

Chapter 1

Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called compilers.

This book is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering.

In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

1.1 Language Processors

Simply stated, a compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

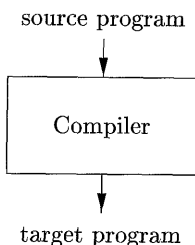


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

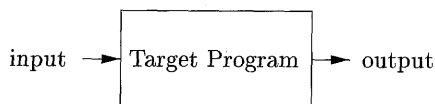


Figure 1.2: Running the target program

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

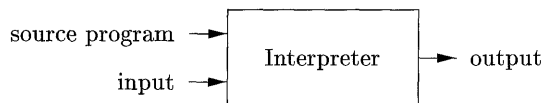


Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Example 1.1: Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input. □

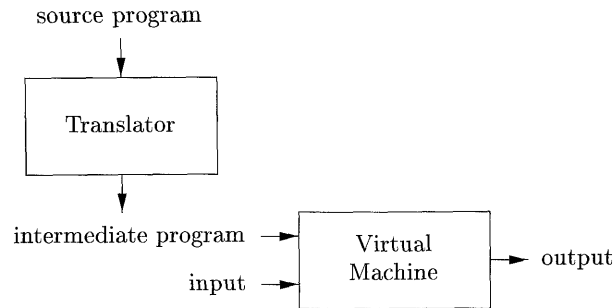


Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

1.1.1 Exercises for Section 1.1

Exercise 1.1.1: What is the difference between a compiler and an interpreter?

Exercise 1.1.2: What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

Exercise 1.1.3: What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

Exercise 1.1.4: A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

Exercise 1.1.5: Describe some of the tasks that an assembler needs to perform.

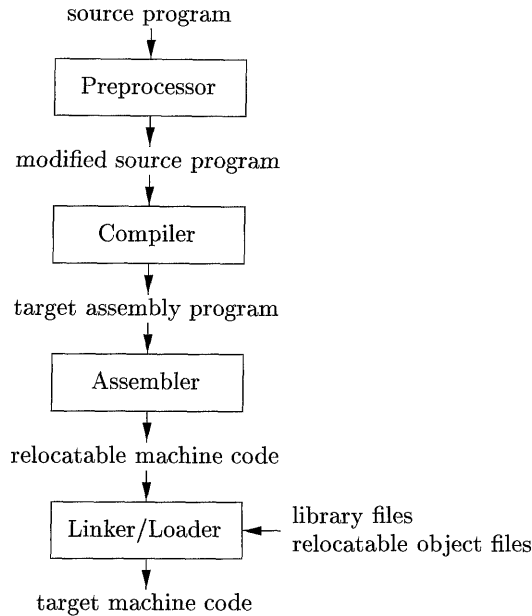


Figure 1.5: A language-processing system

1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*.

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the

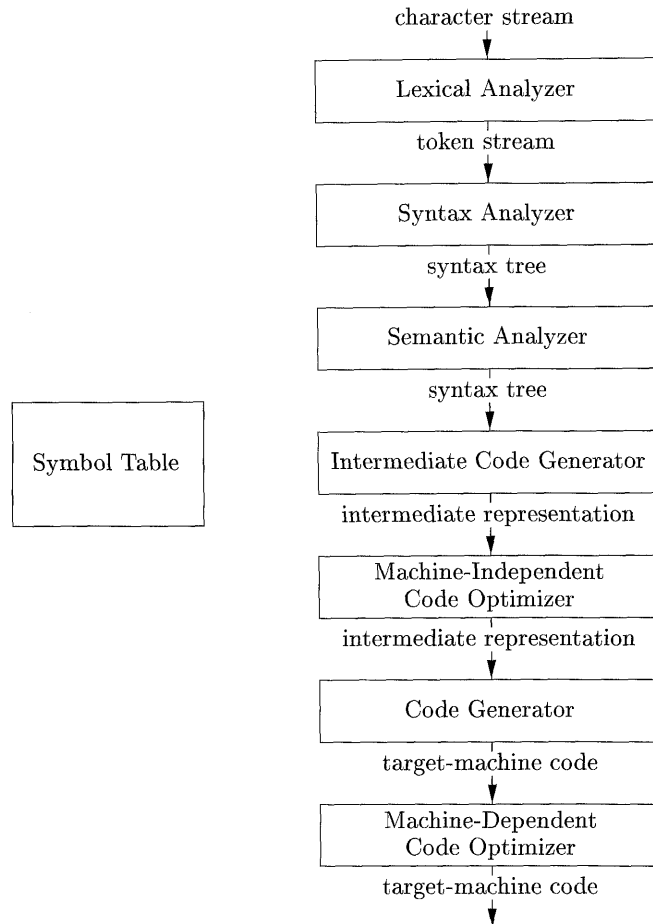


Figure 1.6: Phases of a compiler

entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program

and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces as output a *token* of the form

