Compiling: A High-level Introduction Using Scheme*

Christopher T. Haynes
Computer Science Department
Indiana University
Bloomington, IN 47405
chaynes@indiana.edu

Abstract

Traditional compiler courses use formal methods for parsing, but treat the more important semantic aspects informally. We present a one semester course in which compiler development is reduced to a number of transformation steps, each of which is formally specified, easily tested, and clearly motivated by semantic considerations. Furthermore, the source language is substantial (essentially the host language of the compiler) and the target is a popular RISC architecture.

Introduction

It is customary for compiler courses to begin with substantial application of formal methods to the well-understood parsing problem. Unfortunately code generation has remained ad hoc. Though devices such as attribute grammars may be carefully described, their application to the compilation of practical source languages is quite complex, and the connection between their use and the semantics of the source language is unclear. (Indeed, establishing this connection has been a major topic of research for some time and results of general utility have been elusive.) Furthermore, the complexity of code generation is such that if a compiler is to be completed in one semester, either the source or target language must be simplified to such an extent that the compiler no longer represents a practical tool [1].

In this paper we present an alternative approach, inspired by recent research. It rests on two key elements. First, the host language is Scheme [2]. A very-high level symbol-manipulation language such as Scheme provides a critical advantage that allows a compiler with both source and target languages that are realistic to be implemented in one semester. (It also happens that our source language is a substantial subset of Scheme, but this is not critical. The approach proposed here would, with minor modification, work for a variety of source languages.)

Second, compilation is broken into a series of transformations. Each transformation step is *formally specified, easily tested, and clearly motivated by the source language semantics.* Students are thus able to focus on conceptually manageable units. By representing the output of each transformation in an executable form, the homomorphic (meaning-preserving) character of each transformation is readily understood and tested. The homomorphism requirement, together with formal syntactic specification of the input and output of each transformation, provide a formal basis for compiler development that is comprehensible by average undergraduates.

The general approach to teaching compilers outlined here builds upon more than a decade of course development, primarily by R. Kent Dybvig [6]. This paper emphasizes a number of recent developments not previously reported, including the pedagogic use of executable intermediate languages, a single collection pass, and destination-driven code generation. These developments have been tested both in class and in a summer workshop attended by college teachers, who were uniformly enthusiastic about this approach.[1]

The use of executable intermediate languages was inspired by Clinger's TwoBit Scheme compiler [3]. The destination-driven code generation technique used in the final step is that of Dybvig, Hieb, and Butler [4], who use it in the Chez Scheme compiler. In general, the techniques employed in this course are representative of those used in industrial-quality compilers.

The next section reviews our general approach and considers the benefits of formal methods in this context. Following sections review in some detail the transformations that form the basis for all but the parsing assignments, briefly review our approach to teaching parsing technique, indicate possible extensions and optimizations that may be used if time permits, and provide concluding remarks. The reader is assumed to be familiar with traditional compilation technique. Some

*This research supported by NSF grant CDA-9312614.

[1] This brief paper gives a general impression of our approach, but must necessarily omit many details that are documented in the workshop web structure [7].

[0] Submitted to SIGCSE'97. Copyright © 1996 by Christopher Haynes

familiarity with Scheme would be helpful, but not necessary.

Formal Methods in Compilation

The core of the course is a cumulative series of assignments. This breaks the work into moderate-sized chunks, while maintaining the experience of completing a large project. A solution to each assignment is provided when it is due so that students are not penalized repeatedly for failure to complete a assignment.

Compilers of modern design, especially those for languages amenable to global optimization, are typically organized in a sequence of stages. The stages progressively transform intermediate representations of the program being compiled into increasingly low-level forms. The assignments in this course each represents a transformation step of the compiler under construction.

With the exception of a parsing assignment or two (described in a later section), each assignment is to write a Scheme program that takes a datum satisfying an input grammar and transforms it into a datum satisfying an output grammar. The output grammar of each transformation is the input grammar of the next. The formal specification of each assignment is completed with the requirement that each transformation be *value preserving*.

