Preface

Vision

Compiler construction brings together techniques from disparate parts of Computer Science. The compiler deals with many big-picture issues. At its simplest, a compiler is just a computer program that takes as input one potentially executable program and produces as output another, related, potentially executable program. As part of this translation, the compiler must perform syntax analysis to determine if the input program is valid. To map that input program onto the finite resources of a target computer, the compiler must manipulate several distinct name spaces, allocate several different kinds of resources, and synchronize the behavior of different run-time components. For the output program to have reasonable performance, it must manage hardware latencies in functional units, predict the flow of execution and the demand for memory, and reason about the independence and dependence of different machine-level operations in the program.

Open up a compiler and you are likely to find greedy heuristic searches that explore large solution spaces, finite automata that recognize words in the input, fixed-point algorithms that help reason about program behavior, simple theorem provers and algebraic simplifiers that try to predict the values of expressions, pattern-matchers for both strings and trees that match abstract computations to machine-level operations, solvers for diophantine equations and Pressburger arithmetic used to analyze array subscripts, and techniques such as hash tables, graph algorithms, and sparse set implementations used in myriad applications,

The lore of compiler construction includes both amazing success stories about the application of theory to practice and humbling stories about the limits of what we can do. On the success side, modern scanners are built by applying the theory of regular languages to automatic construction of recognizers. LR parsers use the same techniques to perform the handle-recognition that drives a shift-reduce parser. Data-flow analysis (and its cousins) apply lattice theory to the analysis of programs in ways that are both useful and clever. Some of the problems that a compiler faces are truly hard; many clever approximations and heuristics have been developed to attack these problems.

On the other side, we have discovered that some of the problems that compilers must solve are quite hard. For example, the back end of a compiler for a modern superscalar machine must approximate the solution to two or more

interacting NP-complete problems (instruction scheduling, register allocation, and, perhaps, instruction and data placement). These NP-complete problems, however, look easy next to problems such as algebraic reassociation of expressions. This problem admits a huge number of solutions; to make matters worse, the desired solution is somehow a function of the other transformations being applied in the compiler. While the compiler attempts to solve these problems (or approximate their solution), we constrain it to run in a reasonable amount of time and to consume a modest amount of space. Thus, a good compiler for a modern superscalar machine is an artful blend of theory, of practical knowledge, of engineering, and of experience.

This text attempts to convey both the art and the science of compiler construction. We have tried to cover a broad enough selection of material to show the reader that real tradeoffs exist, and that the impact of those choices can be both subtle and far-reaching. We have limited the material to a manageable amount by omitting techniques that have become less interesting due to changes in the marketplace, in the technology of languages and compilers, or in the availability of tools. We have replaced this material with a selection of subjects that have direct and obvious importance today, such as instruction scheduling, global register allocation, implementation object-oriented languages, and some introductory material on analysis and transformation of programs.

Target Audience

The book is intended for use in a first course on the design and implementation of compilers. Our goal is to lay out the set of problems that face compiler writers and to explore some of the solutions that can be used to solve these problems. The book is not encyclopedic; a reader searching for a treatise on *Earley's algorithm* or *left-corner parsing* may need to look elsewhere. Instead, the book presents a pragmatic selection of practical techniques that you might use to build a modern compiler.

Compiler construction is an exercise in engineering design. The compiler writer must choose a path through a decision space that is filled with diverse alternatives, each with distinct costs, advantages, and complexity. Each decision has an impact on the resulting compiler. The quality of the end product depends on informed decisions at each step of way.

Thus, there is no *right* answer for these problems. Even within "well understood" and "solved" problems, nuances in design and implementation have an impact on both the behavior of the compiler and the quality of the code that it produces. Many considerations play into each decision. As an example, the choice of an intermediate representation (IR) for the compiler has a profound impact on the rest of the compiler, from space and time requirements through the ease with which different algorithms can be applied. The decision, however, is given short shrift in most books (and papers). Chapter 6 examines the space of IRs and some of the issues that should be considered in selecting an IR. We raise the issue again at many points in the book—both directly in the text and indirectly in the questions at the end of each chapter.

This book tries to explore the design space – to present some of the ways problems have been solved and the constraints that made each of those solutions attractive at the time. By understanding the parameters of the problem and their impact on compiler design, we hope to convey both the breadth of possibility and the depth of the problems.

