Chapter 6

Code Generation

6.1 Introduction to Code Generation

Up to this point we have ignored the architecture of the machine for which we are building the compiler, i.e. the target machine. By ***architecture***, we mean the definition of the computer's central processing unit as seen by a machine language programmer. Specifications of instruction-set operations, instruction formats, addressing modes, data formats, CPU registers, input/output instructions, etc. all make up what is sometime called the ***conventional machine language*** architecture (to distinguish it from the microprogramming level architecture which many computers have; see, for example, Tanenbaum [1990]). Once these are all clearly and precisely defined, we can complete the compiler by implementing the ***code generation*** phase. This is the phase which accepts as input the syntax trees or stream of atoms as put out by the syntax phase, and produces, as output, the object language program in binary coded instructions in the proper format.

The primary objective of the ***code generator*** is to convert atoms or syntax trees to instructions. In the process, it is also necessary to handle register allocation for machines that have several general purpose CPU registers. Label atoms must be converted to memory addresses. For some languages, the compiler has to check data types and call the appropriate type conversion routines if the programmer has mixed data types in an expression or assignment.

Note that if we are developing a new computer, we don't need a working model of that computer in order to complete the compiler; all we need are the specifications, or architecture, of that computer. Many designers view the construction of compilers as made up of two logical parts – the front end and the back end. The ***front end*** consists of lexical and syntax analysis and is machine-independent. The ***back end*** consists of code generation and optimization and is very machine-dependent, consequently this chapter

commences our discussion of the back end, or machine-dependendent, phases of the compiler.

This separation into front and back ends simplifies things in two ways when constructing compilers for new machines or new languages. First, if we are implementing a compiler for a new machine, and we already have compilers for our old machine, all we need to do is write the back end, since the front end is not machine dependent. For example, if we have a Pascal compiler for an IBM PS/2, and we wish to implement Pascal on a new RISC (Reduced Intruction Set Computer) machine, we can use the front end of the existing Pascal compiler (it would have to be recompiled to run on the RISC machine). This means that we need to write only the back end of the new compiler (refer to Figure 1.9, p. 23).

Our life is also simplified when constructing a compiler for a new programming language on an existing computer. In this case, we can make use of the back end already written for our existing compiler. All we need to do is rewrite the front end for the new language, compile it, and link it together with the existing back end to form a complete compiler. Alternatively, we could use an editor to combine the source code of our new front end with the source code of the back end of the existing compiler, and compile it all at once.

For example, suppose we have a Pascal compiler for the Macintosh, and we wish to construct an Ada compiler for the Macintosh. First, we understand that the front end of each compiler translates source code to a string of atoms (call this language Atoms), and the back end translates Atoms to Mac machine language (Motorola 680x0 instructions).

The compilers we have are $\mathbf{C}^{\mathtt{Pas} \rightarrow \mathtt{Mac}}_{\mathtt{Pas}}$ and $\mathbf{C}^{\mathtt{Pas} \rightarrow \mathtt{Mac}}_{\mathtt{Mac}}$, the compiler we want is

$\mathbf{C}^{\mathtt{Ada} \rightarrow \mathtt{Mac}}_{\mathtt{Mac}}$, and each is composed of two parts, as shown in Figure 6.1. We write

$\mathbf{C}^{\mathtt{Ada} \rightarrow \mathtt{Atoms}}_{\mathtt{Pas}}$ which is the front end of an Ada compiler and is also shown in Figure 6.1.

We then compile the front end of our Ada compiler as shown in Figure 6.2 (a) and link it with the back end of our Pascal compiler to form a complete Ada compiler for the Mac, as shown in Figure 6.2 (b).

The back end of the compiler consists of the code generation phase, which we will discuss in this chapter, and the optimization phases, which will be discussed in Chapter 7. Code generation is probably the least intensively studied phase of the compiler. Much of it is straightforward and simple; there is no need for extensive research in this area. Most of the research that has been done is concerned with methods for specifying target machine architectures, so that this phase of the compiler can be produced automatically, as in a compiler-compiler.

<u>We have the source code for a Pascal Compiler:</u>

$$C^{Pas \rightarrow Mac}_{Pas} = C^{Pas \rightarrow Atoms}_{Pas} + C^{Atoms \rightarrow Mac}_{Pas}$$

<u>We have the Pascal compiler which runs on the Mac:</u>

$$C^{Pas \rightarrow Mac}_{Mac} = C^{Pas \rightarrow Atoms}_{Mac} + C^{Atoms \rightarrow Mac}_{Mac}$$

<u>We want an Ada Compiler which runs on the Mac:</u>

$$C^{Ada \rightarrow Mac}_{Mac} = C^{Ada \rightarrow Atoms}_{Mac} + C^{Atoms \rightarrow Mac}_{Mac}$$

<u>We write the front end of the Ada compiler in Pascal:</u>

$$C^{Ada \rightarrow Atoms}_{Pas}$$

Figure 6.1 Using a Pascal Compiler to Construct an Ada Compiler

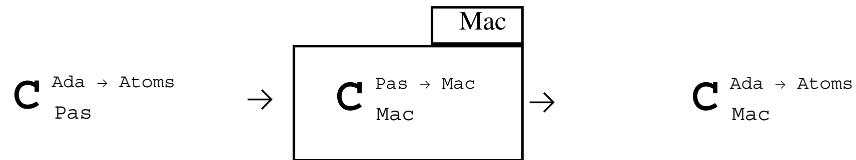


Figure 6.2 (a) Compile the Front End of the Ada Compiler on the Mac

$$C^{Ada \rightarrow Atoms}_{Mac} + C^{Atoms \rightarrow Mac}_{Mac} = C^{Ada \rightarrow Mac}_{Mac}$$

Figure 6.2 (b) Link the Front End of the Ada Compiler with the Back End of the Pascal Compiler to Produce a Complete Ada Compiler.

