Chapter 1

Introduction

Recently the phrase ***user interface*** has received much attention in the computer industry. A *user interface* is the mechanism through which the user of a device communicates with the device. Since digital computers are programmed using a complex system of binary codes and memory addresses, we have developed sophisticated user interfaces, called programming languages, which enable us to specify computations in ways that seem more natural. This book will describe the implementation of this kind of interface, the rationale being that even if you never need to design or implement a programming language, the lessons learned here will still be valuable to you. You will be a better programmer as a result of understanding how programming languages are implemented, and you will have a greater appreciation for programming languages. In addition, the techniques which are presented here can be used in the construction of other user interfaces, such as the query language for a database management system.

1.1 What is a Compiler?

Recall from your study of assembly language or computer organization the kinds of instructions that the computer's CPU is capable of executing. In general, they are very simple, primitive operations. For example, there are often instructions which do the following kinds of operations: (1) add two numbers stored in memory, (2) move numbers from one location in memory to another, (3) move information between the CPU and memory. But there is certainly no single instruction capable of computing an arbitrary expression such as $((x-x_0)^2 + (x-x_1)^2)^{1/2}$, and there is no way to do the following with a single instruction:

```
if (array6[loc]<MAX) sum = 0; else array6[loc] = 0;
```

These capabilities are implemented with a software translator, known as a ***compiler***. The function of the compiler is to accept statements such as those above and translate them into sequences of machine language operations which, if loaded into memory and executed, would carry out the intended computation. It is important to bear in mind that when processing a statement such as x = x * 9; the compiler does not perform the multiplication. The compiler generates, as output, a sequence of instructions, including a "multiply" instruction.

Languages which permit complex operations, such as the ones above, are called ***high-level languages***, or ***programming languages***. A compiler accepts as input a program written in a particular high-level language and produces as output an equivalent program in machine language for a particular machine called the ***target*** machine. We say that two programs are ***equivalent*** if they always produce the same output when given the same input. The input program is known as the ***source program***, and its language is the ***source language***. The output program is known as the ***object program***, and its language is the ***object language.*** A compiler translates source language programs into equivalent object language programs. Some examples of compilers are:

A Java compiler for the Apple Macintosh
A COBOL compiler for the SUN
A C++ compiler for the Apple Macintosh

If a portion of the input to a Java compiler looked like this:

```
A = B + C * D;
```

the output corresponding to this input might look something like this:

```
LOD  R1,C        // Load the value of C into reg 1
MUL  R1,D        // Multiply the value of D by reg 1
STO  R1,TEMP1    // Store the result in TEMP1
LOD  R1,B        // Load the value of B into reg 1
ADD  R1,TEMP1    // Add value of Temp1 to register 1
STO  R1,TEMP2    // Store the result in TEMP2
MOV  A,TEMP2     // Move TEMP2 to A, the final result
```

The compiler must be smart enough to know that the multiplication should be done before the addition even though the addition is read first when scanning the input. The compiler must also be smart enough to know whether the input is a correctly formed program (this is called checking for proper ***syntax***), and to issue helpful error messages if there are syntax errors.

Note the somewhat convoluted logic after the Test instruction in Sample Problem 1.1(a) . Why didn't it simply branch to `L3` if the condition code indicated that the first operand (`X`) was greater than or equal to the second operand (`Temp1`), thus eliminating an unnecessary branch instruction and label? Some compilers might actually do this, but the point is that even if the architecture of the target machine permits it, many compilers

Sample Problem 1.1 (a)

Show the output of a Java native code compiler, in any typical assembly language, for the following Java input string:

```
while (x<a+b) x = 2*x;
```

Solution:

```
L1: LOD   R1,A        // Load A into reg. 1
    ADD   R1,B        // Add B to reg. 1
    STO   R1,Temp1    // Temp1 = A + B
    CMP   X,Temp1     // Test for while condition
    BL    L2          // Continue with loop if X<Temp1
    B     L3          // Terminate loop
L2: LOD   R1,='2'
    MUL   R1,X
    STO   R1,X        // X = 2*X
    B     L1          // Repeat loop
L3:
```

will not generate optimal code. In designing a compiler, the primary concern is that the object program be semantically equivalent to the source program (i.e. that they mean the same thing, or produce the same output for a given input). Object program efficiency is important, but not as important as correct code generation.

What are the advantages of a high-level language over machine or assembly language? (1) Machine language (and even assembly language) is difficult to work with and difficult to maintain. (2) With a high-level language you have a much greater degree of machine independence and portability from one kind of computer to another (as long as the other machine has a compiler for that language). (3) You don't have to retrain application programmers every time a new machine (with a new instruction set) is introduced. (4) High-level languages may support data abstraction (through data structures) and program abstraction (procedures and functions).

What are the disadvantages of high-level languages? (1) The programmer doesn't have complete control of the machine's resources (registers, interrupts, I/O buffers). (2) The compiler may generate inefficient machine language programs. (3) Additional software – the compiler – is needed in order to use a high-level language. As compiler development and hardware have improved over the years, these disadvantages have become less problematic. Consequently, most programming today is done with high-level languages.

An ***interpreter*** is software which serves a purpose very similar to that of a compiler. The input to an interpreter is a program written in a high-level language, but rather than generating a machine language program, the interpreter actually carries out the

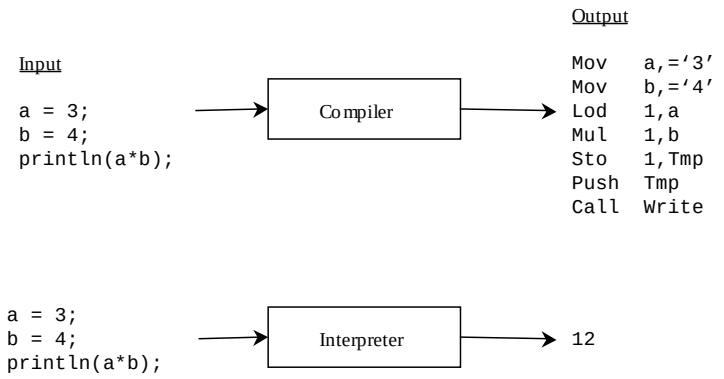


Figure 1.1 A Compiler and Interpreter Producing Very Different Output for the Same Input

computations specified in the source program. In other words, the output of a compiler is a program, whereas the output of an interpreter is the source program's output. Figure 1.1 shows that although the input may be identical, compilers and interpreters produce very different output. Nevertheless, many of the techniques used in designing compilers are also applicable to interpreters.