This book departs from some of the accepted conventions for compiler construction textbooks. For example, we use several different programming languages in the examples. It makes little sense to describe call-by-name parameter passing in C, so we use Algol-60. It makes little sense to describe tail-recursion in Fortran, so we use Scheme. This multi-lingual approach is realistic; over the course of the reader's career, the "language of the future" will change several times. (In the past thirty years, Algol-68, APL, PL/I, Smalltalk, C, Modula-3, C++, and even ADA have progressed from being "the language of the future" to being the "language of the future of the past.") Rather than provide ten to twenty homework-level questions at the end of each chapter, we present a couple of questions suitable for a mid-term or final examination. The questions are intended to provoke further thought about issues raised in the chapter. We do not provide solutions, because we anticipate that the best answer to any interesting question will change over the timespan of the reader's career.

Our Focus

In writing this book, we have made a series of conscious decisions that have a strong impact on both its style and its content. At a high level, our focus is to prune, to relate, and to engineer.

Prune Selection of material is an important issue in the design of a compiler construction course today. The sheer volume of information available has grown dramatically over the past decade or two. David Gries' classic book (*Compiler Construction for Digital Computers, John Wiley, 1971*) covers code optimization in a single chapter of less than forty pages. In contrast, Steve Muchnick's recent book (*Advanced Compiler Design and Implementation, Morgan Kauffman, 1997*) devotes thirteen chapters and over five hundred forty pages to the subject, while Bob Morgan's recent book (*Building an Optimizing Compiler, Digital Press, 1998*) covers the material in thirteen chapters that occupy about four hundred pages.

In laying out *Engineering a Compiler*, we have selectively pruned the material to exclude material that is redundant, that adds little to the student's insight and experience, or that has become less important due to changes in languages, in compilation techniques, or in systems architecture. For example, we have omitted operator precedence parsing, the LL(1) table construction algorithm, various code generation algorithms suitable for the PDP-11, and the UNION-FIND-based algorithm for processing Fortran `Equivalence` statements. In their place, we have added coverage of topics that include instruction scheduling, global register allocation, implementation of object-oriented languages, string manipulation, and garbage collection.

Relate Compiler construction is a complex, multifaceted discipline. The solutions chosen for one problem affect other parts of the compiler because they shape the input to subsequent phases and the information available in those phases. Current textbooks fail to clearly convey these relationships.

To make students aware of these relationships, we expose some of them directly and explicitly in the context of practical problems that arise in commonly-used languages. We present several alternative solutions to most of the problems that we address, and we discuss the differences between the solutions and their overall impact on compilation. We try to select examples that are small enough to be grasped easily, but large enough to expose the student to the full complexity of each problem. We reuse some of these examples in several chapters to provide continuity and to highlight the fact that several different approaches can be used to solve them.

Finally, to tie the package together, we provide a couple of questions at the end of each chapter. Rather than providing homework-style questions that have algorithmic answers, we ask exam-style questions that try to engage the student in a process of comparing possible approaches, understanding the tradeoffs between them, and using material from several chapters to address the issue at hand. The questions are intended as a tool to make the reader think, rather than acting as a set of possible exercises for a weekly homework assignment. (We believe that, in practice, few compiler construction courses assign weekly homework. Instead, these courses tend to assign laboratory exercises that provide the student with hands-on experience in language implementation.)

Engineer Legendary compilers, such as the BLISS-11 compiler or the Fortran-H compiler, have done several things well, rather than doing everything in moderation. We want to show the design issues that arise at each stage and how different solutions affect the resulting compiler and the code that it generates.

For example, a generation of students studied compilation from books that assume stack allocation of activation records. Several popular languages include features that make stack allocation less attractive; a modern textbook should present the tradeoffs between keeping activation records on the stack, keeping them in the heap, and statically allocating them (when possible).

When the most widely used compiler-construction books were written, most computers supported byte-oriented load and store operations. Several of them had hardware support for moving strings of characters from one memory location to another (the move character long instruction – `mvcl`). This simplified the treatment of character strings, allowing them to be treated as vectors of bytes (sometimes, with an implicit loop around the operation). Thus, compiler books scarcely mentioned support for strings.