Sample Problem 6.1

Assume we have a Pascal compiler for a Mac (both source and executable code) as shown in Figure 6.1. We are constructing a completely new machine called a RISC, for which we wish to construct a Pascal compiler. Show how this can be done without writing the entire compiler and without writing any machine or assembly language.

Solution:

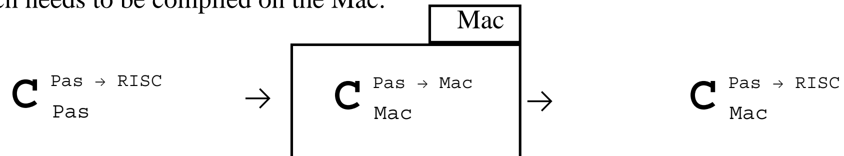
We want $C^{Ada \rightarrow RISC}_{RISC}$

Write (in Pascal) the back end of a compiler for the RISC machine:

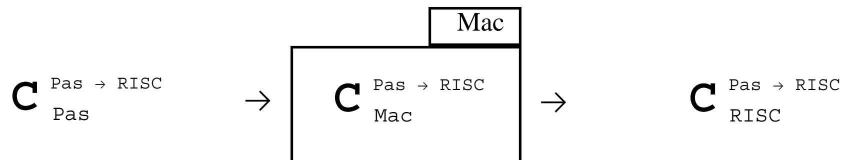
$C^{Atoms \rightarrow RISC}_{Pas}$

We now have $C^{Pas \rightarrow RISC}_{Pas} = C^{Pas \rightarrow Atoms}_{Pas} + C^{Atoms \rightarrow RISC}_{Pas}$

which needs to be compiled on the Mac:



But this is still not what we want, so we load the output into the Mac's memory and compile again:



and the output is the compiler that we wanted to generate.

Exercises 6.1

1. Show the big C notation for each of the following compilers (assume that each uses an intermediate form called "Atoms"):

(a) The back end of a compiler for the Sun computer.

(b) The source code, in Pascal, for a COBOL compiler whose target machine is the PC.

(c) The souce code, in Pascal, for the back end of a FORTRAN compiler for the Sun.

2. Show how to generate $C^{\text{Lisp} \rightarrow \text{PC}}_{\text{PC}}$

without writing any more programs, given a PC machine and each of the following collections of compilers:

(a) $C^{\text{Lisp} \rightarrow \text{PC}}_{\text{Pas}}$ $C^{\text{Pas} \rightarrow \text{PC}}_{\text{PC}}$

(b) $C^{\text{Lisp} \rightarrow \text{Atoms}}_{\text{Pas}}$ $C^{\text{Pas} \rightarrow \text{Atoms}}_{\text{Pas}}$

$C^{\text{Atoms} \rightarrow \text{PC}}_{\text{Pas}}$ $C^{\text{Pas} \rightarrow \text{PC}}_{\text{PC}}$

(c) $C^{\text{Lisp} \rightarrow \text{Atoms}}_{\text{PC}}$ $C^{\text{Atoms} \rightarrow \text{PC}}_{\text{PC}}$

3. Given a Sparc computer and the following compilers, show how to generate a Pascal (Pas) compiler for the MIPS machine without doing any more programming. (Unfortunately, you can't afford to buy a MIPS computer.)

$$C_{Pas}^{Pas \rightarrow Sparc} = C_{Pas}^{Pas \rightarrow Atoms} + C_{Pas}^{Atoms \rightarrow Sparc}$$

$$C_{Sparc}^{Pas \rightarrow Sparc} = C_{Sparc}^{Pas \rightarrow Atoms} + C_{Sparc}^{Atoms \rightarrow Sparc}$$

$$C_{Pas}^{Atoms \rightarrow MIPS}$$

6.2 Converting Atoms to Instructions

If we temporarily ignore the problem of forward references (of Jump or Branch instructions), the process of converting atoms to instructions is relatively simple. For the most part all we need is some sort of case, switch, or multiple destination branch based on the class of the atom being translated. Each atom class would result in a different instruction or sequence of instructions. If the CPU of the target machine requires that all arithmetic be done in registers, then an example of a translation of an ADD atom would be as shown, below, in Figure 6.3; i.e., an ADD atom is translated into a LOD (Load Into Register) instruction, followed by an ADD instruction, followed by a STO (Store Register To Memory) instruction.

```
(ADD, A, B, T1)   →     LOD   R1,A
                        ADD   R1,B
                        STO   R1,T1
```

Figure 6.3 Translation of an `ADD` Atom to Instructions

Most of the atom classes would be implemented in a similar way. Conditional Branch atoms (called TST atoms in our examples) would normally be implemented as a Load, Compare, and Branch, depending on the architecture of the target machine. The `MOV` (move data from one memory location to another) atom could be implemented as a `MOV` (Move) instruction, if permitted by the target machine architecture; otherwise it would be implemented as a Load followed by a Store.

Operand addresses which appear in atoms must be appropriately coded in the target machine's instruction format. For example, many target machines require operands to be addressed with a base register and an offset from the contents of the base register. If this is the case, the code generator must be aware of the presumed contents of the base register, and compute the offset so as to produce the desired operand address. For example, if we know that a particular operand is at memory location `1E` (hex), and the contents of the base register is `10` (hex), then the offset would be `0E`, because `10 + 0E = 1E`. In other words, the contents of the base register, when added to the offset, must equal the operand address.

