Chapter 8

Code Generation

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the *back end*, may make multiple passes over the IR before generating the target program. Code optimization is discussed in detail in Chapter 9. The techniques presented in this chapter can be used whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: instruction selection, register

Figure 8.1: Position of code generator

allocation and assignment, and instruction ordering. The importance of these tasks is outlined in Section 8.1. Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

This chapter presents algorithms that code generators can use to translate the IR into a sequence of target language instructions for simple register machines. The algorithms will be illustrated by using the machine model in Section 8.2. Chapter 10 covers the problem of code generation for complex modern machines that support a great deal of parallelism within a single instruction.

After discussing the broad issues in the design of a code generator, we show what kind of target code a compiler needs to generate to support the abstractions embodied in a typical source language. In Section 8.3, we outline implementations of static and stack allocation of data areas, and show how names in the IR can be converted into addresses in the target code.

Many code generators partition IR instructions into "basic blocks," which consist of sequences of instructions that are always executed together. The partitioning of the IR into basic blocks is the subject of Section 8.4. The following section presents simple local transformations that can be used to transform basic blocks into modified basic blocks from which more efficient code can be generated. These transformations are a rudimentary form of code optimization, although the deeper theory of code optimization will not be taken up until Chapter 9. An example of a useful, local transformation is the discovery of common subexpressions at the level of intermediate code and the resultant replacement of arithmetic operations by simpler copy operations.

Section 8.6 presents a simple code-generation algorithm that generates code for each statement in turn, keeping operands in registers as long as possible. The output of this kind of code generator can be readily improved by peephole optimization techniques such as those discussed in the following Section 8.7.

The remaining sections explore instruction selection and register allocation.

8.1 Issues in the Design of a Code Generator

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

8.1.1 Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's. Many of the algorithms in this chapter are couched in terms of the representations considered in Chapter 6: three-address code, trees, and DAG's. The techniques we discuss can be applied, however, to the other intermediate representations as well.

In this chapter, we assume that the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type-conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

8.1.2 The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The inter-

preter provides software compatibility across multiple platforms, a major factor in the success of Java.

To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.

Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

Producing a relocatable machine-language program (often called an *object module*) as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

In this chapter, we shall use a very simple RISC-like computer as our target machine. We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines. For readability, we use assembly code as the target language. As long as addresses can be calculated from offsets and other information stored in the symbol table, the code generator can produce relocatable or absolute addresses for names just as easily as symbolic addresses.

8.1.3 Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs

further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form `x = y + z`, where `x`, `y`, and `z` are statically allocated, can be translated into the code sequence

```
LD  R0, y       // R0 = y        (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z to R0)
ST  x, R0       // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  R0, b       // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0       // a = R0
LD  R0, a       // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0       // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if `a` is not subsequently used.

The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an "increment" instruction (`INC`), then the three-address statement `a = a + 1` may be implemented more efficiently by the single instruction `INC a`, rather than by a more obvious sequence that loads `a` into a register, adds one to the register, and then stores the result back into `a`:

```
LD  R0, a       // R0 = a
ADD R0, R0, #1  // R0 = R0 + 1
ST  a, R0       // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

In Section 8.9 we shall see that instruction selection can be modeled as a tree-pattern matching process in which we represent the IR and the machine instructions as trees. We then attempt to "tile" an IR tree with a set of subtrees that correspond to machine instructions. If we associate a cost with each machine-instruction subtree, we can use dynamic programming to generate optimal code sequences. Dynamic programming is discussed in Section 8.11.

8.1.4 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.

2. *Register assignment*, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Example 8.1 : Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M x, y
```

where `x`, the multiplicand, is the odd register of an even/odd register pair and `y`, the multiplier, can be anywhere. The product occupies the entire even/odd register pair. The division instruction is of the form

```
D x, y
```

where the dividend occupies an even/odd register pair whose even register is `x`; the divisor is `y`. After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
```

(a) (b)

Figure 8.2: Two three-address code sequences

```
L   R1,a           L    R0, a
A   R1,b           A    R0, b
M   R0,c           A    R0, c
D   R0,d           SRDA R0, 32
ST  R1,t           D    R0, d
                   ST   R1, t
```

(a) (b)

Figure 8.3: Optimal machine-code sequences

`R`i stands for register i. `SRDA` stands for Shift-Right-Double-Arithmetic and `SRDA R0,32` shifts the dividend into `R1` and clears `R0` so all bits equal its sign bit. `L`, `ST`, and `A` stand for load, store, and add, respectively. Note that the optimal choice for the register into which `a` is to be loaded depends on what will ultimately happen to `t`. □

Strategies for register allocation and assignment are discussed in Section 8.8. Section 8.10 shows that for certain classes of machines we can construct code sequences that evaluate expressions using as few registers as possible.

8.1.5 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid

the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator. In Chapter 10, we shall study code scheduling for pipelined machines that can execute several operations in a single clock cycle.

8.2 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine. In this chapter, we shall use as a target language assembly code for a simple computer that is representative of many register machines. However, the code-generation techniques presented in this chapter can be used on many other classes of machines as well.

8.2.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with n general-purpose registers, `R0`, `R1`, ..., `R`$n-1$. A full-fledged assembly language would have scores of instructions. To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction. We assume the following kinds of instructions are available:

- *Load* operations: The instruction `LD` *dst, addr* loads the value in location *addr* into location *dst*. This instruction denotes the assignment $dst = addr$. The most common form of this instruction is `LD` r, x which loads the value in location x into register r. An instruction of the form `LD` r_1, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_1.

- *Store* operations: The instruction `ST` x, r stores the value in register r into the location x. This instruction denotes the assignment $x = r$.

- *Computation* operations of the form *OP dst*, src_1, src_2, where *OP* is a operator like `ADD` or `SUB`, and *dst*, src_1, and src_2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP* to the values in locations src_1 and src_2, and place the result of this operation in location *dst*. For example, `SUB` r_1, r_2, r_3 computes $r_1 = r_2 - r_3$. Any value formerly stored in r_1 is lost, but if r_1 is r_2 or r_3, the old value is read first. Unary operators that take only one operand do not have a src_2.

- *Unconditional jumps*: The instruction `BR` L causes control to branch to the machine instruction with label L. (`BR` stands for *branch*.)

- *Conditional jumps* of the form `B`*cond* r, L, where r is a register, L is a label, and *cond* stands for any of the common tests on values in the register r. For example, `BLTZ` r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the l-value of x).

- A location can also be an indexed address of the form $a(r)$, where a is a variable and r is a register. The memory location denoted by $a(r)$ is computed by taking the l-value of a and adding to it the value in register r. For example, the instruction `LD R1,a(R2)` has the effect of setting `R1` = *contents*(`a` + *contents*(`R2`)), where *contents*(x) denotes the contents of the register or memory location represented by x. This addressing mode is useful for accessing arrays, where a is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

- A memory location can be an integer indexed by a register. For example, `LD R1,100(R2)` has the effect of setting `R1` = *contents*(100 + *contents*(`R2`)), that is, of loading into `R1` the value in the memory location obtained by adding 100 to the contents of register `R2`. This feature is useful for following pointers, as we shall see in the example below.

- We also allow two indirect addressing modes: `*`r means the memory location found in the location represented by the contents of register r and `*100(`r`)` means the memory location found in the location obtained by adding 100 to the contents of r. For example, `LD R1,*100(R2)` has the effect of setting `R1` = *contents*(*contents*(100 + *contents*(`R2`))), that is, of loading into `R1` the value in the memory location stored in the memory location obtained by adding 100 to the contents of register `R2`.

- Finally, we allow an immediate constant addressing mode. The constant is prefixed by `#`. The instruction `LD R1,#100` loads the integer 100 into register `R1`, and `ADD R1,R1,#100` adds the integer 100 into register `R1`.

Comments at the end of instructions are preceded by `//`.

Example 8.2: The three-address statement `x = y - z` can be implemented by the machine instructions:

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2     // R1 = R1 - R2
ST  x, R1          // x = R1
```

We can do better, perhaps. One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible. For example, `y` and/or `z` may have been computed in a register, and if so we can avoid the `LD` step(s). Likewise, we might be able to avoid ever storing `x` if its value is used within the register set and is not subsequently needed.

Suppose `a` is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of `a` are indexed starting at 0. We may execute the three-address instruction `b = a[i]` by the machine instructions:

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

That is, the second step computes $8i$, and the third step places in register `R2` the value in the ith element of `a` — the one found in the location that is $8i$ bytes past the base address of the array `a`.

Similarly, the assignment into the array `a` represented by three-address instruction `a[j] = c` is implemented by:

```
LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contents(a +  contents(R2)) = R1
```

To implement a simple pointer indirection, such as the three-address statement `x = *p`, we can use machine instructions like:

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST  x, R2          // x = R2
```

The assignment through a pointer `*p = y` is similarly implemented in machine code by:

```
LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contents(0 + contents(R1)) = R2
```

Finally, consider a conditional-jump three-address instruction like

```
if x < y goto L
```

The machine-code equivalent would be something like:

```
LD   R1, x          // R1 = x
LD   R2, y          // R2 = y
SUB  R1, R1, R2     // R1 = R1 - R2
BLTZ R1, M          // if R1 < 0 jump to M
```

Here, `M` is the label that represents the first machine instruction generated from the three-address instruction that has label `L`. As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored. □

8.2.2 Program and Instruction Costs

We often associate a cost with compiling and running a program. Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are NP-hard. As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

For the remainder of this chapter, we shall assume each target-language instruction has an associated cost. For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction. Some examples:

- The instruction `LD R0, R1` copies the contents of register `R1` into register `R0`. This instruction has a cost of one because no additional memory words are required.

- The instruction `LD R0, M` loads the contents of memory location `M` into register `R0`. The cost is two since the address of memory location `M` is in the word following the instruction.

- The instruction `LD R1, *100(R2)` loads into register `R1` the value given by *contents*(*contents*(100 + *contents*(`R2`))). The cost is two because the constant 100 is stored in the word following the instruction.

In this chapter we assume the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input. Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs. We shall see that in some situations we can actually generate optimal code for expressions on certain classes of register machines.