Some RISC machines have weakened support for sub-word quantities; the compiler must worry about alignment; it may need to mask a character into a word using boolean operations. The advent of register-to-register load-store machines eliminated instructions like `mvcl`; today's RISC machine expects the compiler to optimize such operations and work together with the operating system to perform them efficiently.

Trademark Notices

In the text, we have used the registered trademarks of several companies.

IBM is a trademark of International Business Machines, Incorporated.

Intel and IA-64 are trademarks of Intel Corporation.

370 is a trademark of International Business Machines, Incorporated.

MC68000 is a trademark of Motorola, Incorporated.

PostScript is a registered trademark of Adobe Systems.

PowerPC is a trademark of (?Motorola or IBM?)

PDP-11 is a registered trademark of Digital Equipment Corporation, now a part of Compaq Computer.

Unix is a registered trademark of someone or other (maybe Novell).

VAX is a registered trademark of Digital Equipment Corporation, now a part of Compaq Computer.

Java may or may not be a registered trademark of SUN Microsystems, Incorporated.

Acknowledgements

We particularly thank the following people who provided us with direct and useful feedback on the form, content, and exposition of this book: Preston Briggs, Timothy Harvey, L. Taylor Simpson, Dan Wallach.

Contents

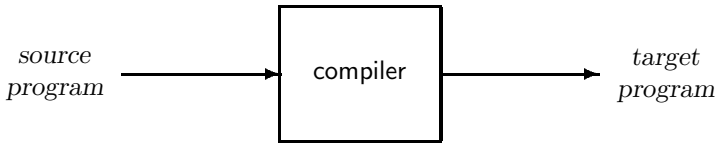
Chapter 1

An Overview of Compilation

1.1 Introduction

The role of computers in daily life is growing each year. Modern microprocessors are found in cars, microwave ovens, dishwashers, mobile telephones, GPSS navigation systems, video games and personal computers. Each of these devices must be programmed to perform its job. Those programs are written in some "programming" language – a formal language with mathematical properties and well-defined meanings – rather than a natural language with evolved properties and many ambiguities. Programming languages are designed for expressiveness, conciseness, and clarity. A program written in a programming language must be translated before it can execute directly on a computer; this translation is accomplished by a software system called a *compiler*. This book describes the mechanisms used to perform this translation and the issues that arise in the design and construction of such a translator.

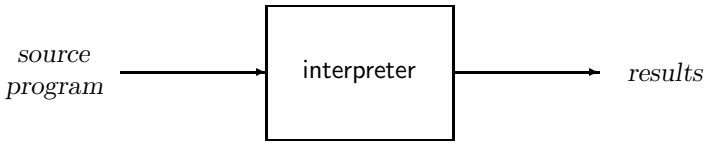
A compiler is just a computer program that takes as input an executable program and produces as output an equivalent executable program.



In a traditional compiler, the input language is a programming language and the output language is either assembly code or machine code for some computer system. However, many other systems qualify as compilers. For example, a typesetting program that produces PostScript can be considered a compiler. It takes as input a specification for how the document should look on the printed

page and it produces as output a PostScript file. PostScript is simply a language for describing images. Since the typesetting program takes an executable specification and produces another executable specification, it is a compiler.

The code that turns PostScript into pixels is typically an interpreter, not a compiler. An interpreter takes as input an executable specification and produces as output the results of executing the specification.



Interpreters and compilers have much in common. From an implementation perspective, interpreters and compilers perform many of the same tasks. For example, both must analyze the source code for errors in either syntax or meaning. However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the results. This book focuses on the problems that arise in building compilers. However, an implementor of interpreters may find much of the material relevant.

The remainder of this chapter presents a high-level overview of the translation process. It addresses both the problems of translation—what issues must be decided along the way—and the structure of a modern compiler—where in the process each decision should occur. Section 1.2 lays out two fundamental principles that every compiler must follow, as well as several other properties that might be desirable in a compiler. Section 1.3 examines the tasks that are involved in translating code from a programming language to code for a target machine. Section 1.4 describes how compilers are typically organized to carry out the tasks of translation.

1.2 Principles and Desires

Compilers are engineered objects—software systems built with distinct goals in mind. In building a compiler, the compiler writer makes myriad design decisions. Each decision has an impact on the resulting compiler. While many issues in compiler design are amenable to several different solutions, there are two principles that should not be compromised. The first principle that a well-designed compiler must observe is inviolable.