Sample Problem 6.2

The Java statement `if (A>B) A = B * C;` might result in the following sequence of atoms:

```
(TST, A, B,, 4, L1)     // Branch to L1 if A<=B
(MUL, B, C, A)
(LBL L1)
```

Translate these atoms to instructions for a Load/Store architecture. Assume that the operations are LOD (Load), STO (Store), ADD, SUB, MUL, DIV, CMP (Compare), and JMP (Conditional Branch). The Compare instruction will set a flag for the Jump instruction, and a comparison code of 0 always sets the flag to True, which results in an Unconditional Branch. Assume that variables and labels may be represented by symbolic addresses.

Solution:

```
      LOD   R1,A              // Load A into Reg. R1
      CMP   R1,B,4            // Compare A <= B ?
      JMP   L1                // Branch if true
      LOD   R1,B
      MUL   R1,C              // R1 = B * C
      STO   R1,A              // A = B * C
L1:
```

Exercises 6.2

1. For each of the following Java statements we show the atom string produced by the parser. Translate each atom string to *instructions*, as in the sample problem for this section. You may assume that variables and labels are represented by symbolic addresses.

(a)

```
{     a = b + c * (d - e) ;
      b = a;
}

(SUB, d, e, T1)
(MUL, c, T1, T2)
(ADD, b, T2, T3)
(MOV, T3,, a)
(MOV, a,, b)
```

(b) `for (i=1; i<=10; i++) j = j/3 ;`

```
(MOV, 1,, i)
(LBL, L1)
(TST, i, 10,, 3, L4)      // Branch if i>10
(JMP, L3)
(LBL, L5)
(ADD, 1, i, i)            // i++
(JMP, L1)                 // Repeat the loop
(LBL, L3)
(DIV, j, 3, T2)           // T2 = j / 3;
(MOV, T2,, j)             // j = T2;
(JMP, L5)
(LBL, L4)                 // End of loop
```

(c) `if (a!=b+3) a = 0; else b = b+3;`

```
(ADD, b, 3, T1)
(TST, a, T1,, 1, L1)      // Branch if a==b+3
(MOV, 0,, a)              // a = 0
(JMP, L2)
(LBL, L1)
(ADD, b, 3, T2)           // T2 = b + 3
(MOV, T2,, b)             // b = T2
(LBL, L2)
```

2. How many instructions correspond to each of the following atom classes on a Load/Store architecture, as in the sample problem of this section?

(a) `ADD` (b) `DIV` (c) `MOV`
(d) `TST` (e) `JMP` (f) `LBL`

3. Why is it important for the code generator to know how many instructions correspond to each atom class?

4. How many machine language instructions would correspond to an ADD atom on each of the following architectures?

(a) Zero address architecture (a stack machine)

(b) One address architecture

(c) Two address architecture

(d) Three address architecture

6.3 Single Pass vs. Multiple Passes

There are several different ways of approaching the design of the code generation phase. The difference between these approaches is generally characterized by the number of passes which are made over the input file. For simplicity, we will assume that the input file is a file of atoms, as specified in Chapters 4 and 5. A code generator which scans this file of atoms once is called a ***single pass*** code generator, and a code generator which scans it more than once is called a ***multiple pass*** code generator.

The most significant problem relevant to deciding whether to use a single or multiple pass code generator has to do with forward jumps. As atoms are encountered, instructions can be generated, and the code generator maintains a memory address counter, or program counter. When a Label atom is encountered, a memory address value can be assigned to that Label atom (a table of labels is maintained, with a memory address assigned to each label as it is defined). If a Jump atom is encountered with a destination that is a higher memory address than the Jump instruction (i.e. a forward jump), the label to which it is jumping has not yet been encountered, and it will not be possible to generate the Jump instruction completely at this time. An example of this situation is shown, below, in Figure 6.4 in which the jump to Label L1 cannot be generated because at the time the JMP atom is encountered the code generator has not encountered the definition of the Label L1, which will have the value 9.

```
Atom                  Location    Instruction
(ADD, A, B, T1)       4           LOD   R1,A
                      5           ADD   R1,B
                      6           STO   R1,T1
(JMP,L1)              7           CMP   0,0,0
                      8           JMP   ?
(LBL, L1)                                     (L1 = 9)
```

Figure 6.4 Problem in Generating a Jump to a Forward Destination

A JMP atom results in a CMP (Compare instruction) followed by a JMP (Jump instruction), to be consistent with the sample architecture presented in Section 6.5, below.

There are two fundamental ways to resolve the problem of forward jumps. Single pass compilers resolve it by keeping a table of Jump instructions which have forward destinations. Each Jump instruction with a forward reference is generated incompletely (i.e., without a destination address) when encountered, and each is also entered into a ***fixup table***, along with the Label to which it is jumping. As each Label definition is encountered, it is entered into a table of Labels, along with its address value. When all of the atoms have been read, all of the Label atoms will have been defined, and, at this time, the code generator can revisit all of the Jump instructions in the Fixup table and fill in their destination addresses. This is shown in Figure 6.5, below, for the same atom sequence shown in Figure 6.4. Note that when the (JMP, L1) atom is encountered,

the Label L1 has not yet been defined, so the location of the Jump (8) is entered into the Fixup table. When the (LBL, L1) atom is encountered, it is entered into the Label table, because the target machine address corresponding to this Label (9) is now known. When the end of file (EOF) is encountered, the destination of the Jump instruction at location 8 is changed, using the Fixup table and the Label table, to 9.