Students are often confused about the difference between a compiler and an interpreter. Many commercial compilers come packaged with a built-in edit-compile-run front end. In effect, the student is not aware that after compilation is finished, the object program must be loaded into memory and executed, because this all happens automatically. As larger programs are needed to solve more complex problems, programs are divided into manageable source modules, each of which is compiled separately to an object module. The object modules can then be linked to form a single, complete, machine language program. In this mode, it is more clear that there is a distinction between ***compile time***, the time at which a source program is compiled, and ***run time,*** the time at which the resulting object program is loaded and executed. Syntax errors are reported by the compiler at compile time and are shown at the left, below, as compile-time errors. Other kinds of errors not generally detected by the compiler are called ***run-time errors*** and are shown at the right below:

Compile-Time Errors

```
a = ((b+c)*d;

if x<b fn1();
   else fn2();
```

Run-Time Errors

```
x = a-a;
y = 100/x;    // division by 0

Integer n[] = new Integer[7];
n[8] = 16; // invalid subscript
```

Sample Problem 1.1 (b)

Show the compiler output and the interpreter output for the following Java source code:

```
for (i=1; i<=4; i++) System.out.println (i*3);
```

Solution:

Compiler

```
    LOD     R1,='4'
    STO     R1,Temp1
    MOV     i,='1'
L1: CMP     i,Temp1
    BH      L2      {Branch if i>Temp1}
    LOD     R1,i
    MUL     R1,='3'
    STO     R1,Temp2
    PUSH    Temp2
    CALL    Println
    ADD     i,='1'  {Add 1 to i}
    B       L1
L2:
```

Interpreter

```
3  6  9  12
```

It is important to remember that a compiler is a program, and it must be written in some language (machine, assembly, high-level). In describing this program, we are dealing with three languages: (1) the ***source*** language, i.e. the input to the compiler, (2) the ***object*** language, i.e. the output of the compiler, and (3) the language in which the compiler is written, or the language in which it exists, since it might have been translated into a language foreign to the one in which it was originally written. For example, it is possible to have a compiler that translates Java programs into Macintosh machine language. That compiler could have been written in the C language, and translated into Macintosh (or some other) machine language. Note that if the language in which the compiler is written is a machine language, it need not be the same as the object language. For example, a compiler that produces Macintosh machine language could run on a Sun computer. Also, the object language need not be a machine or assembly language, but could be a high-level language. A concise notation describing compilers is given by Aho[1986] and is shown in Figure 1.2. In these diagrams, the large C stands for Compiler (not the C programming language), the superscript describes the intended translation of the compiler, and the subscript shows the language in which the compiler exists. Figure 1.2 (a) shows a Java compiler for the Macintosh. Figure 1.2 (b) shows a compiler which translates Java programs into equivalent Macintosh machine language, but it exists in Sun machine language, and consequently it will run only on a Sun. Figure 1.2 (c) shows a compiler which translates PC machine language programs into equivalent Java programs. It is written in Ada and will not run in that form on any machine.

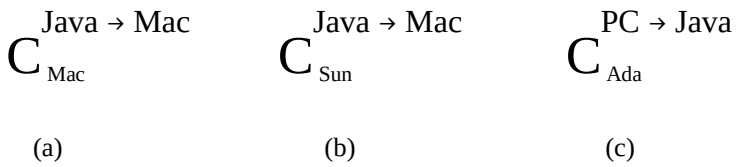


Figure 1.2 Big C notation for compilers: (a) A Java compiler that runs on the Mac (b) A Java compiler that generates Mac programs and runs on a Sun computer (c) A compiler that translates PC programs into Java and is written in Ada.

In this notation the name of a machine represents the machine language for that machine; i.e. Sun represents Sun machine language, and PC represents PC machine language (i.e. Intel Pentium).

Sample Problem 1.1 (c)

Using the big C notation of Figure 1.2, show each of the following compilers:

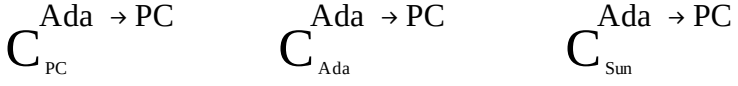
(a) An Ada compiler which runs on the PC and compiles to the PC machine language.

(b) An Ada compiler which compiles to the PC machine language, but which is written in Ada.

(c) An Ada compiler which compiles to the PC machine language, but runs on a Sun.

Solution:

(a) (b) (c)



Exercises 1.1

1. Show *assembly language* for a machine of your choice, corresponding to each of the following Java statements:

(a)
```
a = b + c;
```
(b)
```
a = (b+c) * (c-d);
```
(c)
```
for (i=1; i<=10; i++) a = a+i;
```

2. Show the difference between compiler output and interpreter output for each of the following source inputs:

(a)
```
a = 12;
b = 6;
c = a+b;
println (c+a+b);
```

(b)
```
a = 12;
b = 6;
if (a<b) println (a);
     else println (b);
```

(c)
```
a = 12;
b = 6;
while (b<a)
     {  a = a-1;
        println (a+b);
     }
```

3. Which of the following Java source errors would be detected at compile time, and which would be detected at run time?

(a)
```
a = b+c = 3;
```

(b)
```
if (x<3) a = 2
      else a = x;
```

(c)
```
if (a>0) x = 20;
     else if (a<0) x = 10;
          else x = x/a;
```

(d)
```
MyClass x [] = new MyClass[100];
x[100] = new MyClass();
```

4. Using the big C notation, show the symbol for each of the following:

(a) A compiler which translates COBOL source programs to PC machine language and runs on a PC.
(b) A compiler, written in Java, which translates FORTRAN source programs to Mac machine language.
(c) A compiler, written in Java, which translates Sun machine language programs to Java.