The compiler must preserve the meaning of the program being compiled

The code produced by the compiler must faithfully implement the "meaning" of the source-code program being compiled. If the compiler can take liberties with meaning, then it can always generate the same code, independent of input. For example, the compiler could simply emit a `nop` or a `return` instruction.

The second principle that a well-designed compiler must observe is quite practical.

The compiler must improve the source code in some discernible way.

If the compiler does not improve the code in some way, why should anyone invoke it? A traditional compiler improves the code by making it directly executable on some target machine. Other "compilers" improve their input in different ways. For example, `tpic` is a program that takes the specification for a drawing written in the graphics language `pic`, and converts it into LaTeX; the "improvement" lies in LaTeX's greater availability and generality. Some compilers produce output programs in the same language as their input; we call these "source-to-source" translators. In general, these systems try to restate the program in a way that will lead, eventually, to an improvement.

These are the two fundamental principles of compiler design.

This is an exciting era in the design and implementation of compilers. In the 1980's almost all compilers were large, monolithic systems. They took as input one of a handful of languages—typically Fortran or C—and produced assembly code for some particular computer. The assembly code was pasted together with the code produced by other compiles—including system libraries and application libraries—to form an executable. The executable was stored on a disk; at the appropriate time, the final code was moved from disk to main memory and executed.

Today, compiler technology is being applied in many different settings. These diverse compilation and execution environments are challenging the traditional image of a monolithic compiler and forcing implementors to reconsider many of the design tradeoffs that seemed already settled.

- Java has reignited interest in techniques like "just-in-time" compilation and "throw-away code generation." Java applets are transmitted across the Internet in some internal form, called Java bytecodes; the bytecodes are then interpreted or compiled, loaded, and executed on the target machine. The performance of the tool that uses the applet depends on the total time required to go from bytecodes on a remote disk to a completed execution on the local machine.

- Many techniques developed for large, monolithic compilers are being applied to analyze and improve code at link-time. In these systems, the compiler takes advantage of the fact that the entire program is available at link-time. The "link-time optimizer" analyzes the assembly code to derive knowledge about the run-time behavior of the program and uses that knowledge to produce code that runs faster.

- Some compilation techniques are being delayed even further—to run-time. Several recent systems invoke compilers during program execution to generate customized code that capitalizes on facts that cannot be known any

earlier. If the compile time can be kept small and the benefits are large, this strategy can produce noticeable improvements.

In each of these settings, the constraints on time and space differ, as do the expectations with regard to code quality.

The priorities and constraints of a specific project may dictate specific solutions to many design decisions or radically narrow the set of feasible choices. Some of the issues that may arise are:

1. *Speed:* At any point in time, there seem to be applications that need more performance than they can easily obtain. For example, our ability to simulate the behavior of digital circuits, like microprocessors, always lags far behind the demand for such simulation. Similarly, large physical problems such as climate modeling have an insatiable demand for computation. For these applications, the runtime performance of the compiled code is a critical issue. Achieving predictably good performance requires additional analysis and transformation at compile-time, typically resulting in longer compile times.

2. *Space:* Many applications impose tight restrictions on the size of compiled code. Usually, the constraints arise from either physical or economic factors; for example, power consumption can be a critical issue for any battery-powered device. Embedded systems outnumber general purpose computers; many of these execute code that has been committed permanently to a small "read-only memory" (ROM). Executables that must be transmitted between computers also place a premium on the size of compiled code. This includes many Internet applications, where the link between computers is slow relative to the speed of computers on either end.

3. *Feedback:* When the compiler encounters an incorrect program, it must report that fact back to the user. The amount of information provided to the user can vary widely. For example, the early Unix compilers often produced a simple and uniform message "`syntax error`." At the other end of the spectrum the Cornell PL/C system, which was designed as a "student" compiler, made a concerted effort to correct every incorrect program and execute it [23].

4. *Debugging:* Some transformations that the compiler might use to speed up compiled code can obscure the relationship between the source code and the target code. If the debugger tries to relate the state of the broken executable back to the source code, the complexities introduced by radical program transformations can cause the debugger to mislead the programmer. Thus, both the compiler writer and the user may be forced to choose between efficiency in the compiled code and transparency in the debugger. This is why so many compilers have a "debug" flag that causes the compiler to generate somewhat slower code that interacts more cleanly with the debugger.