			Fixup Table		Label Table	
Atom	Loc	Instruction	Loc	Label	Label	Value
(ADD,A,B,T1)	4	LOD R1,A				
	5	ADD R1,B				
	6	STO R1,T1				
(JMP,L1)	7	CMP 0,0,0				
	8	JMP 0	8	L1		
(LBL,L1)					L1	9
...						
EOF						
	8	JMP 9				

Figure 6.5 Use of the Fixup Table and Label Table in a Single Pass Code Generator for Forward Jumps

Multiple pass code generators do not require a Fixup table. In this case, the first pass of the code generator does nothing but build the table of Labels, storing a memory address for each Label. Then, in the second pass, all the Labels will have been defined, and each time a Jump is encountered its destination Label will be in the table, with an assigned memory address. This method is shown in Figure 6.6 which, again, uses the atom sequence given in Figure 6.4.

Note that, in the first pass, the code generator needs to know how many machine language instructions correspond to an atom (three to an ADD atom and two to a JMP atom), though it does not actually generate the instructions. It can then assign a memory address to each Label in the Label table.

A single pass code generator could be implemented as a subroutine to the parser. Each time the parser generates an atom, it would call the code generator to convert the atom to machine language and put out the instruction(s) corresponding to that atom. A multiple pass code generator would have to read from a file of atoms, created by the parser, and this is the method we use in our sample code generator in Section 6.5.

Sample Problem 6.3

The following atom string resulted from the Java statement `while (i<=x) { x = x+2; i = i*3; }`. Translate it into instructions as in (1) a single pass code generator using a Fixup table and (2) a multiple pass code generator.

```
Begin First Pass:                                          Label Table
Atom                 Loc     Instruction                   Label   Value

(ADD,A,B,T1)         4-6


(JMP,L1)             7-8

(LBL,L1)                                                   L1      9

...
EOF

Begin Second Pass:
Atom                 Loc     Instruction

(ADD, A, B, T1)      4       LOD   R1,A
                     5       ADD   R1,B
                     6       STO   R1,T1
(JMP,L1)             7       CMP   0,0
                     8       JMP   9
(LBL, L1)

...
EOF
```

Figure 6.6 Forward Jumps Handled by a Multiple Pass Code Generator

```
(LBL, L1)
(TST, i, x,, 3, L2)              // Branch if T1 is false
(ADD, x, 2, T1)
(MOV, T1, , x)
(MUL, i, 3, T2)
(MOV, T2, , i)
(JMP, L1)                        // Repeat the loop
(LBL, L2)                        // End of loop
```

Solution:

(1) Single Pass

Atom	Loc	Instruction	Fixup Table Loc	Fixup Table Label	Label Table Label	Label Table Value
(LBL, L1)	0				L1	0
(TST,i,x,,3,L2)	0	CMP i,x,3				
	1	JMP ?	1	L2		
(ADD, X, 2, T1)	2	LOD R1,x				
	3	ADD R1,2				
	4	STO R1,T1				
(MOV, T1,, x)	5	LOD R1,T1				
	6	STO R1,x				
(MUL, i, 3, T2)	7	LOD R1,i				
	8	MUL R1,3				
	9	STO R1,T2				
(MOV, T2,, i)	10	LOD R1,T2				
	11	STO R1,i				
(JMP, L1)	12	CMP 0,0,0				
	13	JMP 0				
(LBL, L2)	14				L2	14
...						
	1	JMP 14				

(2) Multiple passes

Atom	Loc	Instruction	Label Table Label	Label Table Value
Begin First Pass:				
(LBL, L1)	0		L1	2
(TST,i,x,,3,L2)	0			
(ADD, X, 2, T1)	2			
(MOV, T1,, x)	5			
(MUL, i, 3, T2)	7			
(MOV, T2,, i)	10			
(JMP, L1)	12			
(LBL, L2)	14		L2	14
...				

Begin Second Pass:

```
Atom                Loc   Instruction

(LBL, L1)           0
(TST,i,x,,3,L2)     0  CMP  i,x,3
                    1  JMP  14
(ADD, X, 2, T1)     2  LOD  R1,x
                    3  ADD  R1,2
                    4  STO  R1,T1
(MOV, T1,, x)       5  LOD  R1,T1
                    6  STO  R1,x
(MUL, i, 3, T2)     7  LOD  R1,i
                    8  MUL  R1,3
                    9  STO  R1,T2
(MOV, T2,, i)       10 LOD  R1,T2
                    11 STO  R1,i
(JMP, L1)           12 CMP  0,0,0
                    13 JMP  0
(LBL, L2)           14
```

Exercises 6.3

1. The following atom string resulted from the Java statement:
`for (i=a; i<b+c; i++) b = b/2;`
Translate the atoms to *instructions* as in the sample problem for this section using two methods: (1) a *single pass* method with a Fixup table for forward Jumps and (2) a *multiple pass* method. Refer to the variables `a,b,c` symbolically.

```
(MOV, a,, i)
(LBL, L1)
(ADD, b, c, T1)          // T1 = b+c
(TST, i, T1,, 4, L2)     // If i>=b+c, exit loop
(JMP, L3)                // Exit loop
(LBL, L4)
(ADD, i, 1, i)           // Increment i
(JMP, L1)                // Repeat loop
(LBL, L3)
(DIV, b, ='2', T3)       // Loop Body
(MOV, T3,, b)
```

```
(JMP, L4)                    // Jump to increment
(LBL, L2)
```

2. Repeat Problem 1 for the atom string resulting from the Java statement:

```
if (a==(b-33)*2) a = (b-33)*2;
     else a = x+y;

(SUB, b, ='33', T1)
(MUL, T1, ='2', T2)          // T2 = (b-33)*2
(TST, a, T2,, 6, L1)         // Branch if a!=T2
(SUB, b, ='33', T3)
(MUL, T3, ='2', T4)
(MOV, T4,, a)
(JMP, L2)                    // Skip else part
(LBL, L1)                    // else part
(ADD, x, y, T5)
(MOV, T5,, a)
(LBL, L2)
```

3. (a) What are the advantages of a *single pass* method of code generation over a multiple pass method?