1.2 The Phases of a Compiler

The student is reminded that the input to a compiler is simply a string of characters. Students often assume that a particular interpretation is automatically "understood" by the computer (`sum = sum + 1;` is obviously an assignment statement, but the computer must be programmed to determine that this is the case).

In order to simplify the compiler design and construction process, the compiler is implemented in phases. In general, a compiler consists of at least three phases: (1) lexical analysis, (2) syntax analysis, and (3) code generation. In addition, there could be other optimization phases employed to produce efficient object programs.

1.2.1 Lexical Analysis (Scanner) – Finding the Word Boundaries

The first phase of a compiler is called ***lexical analysis*** (and is also known as a ***lexical scanner***). As implied by its name, lexical analysis attempts to isolate the "words" in an input string. We use the word "word" in a technical sense. A word, also known as a ***lexeme***, a lexical ***item***, or a lexical ***token***, is a string of input characters which is taken as a unit and passed on to the next phase of compilation. Examples of words are:

(1) ***key words*** - `while`, `void`, `if`, `for`, ...
(2) ***identifiers*** - declared by the programmer
(3) ***operators*** - `+`, `-`, `*`, `/`, `=`, `==`, ...
(4) ***numeric constants*** - numbers such as `124`, `12.35`, `0.09E-23`, etc.
(5) ***character constants*** - single characters or strings of characters enclosed in quotes.
(6) ***special characters*** - characters used as delimiters such as `. ( ) , ; :`
(7) ***comments*** - ignored by subsequent phases. These must be identified by the scanner, but are not included in the output.

The output of the lexical phase is a stream of ***tokens*** corresponding to the words described above. In addition, this phase builds tables which are used by subsequent phases of the compiler. One such table, called the ***symbol table***, stores all identifiers used in the source program, including relevant information and attributes of the identifiers. In block-structured languages it may be preferable to construct the symbol table during the syntax analysis phase because program blocks (and identifier scopes) may be nested.

1.2.2 Syntax Analysis Phase

The ***syntax analysis phase*** is often called the ***parser***. This term is critical to understanding both this phase and the study of languages in general. The parser will check for proper syntax, issue appropriate error messages, and determine the underlying structure of the source program. The output of this phase may be a stream of ***atoms*** or a collection of ***syntax trees***. An atom is an atomic operation, or one that is generally available with one (or just a few) machine language instruction(s) on most target machines. For example, `MULT`, `ADD`, and `MOVE` could represent atomic operations for multiplication, addition,

Sample Problem 1.2 (a)

Show the token classes, or "words", put out by the lexical analysis phase corresponding to this Java source input:

```
sum = sum + unit * /* accumulate sum */ 1.2e-12 ;
```

Solution:

```
identifier              (sum)
assignment              (=)
identifier              (sum)
operator                (+)
identifier              (unit)
operator                (*)
numeric constant        (1.2e-12)
```

and moving data in memory. Each operation could have 0 or more operands also listed in the atom: (operation, operand1, operand2, operand3). The meaning of the following atom would be to add `A` and `B`, and store the result into `C`:

```
(ADD, A, B, C)
```

In Sample Problem 1.2 (b), below, each atom consists of three or four parts: an operation, one or two operands, and a result. Note that the compiler must put out the MULT atom before the ADD atom, despite the fact that the addition is encountered first in the source statement.

To implement transfer of control, we could use label atoms, which serve only to mark a spot in the object program to which we might wish to branch in implementing a control structure such as `if` or `while`. A label atom with the name `L1` would be `(LBL,L1)`. We could use a jump atom for an unconditional branch, and a test atom for a conditional branch: The atom `(JMP, L1)` would be an unconditional branch to the label

Sample Problem 1.2 (b)

Show atoms corresponding to the following Java statement:

```
A = B + C * D ;
```

Solution:

```
(MULT,C,D,TEMP1)
(ADD,B,TEMP1,TEMP2)
(MOVE,TEMP2,A)
```

Sample Problem 1.2 (c)

Show atoms corresponding to the Java statement:

```
while (A<=B) A = A + 1;
```

Solution:

```
(LBL, L1)
(TEST, A, <=, B, L2)
(JMP, L3)
(LBL, L2)
(ADD, A, 1, A)
(JMP, L1)
(LBL, L3)
```

`L1`. The atom `(TEST, A, <=, B, L2)` would be a conditional branch to the label `L2`, if `A<=B` is true.

Some parsers put out syntax trees as an intermediate data structure, rather than atom strings. A syntax tree indicates the structure of the source statement, and object code can be generated directly from the syntax tree. A syntax tree for the expression `A = B + C * D` is shown in Figure 1.3, below.

In syntax trees, each interior node represents an operation or control structure and each leaf node represents an operand. A statement such as `if (Expr) Stmt1 else Stmt2` could be implemented as a node having three children – one for the conditional expression, one for the true part (`Stmt1`), and one for the `else` statement (`Stmt2`). The `while` control structure would have two children – one for the loop condition, and one for the statement to be repeated. The compound statement could be treated a few different ways. The compound statement could have an unlimited number of children, one for each statement in the compound statement. The other way would be to treat the semicolon like a statement concatenation operator, yielding a binary tree.

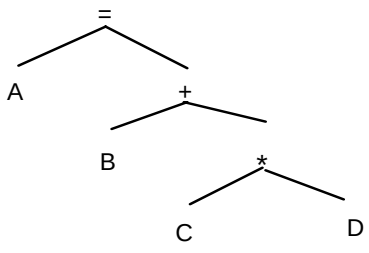


Figure 1.3 A Syntax Tree for `A = B + C * D`

Once a syntax tree has been created, it is not difficult to generate code from the syntax tree; a *postfix* traversal of the tree is all that is needed. In a ***postfix traversal***, for each node, N, the algorithm visits all the subtrees of N, and visits the node N last, at which point the instruction(s) corresponding to node N can be generated.