For the final assembly form, evaluation is by the target machine or an emulator that is provided. For all other forms, evaluation is performed by the host Scheme implementation. All but the last form are subsets of the set of Scheme expressions (with the addition of three simple syntactic extensions in the case of code-generation form). Thus a highly-useful check on the correctness of the output of each transformation is provided by evaluating for a number of test cases both the input and the output, and verifying that the same result is obtained in both cases.

Value preserving transformations are of course much used in the formal study of programming languages. They are, however, very seldom used in practice with well-specified correctness criteria or convenient means of testing. This is perhaps the single most unique and valuable aspect of the approach taken in this course.

This is a powerful example of how formal methods, when applied in a natural way, can greatly enhance learning. Beyond classical parsing theory, traditional compiler writing technique is notoriously informal. This informality leaves the connection between language semantics and compilation unclear, which by extension leaves the meaning of source language programs unclear. Students are typically shown examples of traditional compilation techniques and then asked to figure out the application of these techniques to an assigned problem on their own. The lack of systematic understanding leads both to bugs and discomfort. The weaker students are often those who most appreciate the security and guidance afforded by the suitable application of formal methods.

This great advantage has its (small) costs. For example, the embedding of quoted lists of free and assigned variables in lambda-expression bodies is admittedly at contrivance. Referring to the attempt to keep each form executable as "playing the game," it is not hard to convince students that it is well worth playing along.

Fully formal compiler development would require that each transformation be proved correct *in general*, not just for test examples. This is a major area of research, well beyond the scope of this course. To the author's knowledge, only one practical compiler has been proved correct in this (or any other) way: that of the VLISP[2] project [5]. Nonetheless, this course is believed to make much more use of practical formal techniques than other approaches to teaching compilation.

Transformations

The outermost structure of each transformation program is a dispatch over each expression form of the input grammar (with the last transformation including a dispatch over the primitive procedures). It is recommended that programs be developed and tested incrementally, adding additional cases a few at a time. This allows progress to be measured, provides early recognition of general conceptual difficulties, and assures that weaker students produce some working code.

The transformation assignments are described in the following subsections. The figures contain extended BNF grammars defining each of the transformation input and output forms (except for assembly form, which is omitted for lack of space). The first grammar assumes the entire R4RS[3] Scheme grammar, to which only the indicated modifications are made. Each of the subsequent forms is assumed to inherit the grammar of the preceeding form, to which the specified modifications are made.

Core Form

This assignment is to write a program that accept a program in source form and returns an equivalent program in core form. Programs not in source form should be rejected.

Source form (Figure 1) defines a subset of Scheme that is sufficiently powerful to reasonably write programs such as the compiler itself. One requirement is not reflected in the grammar: variables that occur free in programs are restricted to primitive procedure references in operator position. The other differences from full R4RS Scheme syntax are:

- There are no top-level definitions: hence interactive program development is not supported.
- Only integer numbers are supported: integers suffice for programs such as compilers.

[2] VLISP is a dialect of Scheme.

[3] R4RS refers to the unofficial Scheme standard [2]. Students benefit from exposure to the entire specification of a practical language, which in this case is only 46 pages long and freely available.

R4RS grammar, but with:

⟨program⟩ ⟶ ⟨expression⟩
⟨expression⟩ ⟶ ⟨reference⟩ | ⟨literal⟩ | ⟨procedure call⟩
| ⟨lambda expression⟩ | ⟨conditional⟩ | ⟨assignment⟩
| ⟨begin expression⟩ | ⟨derived expression⟩
⟨reference⟩ ⟶ ⟨variable⟩
⟨begin expression⟩ ⟶ `(begin` ⟨sequence⟩`)`
⟨body⟩ ⟶ ⟨sequence⟩
⟨formals⟩ ⟶ `(`⟨variable⟩`*)`
⟨number⟩ ⟶ ⟨sign⟩ ⟨digit⟩$^+$
⟨derived expression⟩ ⟶
`(let ((`⟨binding spec⟩`*)` ⟨body⟩`)`
| `(letrec ((`⟨binding spec⟩`*)` ⟨body⟩`)`