(b) What are the advantages of a *multiple pass* method of code generation over a single pass method?

6.4 Register Allocation

Some computers (such as the DEC PDP-8) are designed with a single arithmetic register, called an accumulator, in which all arithmetic operations are performed. Other computers (such as the Intel 8086) have only a few CPU registers, and they are not general purpose registers; i.e., each one has a limited range of uses or functions. In these cases the allocation of registers is not a problem.

However, most modern architectures have many CPU registers; the DEC Vax, IBM mainframe, and Motorola 680x0 architectures each has sixteen general purpose registers, for example, and the RISC (Reduced Instruction Set Computer) architectures, such as the SUN SPARC and MIPS, generally have about 500 CPU registers (though only 32 are used at a time). In this case, register allocation becomes an important problem. ***Register allocation*** is the process of assigning a purpose to a particular register, or binding a register to a programmer variable or compiler variable, so that for a certain range or scope of instructions that register has the specified purpose or binding and is used for no other purposes. The code generator must maintain information on which registers are used for which purposes, and which registers are available for reuse. The main objective in register allocation is to maximize utilization of the CPU registers, and to minimize references to memory locations.

It might seem that register allocation is more properly a topic in the area of code optimization, since code generation could be done with the assumption that there is only one CPU register (resulting in rather inefficient code). Nevertheless, register allocation is always handled (though perhaps not in an optimal way) in the code generation phase. A well chosen register allocation scheme can not only reduce the number of instructions required, but it can also reduce the number of memory references. Since operands which are used repeatedly can be kept in registers, the operands do not need to be recomputed, nor do they need to be loaded from memory. It is especially important to minimize memory references in compilers for RISC machines, in which the objective is to execute one instruction per machine cycle, as described in Tanenbaum [1990].

An example, showing the importance of smart register allocation, is shown in Figure 6.7 for the two statement program segment:

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

The smart register allocation scheme takes advantage of the fact that `C*D` is a common subexpression, and that the variable `A` is bound, temporarily, to register `R2`. If no attention is paid to register allocation, the two statements in Figure 6.7 are translated into twelve instructions, involving a total of twelve memory references. With smart register allocation, however, the two statements are translated into seven instructions, with only five memory references. (Some computers, such as the VAX, permit arithmetic on memory operands, in which case register allocation takes on lesser importance.)

An algorithm which takes advantage of repeated subexpressions will be discussed in Section 7.2. Here, we will discuss an algorithm which determines how many

```
Simple Register Allocation        Smart Register Allocation
LOD   R1,C                        LOD   R1,C
MUL   R1,D                        MUL   R1,D        C*D
STO   R1,Temp1                    LOD   R2,B
LOD   R1,B                        ADD   R2,R1       B+C*D
ADD   R1,Temp1                    STO   R2,A
STO   R1,A                        SUB   R2,R1       A-C*D
LOD   R1,C                        STO   R2,B
MUL   R1,D
STO   R1,Temp2
LOD   R1,A
SUB   R1,Temp2
STO   R1,B
```

Figure 6.7 Register Allocation, Simple and Smart, for a Two Statement Program

registers will be needed to evaluate an expression without storing subexpressions to temporary memory locations. This algorithm will also determine the sequence in which subexpressions should be evaluated to minimize register usage.

This register allocation algorithm will require a syntax tree for an expression to be evaluated. Each node of the syntax tree will have a weight associated with it which tells us how many registers will be needed to evaluate each subexpression without storing to temporary memory locations. Each leaf node which is a left operand will have a weight of one, and each leaf node which is a right operand will have a weight of zero. The weight of each interior node will be computed from the weights of its two children as follows: If the two children have different weights, the parent's weight is the maximum of the two children. If the two children have the same weight, w, then the parent's weight is w+1. As an example, the weighted syntax tree for the expression `a*b - (c+d) * (e+f)` is shown in Figure 6.8 from which we can see that the entire expression should require two registers.

Intuitively, if two expressions representing the two children of a node, N, in a syntax tree require the same number of registers, we will need an additional register to store the result for node N, regardless of which subexpression is evaluated first. In the other case, if the two subexpressions do not require the same number of registers, we can evaluate the one requiring more registers first, at which point those registers are freed for other use.

We can now generate code for this expression. We do this by evaluating the operand having greater weight, first. If both operands of an operation have the same weight, we evaluate the left operand first. For our example in Figure 6.8 we generate the code shown in Figure 6.9. We assume that there are register-register instructions (i.e., instructions in which both operands are contained in registers) for the arithmetic operations in the target machine architecture. Note that if we had evaluated `a*b` first we would have needed either an additional register or memory references to a temporary location.