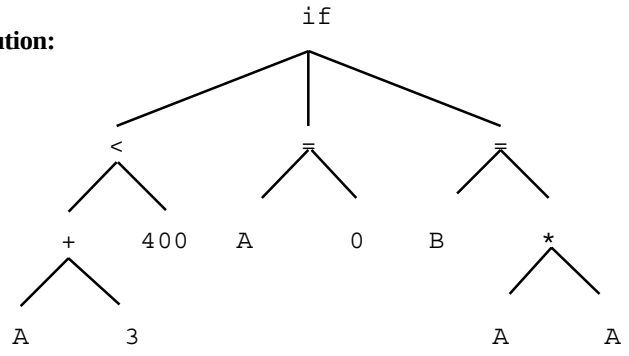
Sample Problem 1.2 (d)

Show a syntax tree for the Java statement

```
if (A+3<400) A = 0; else B = A*A;
```

Assume that an `if` statement consists of three subtrees, one for the condition, one for the consequent statement, and one for the `else` statement, if necessary.

Solution:



Many compilers also include a phase for ***semantic analysis***. In this phase the data types are checked, and type conversions are performed when necessary. The compiler may also be able to detect some semantic errors, such as division by zero, or the use of a null pointer.

1.2.3 Global Optimization

The ***global optimization*** phase is optional. Its purpose is simply to make the object program more efficient in space and/or time. It involves examining the sequence of atoms put out by the parser to find redundant or unnecessary instructions or inefficient code. Since it is invoked before the code generator, this phase is often called machine-independent optimization. For example, in the following program segment:

```
          stmt1
          go to label1
          stmt2
          stmt3
label2:   stmt4
```

`stmt2` and `stmt3` can never be executed. They are unreachable and can be eliminated from the object program. A second example of global optimization is shown below:

```
for (i=1; i<=100000; i++)
   {  x = Math.sqrt (y);      // square root method
      System.out.println (x+i);
   }
```

In this case, the assignment to `x` need not be inside the loop since `y` doesn't change as the loop repeats (it is a ***loop invariant***). In the global optimization phase, the compiler would move the assignment to `x` out of the loop in the object program:

```
x = Math.sqrt (y);                  // loop invariant
for (i=1; i<=100000; i++)
      System.out.println (x+i);
```

This would eliminate 99,999 unnecessary calls to the `sqrt` method at run time.

The reader is cautioned that global optimization can have a serious impact on run-time debugging. For example, if the value of `y` in the above example was negative, causing a run-time error in the `sqrt` function, the user would be unaware of the actual location of that portion of code which called the `sqrt` function, because the compiler would have moved the offending statement (usually without informing the programmer). Most compilers that perform global optimization also have a switch with which the user can turn optimization on or off. When debugging the program, the switch would be off. When the program is correct, the switch would be turned on to generate an optimized version for the user. One of the most difficult problems for the compiler writer is making sure that the compiler generates optimized and unoptimized object modules, from the same source module, which are equivalent.

1.2.4 Code Generation

Most Java compilers produce an intermediate form, known as Byte Code, which can be interpreted by the Java run-time environment. In this book we will be assuming that our compiler is to produce native code for a particular machine.

It is assumed that the student has had some experience with assembly language and machine language, and is aware that the computer is capable of executing only a limited number of primitive operations on operands with numeric memory addresses, all encoded as binary values. In the ***code generation*** phase, atoms or syntax trees are translated to machine language (binary) instructions, or to assembly language, in which case the assembler is invoked to produce the object program. Symbolic addresses (statement labels) are translated to relocatable memory addresses at this time.

For target machines with several CPU registers, the code generator is responsible for ***register allocation***. This means that the compiler must be aware of which registers are

being used for particular purposes in the generated program, and which become available as code is generated.

For example, an ADD atom might be translated to three machine language instructions: (1) load the first operand into a register, (2) add the second operand to that register, and (3) store the result, as shown for the atom (ADD, A, B, Temp):

```
LOD    R1,A        // Load A into reg. 1
ADD    R1,B        // Add B to reg. 1
STO    R1,Temp     // Store reg. 1 in Temp
```

In Sample Problem 1.2 (e), below, the destination for the MOV instruction is the first operand, and the source is the second operand, which is the reverse of the operand positions in the MOVE atom.

Sample Problem 1.2 (e)

Show assembly language instructions corresponding to the following atom string:

```
(ADD, A, B, Temp1)
(TEST, A, ==, B, L1)
(MOVE, Temp1, A)
(LBL, L1)
(MOVE, Temp1, B)
```

Solution:

```
      LOD      R1,A
      ADD      R1,B
      STO      R1,Temp1        // ADD, A, B, Temp1
      CMP      A,B
      BE       L1              // TEST, A, ==, B, L1
      MOV      A,Temp1         // MOVE, Temp1, A
L1:   MOV      B,Temp1         // MOVE, Temp1, B
```

It is not uncommon for the object language to be another high-level language. This is done in order to improve portablility of the language being implemented.

1.2.5 Local Optimization

The ***local optimization*** phase is also optional and is needed only to make the object program more efficient. It involves examining sequences of instructions put out by the code generator to find unnecessary or redundant instructions. For this reason, local optimization is often called machine-dependent optimization. An *addition operation* in

the source program might result in three instructions in the object program: (1) Load one operand into a register, (2) add the other operand to the register, and (3) store the result. Consequently, the expression `A + B + C` in the source program might result in the following instructions as code generator output:

```
LOD   R1,A          // Load A into register 1
ADD   R1,B          // Add B to register 1
STO   R1,TEMP1      // Store the result in TEMP1*
LOD   R1,TEMP1      // Load result into reg 1*
ADD   R1,C          // Add C to register 1
STO   R1,TEMP2      // Store the result in TEMP2
```

Note that some of these instructions (those marked with * in the comment) can be eliminated without changing the effect of the program, making the object program both smaller and faster:

```
LOD   R1,A          // Load A into register 1
ADD   R1,B          // Add B to register 1
ADD   R1,C          // Add C to register 1
STO   R1,TEMP       // Store the result in TEMP
```

A diagram showing the phases of compilation and the output of each phase is shown in Figure 1.4, at right. Note that the optimization phases may be omitted (i.e. the atoms may be passed directly from the Syntax phase to the Code Generator, and the instructions may be passed directly from the Code Generator to the compiler output file.)

