semester to implement the compiler optimizations, run-time system extensions, and language features discussed in lecture. The topics can be broadly classified as either compile-time or run-time.

4.1 Compile-time Topics

The following compile-time topics have been successfully covered in a follow-up course:

- macro expansion [9],
- destination-driven code generation [10],
- copy propagation and constant folding [1],
- register allocation [6], and
- type check elimination by abstract interpretation [16, 4].

With the exception of macro expansion, the compile-time topics are about compiler optimizations. To motivate them, an assignment is given early in which the students are told to hand-optimize a program that solves the eight queens problem. There is no constraint on the optimizations that may be applied, and optimizations to both the source and target code are allowed. Students typically perform procedure integration, constant folding, peephole optimization, and register allocation.

Some of the optimizations done by hand are covered in class and then implemented. The compiler's code generator is rewritten to use destination-driven code generation, which is a one-pass code generation technique that achieves most of the benefits of peephole optimization. For register allocation, traditional techniques are covered in lecture, including register assignment and save/restore placement. The students implement a three-pass algorithm that uses caller-save registers and does a good job of save/restore placement. The algorithm does not, however, try to optimize register assignment. The discussion of copy propagation, procedure integration, and constant folding leads to (on-line) partial evaluation. The students implement their own compiler passes to perform these transformations with the constraint that the passes must terminate.

4.2 Run-time Topics

Run-time topics are treated like their compile-time counterparts: theory and possible implementation strategies for each topic are discussed in class and a particular strategy is chosen for implementation. Some of the topics covered in a past course have been:

- separate compilation,
- buffered I/O,
- first-class continuations [14], and
- garbage collection.

The bulk of the run-time support code is written in Scheme using a set of low-level primitives supported by the compiler's code generator. One strategy for including the support code with user code is to simply wrap the support code (such as a definition of *string−>symbol* or a garbage-collection routine) around the user code using a **let** expression and input the combination to the compiler as one monolithic program. While simple, this approach has the drawback that it is difficult to debug the compiler since even small programs are combined with a substantial amount of run-time support code.

An alternative strategy is to use a separate compilation facility. Run-time support code is separately compiled to produce a library. User code input to the compiler is then compiled and linked against the library to produce stand-alone assembly code as before. An early project in the semester is to implement a simple separate compilation facility that supports this model of program development. The details of the implementation can be found in the extended version of this paper [15].

With a separate compilation facility in hand, it is possible to implement substantial projects to enhance the run-time system. One possible set of topics is first-class continuations, garbage collection, and input/output. While others are possible, this choice allows students to bootstrap their compiler by the end of the semester.

To support first-class continuations, the stack model is generalized to a series of linked stack segments. The model supports stack overflow gracefully by treating overflow as an implicit continuation capture in which a new stack is allocated and linked to the overflowed segment.

First-class continuations simplify memory management techniques. In lecture, various types of memory management are presented, including mark-sweep, reference counting, copying, and generational. The students implement a copying collector in Scheme as a procedure of no arguments that immediately reifies its current continuation. Since registers are caller-save and the collector is a procedure of no arguments, the continuation is the root of the collection. The collector is an iterative Scheme program that performs a bounded amount of allocation. To detect when a garbage collection must occur, heap overflow checks are inserted by the compiler before each allocation.

Buffered I/O is a straightforward topic. The implementation requires code in the C stub to interface to the operating system as well as primitives supported by the code generator that call the C routines in the stub.

4.3 Discussion

The second semester can be taught with varying degrees of depth and breadth. If many topics are covered, the course has a "concepts" feel to it. For such a course, the time needed to implement the projects limits the number of topics that can be covered. Six or seven serious projects can be completed in a semester, but there is enough lecture time to cover more topics.

Regardless, six or seven projects is an ambitious undertaking for one semester. The fact that the students can complete this many projects is due to the framework developed in the first semester: the compiler is simple but realistic, and it is written in Scheme and developed in an incremental programming environment. The projects would take much longer if the compiler or any of the run-time support code, *e.g.*, the garbage collector, were implemented in a lower-level language like C.

As an experiment to further reduce debugging time, a simulator for an appropriate subset of symbolic Alpha assembly code has been recently introduced. The simulator is embedded in Scheme and thus allows for the intermixing of Scheme expressions and Alpha code to provide a more robust debugging environment than `gdb`. Whether the simulator actually reduced debugging time, however, has not been studied.

In the past, compile-time topics have been discussed first, followed by run-time topics. Perhaps a better organization would be to address run-time topics first, particularly if a goal is to have the students bootstrap their compilers.

When compilers are bootstrapped, students can use their own compilers as a non-trivial test of their systems. This makes the benefits of compiler optimizations later in the course much more dramatic. Furthermore, students learn first-hand the tradeoffs between the cost of compile-time optimizations versus their benefits. For example, adding a register allocator might add three passes to the compiler, but it still speeds up the bootstrapped compiler since it is compiled using register allocation.

5 Conclusions

This paper outlines a course in compiler construction. The implementation and source language is Scheme, and the target language is assembly code. This choice of languages allows a direct-style, stack-based compiler to be implemented by an undergraduate in one semester that touches on more aspects of compilation than a student is likely to see in a compiler course for more traditional languages. Furthermore, expressiveness is barely sacrificed; the compiler can be bootstrapped provided there is enough run-time support.

Besides covering basic compilation issues, the course yields an implemented compiler that can serve as a testbed for advanced coursework in language implementation. The compiler has been used, for example, to study advanced topics such as the implementation of first-class continuations and register allocation.

A technical report [15] giving more details is in preparation. The report describes the implementation of the compiler in considerably more detail. Furthermore, curricular materials such as the Alpha simulator and the **synlambda**, **declare-table**, and **state-case** macros are also described.

Acknowledgments

Carl Bruggeman, Robert Hieb and Suresh Srinivas helped design the Scheme subset used in this course. Chris Haynes provided useful feedback on various aspects of the course.

References

[1] Alfred D. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appell and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. To appear in *Journal of Functional Programming.*

[4] J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. To appear in *Proceedings of the ACM Symposium on Principles of Programming Languages, 1996.*

[5] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

[6] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, 1995.

[7] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 5(3):1–55, July-September 1991.

[8] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139, 1994.

[9] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[10] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Indiana University, February 1990.

[11] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[12] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[13] Joshua D. Guttman and Mitchell Wand, editors. *VLISP: A Verified Implementation of Scheme.* Kluwer, Boston, 1995. Originally published as a special double issue of the journal *Lisp and Symbolic Computation* (Volume 8, Issue 1/2).

[14] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[15] Erik Hilsdale and Daniel P. Friedman. A Scheme-based course on compiler construction. In Preparation.

[16] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding runtime checks. In *Proceedings of the 1995 International Static Analysis Symposium*, 1995.

[17] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[18] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, J. Philbin, and Norman I. Adams. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 21(7):219–233, 1986.

[19] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 130–161, 1994.

[20] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[21] Michael Sperber and Peter Thiemann. The essence of LR parsing. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '95*, pages 146–155, 1995.

[22] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, M.I.T (A.I. LAB.), Massachusetts, U.S.A, 1978. Also available as MIT AI Memo 474.

[23] Peter A. Steenkiste. The implementation of tags and run-time type checking. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 3–24. MIT Press, 1991.