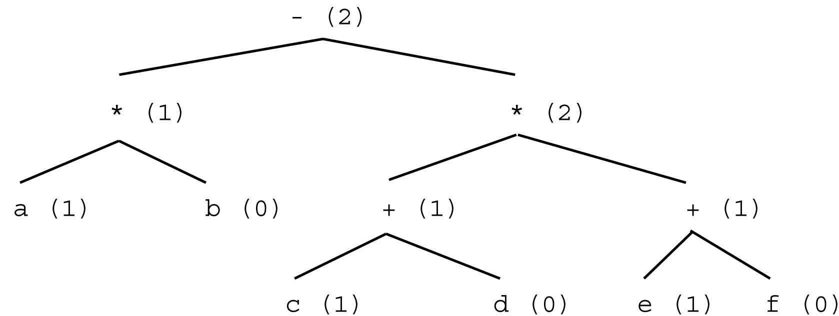


Figure 6.8 Weighted Syntax Tree for `a*b-(c+d)*(e+f)`, with Weights Shown in Parentheses

This problem would have had a more interesting solution if the expression had been `e+f - (c+d)*(e+f)` because of the repeated subexpression `e+f`. If the value of `e+f` were left in a register, it would not have to be recomputed. There are algorithms which handle this kind of problem, but they will be covered in the chapter on optimization.

```
LOD   R1,c
ADD   R1,d        R1 = c + d
LOD   R2,e
ADD   R2,f        R2 = e + f
MUL   R1,R2       R1 = (c+d) * (e+f)
LOD   R2,a
MUL   R2,b        R2 = a * b
SUB   R2,R1       R2 = a*b - (c+d)*(e+f
```

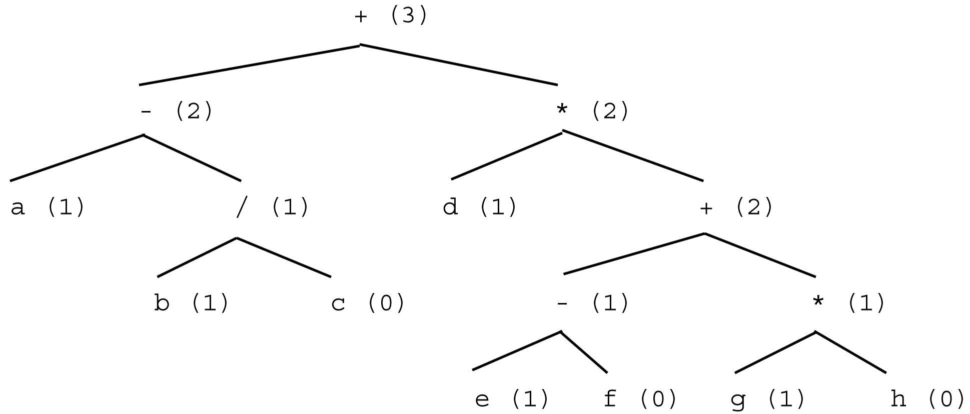
Figure 6.9 Code Generated for `a*b-(c+d)*(e+f)`, Using Figure 6.8

Sample Problem 6.4

Use the register allocation algorithm of this section to show a weighted syntax tree for the expression `a - b/c + d * (e-f + g*h)`, and show the resulting instructions, as in Figure 6.9.

Solution:

```
LOD   R1,a
LOD   R2,b
DIV   R2,c        b/c
SUB   R1,R2       a - b/c
```




```
LOD   R2,e
SUB   R2,f        e - f
LOD   R3,g
MUL   R3,h        g * h
ADD   R2,R3       e - f + g * h
LOD   R3,d
MUL   R3,R2       d * (e-f + g*h)
ADD   R1,R3       a - b/c + d * (e-f + g*h)
```

Exercises 6.4

1. Use the register allocation algorithm given in this section to construct a *weighted syntax tree* and generate code for each of the given expressions, as done in Sample Problem 6.4. Do not attempt to optimize for common subexpressions.

(a) `a + b * c - d`

(b) `a + (b + (c + (d + e)))`

(c) `(a + b) * (c + d) - (a + b) * (c + d)`

(d) `a / (b + c) - (d + (e - f)) + (g - h * i) * (j * (k / m))`

2. Show an expression different in structure from those in Problem 1 which requires:

(a) two registers (b) three registers

As in Problem 1, assume that common subexpressions are not detected and that Loads and Stores are minimized.

3. Show how the code generated in Problem 1 (c) can be improved by making use of common subexpressions.

6.5 Case Study: A Code Generator for the Mini Architecture

When working with code generators, at some point it becomes necessary to choose a target machine. Up to this point we have been reluctant to do so because we wanted the discussion to be as general as possible, so that the concepts could be applied to a variety of architectures. However, in this section we will work with an example of a code generator, and it now becomes necessary to specify a target machine architecture. It is tempting to choose a popular machine such as a RISC, Intel, Motorola, IBM, or Sparc CPU. If we did so, the student who had access to that processor could conceivably generate executable code for that machine. But what about those who do not have access to the chosen processor? Also, there would be details, such as Object formats (the input to the linker), and supervisor or system calls for certain operations, which we have not explained.

For these reasons, we choose our own *simulated* machine. This is an architecture which we will specify for the student. We also provide a simulator for this machine, written in the C language. Thus, anyone who has a C compiler has access to our simulated machine, regardless of the actual platform on which it is running. Another advantage of a simulated architecture is that we can make it as simple as necessary to illustrate the concepts of code generation. We don't need to be concerned with efficiency or completeness. The architecture will be relatively simple and not cluttered with unnecessary features.

6.5.1 Mini: The Simulated Architecture

In this section we devise a completely fictitious computer, and we provide a simulator for that computer so that the student will be able to generate and execute machine language programs. We call our machine Mini, not because it is supposed to be a "minicomputer," but because it is really a minimal computer. We have described and implemented just enough of the architecture to enable us to implement a fairly simple code generator. The student should feel free to implement additional features in the Mini architecture. For example, the Mini architecture contains no integer arithmetic; all arithmetic is done with floating-point values, but the instruction set could easily be extended to include integer arithmetic.