A word needs to be said about the flow of control between phases. One way to handle this is for each phase to run from start to finish separately, writing output to a disk file. For example, lexical analysis is started and creates a file of tokens. Then, after the entire source program has been scanned, the syntax analysis phase is started, reads the entire file of tokens, and creates a file of atoms. The other phases continue in this manner; this would be a ***multiple pass compiler*** since the input is scanned several times.

Another way for flow of control to proceed would be to start up the syntax analysis phase first. Each time it needs a token it calls the lexical analysis phase as a subroutine, which reads enough source characters to produce one token, and returns it to the parser. Whenever the parser has scanned enough source code to produce an atom, the atom is con-

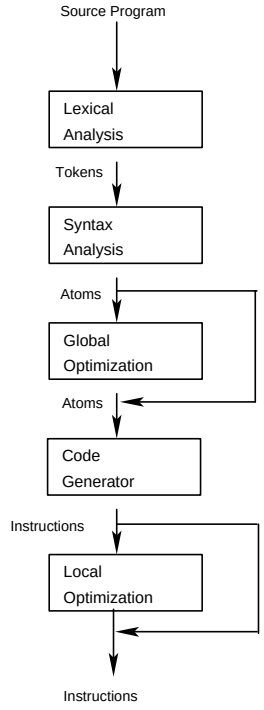


Figure 1.4 The Phases of a Compiler

verted to object code by calling the code generator as a subroutine; this would be a ***single pass compiler***.

Exercises 1.2

1. Show the *lexical tokens* corresponding to each of the following Java source inputs:

(a)
```
for (i=1; i<5.1e3; i++) func1(x);
```
(b)
```
if (sum!=133) /* sum = 133 */
```
(c)
```
) while ( 1.3e-2 if &&
```
(d)
```
if 1.2.3 < 6
```

2. Show the sequence of atoms put out by the parser, and show the *syntax tree* corresponding to each of the following Java source inputs:

(a)
```
a = (b+c) * d;
```
(b)
```
if (a<b) a = a + 1;
```
(c)
```
while (x>1)
    { x = x/2;
      i = i+1;
    }
```
(d)
```
a = b - c - d/a + d * a;
```

3. Show an example of a *Java statement* which indicates that the order in which the two operands of an `ADD` are evaluated can cause different results:

```
operand1 + operand2
```

4. Show how each of the following *Java source inputs* can be optimized using global optimization techniques:

(a)
```
for (i=1; i<=10; i++)
   { x = i + x;
     a[i] = a[i-1];
     y = b * 4;
   }
```

(b)
```
for (i=1; i<=10; i++)
   { x = i;
     y = x/2;
     a[i] = x;
   }
```

(c)
```
if (x>0) {x = 2; y = 3;}
   else {y = 4; x = 2;}
```

(d)
```
if (x>0) x = 2;
else if (x<=0) x = 3;
  else x = 4;
```

5. Show, in *assembly language* for a machine of your choice, the output of the code generator for the following atom string:

```
(ADD,A,B,Temp1)
(SUB,C,D,Temp2)
(TEST,Temp1,<,Temp2,L1)
(JUMP,L2)
(LBL,L1)
(MOVE,A,B)
(JUMP,L3)
(LBL,L2)
(MOVE,B,A)
(LBL,L3)
```

6. Show a *Java source statement* which might have produced the atom string in Problem 5, above.

7. Show how each of the following *object code segments* could be optimized using local optimization techniques:

(a)
```
        LD   R1,A
        MULT R1,B
        ST   R1,Temp1
        LD   R1,Temp1
        ADD  R1,C
        ST   R1,Temp2
```

(b)
```
        LD   R1,A
        ADD  R1,B
        ST   R1,Temp1
        MOV  C,Temp1
```

(c)
```
        CMP  A,B
        BH   L1
        B    L2
L1:     MOV  A,B
        B    L3
L2:     MOV  B,A
L3:
```

1.3 Implementation Techniques

By this point it should be clear that a compiler is not a trivial program. A new compiler, with all optimizations, could take over a person-year to implement. For this reason, we are always looking for techniques or shortcuts which will speed up the development process. This often involves making use of compilers, or portions of compilers, which have been developed previously. It also may involve special compiler generating tools, such as ***lex*** and ***yacc*** , which are part of the Unix environment, or newer tools such as JavaCC or SableCC.

In order to describe these implementation techniques graphically, we use the method shown below, in Figure 1.5, in which the computer is designated with a rectangle, and its name is in a smaller rectangle sitting on top of the computer. In all of our examples the program loaded into the computer's memory will be a compiler. It is important to remember that a computer is capable of running only programs written in the machine language of that computer. The input and output (also compilers in our examples) to the program in the computer are shown to the left and right, respectively.

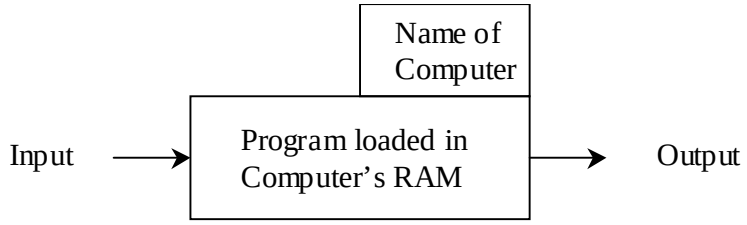


Figure 1.5 Notation for a Program Running on a Computer

Since a compiler does not change the purpose of the source program, the superscript on the output is the same as the superscript on the input ($X \rightarrow Y$), as shown in Figure 1.6, below. The subscript language (the language in which it exists) of the executing compiler (the one inside the computer), M, must be the machine language of the computer on which it is running. The subscript language of the input, S, must be the same as the source language of the executing compiler. The subscript language of the output, O, must be the same as the object language of the executing compiler.