8.2.3 Exercises for Section 8.2

Exercise 8.2.1: Generate code for the following three-address statements assuming all variables are stored in memory locations.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

Exercise 8.2.2: Generate code for the following three-address statements assuming a and b are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

Exercise 8.2.3: Generate code for the following three-address sequence assuming that `p` and `q` are in memory locations:

```
    y = *q
    q = q + 4
    *p = y
    p = p + 4
```

Exercise 8.2.4: Generate code for the following sequence assuming that `x`, `y`, and `z` are in memory locations:

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

Exercise 8.2.5: Generate code for the following sequence assuming that `n` is in a memory location:

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

Exercise 8.2.6: Determine the costs of the following instruction sequences:

a)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

b)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

c)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

d)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

e)
```
LD R0, p
LD R1, x
ST 0(R0), R1
```

f)
```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 Addresses in the Target Code

In this section, we show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation. In Section 7.1, we described how each executing program runs in its own logical address space that was partitioned into four code and data areas:

1. A statically determined area *Code* that holds the executable target code. The size of the target code can be determined at compile time.

2. A statically determined data area *Static* for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

3. A dynamically managed area *Heap* for holding data objects that are allocated and freed during program execution. The size of the *Heap* cannot be determined at compile time.

4. A dynamically managed area *Stack* for holding activation records as they are created and destroyed during procedure calls and returns. Like the *Heap*, the size of the *Stack* cannot be determined at compile time.

8.3.1 Static Allocation

To illustrate code generation for simplified procedure calls and returns, we shall focus on the following three-address statements:

- `call` *callee*
- `return`
- `halt`
- `action`, which is a placeholder for other three-address statements.

The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table. We shall first illustrate how to store the return address in an activation record on a procedure

call and how to return control to it after the procedure call. For convenience, we assume the first location in the activation record holds the return address.

Let us first consider the code needed to implement the simplest case, static allocation. Here, a `call` *callee* statement in the intermediate code can be implemented by a sequence of two target-machine instructions:

```
ST   callee.staticArea, #here + 20
BR   callee.codeArea
```

The `ST` instruction saves the return address at the beginning of the activation record for *callee*, and the `BR` transfers control to the target code for the called procedure *callee*. The attribute *callee.staticArea* is a constant that gives the address of the beginning of the activation record for *callee*, and the attribute *callee.codeArea* is a constant referring to the address of the first instruction of the called procedure *callee* in the *Code* area of the run-time memory.

The operand `#`*here* + 20 in the `ST` instruction is the literal return address; it is the address of the instruction following the `BR` instruction. We assume that `#`*here* is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of 5 words or 20 bytes.

The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is `HALT`, which returns control to the operating system. A `return` statement can be implemented by a simple jump instruction

```
BR   *callee.staticArea
```

which transfers control to the address saved at the beginning of the activation record for *callee*.

Example 8.3 : Suppose we have the following three-address code:

```
                    // code for c
action1
call p
action2
halt
                    // code for p
action3
return
```

Figure 8.4 shows the target program for this three-address code. We use the pseudoinstruction `ACTION` to represent the sequence of machine instructions to execute the statement `action`, which represents three-address code that is not relevant for this discussion. We arbitrarily start the code for procedure `c` at address 100 and for procedure `p` at address 200. We assume that each `ACTION` instruction takes 20 bytes. We further assume that the activation records for these procedures are statically allocated starting at locations 300 and 364, respectively.

The instructions starting at address 100 implement the statements

```
action1; call p; action2; halt
```

of the first procedure `c`. Execution therefore starts with the instruction $\texttt{ACTION}_1$ at address 100. The `ST` instruction at address 120 saves the return address 140 in the machine-status field, which is the first word in the activation record of `p`. The `BR` instruction at address 132 transfers control the first instruction in the target code of the called procedure `p`.

```
                        // code for c
100:  ACTION1           // code for action1
120:  ST 364, #140      // save return address 140 in location 364
132:  BR 200            // call p
140:  ACTION2
160:  HALT              // return to operating system
      ...
                        // code for p
200:  ACTION3
220:  BR *364           // return to address saved in location 364
      ...
                        // 300-363 hold activation record for c
300:                    // return address
304:                    // local data for c
      ...
                        // 364-451 hold activation record for p
364:                    // return address
368:                    // local data for p
```

Figure 8.4: Target code for static allocation

After executing $\texttt{ACTION}_3$, the jump instruction at location 220 is executed. Since location 140 was saved at address 364 by the call sequence above, `*364` represents 140 when the `BR` statement at address 220 is executed. Therefore, when procedure `p` terminates, control returns to address 140 and execution of procedure `c` resumes. □

8.3.2 Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, however, the position of an activation record for a procedure is not known until run time. This position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. The indexed address mode of our target machine is convenient for this purpose.

Relative addresses in an activation record can be taken as offsets from any known position in the activation record, as we saw in Chapter 7. For conve-

nience, we shall use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on top of the stack. When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure. After control returns to the caller, we decrement SP, thereby deallocating the activation record of the called procedure.

The code for the first procedure initializes the stack by setting SP to the start of the stack area in memory:

```
LD   SP, #stackStart          // initialize the stack
code for the first procedure
HALT                          // terminate execution
```

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD  SP, SP, #caller.recordSize    // increment stack pointer
ST   0(SP), #here + 16             // save return address
BR   callee.codeArea               // jump to the callee
```

The operand #*caller.recordSize* represents the size of an activation record, so the ADD instruction makes SP point to the next activation record. The operand #*here* + 16 in the ST instruction is the address of the instruction following BR; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

```
BR  *0(SP)                    // return to caller
```

The reason for using *0(SP) in the BR instruction is that we need two levels of indirection: 0(SP) is the address of the first word in the activation record and *0(SP) is the return address saved there.

The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

```
SUB  SP, SP, #caller.recordSize    // decrement stack pointer
```

Chapter 7 contains a broader discussion of calling sequences and the tradeoffs in the division of labor between the calling and called procedures.

Example 8.4: The program in Fig. 8.5 is an abstraction of the quicksort program in the previous chapter. Procedure q is recursive, so more than one activation of q can be alive at the same time.

Suppose that the sizes of the activation records for procedures m, p, and q have been determined to be *msize*, *psize*, and *qsize*, respectively. The first word in each activation record will hold a return address. We arbitrarily assume that the code for these procedures starts at addresses 100, 200, and 300, respectively,

```
                    // code for m
action1
call q
action2
halt
                    // code for p
action3
return
                    // code for q
action4
call p
action5
call q
action6
call q
return
```

Figure 8.5: Code for Example 8.4

and that the stack starts at address 600. The target program is shown in Figure 8.6.

We assume that $\texttt{ACTION}_4$ contains a conditional jump to the address 456 of the return sequence from q; otherwise, the recursive procedure q is condemned to call itself forever.

Let *msize*, *psize*, and *qsize* be 20, 40, and 60, respectively. The first instruction at address 100 initializes the SP to 600, the starting address of the stack. SP holds 620 just before control transfers from m to q, because *msize* is 20. Subsequently, when q calls p, the instruction at address 320 increments SP to 680, where the activation record for p begins; SP reverts to 620 after control returns to q. If the next two recursive calls of q return immediately, the maximum value of SP during this execution is 680. Note, however, that the last stack location used is 739, since the activation record of q starting at location 680 extends for 60 bytes. □

8.3.3 Run-Time Addresses for Names

The storage-allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed. In Chapter 6, we assumed that a name in a three-address statement is really a pointer to a symbol-table entry for that name. This approach has a significant advantage; it makes the compiler more portable, since the front end need not be changed even when the compiler is moved to a different machine where a different run-time organization is needed. On the other hand, generating the specific sequence of access steps while generating intermediate code can be of

```
                                // code for m
100:  LD SP, #600               // initialize the stack
108:  ACTION1                   // code for action1
128:  ADD SP, SP, #msize        // call sequence begins
136:  ST 0(SP), #152            // push return address
144:  BR 300                    // call q
152:  SUB SP, SP, #msize        // restore SP
160:  ACTION2
180:  HALT
      ...
                                // code for p
200:  ACTION3
220:  BR *0(SP)                 // return
      ...
                                // code for q
300:  ACTION4                   // contains a conditional jump to 456
320:  ADD SP, SP, #qsize
328:  ST 0(SP), #344            // push return address
336:  BR 200                    // call p
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  ST 0(SP), #396            // push return address
388:  BR 300                    // call q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST 0(SP), #440            // push return address
440:  BR 300                    // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                 // return
      ...
600:                            // stack starts here
```

Figure 8.6: Target code for stack allocation

significant advantage in an optimizing compiler, since it lets the optimizer take advantage of details it would not see in the simple three-address statement.

In either case, names must eventually be replaced by code to access storage locations. We thus consider some elaborations of the simple three-address copy statement `x = 0`. After the declarations in a procedure are processed, suppose the symbol-table entry for `x` contains a relative address 12 for `x`. For example, consider the case in which `x` is in a statically allocated area beginning at address *static*. Then the actual run-time address of `x` is *static* + 12. Although the compiler can eventually determine the value of *static* + 12 at compile time, the position of the static area may not be known when intermediate code to access the name is generated. In that case, it makes sense to generate three-address code to "compute" *static* + 12, with the understanding that this computation will be carried out during the code generation phase, or possibly by the loader, before the program runs. The assignment `x = 0` then translates into

```
static[12] = 0
```

If the static area starts at address 100, the target code for this statement is

```
LD 112, #0
```

8.3.4 Exercises for Section 8.3

Exercise 8.3.1: Generate code for the following three-address statements assuming stack allocation where register `SP` points to the top of the stack.

```
call p
call q
return
call r
return
return
```

Exercise 8.3.2: Generate code for the following three-address statements assuming stack allocation where register `SP` points to the top of the stack.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

Exercise 8.3.3: Generate code for the following three-address statements again assuming stack allocation and assuming `a` and `b` are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

8.4 Basic Blocks and Flow Graphs

This section introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. Code generation benefits from context. We can do a better job of register allocation if we know how values are defined and used, as we shall see in Section 8.8. We can do a better job of instruction selection by looking at sequences of three-address statements, as we shall see in Section 8.9.

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that

   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

   (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

The Effect of Interrupts

The notion that control, once it reaches the beginning of a basic block is certain to continue through to the end requires a bit of thought. There are many reasons why an interrupt, not reflected explicitly in the code, could cause control to leave the block, perhaps never to return. For example, an instruction like `x = y/z` appears not to affect control flow, but if z is 0 it could actually cause the program to abort.

We shall not worry about such possibilities. The reason is as follows. The purpose of constructing basic blocks is to optimize the code. Generally, when an interrupt occurs, either it will be handled and control will come back to the instruction that caused the interrupt, as if control had never deviated, or the program will halt with an error. In the latter case, it doesn't matter how we optimized the code, even if we depended on control reaching the end of the basic block, because the program didn't produce its intended result anyway.