Figure 1: Source Form

Source form, but with:

⟨begin expression⟩ ⟶ `(begin` ⟨expression⟩ ⟨expression⟩`)`
⟨alternate⟩ ⟶ ⟨expression⟩
⟨body⟩ ⟶ ⟨expression⟩
⟨literal⟩ ⟶ ⟨quotation⟩
⟨derived expression⟩ ⟶ ⟨empty⟩

Figure 2: Core Form

- Variable-arity procedures are not supported: they are sometimes convenient, but not necessary.
- Only the most useful derived forms are supported: others may be added with low to moderate difficulty.
- Begin expressions are not treated as derived forms and a new syntactic category is created for references: these grammar variations do not effect the core-form language and suit our purposes.

Core form (Figure 2) eliminates derived expressions, begin expressions with other than two subexpressions, and one-armed conditionals. This is accomplished via simple syntactic transformations which students are shown both by examples and as abstract patterns transformations.

Analyzed Form

This transformation performs all the bottom-up propagation of information required by later steps. (This information corresponds to derived attributes.) Specifically, in *analyzed form* (Figure 3) each lambda expression is annotated with a list of the variables that occur free in its body and a list of the variables that are bound in its formals list and assigned in its body. Also, all literals other than immediate values are replaced by newly-generated variables referring to bindings that are provided by an outermost let expression.

The need to return multiple values makes the "boiler plate" structure of a transformation that propagates information upward somewhat cumbersome. It would be conceptually simpler if each form of information collection were performed

Core form, but with:

⟨lambda expression⟩ ⟶
`(lambda` ⟨formals⟩
`(quote (assigned` ⟨variable⟩`*))`
`(quote (free` ⟨variable⟩`*))`
⟨body⟩`)`
⟨literal⟩ ⟶ `(quote` ⟨immediate datum⟩`)`
⟨immediate datum⟩ ⟶ `()` | ⟨boolean⟩ | ⟨number⟩
| ⟨character⟩
⟨program⟩ ⟶ ⟨expression⟩
| `(let ((`⟨quotation binding⟩`*)` ⟨expression⟩`)`
⟨quotation binding⟩ ⟶
`(`⟨variable⟩ `(quote` ⟨heap datum⟩`))`
⟨heap datum⟩ ⟶ ⟨symbol⟩ | ⟨string⟩ | `(`⟨datum⟩$^+$`)`
| `(`⟨datum⟩$^+$ `.` ⟨datum⟩`)` | `#(`⟨datum⟩`*)`

Figure 3: Analyzed form

Analyzed form, but with:

⟨assignment⟩ ⟶ ⟨empty⟩
⟨lambda expression⟩ ⟶
`(lambda` ⟨formals⟩ `(quote (free` ⟨variable⟩`* ))`
⟨body⟩`)`

Figure 4: Assignment-less form

in a separate transformation, but in that case this boiler plate would have to be repeated for each pass, resulting in a considerable volume of code. Hence our decision to combine them in one transformation.

A production compiler would probably also combine several of the other transformation passes outlined here (though keep them separate from the analyzed form transformation). In those cases, however, the gains in code reduction and efficiency do not, for our purposes, compensate adequately for the loss in conceptual clarity.

Assignment-less Form

In this transformation to *assignment-less form* (Figure 4), variable assignments are eliminated. This is done by creating new bindings which associate previously assigned variables with heap allocated cells containing the variable values. Previous references to these variables are replaced expressions that dereference the new cells, and variable assignment expressions are replaced by reference assignment expressions.

This is necessitated by the display representation of closures in the run-time model. The general outlines of the run-time model (see [6]) are presented prior to this point in the course to motivate this transformation.