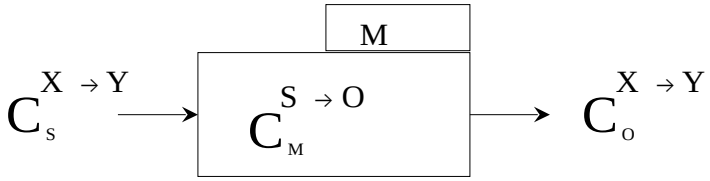
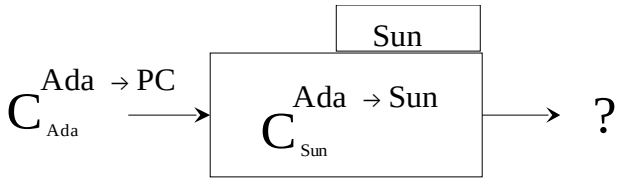


Figure 1.6 Notation for a compiler being translated to a different language

In the following sections it is important to remember that a compiler does not change the purpose of the source program; a compiler ***translates*** the source program into an equivalent program in another language (the object program). The source program could, itself, be a compiler. If the source program is a compiler which translates language A into language B, then the object program will also be a compiler which translates language A into language B.

Sample Problem 1.3

Show the output of the following compilation using the big C notation.



Solution:

$$C^{Ada \rightarrow PC}_{Sun}$$

1.3.1 Bootstrapping

The term ***bootstrapping*** is derived from the phrase "pull yourself up by your bootstraps" and generally involves the use of a program as input to itself (the student may be familiar with *bootstrapping loaders* which are used to initialize a computer just after it has been switched on, hence the expression "to boot" a computer).

In this case, we are talking about bootstrapping a compiler, as shown in Figure 1.7. We wish to implement a Java compiler for the Sun computer. Rather than writing the whole thing in machine (or assembly) language, we instead choose to write two easier programs. The first is a compiler for a subset of Java, written in machine (assembly) language. The second is a compiler for the full Java language written in the Java subset language. In Figure 1.7 the subset language of Java is designated "Sub", and it is simply Java, without several of the superfluous features, such as enumerated types, unions, switch statements, etc. The first compiler is loaded into the computer's memory and the second is used as input. The output is the compiler we want – i.e. a compiler for the full Java language, which runs on a Sun and produces object code in Sun machine language.

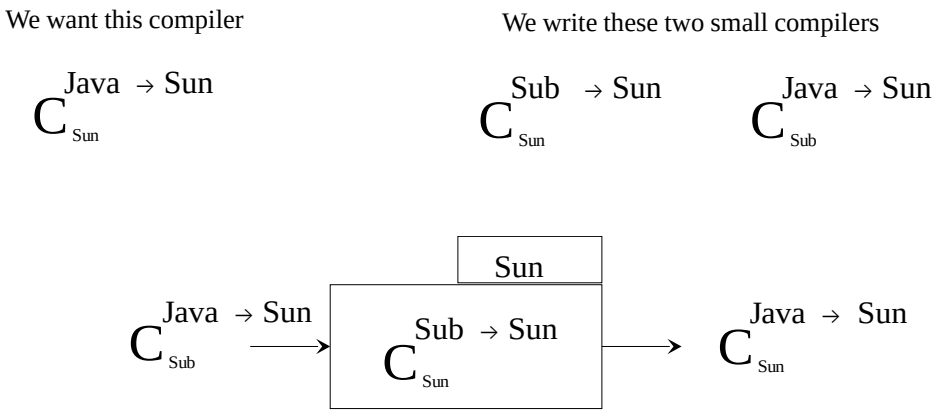


Figure 1.7 Bootstrapping Java onto a Sun Computer

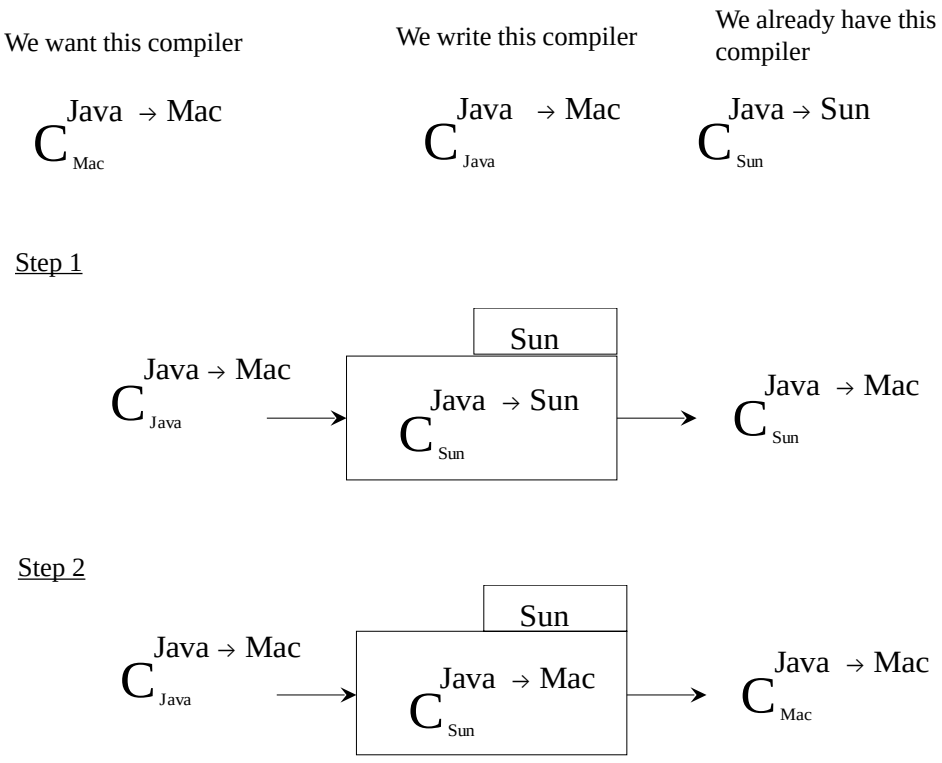


Figure 1.8 Cross compiling Java from a Sun to a Mac computer

In actual practice this is an iterative process, beginning with a small subset of Java, and producing, as output, a slightly larger subset. This is repeated, using larger and larger subsets, until we eventually have a compiler for the complete Java language.