Starting in Chapter 9, we discuss transformations on flow graphs that turn the original intermediate code into "optimized" intermediate code from which better target code can be generated. The "optimized" intermediate code is turned into machine code using the code-generation techniques in this chapter.

8.4.1 Basic Blocks

Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Example 8.6 : The intermediate code in Fig. 8.7 turns a 10×10 matrix `a` into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix `a` is stored in row-major form.

for i from 1 to 10 **do**
    for j from 1 to 10 **do**
        $a[i, j] = 0.0$;
for i from 1 to 10 **do**
    $a[i, i] = 1.0$;

Figure 8.8: Source code for Fig. 8.7

First, instruction 1 is a leader by rule (1) of Algorithm 8.5. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17. □

8.4.2 Next-Use Information

Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Our algorithm to determine liveness and next-use information makes a backward pass over each basic block. We store the information in the symbol table. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 8.5. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Algorithm 8.7: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.

2. In the symbol table, set x to "not live" and "no next use."

3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Here we have used + as a symbol representing any operator. If the three-address statement i is of the form $x = +\, y$ or $x = y$, the steps are the same as above, ignoring z. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z. □

8.4.3 Flow Graphs

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a *predecessor* of C, and C is a *successor* of B.

Often we add two nodes, called the *entry* and *exit*, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

Example 8.8: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B_1, since B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4.

Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. □

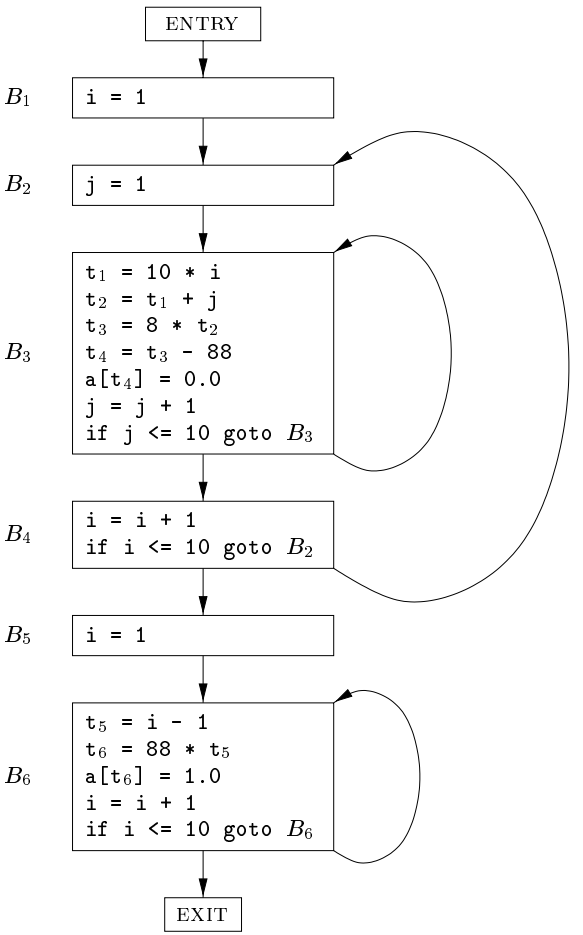


Figure 8.9: Flow graph from Fig. 8.7

8.4.4 Representation of Flow Graphs

First, note from Fig. 8.9 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks. Recall that every conditional or unconditional jump is to the leader of some basic block, and it is to this block that the jump will now refer. The reason for this change is that after constructing the flow graph, it is common to make substantial changes to the instructions in the various basic blocks. If jumps were to instructions, we would have to fix the targets of the jumps every time one of the target instructions was changed.

Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) need their own representation. We might represent the content of a node by a

pointer to the leader in the array of three-address instructions, together with a count of the number of instructions or a second pointer to the last instruction. However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

8.4.5 Loops

Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a *loop* if L contains a node e called the *loop entry*, such that:

1. e is not ENTRY, the entry of the entire flow graph.
2. No node in L besides e has a predecessor outside L. That is, every path from ENTRY to any node in L goes through e.
3. Every node in L has a nonempty path, completely within L, to e.

Example 8.9: The flow graph of Fig. 8.9 has three loops:

1. B_3 by itself.
2. B_6 by itself.
3. $\{B_2, B_3, B_4\}$.

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the last requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a nonempty path to B_2 staying within L. For instance, B_2 has the path $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$. $\Box$

8.4.6 Exercises for Section 8.4

Exercise 8.4.1: Figure 8.10 is a simple matrix-multiplication program.

a) Translate the program into three-address statements of the type we have been using in this section. Assume the matrix entries are numbers that require 8 bytes, and that matrices are stored in row-major order.

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Figure 8.10: A matrix-multiplication algorithm

Exercise 8.4.2: Figure 8.11 is code to count the number of primes from 2 to n, using the sieve method on a suitably large array a. That is, $a[i]$ is `TRUE` at the end only if there is no prime $\sqrt{i}$ or less that evenly divides i. We initialize all $a[i]$ to `TRUE` and then set $a[j]$ to `FALSE` if we find a divisor of j.

a) Translate the program into three-address statements of the type we have been using in this section. Assume integers require 4 bytes.

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) /* i has been found to be a prime */ {
        count++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE; /* no multiple of i is a prime */
    }
```

Figure 8.11: Code to sieve for primes

8.5 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. More thorough *global* optimization, which looks at how information flows among the basic blocks of a program, is covered in later chapters, starting with Chapter 9. It is a complex subject, with many different techniques to consider.

8.5.1 The DAG Representation of Basic Blocks

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). In Section 6.1.1, we introduced the DAG as a representation for single expressions. The idea extends naturally to the collection of expressions that are created within one basic block. We construct a DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, discussed in Section 9.2.5.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.

b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

8.5.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions in Section 6.1.1.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement `c = b + c`, we know that the use of `b` in `b + c` refers to the node of Fig. 8.12 labeled $-$, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

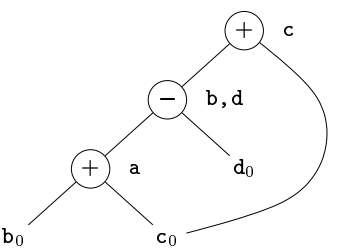


Figure 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement `d = a - d` has the operator $-$ and the nodes with attached variables `a` and $\mathtt{d}_0$ as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled $-$. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 8.12, the basic block in Example 8.10 can be replaced by a block with only three statements. In fact, if `b` is not live on exit from the block, then we do not need to compute that variable, and can use `d` to receive the value represented by the node labeled $-$ in Fig. 8.12. The block then becomes

```
a = b + c
d = a - d
c = d + c
```

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.[1]

Example 8.11: When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a = b + c
b = b - d
c = c + d
e = b + c
```

is the same, namely $b_0 + c_0$. That is, even though b and c both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 8.13, but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, as discussed in Section 8.5.4, may expose the equivalence. □

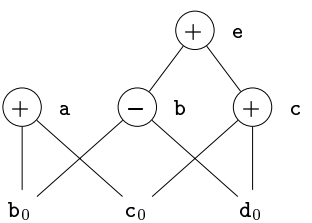


Figure 8.13: DAG for basic block in Example 8.11

8.5.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example 8.12: If, in Fig. 8.13, a and b are live but c and e are not, we can immediately remove the root labeled e. Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached. □

[1] In general, we must be careful, when reconstructing code from DAG's, how we choose the names of variables. If a variable x is defined twice, or if it is assigned once and the initial value x_0 is also used, then we must make sure that we do not change the value of x until we have made all uses of the node whose value x previously held.

8.5.4 The Use of Algebraic Identities

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local *reduction in strength*, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

A third class of related optimizations is *constant folding*. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.[2] Thus the expression $2 * 3.14$ would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that $*$ is commutative; that is, $x*y = y*x$. Before we create a new node labeled $*$ with left child M and right child N, we always check whether such a node already exists. However, because $*$ is commutative, we should then check for a node having operator $*$, left child N, and right child M.

The relational operators such as $<$ and $=$ sometimes generate unexpected common subexpressions. For example, the condition $x > y$ can also be tested by subtracting the arguments and performing a test on the condition code set by the subtraction.[3] Thus, only one node of the DAG may need to be generated for $x - y$ and $x > y$.

Associative laws might also be applicable to expose common subexpressions. For example, if the source code has the assignments

```
a = b + c;
e = c + d + b;
```

the following intermediate code might be generated:

[2] Arithmetic expressions should be evaluated the same way at compile time as they are at run time. K. Thompson has suggested an elegant solution to constant folding: compile the constant expression, execute the target code on the spot, and replace the expression with the result. Thus, the compiler does not need to contain an interpreter.

[3] The subtraction can, however, introduce overflows and underflows while a compare instruction would not.

```
a = b + c
t = c + d
e = t + b
```

If `t` is not needed outside this block, we can change this sequence to

```
a = b + c
e = a + d
```

using both the associativity and commutativity of $+$.

The compiler writer should examine the language reference manual carefully to determine what rearrangements of computations are permitted, since (because of possible overflows or underflows) computer arithmetic does not always obey the algebraic identities of mathematics. For example, the Fortran standard states that a compiler may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Thus, a compiler may evaluate $x * y - x * z$ as $x * (y - z)$, but it may not evaluate $a + (b - c)$ as $(a + b) - c$. A Fortran compiler must therefore keep track of where parentheses were present in the source language expressions if it is to optimize programs in accordance with the language definition.

8.5.5 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three-address statements:

```
x = a[i]
a[j] = y
z = a[i]
```

If we think of `a[i]` as an operation involving `a` and `i`, similar to $a + i$, then it might appear as if the two uses of `a[i]` were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction `z = a[i]` by the simpler `z = x`. However, since `j` could equal `i`, the middle statement may in fact change the value of `a[i]`; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like `x = a[i]`, is represented by creating a node with operator =[] and two children representing the initial value of the array, $\mathtt{a}_0$ in this case, and the index `i`. Variable `x` becomes a label of this new node.