Immediate-literal Form

All non-immediate literals are eliminated in this transformation to *immediate-literal form* (Figure 5). They are replaced

Assignment-less form, but with:

⟨program⟩ ⟶ ⟨symbol-less program⟩
  | `((lambda (`⟨variable⟩$^+$`)` ⟨symbol-less program⟩`)`
    ⟨symbol expression⟩$^+$`)`
⟨symbol expression⟩ ⟶
  `(string->uninterned-symbol`
    `(string` ⟨character⟩$^+$`))`
⟨symbol-less program⟩ ⟶ ⟨expression⟩
  | `((lambda (`⟨variable⟩$^+$`)` ⟨expression⟩`)`
    ⟨expression⟩`)`

Figure 5: Immediate-literal form

Immediate literal form, but with:

⟨reference⟩ ⟶ `(bound` ⟨number⟩ ⟨variable⟩`)`
  | `(free` ⟨number⟩ ⟨variable⟩`)`
⟨lambda expression⟩ ⟶
  `(build-closure`
    `(lambda` ⟨formals⟩ ⟨body⟩`)`
    ⟨reference⟩*`)`

Figure 6: Code-generation form

with expressions that build their structures in the heap using immediate literals and calls to data-construction primitives.

Symbols are represented by boxed values that reference strings. It is critical that all symbols with the same name reference the same string. This is called *interning* the symbols and is arrange by building a symbol table. Symbols may then be created using the simple primitive `string->uninterned-symbol`.

Code-generation Form

In this pass, with output in *code-generation form* (Figure 6), variable references are replaced by expressions indicating whether the reference is a locally-bound variable or a free variable, and the position of the reference in the local lambda expression's formal parameter list or free variable list, respectively. Variable names are retained only for debugging purposes.

Lambda expressions are also replaced by `build-closure` expressions that contain the lambda expression and a sequence of references to its free variables. The syntax of the new expressions closely resembles the runtime structure of display closures.

For test purposes, the new `bound`, `free`, and `build-closure` forms are easily defined in the host language using whatever form of syntactic extension is supported. The patterns of these syntactic extensions are also shown to the students and are a great aid in their understanding.

Assembly Form

The *assembly form* of this transformation's output is a representation of assembler input as structured Scheme data. Details are omitted in the interest of space. They vary somewhat depending on the target architecture, but are straightforward.

By transforming the original source program into code-generation form, we have done everything possible to make this, the final transformation step, as easy as possible. It is still perhaps the most challenging step. It certainly requires the largest volume of code for its solution, but thanks to the previous transformations, this step is conceptually straightforward and clearly connected with the semantics of the source language.

Having tried several approaches with students, we feel that the little-known destination-driven technique [4] is the most satisfactory. The code generation procedure is passed both control and data destinations, as well as an expression in code-generation form.

There are three possible data destinations: a register, a memory location specified by a register and offset, and a token indicating that the result need not be stored. There are also three possible control destinations: a label to jump to, a pair of labels indicating jump locations for true and false values, and a token indicating that control is to continue with a procedure-call return.

Four of the nine possible combinations of control and data destinations are not possible, leaving five combinations to be addressed for each type of expression and each primitive. This results in a very large volume of code unless functional abstraction techniques are used effectively to isolate recurring patterns.

Coding requires considerable attention to detail, but the destination-driven approach provides detailed guidance. It is not hard to teach if a couple examples of each primary type are provided.

Though not our motivation for choosing to use destination-driven code generation, it is a pleasure to discover that a great many local optimizations are performed automatically by following this approach. This is a major advantage in a production compiler.

Running the Code

A simple translation procedure (supplied to the students) may be used to translate assembly-form code into a form suitable for the standard target machine assembler. A startup program written in C is linked to the assembler output to create an executable object file.

Several RISC architectures have been used: 68030, Alpha, and Sparc. Advantage is taken of only a relatively small set of instructions, for which the differences between these and similar architectures is minimal. Unfortunately, the lack of a general-purpose register file makes the Intel architecture a much less satisfactory target.