1.3.2 Cross Compiling

New computers with enhanced (and sometimes reduced) instruction sets are constantly being produced in the computer industry. The developers face the problem of producing a new compiler for each existing programming language each time a new computer is designed. This problem is simplified by a process called ***cross compiling***.

Cross compiling is a two-step process and is shown in Figure 1.8. Suppose that we have a Java compiler for the Sun, and we develop a new machine called a Mac. We now wish to produce a Java compiler for the Mac without writing it entirely in machine (assembly) language; instead, we write the compiler in Java. Step one is to use this compiler as input to the Java compiler on the Sun. The output is a compiler that translates Java into Mac machine language, and which runs on a Sun. Step two is to load this compiler into the Sun and use the compiler we wrote in Java as input once again. This time the output is a Java compiler for the Mac which runs on the Mac, i.e. the compiler we wanted to produce.

Note that this entire process can be completed before a single Mac has been built. All we need to know is the architecture (the instruction set, instruction formats, addressing modes, ...) of the Mac.

1.3.3 Compiling To Intermediate Form

As we mentioned in our discussion of interpreters above, it is possible to compile to an ***intermediate form***, which is a language somewhere between the source high-level language and machine language. The stream of atoms put out by the parser is a possible example of an intermediate form. The primary advantage of this method is that one needs only one translator for each high-level language to the intermediate form (each of these is called a ***front end***) and only one translator (or interpreter) for the intermediate form on each computer (each of these is called a ***back end***). As depicted in Figure 1.9, for three high-level languages and two computers we would need three translators to intermediate form and two code generators (or interpreters) – one for each computer. Had we not used the intermediate form, we would have needed a total of six different compilers. In general, given `n` high-level languages and `m` computers, we would need `n x m` compilers. Assuming that each front end and each back end is half of a compiler, we would need `(n+m)/2` compilers using intermediate form.

A very popular intermediate form for the PDP-8 and Apple II series of computers, among others, called ***p-code***, was developed several years ago at the University of California at San Diego. Today, high-level languages such as C are commonly used as an intermediate form. The Java Virtual Machine (i.e. Java byte code) is another intermediate form which has been used extensively on the Intenet.

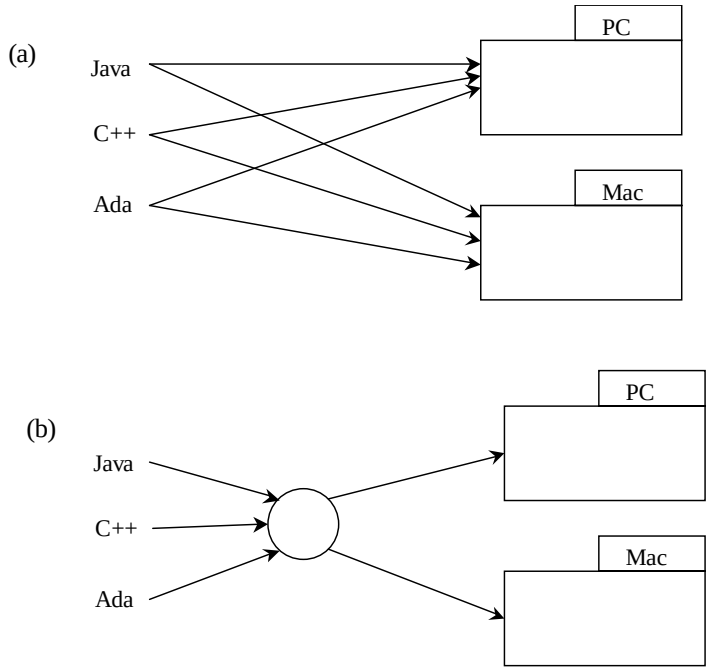


Figure 1.9 (a) Six compilers needed for three languages on two machines (b) Fewer than three compilers using intermediate form needed for the same languages and machines

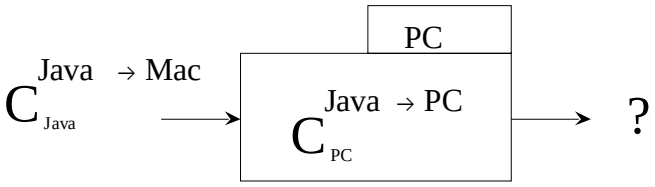
1.3.4 Compiler-Compilers

Much of compiler design is understood so well at this time that the process can be automated. It is possible for the compiler writer to write specifications of the source language and of the target machine so that the compiler can be generated automatically. This is done by a ***compiler-compiler***. We will introduce this topic in Chapters 2 and 5 when we study the *SableCC* public domain software.

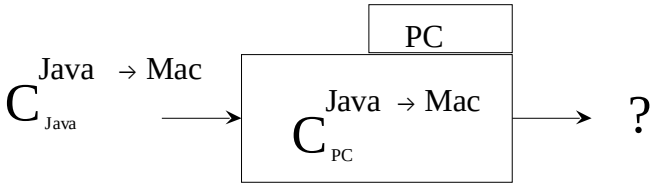
Exercises 1.3

1. Fill in the missing information in the compilations indicated below:

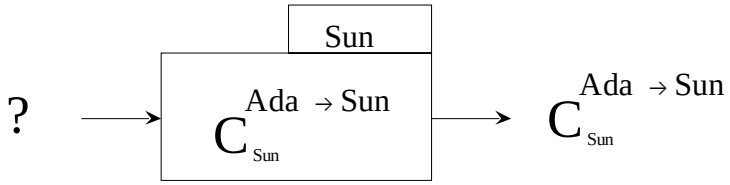
(a)



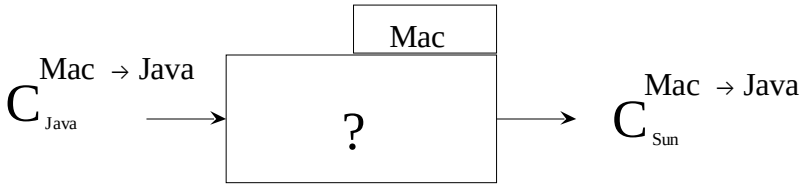
(b)



(c)



(d)



2. How could the compiler generated in part (d) of Question 1 be used?

3. If the only computer you have is a PC (for which you already have a FORTRAN compiler), show how you can produce a FORTRAN compiler for the Mac computer, without writing any assembly or machine language.