2. An assignment to an array, like `a[j] = y`, is represented by a new node with operator []= and three children representing $\mathtt{a}_0$, `j` and `y`. There is no variable labeling this node. What is different is that the creation of

this node *kills* all currently constructed nodes whose value depends on $\texttt{a}_0$. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Example 8.13: The DAG for the basic block

```
x = a[i]
a[j] = y
z = a[i]
```

is shown in Fig. 8.14. The node N for `x` is created first, but when the node labeled `[]=` is created, N is killed. Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands $\texttt{a}_0$ and $\texttt{i}_0$ must be created instead. □

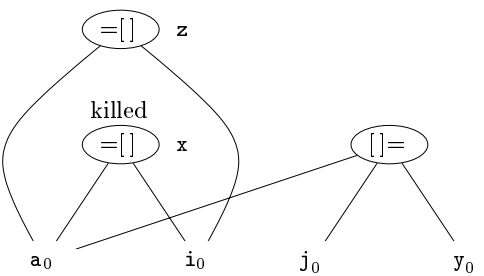


Figure 8.14: The DAG for a sequence of array assignments

Example 8.14: Sometimes, a node must be killed even though none of its children have an array like $\texttt{a}_0$ in Example 8.13 as attached variable. Likewise, a node can kill if it has a descendant that is an array, even though none of its children are array nodes. For instance, consider the three-address code

```
b = 12 + a
x = b[i]
b[j] = y
```

What is happening here is that, for efficiency reasons, `b` has been defined to be a position in an array `a`. For example, if the elements of `a` are four bytes long, then `b` represents the fourth element of `a`. If `j` and `i` represent the same value, then `b[i]` and `b[j]` represent the same location. Therefore it is important to have the third instruction, `b[j] = y`, kill the node with `x` as its attached variable. However, as we see in Fig. 8.15, both the killed node and the node that does the killing have $\texttt{a}_0$ as a grandchild, not as a child. □

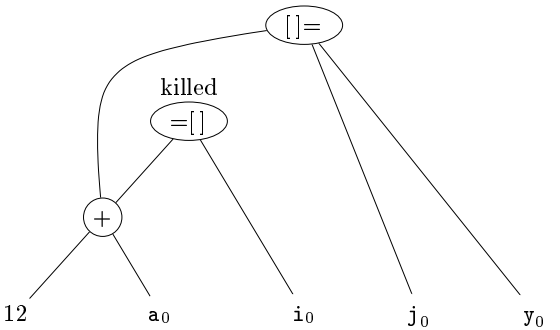


Figure 8.15: A node that kills a use of an array need not have that array as a child

8.5.6 Pointer Assignments and Procedure Calls

When we assign indirectly through a pointer, as in the assignments

```
x = *p
*q = y
```

we do not know what `p` or `q` point to. In effect, `x = *p` is a use of every variable whatsoever, and `*q = y` is a possible assignment to every variable. As a consequence, the operator $=*$ must take all nodes that are currently associated with identifiers as arguments, which is relevant for dead-code elimination. More importantly, the $*=$ operator kills all other nodes so far constructed in the DAG.

There are global pointer analyses one could perform that might limit the set of variables a pointer could reference at a given place in the code. Even local analysis could restrict the scope of a pointer. For instance, in the sequence

```
p = &x
*p = y
```

we know that `x`, and no other variable, is given the value of `y`, so we don't need to kill any node but the node to which `x` was attached.

Procedure calls behave much like assignments through pointers. In the absence of global data-flow information, we must assume that a procedure uses and changes any data to which it has access. Thus, if procedure P is in the scope of variable x, a call to P both uses the node with attached variable x and kills that node.

8.5.7 Reassembling Basic Blocks From DAG's

After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three-address code for the basic block from which we built the DAG. For each

node that has one or more attached variables, we construct a three-address statement that computes the value of one of those variables. We prefer to compute the result into a variable that is live on exit from the block. However, if we do not have global live-variable information to work from, we need to assume that every variable of the program (but not temporaries that are generated by the compiler to process expressions) is live on exit from the block.

If the node has more than one live variable attached, then we have to introduce copy statements to give the correct value to each of those variables. Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

Example 8.15: Recall the DAG of Fig. 8.12. In the discussion following Example 8.10, we decided that if `b` is not live on exit from the block, then the three statements

```
a = b + c
d = a - d
c = d + c
```

suffice to reconstruct the basic block. The third instruction, `c = d + c`, must use `d` as an operand rather than `b`, because the optimized block never computes `b`.

If both `b` and `d` are live on exit, or if we are not sure whether or not they are live on exit, then we need to compute `b` as well as `d`. We can do so with the sequence

```
a = b + c
d = a - d
b = d
c = d + c
```

This basic block is still more efficient than the original. Although the number of instructions is the same, we have replaced a subtraction by a copy, which tends to be less expensive on most machines. Further, it may be that by doing a global analysis, we can eliminate the use of this computation of `b` outside the block by replacing it by uses of `d`. In that case, we can come back to this basic block and eliminate `b = d` later. Intuitively, we can eliminate this copy if wherever this value of `b` is used, `d` is still holding the same value. That situation may or may not be true, depending on how the program recomputes `d`. □

When reconstructing the basic block from a DAG, we not only need to worry about what variables are used to hold the values of the DAG's nodes, but we also need to worry about the order in which we list the instructions computing the values of the various nodes. The rules to remember are

1. The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.

2. Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.

3. Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.

4. Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.

5. Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

That is, when reordering code, no statement may cross a procedure call or assignment through a pointer, and uses of the same array may cross each other only if both are array accesses, but not assignments to elements of the array.

8.5.8 Exercises for Section 8.5

Exercise 8.5.1: Construct the DAG for the basic block

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Exercise 8.5.2: Simplify the three-address code of Exercise 8.5.1, assuming

a) Only a is live on exit from the block.

b) a, b, and c are live on exit from the block.

Exercise 8.5.3: Construct the DAG for the code in block B_6 of Fig. 8.9. Do not forget to include the comparison $i \leq 10$.

Exercise 8.5.4: Construct the DAG for the code in block B_3 of Fig. 8.9.

Exercise 8.5.5: Extend Algorithm 8.7 to process three-statements of the form

a) `a[i] = b`

b) `a = b[i]`

c) `a = *b`

c) `*a = b`

Exercise 8.5.6: Construct the DAG for the basic block

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

on the assumption that

a) `p` can point anywhere.

b) `p` can point only to `b` or `d`.

! **Exercise 8.5.7:** If a pointer or array expression, such as `a[i]` or `*p` is assigned and then used, without the possibility of being changed in the interim, we can take advantage of the situation to simplify the DAG. For example, in the code of Exercise 8.5.6, if `p` cannot point to `d`, then the fourth statement `e = *p` can be replaced by `e = c`. Revise the DAG-construction algorithm to take advantage of such situations, and apply your algorithm to the code of Exercise 8.5.6.

Exercise 8.5.8: Suppose a basic block is formed from the C assignment statements

```
x = a + b + c + d + e + f;
y = a + c + e;
```

a) Give the three-address statements (only one addition per statement) for this block.

b) Use the associative and commutative laws to modify the block to use the fewest possible number of instructions, assuming both `x` and `y` are live on exit from the block.

8.6 A Simple Code Generator

In this section, we shall consider an algorithm that generates code for a single basic block. It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

One of the primary issues during code generation is deciding how to use registers to best advantage. There are four principal uses of registers:

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries — places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.

- Registers are used to hold (*global*) values that are computed in one basic block and used in other blocks, for example, a loop index that is incremented going around the loop and is used several times within the loop.
- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

These are competing needs, since the number of registers available is limited.

The algorithm in this section assumes that some set of registers is available to hold the values that are used within the block. Typically, this set of registers does not include all the registers of the machine, since some registers are reserved for global variables and managing the stack. We assume that the basic block has already been transformed into a preferred sequence of three-address instructions, by transformations such as combining common subexpressions. We further assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form

- `LD` *reg, mem*
- `ST` *mem, reg*
- *OP reg, reg, reg*

8.6.1 Register and Address Descriptors

Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.

In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so. We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored. The desired data structure has the following descriptors:

1. For each available register, a *register descriptor* keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.
2. For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

8.6.2 The Code-Generation Algorithm

An essential part of the algorithm is a function *getReg*(I), which selects registers for each memory location associated with the three-address instruction I. Function *getReg* has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block. We shall discuss *getReg* after presenting the basic algorithm. While we do not know the total number of registers available for local data belonging to a basic block, we assume that there are enough registers so that, after freeing all available registers by storing their values in memory, there are enough registers to accomplish any three-address operation.

In a three-address instruction such as `x = y + z`, we shall treat `+` as a generic operator and `ADD` as the equivalent machine instruction. We do not, therefore, take advantage of commutativity of `+`. Thus, when we implement the operation, the value of `y` must be in the second register mentioned in the `ADD` instruction, never the third. A possible improvement to the algorithm is to generate code for both `x = y + z` and `x = z + y` whenever `+` is a commutative operator, and pick the better code sequence.

Machine Instructions for Operations

For a three-address instruction such as $x = y + z$, do the following:

1. Use *getReg*($x = y + z$) to select registers for x, y, and z. Call these R_x, R_y, and R_z.

2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction `LD` R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).

3. Similarly, if z is not in R_z, issue an instruction `LD` R_z, z', where z' is a location for z.

4. Issue the instruction `ADD` R_x, R_y, R_z.

Machine Instructions for Copy Statements

There is an important special case: a three-address copy statement of the form $x = y$. We assume that *getReg* will always choose the same register for both x and y. If y is not already in that register R_y, then generate the machine instruction `LD` R_y, `y`. If y was already in R_y, we do nothing. It is only necessary that we adjust the register descriptor for R_y so that it includes x as one of the values found there.

Ending the Basic Block

As we have described the algorithm, variables used by the block may wind up with their only location being a register. If the variable is a temporary used only within the block, that is fine; when the block ends, we can forget about the value of the temporary and assume its register is empty. However, if the variable is live on exit from the block, or if we don't know which variables are live on exit, then we need to assume that the value of the variable is needed later. In that case, for each variable x whose address descriptor does not say that its value is located in the memory location for x, we must generate the instruction `ST` x, R, where R is a register in which x's value exists at the end of the block.

Managing Register and Address Descriptors

As the code-generation algorithm issues load, store, and other machine instructions, it needs to update the register and address descriptors. The rules are as follows:

1. For the instruction `LD` R, x
   (a) Change the register descriptor for register R so it holds only x.
   (b) Change the address descriptor for x by adding register R as an additional location.
2. For the instruction `ST` x, R, change the address descriptor for x to include its own memory location.
3. For an operation such as `ADD` R_x, R_y, R_z implementing a three-address instruction $x = y + z$
   (a) Change the register descriptor for R_x so that it holds only x.
   (b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
   (c) Remove R_x from the address descriptor of any variable other than x.
4. When we process a copy statement $x = y$, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):
   (a) Add x to the register descriptor for R_y.
   (b) Change the address descriptor for x so that its only location is R_y.