$$\langle token\text{-}name, attribute\text{-}value \rangle$$

that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

$$\text{position} = \text{initial} + \text{rate} * 60 \quad (1.1)$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. **position** is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for **position**. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol **=** is a lexeme that is mapped into the token $\langle \mathbf{=}, \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. **initial** is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for **initial**.
4. **+** is a lexeme that is mapped into the token $\langle \mathbf{+}, \rangle$.
5. **rate** is a lexeme that is mapped into the token $\langle \mathbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for **rate**.
6. ***** is a lexeme that is mapped into the token $\langle \mathbf{*}, \rangle$.
7. **60** is a lexeme that is mapped into the token $\langle \mathbf{60}, 1 \rangle$.

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$\langle \mathbf{id}, 1 \rangle \langle \mathbf{=}, \rangle \langle \mathbf{id}, 2 \rangle \langle \mathbf{+}, \rangle \langle \mathbf{id}, 3 \rangle \langle \mathbf{*}, \rangle \langle \mathbf{60}, 1 \rangle \quad (1.2)$$

In this representation, the token names **=**, **+**, and ***** are abstract symbols for the assignment, addition, and multiplication operators, respectively.

¹Technically speaking, for the lexeme **60** we should make up a token like $\langle \mathbf{number}, 4 \rangle$, where 4 points to the symbol table for the internal representation of integer 60 but we shall defer the discussion of tokens for numbers until Chapter 2. Chapter 3 discusses techniques for building lexical analyzers.