4. Show how Ada can be bootstrapped in two steps on a Sun, using first a small subset of Ada, `Sub1`, and then a larger subset, `Sub2`. First use `Sub1` to implement `Sub2` (by bootstrapping), then use `Sub2` to implement Ada (again by bootstrapping). `Sub1` is a subset of `Sub2`.

5. You have 3 computers: a PC, a Mac, and a Sun. Show how to generate automatically a Java to FORT translator which will run on a Sun if you also have the four compilers shown below:

$$C_{Mac}^{Java \rightarrow FORT} \qquad C_{Sun}^{FORT \rightarrow Java} \qquad C_{Mac}^{Java \rightarrow Sun} \qquad C_{Java}^{Java \rightarrow FORT}$$

6. In Figure 1.8 suppose we also have $C_{Java}^{Java \rightarrow Sun}$. When we write $C_{Java}^{Java \rightarrow Mac}$, which of the phases of $C_{Java}^{Java \rightarrow Sun}$ can be reused as is?

7. Using the big C notation, show the 11 translators which are represented in figure 1.9. Use "Int" to represent the intermediate form.

1.4 Case Study: Decaf

As we study the various phases of compilation and techniques used to implement those phases, we will show how the concepts can be applied to an actual compiler. For this purpose we will define a language called ***Decaf*** as a relatively simple subset of the Java language. The implementation of Decaf will then be used as a case study, or extended project, throughout the textbook. The last section of each chapter will show how some of the concepts of that chapter can be used in the design of an actual compiler for Decaf.

Decaf is a "bare bones" version of Java. Its only data types are `int` and `float`, and it does not permit arrays, classes, enumerated types, methods, or subprograms. However, it does include `while`, `for`, and `if` control structures, and it is possible to write some useful programs in Decaf. The example that we will use for the case study is the following Decaf program, to compute the cosine function:

```
class Cosine
{ public static void main (String [] args)
{ float cos, x, n, term, eps, alt;
/* compute the cosine of x to within tolerance eps */
/* use an alternating series */

   x = 3.14159;
   eps = 0.0001;
   n = 1;
   cos = 1;
   term = 1;
   alt = -1;
   while (term>eps)
      { term = term * x * x / n / (n+1);
        cos = cos + alt * term;
        alt = -alt;
        n = n + 2;
      }
}
}
```

This program computes the cosine of the value `x` (in radians) using an alternating series which terminates when a term becomes smaller than a given tolerance (`eps`). This series is described in most calculus textbooks and can be written as:

$$\cos(x) = 1 - x^2/2 + x^4/24 - x^6/720 + \ldots$$

Note that in the statement `term = term * x * x / n / (n+1)` the multiplication and division operations associate to the left, so that `n` and `(n+1)` are both in the denominator.

A precise specification of Decaf, similar to a BNF description, is given in Appendix A. The lexical specifications (free format, white space taken as delimiters, numeric constants, comments, etc.) of Decaf are the same as standard C.

When we discuss the back end of the compiler (code generation and optimization) we will need to be concerned with a target machine for which the compiler generates instructions. Rather than using an actual computer as the target machine, we have designed a fictitious computer called Mini as the target machine. This was done for two reasons: (1) We can simplify the architecture of the machine so that the compiler is not unnecessarily complicated by complex addressing modes, complex instruction formats, operating system constraints, etc., and (2) we provide the source code for a simulator for Mini so that the student can compile and execute Mini programs (as long as he/she has a C compiler on his/her computer). The student will be able to follow all the steps in the compilation of the above cosine program, understand its implementation in Mini machine language, and observe its execution on the Mini machine.

The complete source code for the Decaf compiler and the Mini simulator is provided in the appendix and is available through the Internet, as described in the appendix. With this software, the student will be able to make his/her own modifications to the Decaf language, the compiler, or the Mini machine architecture. Some of the exercises in later chapters are designed with this intent.

Exercises 1.4

1. Which of the following are valid program segments in Decaf? Like Java, Decaf programs are free-format (Refer to Appendix A).

(a)
```
for (x = 1; x<10; )
     y = 13;
```

(b)
```
if (a<b) { x =
     2;  y = 3 ;}
```

(c)
```
while (a+b==c) if (a!=c)
     a = a + 1;
```

(d)
```
{
     a = 4 ;
     b = c = 2; ;
}
```

(e)
```
for (i==22; i++; i=3) ;
```

2. Modify the Decaf description given in Appendix A to include a `switch` statement as defined in standard Java.

3. Modify the Decaf description given in Appendix A to include a `do while` statment as defined in standard Java.

1.5 Chapter Summary

This chapter reviewed the concepts of *high-level language* and *machine language* and introduced the purpose of the compiler. The *compiler* serves as a translator from any program in a given high-level language (the source program) to an equivalent program in a given machine language (the object program). We stressed the fact that the output of a compiler is a program, and contrasted compilers with interpreters, which carry out the computations specified by the source program.

We introduced the phases of a compiler: (1) The *lexical scanner* finds word boundaries and produces a token corresponding to each word in the source program. (2) The *syntax phase*, or *parser*, checks for proper syntax and, if correct, puts out a stream of atoms or syntax trees which are similar to the primitive operations found in a typical target machine. (3) The *global optimization phase* is optional, eliminates unnecessary atoms or syntax tree elements, and improves efficiency of loops if possible. (4) The *code generator* converts the *atoms* or *syntax trees* to instructions for the target machine. (5) The *local optimization phase* is also optional, eliminates unnecessary instructions, and uses other techniques to improve the efficiency of the object program.

We discussed some compiler implementation techniques. The first implementation technique was *bootstrapping*, in which a small subset of the source language is implemented and used to compile a compiler for the full source language, written in the source language itself. We also discussed *cross compiling*, in which an existing compiler can be used to implement a compiler for a new computer. We showed how the use of an intermediate form can reduce the workload of the compiler writer.

Finally, we examined a language called *Decaf*, a small subset of the C language, which will be used for a case study compiler throughout the textbook.