Example 8.16 : Let us translate the basic block consisting of the three-address statements

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

Here we assume that `t`, `u`, and `v` are temporaries, local to the block, while `a`, `b`, `c`, and `d` are variables that are live on exit from the block. Since we have not yet discussed how the function *getReg* might work, we shall simply assume that there are as many registers as we need, but that when a register's value is no longer needed (for example, it holds only a temporary, all of whose uses have been passed), then we reuse its register.

A summary of all the machine-code instructions generated is in Fig. 8.16. The figure also shows the register and address descriptors before and after the translation of each three-address instruction.

	R1	R2	R3	a	b	c	d	t	u	v
				a	b	c	d			
`t = a - b`										
`LD R1, a`										
`LD R2, b`										
`SUB R2, R1, R2`										
	a	t		a, R1	b	c	d	R2		
`u = a - c`										
`LD R3, c`										
`SUB R1, R1, R3`										
	u	t	c	a	b	c, R3	d	R2	R1	
`v = t + u`										
`ADD R3, R2, R1`										
	u	t	v	a	b	c	d	R2	R1	R3
`a = d`										
`LD R2, d`										
	u	a, d	v	R2	b	c	d, R2		R1	R3
`d = v + u`										
`ADD R1, R3, R1`										
	d	a	v	R2	b	c	R1			R3
exit										
`ST a, R2`										
`ST d, R1`										
	d	a	v	a, R2	b	c	d, R1			R3

Figure 8.16: Instructions generated and the changes in the register and address descriptors

For the first three-address instruction, `t = a - b` we need to issue three instructions, since nothing is in a register initially. Thus, we see `a` and `b` loaded

into registers R1 and R2, and the value t produced in register R2. Notice that we can use R2 for t because the value b previously in R2 is not needed within the block. Since b is presumably live on exit from the block, had it not been in its own memory location (as indicated by its address descriptor), we would have had to store R2 into b first. The decision to do so, had we needed R2, would be taken by *getReg*.

The second instruction, u = a - c, does not require a load of a, since it is already in register R1. Further, we can reuse R1 for the result, u, since the value of a, previously in that register, is no longer needed within the block, and its value is in its own memory location if a is needed outside the block. Note that we change the address descriptor for a to indicate that it is no longer in R1, but is in the memory location called a.

The third instruction, v = t + u, requires only the addition. Further, we can use R3 for the result, v, since the value of c in that register is no longer needed within the block, and c has its value in its own memory location.

The copy instruction, a = d, requires a load of d, since it is not in a register. We show register R2's descriptor holding both a and d. The addition of a to the register descriptor is the result of our processing the copy statement, and is not the result of any machine instruction.

The fifth instruction, d = v + u, uses two values that are in registers. Since u is a temporary whose value is no longer needed, we have chosen to reuse its register R1 for the new value of d. Notice that d is now in only R1, and is not in its own memory location. The same holds for a, which is in R2 and not in the memory location called a. As a result, we need a "coda" to the machine code for the basic block that stores the live-on-exit variables a and d into their memory locations. We show these as the last two instructions. □

8.6.3 Design of the Function *getReg*

Lastly, let us consider how to implement *getReg*(I), for a three-address instruction I. There are many options, although there are also some absolute prohibitions against choices that lead to incorrect code due to the loss of the value of one or more live variables. We begin our examination with the case of an operation step, for which we again use $x = y + z$ as the generic example. First, we must pick a register for y and a register for z. The issues are the same, so we shall concentrate on picking register R_y for y. The rules are as follows:

1. If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.

2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.

3. The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse. Let R be a candidate

register, and suppose v is one of the variables that the register descriptor for R says is in R. We need to make sure that v's value either is not really needed, or that there is somewhere else we can go to get the value of v. The possibilities are:

(a) If the address descriptor for v says that v is somewhere besides R, then we are OK.

(b) If v is x, the variable being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.

(c) Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.

(d) If we are not OK by one of the first three cases, then we need to generate the store instruction `ST` v, R to place a copy of v in its own memory location. This operation is called a *spill*.

Since R may hold several variables at the moment, we repeat the above steps for each such variable v. At the end, R's "score" is the number of store instructions we needed to generate. Pick one of the registers with the lowest score.

Now, consider the selection of the register R_x. The issues and options are almost as for y, so we shall only mention the differences.

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x. This statement holds even if x is one of y and z, since our machine instructions allows two registers to be the same in one instruction.

2. If y is not used after instruction I, in the sense described for variable v in item (3c), and R_y holds only y after being loaded, if necessary, then R_y can also be used as R_x. A similar option holds regarding z and R_z.

The last matter to consider specially is the case when I is a copy instruction $x = y$. We pick the register R_y as above. Then, we always choose $R_x = R_y$.

8.6.4 Exercises for Section 8.6

Exercise 8.6.1: For each of the following C assignment statements

a) `x = a + b*c;`

b) `x = a/(b+c) - d*(e+f);`

c) `x = a[i] + 1;`

d) `a[i] = b[c[i]];`

e) `a[i][j] = b[i][k] + c[k][j];`

f) `*p++ = *q++;`

generate three-address code, assuming that all array elements are integers taking four bytes each. In parts (d) and (e), assume that `a`, `b`, and `c` are constants giving the location of the first (0th) elements of the arrays with those names, as in all previous examples of array accesses in this chapter.

! **Exercise 8.6.2:** Repeat Exercise 8.6.1 parts (d) and (e), assuming that the arrays `a`, `b`, and `c` are located via pointers, `pa`, `pb`, and `pc`, respectively, pointing to the locations of their respective first elements.

Exercise 8.6.3: Convert your three-address code from Exercise 8.6.1 into machine code for the machine model of this section. You may use as many registers as you need.

Exercise 8.6.4: Convert your three-address code from Exercise 8.6.1 into machine code, using the simple code-generation algorithm of this section, assuming three registers are available. Show the register and address descriptors after each step.

Exercise 8.6.5: Repeat Exercise 8.6.4, but assuming only two registers are available.

8.7 Peephole Optimization

While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may

spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit. In this section, we shall give the following examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

8.7.1 Eliminating Redundant Loads and Stores

If we see the instruction sequence

```
LD R0, a
ST a, R0
```

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of `a` has already been loaded into register `R0`. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

Redundant loads and stores of this nature would not be generated by the simple code generation algorithm of the previous section. However, a naive code generation algorithm like the one in Section 8.1.3 would generate redundant sequences such as these.

8.7.2 Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable `debug` is equal to 1. In the intermediate representation, this code may look like

```
      if debug == 1 goto L1
      goto L2
L1:   print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of `debug`, the code sequence above can be replaced by

```
        if debug != 1 goto L2
        print debugging information
L2:
```

If `debug` is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
        if 0 != 1 goto L2
        print debugging information
L2:
```

Now the argument of the first statement always evaluates to *true*, so the statement can be replaced by `goto L2`. Then all statements that print debugging information are unreachable and can be eliminated one at a time.

8.7.3 Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
        goto L1
        ...
L1:     goto L2
```

by the sequence

```
        goto L2
        ...
L1:     goto L2
```

If there are now no jumps to `L1`, then it may be possible to eliminate the statement `L1: goto L2` provided it is preceded by an unconditional jump.

Similarly, the sequence

```
        if a < b goto L1
        ...
L1:     goto L2
```

can be replaced by the sequence

```
        if a < b goto L2
        ...
L1:     goto L2
```

Finally, suppose there is only one jump to `L1` and `L1` is preceded by an unconditional goto. Then the sequence

```
        goto L1
        . . .
L1:     if a < b goto L2
L3:
```

may be replaced by the sequence

```
        if a < b goto L2
        goto L3
        . . .
L3:
```

While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

8.7.4 Algebraic Simplification and Reduction in Strength

In Section 8.5 we discussed algebraic identities that could be used to simplify DAG's. These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