The Mini architecture has a 32-bit word size, with 32-bit registers, and a word addressable memory consisting of, at most, 4 G (32 bit) words (the simulator defines a memory of 64 K words, though this is easily extended). There are two addressing modes in the Mini architecture: ***absolute*** and ***register-displacement*.** In absolute mode, the memory address is stored in the instruction as a 20-bit quantity (in this mode it is only possible to address the lowest megaword of memory). In register-displacement mode, the memory address is computed by adding the contents of the specified general register to the value of the 16-bit offset, or displacement, in the instruction (in this mode it is possible to address all of memory).

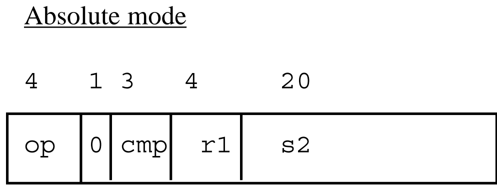
The CPU has sixteen general purpose registers and sixteen floating-point registers. All floating-point arithmetic must be done in the floating-point registers

(floating-point data are stored in the format of the simulator's host machine, so the student need not be concerned with the specifics of floating-point data formats). There is also a 1-bit flag in the CPU which is set by the compare (CMP) instruction and tested by the conditional branch (JMP) instruction. There is also a 32-bit program counter register (PC). The Mini processor has two instruction formats corresponding to the two addressing modes, as shown in Figure 6.10.

The ***absolute mode*** instruction can be described as:

```
fpreg[r1]  ← fpreg[r1] op memory[s2]
```

Absolute mode



Register-displacement mode

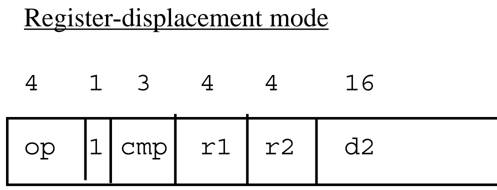


Figure 6.10 Mini Instruction Formats

and the ***register-displacement mode*** instruction can be described as

```
fpreg[r1]  ← fpreg[r1] op memory[reg[r2]+d2].
```

The operation codes (specified in the `op` field) are shown below:

```
0  CLR  fpreg[r1] ← 0                           Clear Floating-Point Reg.
1  ADD  fpreg[r1] ← fpreg[r1] + memory[s2]      Floating-Point Add
2  SUB  fpreg[r1] ← fpreg[r1] - memory[s2]      Floating-Point Subtract
3  MUL  fpreg[r1] ← fpreg[r1] * memory[s2]      Floating-Point Multiply
4  DIV  fpreg[r1] ← fpreg[r1] / memory[s2]      Floating-Point Division
5  JMP  PC ← s2 if flag is true                 Conditional Branch
6  CMP  flag ←  r1 cmp memory[s2]               Compare, Set Flag
7  LOD  fpreg[r1] ← memory[s2]                  Load Floating-Point Register
8  STO  memory[s2] ← fpreg[r1]                  Store Floating-Point Register
9  HLT                                          Halt Processor
```

The Compare field in either instruction format (`cmp`) is used only by the Compare instruction to indicate the kind of comparison to be done on arithmetic data. In addition to a code of 0, which always sets the flag to True, there are six valid comparison codes as shown below:

1	==	4	<=
2	<	5	>=
3	>	6	!=

The following example of a Mini program will replace the memory word at location 0 with its absolute value. The memory contents are shown in hexadecimal, and program execution is assumed to begin at memory location 1.

Loc	Contents				
0	00000000	Data	0		
1	00100000		CLR	R1	Put 0 into Register R1.
2	64100000		CMP	R1,Data,4	Is 0 <= Data?
3	50000006		JMP	Stop	If so, finished.
4	20100000		SUB	R1,Data	If not, find 0-Data.
5	80100000		STO	R1,Data	
6	90000000	Stop	HLT		Halt processor

The simulator for the Mini architecture is shown in Appendix C.

6.5.2 The Input to the Code Generator

In our example, the input to the code generator will be a file in which each record is an atom, as discussed in Chapters 4 and 5. Here we specify the meaning of the atoms more precisely in the table below:

Class	Name	Operands				Meaning
1	ADD	left	right	result		result ← left + right
2	SUB	left	right	result		result ← left - right
3	MUL	left	right	result		result ← left * right
4	DIV	left	right	result		result ← left / right
5	JMP	—	—	—	—	dest → dest
10	NEG	left	—	result		result ← - left
11	LBL	—	—	—	—	dest (no action)
12	TST	left	right	-	cmp	dest → dest if left cmp right is true
13	MOV	left	—	result		— — result ← left

Each atom class is specified with an integer code, and each record may have up to six fields specifying the atom class, the location of the left operand, the location of the right

operand, the location of the result, a comparison code (for TST atoms only), and a destination (for JMP, LBL, and TST atoms only). Note that a JMP atom is an unconditional branch, whereas a JMP instruction is a conditional branch. An example of an input file of atoms which would replace the value of `Data` with its absolute value is shown below:

```
TST    0      Data   4      L1     —              Branch to L1 if 0 <= Data
NEG    Data   —      Data   —      —              Data ← - Data
LBL    L1     —      —      —      —
```

6.5.3 The Code Generator for Mini

The complete code generator is shown in Appendix B, in which the function name is `code_gen()`. In this section we explain the techniques used and the design of that program. The code generator reads from a file of atoms, and it is designed to work in two passes. Since instructions are 32 bits, the code generator declares integer quantities as long (assuming that the host machine will implement these in 32 bits).

In the first pass it builds a table of Labels, assigning each Label a value corresponding to its ultimate machine address; the table is built by the function `build_labels()`, and the name of the table is `labels`. It is simply an array of integers holding the value of each Label. The integer variable `pc` is used to maintain a hypothetical program counter as the atoms are read, incremented by two for MOV and JMP atoms and incremented by three for all other atoms. The global variable `end_data` indicates the memory location where the program instructions will begin, since all constants and program variables are stored, beginning at memory location 0, by a function called `out_mem()` and precede the instructions.