5. *Compile-time efficiency:* Compilers are invoked frequently. Since the user usually waits for the results, compilation speed can be an important issue. In practice, no one likes to wait for the compiler to finish. Some users will be more tolerant of slow compiles, especially when code quality is a serious issue. However, given the choice between a slow compiler and a fast compiler that produces the same results, the user will undoubtedly choose the faster one.

Before reading the rest of this book, you should write down a prioritized list of the qualities that you want in a compiler. You might apply the ancient standard from software engineering—evaluate features as if you were paying for them with your own money! Examining your list will tell you a great deal about how you would make the various tradeoffs in building your own compiler.

1.3 High-level View of Translation

To gain a better understanding of the tasks that arise in compilation, consider what must be done to generate executable code for the following expression:

```
w ← w × 2 × x × y × z.
```

Let's follow the expression through compilation to discover what facts must be discovered and what questions must be answered.

1.3.1 Understanding the Input Program

The first step in compiling our expression is to determine whether or not

```
w ← w × 2 × x × y × z.
```

is a legal sentence in the programming language. While it might be amusing to feed random words to an English to Italian translation system, the results are unlikely to have meaning. A compiler must determine whether or not its input constitutes a well-constructed sentence in the source language. If the input is well-formed, the compiler can continue with translation, optimization, and code generation. If it is not, the compiler should report back to the user with a clear error message that isolates, to the extent possible, the problem with the sentence.

Syntax In a compiler, this task is called *syntax analysis*. To perform syntax analysis efficiently, the compiler needs:

1. a formal definition of the source language,
2. an efficient membership test for the source language, and
3. a plan for how to handle illegal inputs.

Mathematically, the source language is a set, usually infinite, of strings defined by some finite set of rules. The compiler's front end must determine whether the source program presented for compilation is, in fact, an element in that set of valid strings. In engineering a compiler, we would like to answer this membership question efficiently. If the input program is not in the set, and therefore not in the language, the compiler should provide useful and detailed feedback that explains where the input deviates from the rules.

To keep the set of rules that define a language small, the rules typically refer to words by their syntactic categories, or parts-of-speech, rather than individual words. In describing English, for example, this abstraction allows us to state that many sentences have the form

$$\textit{sentence} \rightarrow \quad \textit{subject} \quad \textit{verb} \quad \textit{object} \quad \textit{period}$$

rather than trying to enumerate the set of all sentences. For example, we use a syntactic variable, *verb*, to represent all possible verbs. With English, the reader generally recognizes many thousand words and knows the possible parts-of-speech that each can fulfill. For an unfamiliar string, the reader consults a dictionary. Thus, the syntactic structure of the language is based on a set of rules, or a grammar, and a system for grouping characters together to form words and for classifying those words into their syntactic categories.

This description-based approach to specifying a language is critical to compilation. We cannot build a software system that contains an infinite set of rules, or an infinite set of sentences. Instead, we need a finite set of rules that can generate (or specify) the sentences in our language. As we will see in the next two chapters, the finite nature of the specification does not limit the expressiveness of the language.

To understand whether the sentence "Compilers are engineered objects." is, in fact, a valid English sentence, we first establish that each word is valid. Next, each word is replaced by its syntactic category to create a somewhat more abstract representation of the sentence–

$$\textit{noun verb adjective noun period}$$

Finally, we try to fit this sequence of abstracted words into the rules for an English sentence. A working knowledge of English grammar might include the following rules:

1	*sentence*	$\rightarrow$	*subject verb object*
2	*subject*	$\rightarrow$	*noun*
3	*subject*	$\rightarrow$	*modifier noun*
4	*object*	$\rightarrow$	*noun*
5	*object*	$\rightarrow$	*modifier noun*
6	*modifier*	$\rightarrow$	*adjective*
7	*modifier*	$\rightarrow$	*adjectival phrase*
	...		

Here, the symbol → reads "derives" and means that an instance of the right hand side can be abstracted to the left hand side. By inspection, we can discover the following *derivation* for our example sentence.