It is valuable for the students to see the result of their efforts run on stock hardware, but debugging at this level

is very difficult. Thus we provide an emulator, written in Scheme, that executes programs in our assembly form directly. (The emulator is a straightforward program, complicated only by the need to do exact 32-bit arithmetic and logical operations in Scheme.)

Parsing

So far we have not dealt with parsing: a topic which consumes roughly a quarter of popular compiler texts, and often half of a compiler course. Though context-free parsing theory was one of the earliest and most successful applications of formal methods in computer science, it is not felt to be as instructive as the use of formal methods practiced in this course.

The Scheme `read` procedure is a parser for the Scheme datum syntax (which properly contains the Scheme language syntax). The regular syntax is approximately as complex as that of other general purpose languages. For this we teach scanner construction in the traditional way, deriving an NFA and DFA from regular BNF productions. Rather than using a traditional table-driven DFA implementation, we suggest students translate the DFA directly into a set of mutually-recursive procedures (a well-known approach made possible by proper tail-recursion). The read procedure is completed with a very simple recursive-descent parser.

More complicated LL(1) recursive-descent parsing is also taught with an added assignment. Students are asked to write a parser for an alternate Pascal-like syntax for Scheme, with output in our source form. (This assignment also serves to emphasize that the techniques taught in the course are not peculiar to the Scheme language.)

Altogether, parsing and scanning issues consume less than a quarter of the course. Only brief mention is made of LR parsing technique. We strongly believe the other topics in this course are of more value. In the unlikely event that they are needed, LR techniques can be learned through self study. Students are unlikely, on the other hand, to learn semantics-based formal methods on their own.

Extensions and Optimizations

A number of extensions and optimizations are possible if time permits in a one semester course, or in the context of a second course. The most important optimization is for the code generator to recognize calls in which the operator is a lambda expression. It is then possible to avoid closure creation and call, as when let expressions are in the core [6].

For a fully-functional run-time environment capable of, say, running the compiler itself, it is necessary to implement a garbage collector and the procedures `read` and `write`. The code for the read procedure developed in the parsing section of the course may be converted into source form and compiled to obtain a preamble for future compilation. A similar bootstrap process may be used to implement `write` (which is considerably smaller than `read`). The garbage collector may also be written in Scheme (with a few hidden memory access hooks) and similarly bootstrapped.

Constant folding and copy propagation optimization are natural as core-form to core-form transformations. The lambda-lifting optimization may be done with the help of the information available in analyzed form.

Conclusion

A compilers course is widely regarded as an effective capstone of undergraduate computer science education. This is especially true with the approach outlined here. The simple yet powerful techniques employed span, with comprehension, the chasm between high-level abstraction (a very-high-level language) and the nitty-gritty of assembly language, while simultaneously bridging formal methods and major system development.

Acknowledgment

Eric Hilsdale assisted in developing the course materials described here.

References

[1] AIKEN, A. Cool: a portable project for teaching compiler construction. *SIGPLAN Notices 31*, 7 (July 1996), 19–24.

[2] CLINGER, W., AND (EDITORS), J. R. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers 5*, 3 (July-September 1991), 1–55.

[3] CLINGER, W. D., AND HANSEN, L. T. Lambda, the ultimate label, or a simple optimizing compiler for scheme. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (1994), pp. 128–139.

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

[5] GUTTMAN, J. D., AND WAND, M., Eds. *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).

[6] HILSDALE, E., ASHLEY, J. M., DYBVIG, R. K., AND FRIEDMAN, D. P. Compiler construction using scheme. In *Functional programming languages in education (FPLE), LNCS 1022* (Nijmegen, The Netherlands, Dec 1995), P. H. Hartel and M. J. Plasmeijer, Eds., Springer-Verlag, Heidelberg, pp. 251–268.

[7] URL withheld for anonymity, to appear in final paper.