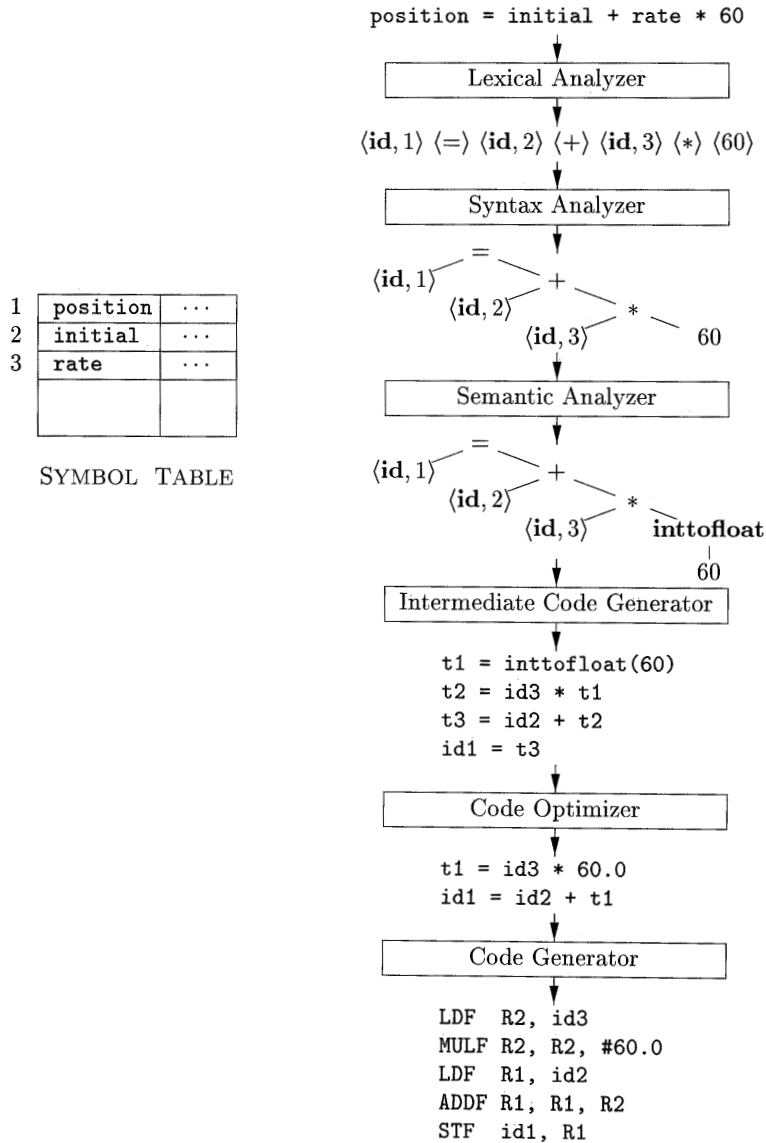


Figure 1.7: Translation of an assignment statement

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment

```
position = initial + rate * 60
```

are to be performed. The tree has an interior node labeled `*` with `(id, 3)` as its left child and the integer 60 as its right child. The node `(id, 3)` represents the identifier `rate`. The node labeled `*` makes it explicit that we must first multiply the value of `rate` by 60. The node labeled `+` indicates that we must add the result of this multiplication to the value of `initial`. The root of the tree, labeled `=`, indicates that we must store the result of this addition into the location for the identifier `position`. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. In Chapter 4 we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars. In Chapters 2 and 5 we shall see that syntax-directed definitions can help specify the translation of programming language constructs.

1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called *coercions*. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme `60` by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator `*` is applied to a floating-point number `rate` and an integer `60`. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **`inttfloat`**, which explicitly converts its integer argument into a floating-point number. Type checking and semantic analysis are discussed in Chapter 6.

1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

In Chapter 6, we consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
t1 = inttfloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some “three-address instructions” like the first and last in the sequence (1.3), above, have fewer than three operands.

In Chapter 6, we cover the principal intermediate representations used in compilers. Chapter 5 introduces techniques for syntax-directed translation that are applied in Chapter 6 to type checking and intermediate-code generation for typical programming language constructs such as expressions, flow-of-control constructs, and procedure calls.

1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, **t3** is used only once to transmit its value to **id1** so the optimizer can transform (1.3) into the shorter sequence

```
t1 = id3 * 60.0  
id1 = id2 + t1
```

(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called “optimizing compilers,” a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. The chapters from 8 on discuss machine-independent and machine-dependent optimizations in detail.

1.2.6 Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers **R1** and **R2**, the intermediate code in (1.4) might get translated into the machine code

```
LDF  R2,  id3  
MULF R2,  R2, #60.0  
LDF  R1,  id2  
ADDF R1,  R1, R2  
STF  id1, R1
```

(1.5)

The first operand of each instruction specifies a destination. The **F** in each instruction tells us that it deals with floating-point numbers. The code in

(1.5) loads the contents of address `id3` into register `R2`, then multiplies it with floating-point constant `60.0`. The `#` signifies that `60.0` is to be treated as an immediate constant. The third instruction moves `id2` into register `R1` and the fourth adds to it the value previously computed in register `R2`. Finally, the value in register `R1` is stored into the address of `id1`, so the code correctly implements the assignment statement (1.1). Chapter 8 covers code generation.

This discussion of code generation has ignored the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapter 2.

1.2.8 The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine. With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.
2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.
4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.

1.3 The Evolution of Programming Languages

The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

1.3.1 The Move to Higher-level Languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today.

In the following decades, many more languages were created with innovative features to help make programming easier, more natural, and more robust. Later in this chapter, we shall discuss some key features that are common to many modern programming languages.

Today, there are thousands of programming languages. They can be classified in a variety of ways. One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java. *Fourth-generation languages* are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting. The term *fifth-generation language* has been applied to logic- and constraint-based languages like Prolog and OPS5.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done. Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state. Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture. Many of today's languages, such as Fortran and C are von Neumann languages.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Simula 67 and Smalltalk are the earliest major object-oriented languages. Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

Scripting languages are interpreted languages with high-level operators designed for "gluing together" computations. These computations were originally

called “scripts.” Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages. Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

1.3.2 Impacts on Compilers

Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers. They had to devise algorithms and representations to translate and support the new language features. Since the 1940’s, computer architecture has evolved as well. Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users’ applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Compiler writing is challenging. A compiler by itself is a large program. Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code. Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.

A compiler must translate correctly the potentially infinite set of programs that could be written in the source language. The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

A study of compilers is also a study of how theory meets practice, as we shall see in Section 1.4.