Rule	*Prototype Sentence*
—	*sentence*
1	*subject verb object period*
2	*noun verb object period*
5	*noun verb modifier noun period*
6	*noun verb adjective noun period*

At this point, the prototype sentence generated by the derivation matches the abstract representation of our input sentence. Because they match, at this level of abstraction, we can conclude that the input sentence is a member of the language described by the grammar. This process of discovering a valid derivation for some stream of tokens is called parsing.

If the input is not a valid sentence, the compiler must report the error back to the user. Some compilers have gone beyond diagnosing errors; they have attempted to correct errors. When an error-correcting compiler encounters an invalid program, it tries to discover a "nearby" program that is well-formed. The classic game to play with an error-correcting compiler is to feed it a program written in some language it does not understand. If the compiler is thorough, it will faithfully convert the input into a syntactically correct program and produce executable code for it. Of course, the results of such an automatic (and unintended) transliteration are almost certainly meaningless.

Meaning A critical observation is that syntactic correctness depended entirely on the parts of speech, not the words themselves. The grammatical rules are oblivious to the difference between the noun "compiler" and the noun "tomatoes". Thus, the sentence "Tomatoes are engineered objects." is grammatically indistinguishable from "Compilers are engineered objects.", even though they have significantly different meanings. To understand the difference between these two sentences requires contextual knowledge about both compilers and vegetables.

Before translation can proceed, the compiler must determine that the program has a well-defined meaning. Syntax analysis can determine that the sentences are well-formed, at the level of checking parts of speech against grammatical rules. Correctness and meaning, however, go deeper than that. For example, the compiler must ensure that names are used in a fashion consistent with their declarations; this requires looking at the words themselves, not just at their syntactic categories. This analysis of meaning is often called either *semantic analysis* or *context-sensitive analysis*. We prefer the latter term, because it emphasizes the notion that the correctness of some part of the input, at the level of meaning, depends on the context that both precedes it and follows it.

A well-formed computer program specifies some computation that is to be performed when the program executes. There are many ways in which the expression

```
w ← w × 2 × x × y × z
```

might be ill-formed, beyond the obvious, syntactic ones. For example, one or more of the names might not be defined. The variable `x` might not have a value when the expression executes. The variables `y` and `z` might be of different types that cannot be multiplied together. Before the compiler can translate the expression, it must also ensure that the program has a well-defined meaning, in the sense that it follows some set of additional, extra-grammatical rules.

Compiler Organization The compiler's *front end* performs the analysis to check for syntax and meaning. For the restricted grammars used in programming languages, the process of constructing a valid derivation is easily automated. For efficiency's sake, the compiler usually divides this task into *lexical analysis,* or *scanning,* and *syntax analysis,* or *parsing.* The equivalent skill for "natural" languages is sometimes taught in elementary school. Many English grammar books teach a technique called "diagramming" a sentence—drawing a pictorial representation of the sentence's grammatical structure. The compiler accomplishes this by applying results from the study of formal languages [1]; the problems are tractable because the grammatical structure of programming languages is usually more regular and more constrained than that of a natural language like English or Japanese.

Inferring meaning is more difficult. For example, are `w`, `x`, `y`, and `z` declared as variables and have they all been assigned values previously? Answering these questions requires deeper knowledge of both the surrounding context and the source language's definition. A compiler needs an efficient mechanism for determining if its inputs have a legal meaning. The techniques that have been used to accomplish this task range from high-level, rule-based systems through *ad hoc* code that checks specific conditions.

Chapters 2 through 5 describe the algorithms and techniques that a compiler's front end uses to analyze the input program and determine whether it is well-formed, and to construct a representation of the code in some internal form. Chapter 6 and Appendix B, explore the issues that arise in designing and implementing the internal structures used throughout the compiler. The front end builds many of these structures.

1.3.2 Creating and Maintaining the Runtime Environment

Our continuing example concisely illustrates how programming languages provides their users with abstractions that simplify programming. The language defines a set of facilities for expressing computations; the programmer writes code that fits a model of computation implicit in the language definition. (Implementations of QuickSort in scheme, Java, and Fortran would, undoubtedly, look quite different.) These abstractions insulate the programmer from low-level details of the computer systems they use. One key role of a compiler is to put in place mechanisms that efficiently create and maintain these illusions. For example, assembly code is a convenient fiction that allows human beings to read and write short mnemonic strings rather than numerical codes for operations;