After the first pass is complete, the file of atoms is closed and reopened to begin reading atoms for the second pass. The control structure for the second pass is a `switch` statement that uses the atom class to determine flow of control. Each atom class is handled by two or three calls to a function that actually generates an instruction - `gen()`. Label definitions can be ignored in the second pass.

The function which generates instructions takes four arguments:

```
gen (op, r, add, cmp)
```

where `op` is the operation code of the instruction, `r` is the register for the first operand, `add` is the absolute address for the second operand, and `cmp` is the comparison code for Compare instructions. For simplicity, the addressing mode is assumed always to be absolute (this limits us to a one megaword address space). As an example, Figure 6.11 shows that a Multiply atom would be translated by three calls to the `gen()` function to generate LOD, MUL, and STO instructions.

In Figure 6.11, the function `reg()` returns an available floating-point register. For simplicity, our implementation of `reg()` always returns a 1, which means that floating-point values are always kept in floating-point register 1. The structure `inp` is used to hold the atom which is being processed. The `dest` field of an atom is the destination label for jump instructions, and the actual address is obtained from the labels table by a function called `lookup()`. The code generator sends all instructions to the

Multiply atom to compute A*B, putting result into T1:

class	left	right	result	cmp	dest
MUL	A	B	T1	—	—

```
                                        Loc   Contents
                                        0                 A
                                        1                 B
gen (LOD, r = reg(), inp.left);         2     70100000
gen (MUL, r, inp.right);                3     30100001
gen (STO, r, inp.result);               4     80100010
                                        ...
                                        10                T1
```

Figure 6.11 Translation of a Multiply Atom

standard output file as hex characters, so that the user has the option of discarding them, storing them in a file, or piping them directly into the Mini simulator. The generated instructions are shown to the right in Figure 6.11.

The student is encouraged to use, abuse, modify and/or distribute (but not for profit) the software shown in the Appendix to gain a better understanding of the operation of the code generator.

Sample Problem 6.5

Show the code gerated by the code generator for the following TST atom. Assume that the value of L1 is hex 23 and the variables A and B are stored at locations 0 and 1, respectively.

class	left	right	result	cmp	dest
TST	A	B	—	4	L1

Solution:

```
Loc   Contents
0                       A
1                       B
2     70100000                LOD   R1,A
3     64100001                CMP   R1,B,4
4     50000023                JMP   L1
```

Exercises 6.5

1. How is the compiler's task simplified by the fact that floating-point is the only numeric data type in the Mini architecture?

2. Disassemble the following Mini instructions. Assume that general register 7 contains hex 20, and that the variables `A` and `B` are stored at locations hex 21 and hex 22, respectively.

```
70100021
10300022
18370002
```

3. Show the code, in hex, generated by the code generator for each of the following atom strings. Assume that `A` and `B` are stored at locations 0 and 1, respectively. Allocate space for the temporary value `T1` at the end of the program.

(a)

class	left	right	result	cmp	dest
MUL	A	B	T1	—	—
LBL	—	—	—	—	L1
TST	A	T1	—	2	L1
JMP	—	—	—	—	L2
MOV	T1	—	B	—	—
LBL	—	—	—	—	L2

(b)

class	left	right	result	cmp	dest
NEG	A	—	T1	—	—
LBL	—	—	—	—	L1
MOV	T1	—	B	—	—
TST	B	T1	—	4	L1

(c)

class	left	right	result	cmp	dest
TST	A	B	—	6	L2
JMP	—	—	—	—	L1
LBL	—	—	—	—	L2
TST	A	T1	—	0	L2
LBL	—	—	—	—	L1

6.6 Chapter Summary

This chapter commences our study of the *back end* of a compiler. Prior to this point everything we have studied was included in the *front end*. The *code generator* is the portion of the compiler which accepts *syntax trees* or *atoms* (sometimes referred to as *3 - address code*) created by the front end and converts them to *machine language instructions* for the *target machine*.

It was shown that if the language of syntax trees or atoms (known as an *intermediate form*) is standardized, then, as new machines are constructed, we need only rewrite the back ends of our compilers. Conversely, as new languages are developed, we need only rewrite the front ends of our compilers.

The process of converting atoms to instructions is relatively easy to implement, since each atom corresponds to a small, fixed number of instructions. The main problems to be solved in this process are (1)obtaining memory addresses for *forward references* and (2) *register allocation* . Forward references result from branch instructions to a higher memory address which can be computed by either *single pass* or *multiple pass* methods. With a single pass method, a *fixup table* for forward references is required. For either method a table of labels is used to *bind labels to target machine addresses*.

Register allocation is important for efficient object code in machines which have *several CPU registers*. An *algorithm for allocating registers* from syntax trees are presented. Algorithms which make use of common subexpressions in an expression, or common subexpressions in a block of code, will be discussed in Chapter 7.

This chapter concludes with a *case study code generator*. This code generator can be used for any compiler whose front end puts out atoms as we have defined them. In order to complete the case study, we define a fictitious target machine, called *Mini*. This machine has a very simple 32 bit architecture, which simplifies the code generation task. Since we have a *simulator* for the Mini machine, written in the C language, in Appendix C, anyone with access to a C compiler can run the Mini machine.

It is assumed that all arithmetic is done in floating-point format, which eliminates the need for data conversions. Code is generated by a function with three arguments specifying the operation code and two operands. The code generator, shown in Appendix B.3, uses a two pass method to handle forward references.