The purpose of this text is to teach the methodology and fundamental ideas used in compiler design. It is not the intention of this text to teach all the algorithms and techniques that could be used for building a state-of-the-art language-processing system. However, readers of this text will acquire the basic knowledge and understanding to learn how to build a compiler relatively easily.

1.3.3 Exercises for Section 1.3

Exercise 1.3.1: Indicate which of the following terms:

- | | | |
|----------------------|----------------|---------------------|
| a) imperative | b) declarative | c) von Neumann |
| d) object-oriented | e) functional | f) third-generation |
| g) fourth-generation | h) scripting | |

apply to which of the following languages:

- | | | | | |
|---------|--------|----------|------------|---------|
| 1) C | 2) C++ | 3) Cobol | 4) Fortran | 5) Java |
| 6) Lisp | 7) ML | 8) Perl | 9) Python | 10) VB. |

1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

1.4.1 Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Similarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

1.4.2 The Science of Code Optimization

The term “optimization” in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. “Optimization” is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of “hacks.” Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter 9, how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as

machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimizations introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be performed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

1.5 Applications of Compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, yet have never, strictly speaking, written (even part of) a compiler for a major programming language. Compiler technology has other important uses as well. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

1.5.1 Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Example 1.2: The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution. When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the **register** keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written. \square

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization. In the following, we give an overview on the main language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages. The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type. All array accesses are checked to ensure that they lie within the bounds of the array. Java has no pointers and does not allow pointer arithmetic. It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. While all these features make programming easier, they incur a run-time overhead. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In addition, Java is designed to support portable and mobile code. Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, that is, at run time. Dynamic compilation has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code. In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

1.5.2 Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue

multiple operations in parallel. The Intel IA64 is a well-known example of such an architecture. All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors. Many scientific-computing and engineering applications are computation-intensive and can benefit greatly from parallel processing. Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

Memory Hierarchies

A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond. Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude. The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem. While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Using registers effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures (arrays, typically). It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data. We can also change the layout of code to improve the effectiveness of instruction caches.

1.5.3 Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as CISC (Complex Instruction-Set Computer). For example, CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept. Although the x86 architecture—the most popular microprocessor—has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself. Moreover, the most effective way to use a high-performance x86 machine is to use just its simple instructions.

Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Some of these ideas have made their way into the designs of embedded machines. Since entire systems can fit on a single chip, processors need no longer be prepackaged commodity units, but can be tailored to achieve better cost-effectiveness for a particular application. Thus, in contrast to general-purpose processors, where economies of scale have led computer architectures

to converge, application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

1.5.4 Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

Binary Translation

Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines. In particular, because of the domination of the x86 personal-computer market, most software titles are available as x86 code. Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.

Binary translation can also be used to provide backward compatibility. When the processor in the Apple Macintosh was changed from the Motorola MC 68040 to the PowerPC in 1994, binary translation was used to allow PowerPC processors run legacy MC 68040 code.

Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout. Unlike compilers for programming languages, these tools often take hours optimizing the circuit. Techniques to translate designs at higher levels, such as the behavior or functional level, also exist.

Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query

Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

Compiled Simulation

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements violating a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

In the balance of this section, we shall mention several ways in which program analysis, building upon techniques originally developed to optimize code in compilers, have improved software productivity. Of special importance are techniques that detect statically when a program might have a security vulnerability.

Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of data through a program. For example, if a pointer is assigned null and then immediately dereferenced, the program is clearly in error.

The same technology can be used to catch a variety of security holes, in which an attacker supplies a string or other data that is used carelessly by the program. A user-supplied string can be labeled with a type “dangerous.” If this string is not checked for proper format, then it remains “dangerous,” and if a string of this type is able to influence the control-flow of the code at some point in the program, then there is a potential security flaw.

Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array-bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system. Techniques have been developed to find buffer overflows in programs, but with limited success.

Had the program been written in a safe language that includes automatic range checking, this problem would not have occurred. The same data-flow analysis that is used to eliminate redundant range checks can also be used to locate buffer overflows. The major difference, however, is that failing to eliminate a range check would only result in a small run-time cost, while failing to identify a potential buffer overflow may compromise the security of the system. Thus, while it is adequate to use simple techniques to optimize range checks, sophisticated analyses, such as tracking the values of pointers across procedures, are needed to get high-quality results in error detection tools.

Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., “memory leaks”), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.