```
x = x + 0
```

or

```
x = x * 1
```

in the peephole.

Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be approximated as multiplication by a constant, which may be cheaper.

8.7.5 Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like `x = x + 1`.

8.7.6 Exercises for Section 8.7

Exercise 8.7.1: Construct an algorithm that will perform redundant-instruction elimination in a sliding peephole on target machine code.

Exercise 8.7.2: Construct an algorithm that will do flow-of-control optimizations in a sliding peephole on target machine code.

Exercise 8.7.3: Construct an algorithm that will do simple algebraic simplifications and reductions in strength in a sliding peephole on target machine code.

8.8 Register Allocation and Assignment

Instructions involving only register operands are faster than those involving memory operands. On modern machines, processor speeds are often an order of magnitude or more faster than memory speeds. Therefore, efficient utilization of registers is vitally important in generating good code. This section presents various strategies for deciding at each point in a program what values should reside in registers (register allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in the target program to certain registers. For example, we could decide to assign base addresses to one group of registers, arithmetic computations to another, the top of the stack to a fixed register, and so on.

This approach has the advantage that it simplifies the design of a code generator. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers. Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers, and the like, and to allow the remaining registers to be used by the code generator as it sees fit.

8.8.1 Global Register Allocation

The code generation algorithm in Section 8.6 used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (*globally*). Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop. For the time being, assume that we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block. The next chapter covers techniques for computing this information.

One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. The selected values may be different in different loops. Registers not already allocated may be used to hold values local to one block as in Section 8.6. This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation. Yet the method is simple to implement and was used in Fortran H, the optimizing Fortran compiler developed by IBM for the 360-series machines in the late 1960s.

With early C compilers, a programmer could do some register allocation explicitly by using register declarations to keep certain values in registers for the duration of a procedure. Judicious use of register declarations did speed up many programs, but programmers were encouraged to first profile their programs to determine the program's hotspots before doing their own register allocation.

8.8.2 Usage Counts

In this section we shall assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register. However, if we use the approach in Section 8.6 to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block. Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block. Thus, if x is allocated a register, we count a savings of two for each block in loop L for which x is live on exit and in which x is assigned a value.

On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units. Similarly, for each exit block B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two. However, on the assumption that the loop is iterated many times, we may neglect these debits since they occur only once each time we enter the loop. Thus, an approximate formula for the benefit to be realized from allocating a register for x within loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B) \tag{8.1}$$

where $use(x, B)$ is the number of times x is used in B prior to any definition of x; $live(x, B)$ is 1 if x is live on exit from B and is assigned a value in B, and $live(x, B)$ is 0 otherwise. Note that (8.1) is approximate, because not all blocks in a loop are executed with equal frequency and also because (8.1) is based on the assumption that a loop is iterated many times. On specific machines a formula analogous to (8.1), but possibly quite different from it, would have to be developed.

Example 8.17: Consider the basic blocks in the inner loop depicted in Fig. 8.17, where jump and conditional jump statements have been omitted. Assume registers `R0`, `R1`, and `R2` are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig. 8.17 for convenience, immediately above and below each block, respectively. There are some subtle points about live variables that we address in the next chapter. For example, notice that both `e` and `f` are live at the end of B_1, but of these, only `e` is live on entry to B_2 and only `f` on entry to B_3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

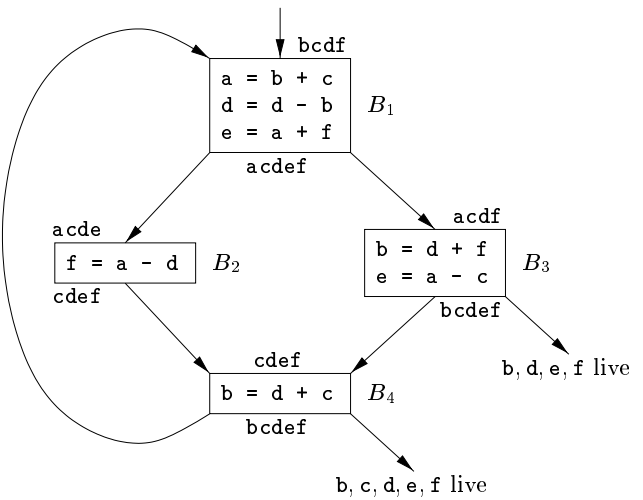


Figure 8.17: Flow graph of an inner loop

To evaluate (8.1) for $x =$ `a`, we observe that `a` is live on exit from B_1 and is assigned a value there, but is not live on exit from B_2, B_3, or B_4. Thus, $\sum_{B \text{ in } L} use(\texttt{a}, B) = 2$. Hence the value of (8.1) for $x =$ `a` is 4. That is, four units of cost can be saved by selecting `a` for one of the global registers. The values of (8.1) for `b`, `c`, `d`, `e`, and `f` are 5, 3, 6, 4, and 4, respectively. Thus, we may select `a`, `b`, and `d` for registers `R0`, `R1`, and `R2`, respectively. Using `R0` for `e` or `f` instead of `a` would be another choice with the same apparent benefit. Figure 8.18 shows the assembly code generated from Fig. 8.17, assuming that the strategy of Section 8.6 is used to generate code for each block. We do not show the generated code for the omitted conditional or unconditional jumps that end each block in Fig. 8.17, and we therefore do not show the generated code as a single stream as it would appear in practice. □

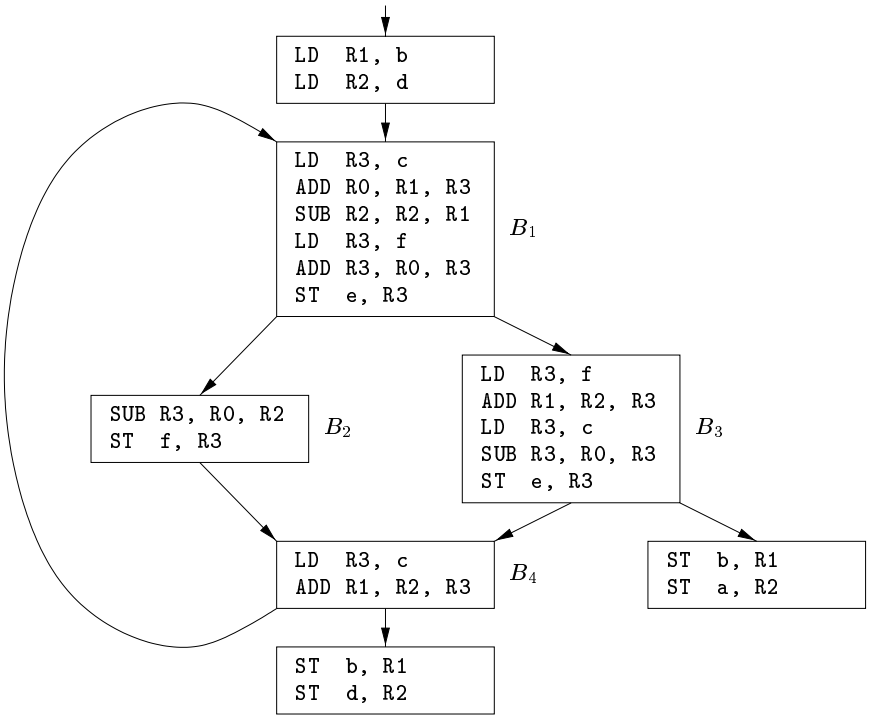


Figure 8.18: Code sequence using global register assignment

8.8.3 Register Assignment for Outer Loops

Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops. If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers in $L_1 - L_2$. However, if we choose to allocate x a register in L_2 but not L_1, we must load x on entrance to L_2 and store x on exit from L_2. We leave as an exercise the derivation of a criterion for selecting names to be allocated registers in an outer loop L, given that choices have already been made for all loops nested within L.

8.8.4 Register Allocation by Graph Coloring

When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (*spilled*) into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

In the method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and

the three-address instructions become machine-language instructions. If access to variables requires instructions that use stack pointers, display pointers, base registers, or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose. Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction. If access is more complex, the access must be broken into several machine instructions, and a temporary symbolic register (or several) may need to be created.

Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of spills.

In the second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined. For example, a register-interference graph for Fig. 8.17 would have nodes for names `a` and `d`. In block B_1, `a` is live at the second statement, which defines `d`; therefore, in the graph there would be an edge between the nodes for `a` and `d`.

An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers. A graph is said to be *colored* if each node has been assigned a color in such a way that no two adjacent nodes have the same color. A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice. Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'. A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.

By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible. At this point a node is spilled by introducing code to store and reload the register. Chaitin has devised several heuristics for choosing the node to spill. A general rule is to avoid introducing spill code into inner loops.

8.8.5 Exercises for Section 8.8

Exercise 8.8.1: Construct the register-interference graph for the program in Fig. 8.17.

Exercise 8.8.2: Devise a register-allocation strategy on the assumption that we automatically store all registers on the stack before each procedure call and restore them after the return.

8.9 Instruction Selection by Tree Rewriting

Instruction selection can be a large combinatorial task, especially for machines that are rich in addressing modes, such as CISC machines, or on machines with special-purpose instructions, say, for signal processing. Even if we assume that the order of evaluation is given and that registers are allocated by a separate mechanism, instruction selection — the problem of selecting target-language instructions to implement the operators in the intermediate representation — remains a large combinatorial task.

In this section, we treat instruction selection as a tree-rewriting problem. Tree representations of target instructions have been used effectively in code-generator generators, which automatically construct the instruction-selection phase of a code generator from a high-level specification of the target machine. Better code might be obtained for some machines by using DAG's rather than trees, but DAG matching is more complex than tree matching.

8.9.1 Tree-Translation Schemes

Throughout this section, the input to the code-generation process will be a sequence of trees at the semantic level of the target machine. The trees are what we might get after inserting run-time addresses into the intermediate representation, as described in Section 8.3. In addition, the leaves of the trees contain information about the storage types of their labels.

Example 8.18: Figure 8.19 contains a tree for the assignment statement `a[i] = b + 1`, where the array `a` is stored on the run-time stack and the variable `b` is a global in memory location M_b. The run-time addresses of locals `a` and `i` are given as constant offsets C_a and C_i from `SP`, the register containing the pointer to the beginning of the current activation record.

The assignment to `a[i]` is an indirect assignment in which the r-value of the location for `a[i]` is set to the r-value of the expression `b + 1`. The addresses of array `a` and variable `i` are given by adding the values of the constant C_a and C_i, respectively, to the contents of register `SP`. We simplify array-address calculations by assuming that all values are one-byte characters. (Some instruction sets make special provisions for multiplications by constants, such as 2, 4, and 8, during address calculations.)

In the tree, the **ind** operator treats its argument as a memory address. As the left child of an assignment operator, the **ind** node gives the location into which the r-value on the right side of the assignment operator is to be stored. If an argument of a + or **ind** operator is a memory location or a register, then the contents of that memory location or register are taken as the value. The leaves in the tree are labeled with attributes; a subscript indicates the value of the attribute. □

The target code is generated by applying a sequence of tree-rewriting rules to reduce the input tree to a single node. Each tree-rewriting rule has the form

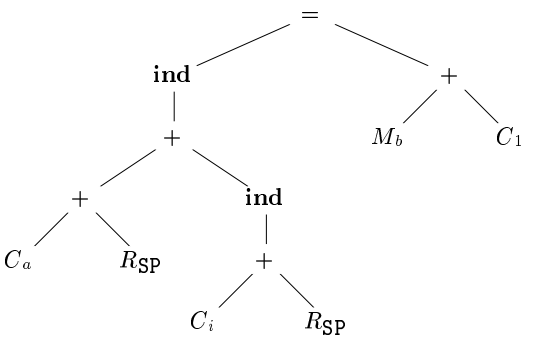


Figure 8.19: Intermediate-code tree for `a[i] = b + 1`

$$\textit{replacement} \leftarrow \textit{template}\ \{\ \textit{action}\ \}$$

where *replacement* is a single node, *template* is a tree, and *action* is a code fragment, as in a syntax-directed translation scheme.

A set of tree-rewriting rules is called a *tree-translation scheme*.

Each tree-rewriting rule represents the translation of a portion of the tree given by the template. The translation consists of a possibly empty sequence of machine instructions that is emitted by the action associated with the template. The leaves of the template are attributes with subscripts, as in the input tree. Sometimes, certain restrictions apply to the values of the subscripts in the templates; these restrictions are specified as semantic predicates that must be satisfied before the template is said to match. For example, a predicate might specify that the value of a constant fall in a certain range.

A tree-translation scheme is a convenient way to represent the instruction-selection phase of a code generator. As an example of a tree-rewriting rule, consider the rule for the register-to-register add instruction:

This rule is used as follows. If the input tree contains a subtree that matches this tree template, that is, a subtree whose root is labeled by the operator + and whose left and right children are quantities in registers i and j, then we can replace that subtree by a single node labeled R_i and emit the instruction `ADD Ri, Ri, Rj` as output. We call this replacement a *tiling* of the subtree. More than one template may match a subtree at a given time; we shall describe shortly some mechanisms for deciding which rule to apply in cases of conflict.

Example 8.19 : Figure 8.20 contains tree-rewriting rules for a few instructions of our target machine. These rules will be used in a running example throughout this section. The first two rules correspond to load instructions, the next two

to store instructions, and the remainder to indexed loads and additions. Note that rule (8) requires the value of the constant to be 1. This condition would be specified by a semantic predicate. □

8.9.2 Code Generation by Tiling an Input Tree

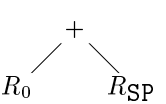
A tree-translation scheme works as follows. Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees. If a template matches, the matching subtree in the input tree is replaced with the replacement node of the rule and the action associated with the rule is done. If the action contains a sequence of machine instructions, the instructions are emitted. This process is repeated until the tree is reduced to a single node, or until no more templates match. The sequence of machine instructions generated as the input tree is reduced to a single node constitutes the output of the tree-translation scheme on the given input tree.

The process of specifying a code generator becomes similar to that of using a syntax-directed translation scheme to specify a translator. We write a tree-translation scheme to describe the instruction set of a target machine. In practice, we would like to find a scheme that causes a minimal-cost instruction sequence to be generated for each input tree. Several tools are available to help build a code generator automatically from a tree-translation scheme.

Example 8.20 : Let us use the tree-translation scheme in Fig. 8.20 to generate code for the input tree in Fig. 8.19. Suppose that the first rule is applied to load the constant C_a into register `R0`:

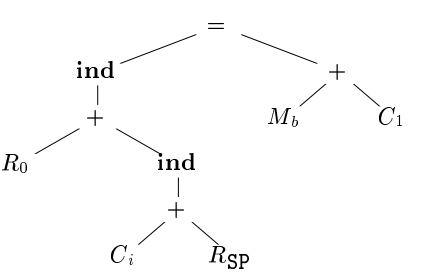
1) $R_0 \leftarrow C_a$ { `LD R0, #a` }

The label of the leftmost leaf then changes from C_a to R_0 and the instruction `LD R0, #a` is generated. The seventh rule now matches the leftmost subtree with root labeled +:

7) $R_0 \leftarrow$ 

{ `ADD R0, R0, SP` }

Using this rule, we rewrite this subtree as a single node labeled R_0 and generate the instruction `ADD R0, R0, SP`. Now the tree looks like



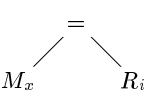
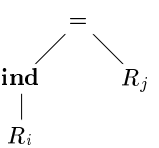
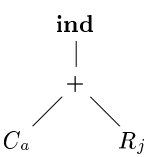
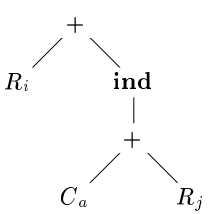
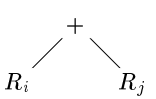
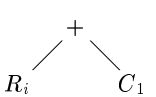
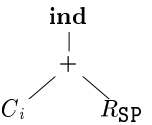
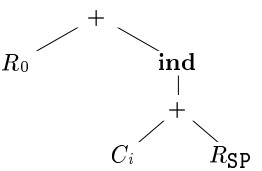
1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ 	{ ST x, Ri }
4)	$M \leftarrow$ 	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ 	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

Figure 8.20: Tree-rewriting rules for some target-machine instructions

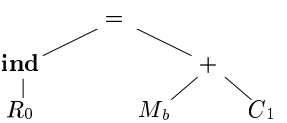
At this point, we could apply rule (5) to reduce the subtree



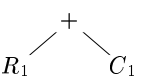
to a single node labeled, say, R_1. We could also use rule (6) to reduce the larger subtree



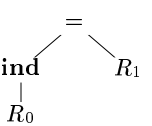
to a single node labeled R_0 and generate the instruction `ADD R0,R0,i(SP)`. Assuming that it is more efficient to use a single instruction to compute the larger subtree rather than the smaller one, we choose rule (6) to get



In the right subtree, rule (2) applies to the leaf M_b. It generates an instruction to load `b` into register `R1`, say. Now, using rule (8) we can match the subtree



and generate the increment instruction `INC R1`. At this point, the input tree has been reduced to



This remaining tree is matched by rule (4), which reduces the tree to a single node and generates the instruction `ST *R0,R1`. We generate the following code sequence:

```
LD  R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD  R1, b
INC R1
ST  *R0, R1
```

in the process of reducing the tree to a single node. □

In order to implement the tree-reduction process in Example 8.18, we must address some issues related to tree-pattern matching:

- How is tree-pattern matching to be done? The efficiency of the code-generation process (at compile time) depends on the efficiency of the tree-matching algorithm.
- What do we do if more than one template matches at a given time? The efficiency of the generated code (at run time) may depend on the order in which templates are matched, since different match sequences will in general lead to different target-machine code sequences, some more efficient than others.

If no template matches, then the code-generation process blocks. At the other extreme, we need to guard against the possibility of a single node being rewritten indefinitely, generating an infinite sequence of register move instructions or an infinite sequence of loads and stores.

To prevent blocking, we assume that each operator in the intermediate code can be implemented by one or more target-machine instructions. We further assume that there are enough registers to compute each tree node by itself. Then, no matter how the tree matching proceeds, the remaining tree can always be translated into target-machine instructions.

8.9.3 Pattern Matching by Parsing

Before considering general tree matching, we consider a specialized approach that uses an LR parser to do the pattern matching. The input tree can be treated as a string by using its prefix representation. For example, the prefix representation for the tree in Fig. 8.19 is

$$= \mathbf{ind} + + C_a \; R_{\mathtt{SP}} \; \mathbf{ind} + C_i \; R_{\mathtt{SP}} + M_b \; C_1$$

The tree-translation scheme can be converted into a syntax-directed translation scheme by replacing the tree-rewriting rules with the productions of a context-free grammar in which the right sides are prefix representations of the instruction templates.

Example 8.21 : The syntax-directed translation scheme in Fig. 8.21 is based on the tree-translation scheme in Fig. 8.20.

The nonterminals of the underlying grammar are R and M. The terminal **m** represents a specific memory location, such as the location for the global variable `b` in Example 8.18. The production $M \rightarrow \mathbf{m}$ in Rule (10) can be thought of as matching M with **m** prior to using one of the templates involving M. Similarly, we introduce a terminal **sp** for register `SP` and add the production $R \rightarrow \mathbf{sp}$. Finally, terminal **c** represents constants.

Using these terminals, the string for the input tree in Fig. 8.19 is

1)	R_i	$\rightarrow$	$\mathbf{c}_a$	{ LD Ri, #a }
2)	R_i	$\rightarrow$	M_x	{ LD Ri, x }
3)	M	$\rightarrow$	$= M_x\ R_i$	{ ST x, Ri }
4)	M	$\rightarrow$	$= \mathbf{ind}\ R_i\ R_j$	{ ST *Ri, Rj }
5)	R_i	$\rightarrow$	$\mathbf{ind} + \mathbf{c}_a\ R_j$	{ LD Ri, a(Rj) }
6)	R_i	$\rightarrow$	$+ R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$	{ ADD Ri, Ri, a(Rj) }
7)	R_i	$\rightarrow$	$+ R_i\ R_j$	{ ADD Ri, Ri, Rj }
8)	R_i	$\rightarrow$	$+ R_i\ \mathbf{c}_1$	{ INC Ri }
9)	R	$\rightarrow$	$\mathbf{sp}$	
10)	M	$\rightarrow$	$\mathbf{m}$	

Figure 8.21: Syntax-directed translation scheme constructed from Fig. 8.20

$$= \mathbf{ind} + + \mathbf{c}_a\ \mathbf{sp}\ \mathbf{ind} + \mathbf{c}_i\ \mathbf{sp} + \mathbf{m}_b\ \mathbf{c}_1$$

□

From the productions of the translation scheme we build an LR parser using one of the LR-parser construction techniques of Chapter 4. The target code is generated by emitting the machine instruction corresponding to each reduction.

A code-generation grammar is usually highly ambiguous, and some care needs to be given to how the parsing-action conflicts are resolved when the parser is constructed. In the absence of cost information, a general rule is to favor larger reductions over smaller ones. This means that in a reduce-reduce conflict, the longer reduction is favored; in a shift-reduce conflict, the shift move is chosen. This "maximal munch" approach causes a larger number of operations to be performed with a single machine instruction.

There are some benefits to using LR parsing in code generation. First, the parsing method is efficient and well understood, so reliable and efficient code generators can be produced using the algorithms described in Chapter 4. Second, it is relatively easy to retarget the resulting code generator; a code selector for a new machine can be constructed by writing a grammar to describe the instructions of the new machine. Third, the the code generated can be made more efficient by adding special-case productions to take advantage of machine idioms.

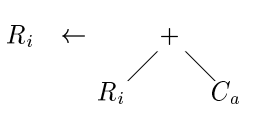
However, there are some challenges as well. A left-to-right order of evaluation is fixed by the parsing method. Also, for some machines with large numbers of addressing modes, the machine-description grammar and resulting parser can become inordinately large. As a consequence, specialized techniques are necessary to encode and process the machine-description grammars. We must also be careful that the resulting parser does not block (has no next move) while parsing an expression tree, either because the grammar does not handle some operator patterns or because the parser has made the wrong resolution of some parsing-action conflict. We must also make sure the parser does not get into an

infinite loop of reductions of productions with single symbols on the right side. The looping problem can be solved using a state-splitting technique at the time the parser tables are generated.

8.9.4 Routines for Semantic Checking

In a code-generation translation scheme, the same attributes appear as in an input tree, but often with restrictions on what values the subscripts can have. For example, a machine instruction may require that an attribute value fall in a certain range or that the values of two attributes be related.

These restrictions on attribute values can be specified as predicates that are invoked before a reduction is made. In fact, the general use of semantic actions and predicates can provide greater flexibility and ease of description than a purely grammatical specification of a code generator. Generic templates can be used to represent classes of instructions and the semantic actions can then be used to pick instructions for specific cases. For example, two forms of the addition instruction can be represented with one template:




```
{ if (a = 1)
      INC Ri
  else
      ADD Ri, Ri, #a }
```

Parsing-action conflicts can be resolved by disambiguating predicates that can allow different selection strategies to be used in different contexts. A smaller description of a target machine is possible because certain aspects of the machine architecture, such as addressing modes, can be factored into the attributes. The complication in this approach is that it may become difficult to verify the accuracy of the translation scheme as a faithful description of the target machine, although this problem is shared to some degree by all code generators.

8.9.5 General Tree Matching

The LR-parsing approach to pattern matching based on prefix representations favors the left operand of a binary operator. In a prefix representation **op** E_1 E_2, the limited-lookahead LR parsing decisions must be made on the basis of some prefix of E_1, since E_1 can be arbitrarily long. Thus, pattern matching can miss nuances of the target-instruction set that are due to right operands.

Instead prefix representation, we could use a postfix representation. But, then an LR-parsing approach to pattern matching would favor the right operand.

For a hand-written code generator, we can use tree templates, as in Fig. 8.20, as a guide and write an ad-hoc matcher. For example, if the root of the input tree is labeled **ind**, then the only pattern that could match is for rule (5); otherwise, if the root is labeled +, then the patterns that could match are for rules (6-8).

For a code-generator generator, we need a general tree-matching algorithm. An efficient top-down algorithm can be developed by extending the string-pattern-matching techniques of Chapter 3. The idea is to represent each template as a set of strings, where a string corresponds to a path from the root to a leaf in the template. We treat all operands equally by including the position number of a child, from left to right, in the strings.

Example 8.22: In building the set of strings for an instruction set, we shall drop the subscripts, since pattern matching is based on the attributes alone, not on their values.

The templates in Fig. 8.22 have the following set of strings from the root to a leaf:

$$
\begin{array}{l}
C \\
+\ 1\ R \\
+\ 2\ \textbf{ind}\ 1\ +\ 1\ C \\
+\ 2\ \textbf{ind}\ 1\ +\ 2\ R \\
+\ 2\ R
\end{array}
$$

The string C represents the template with C at the root. The string $+\,1\,R$ represents the $+$ and its left operand R in the two templates that have $+$ at the root. $\Box$

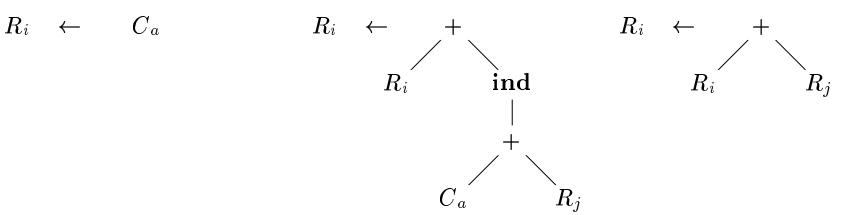


Figure 8.22: An instruction set for tree matching

Using sets of strings as in Example 8.22, a tree-pattern matcher can be constructed by using techniques for efficiently matching multiple strings in parallel.

In practice, the tree-rewriting process can be implemented by running the tree-pattern matcher during a depth-first traversal of the input tree and performing the reductions as the nodes are visited for the last time.

Instruction costs can be taken into account by associating with each tree-rewriting rule the cost of the sequence of machine instructions generated if that rule is applied. In Section 8.11, we discuss a dynamic programming algorithm that can be used in conjunction with tree-pattern matching.

By running the dynamic programming algorithm concurrently, we can select an optimal sequence of matches using the cost information associated with each rule. We may need to defer deciding upon a match until the cost of all alternatives is known. Using this approach, a small, efficient code generator can

be constructed quickly from a tree-rewriting scheme. Moreover, the dynamic programming algorithm frees the code-generator designer from having to resolve conflicting matches or decide upon an order for the evaluation.

8.9.6 Exercises for Section 8.9

Exercise 8.9.1: Construct syntax trees for each of the following statements assuming all nonconstant operands are in memory locations:

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

b) `x[i] = y[j] * z[k];`

c) `x = x + 1;`

Use the tree-rewriting scheme in Fig. 8.20 to generate code for each statement.

Exercise 8.9.2: Repeat Exercise 8.9.1 above using the syntax-directed translation scheme in Fig. 8.21 in place of the tree-rewriting scheme.

! **Exercise 8.9.3:** Extend the tree-rewriting scheme in Fig. 8.20 to apply to while-statements.

! **Exercise 8.9.4:** How would you extend tree rewriting to apply to DAG's?

8.10 Optimal Code Generation for Expressions

We can choose registers optimally when a basic block consists of a single expression evaluation, or if we accept that it is sufficient to generate code for a block one expression at a time. In the following algorithm, we introduce a numbering scheme for the nodes of an expression tree (a syntax tree for an expression) that allows us to generate optimal code for an expression tree when there is a fixed number of registers with which to evaluate the expression.

8.10.1 Ershov Numbers

We begin by assigning to each node of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries. These numbers are sometimes called *Ershov numbers*, after A. Ershov, who used a similar scheme for machines with a single arithmetic register. For our machine model, the rules are:

1. Label all leaves 1.
2. The label of an interior node with one child is the label of its child.
3. The label of an interior node with two children is

(a) The larger of the labels of its children, if those labels are different.

(b) One plus the label of its children if the labels are the same.

Example 8.23: In Fig. 8.23 we see an expression tree (with operators omitted) that might be the tree for expression $(a - b) + e \times (c + d)$ or the three-address code:

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

Each of the five leaves is labeled 1 by rule (1). Then, we can label the interior node for `t1 = a - b`, since both of its children are labeled. Rule (3b) applies, so it gets label one more than the labels of its children, that is, 2. The same holds for the interior node for `t2 = c + d`.

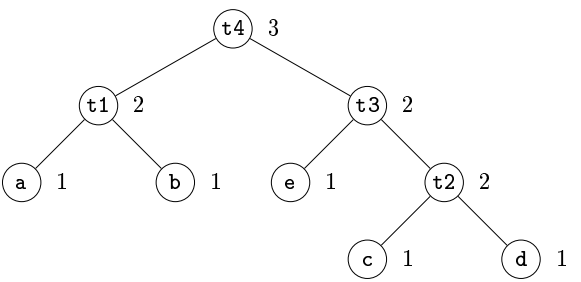


Figure 8.23: A tree labeled with Ershov numbers

Now, we can work on the node for `t3 = e * t2`. Its children have labels 1 and 2, so the label of the node for `t3` is the maximum, 2, by rule (3a). Finally, the root, the node for `t4 = t1 + t3`, has two children with label 2, and therefore it gets label 3. □

8.10.2 Generating Code From Labeled Expression Trees

It can be proved that, in our machine model, where all operands must be in registers, and registers can be used by both an operand and the result of an operation, the label of a node is the fewest registers with which the expression can be evaluated using no stores of temporary results. Since in this model, we are forced to load each operand, and we are forced to compute the result corresponding to each interior node, the only thing that can make the generated code inferior to the optimal code is if there are unnecessary stores of temporaries. The argument for this claim is embedded in the following algorithm for generating code with no stores of temporaries, using a number of registers equal to the label of the root.

Algorithm 8.24: Generating code from a labeled expression tree.

INPUT: A labeled tree with each operand appearing once (that is, no common subexpressions).

OUTPUT: An optimal sequence of machine instructions to evaluate the root into a register.

METHOD: The following is a recursive algorithm to generate the machine code. The steps below are applied, starting at the root of the tree. If the algorithm is applied to a node with label k, then only k registers will be used. However, there is a "base" $b \geq 1$ for the registers used so that the actual registers used are $R_b, R_{b+1}, \ldots, R_{b+k-1}$. The result always appears in R_{b+k-1}.

1. To generate machine code for an interior node with label k and two children with equal labels (which must be $k-1$) do the following:
   (a) Recursively generate code for the right child, using base $b+1$. The result of the right child appears in register R_{b+k-1}.
   (b) Recursively generate code for the left child, using base b; the result appears in R_{b+k-2}.
   (c) Generate the instruction `OP` $R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$, where `OP` is the appropriate operation for the interior node in question.
2. Suppose we have an interior node with label k and children with unequal labels. Then one of the children, which we'll call the "big" child, has label k, and the other child, the "little" child, has some label $m < k$. Do the following to generate code for this interior node, using base b:
   (a) Recursively generate code for the big child, using base b; the result appears in register R_{b+k-1}.
   (b) Recursively generate code for the little child, using base b; the result appears in register R_{b+m-1}. Note that since $m < k$, neither R_{b+k-1} nor any higher-numbered register is used.
   (c) Generate the instruction `OP` $R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$ or the instruction `OP` $R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$, depending on whether the big child is the right or left child, respectively.
3. For a leaf representing operand x, if the base is b generate the instruction `LD` R_b, x.

□

Example 8.25: Let us apply Algorithm 8.24 to the tree of Fig. 8.23. Since the label of the root is 3, the result will appear in R_3, and only R_1, R_2, and R_3 will be used. The base for the root is $b = 1$. Since the root has children of equal labels, we generate code for the right child first, with base 2.

When we generate code for the right child of the root, labeled `t3`, we find the big child is the right child and the little child is the left child. We thus generate code for the right child first, with $b = 2$. Applying the rules for equal-labeled children and leaves, we generate the following code for the node labeled `t2`:

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
```

Next, we generate code for the left child of the right child of the root; this node is the leaf labeled e. Since $b = 2$, the proper instruction is

```
LD  R2, e
```

Now we can complete the code for the right child of the root by adding the instruction

```
MUL R3, R2, R3
```

The algorithm proceeds to generate code for the left child of the root, leaving the result in R_2, and with base 1. The complete sequence of instructions is shown in Fig. 8.24. □

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Figure 8.24: Optimal three-register code for the tree of Fig. 8.23

8.10.3 Evaluating Expressions with an Insufficient Supply of Registers

When there are fewer registers available than the label of the root of the tree, we cannot apply Algorithm 8.24 directly. We need to introduce some store instructions that spill values of subtrees into memory, and we then need to load those values back into registers as needed. Here is the modified algorithm that takes into account a limitation on the number of registers.

Algorithm 8.26: Generating code from a labeled expression tree.

INPUT: A labeled tree with each operand appearing once (i.e., no common subexpressions) and a number of registers $r \geq 2$.

OUTPUT: An optimal sequence of machine instructions to evaluate the root into a register, using no more than r registers, which we assume are $R_1, R_2, \ldots, R_r$.

METHOD: Apply the following recursive algorithm, starting at the root of the tree, with base $b = 1$. For a node N with label r or less, the algorithm is exactly the same as Algorithm 8.24, and we shall not repeat those steps here. However, for interior nodes with a label $k > r$, we need to work on each side of the tree separately and store the result of the larger subtree. That result is brought back from memory just before node N is evaluated, and the final step will take place in registers R_{r-1} and R_r. The modifications to the basic algorithm are as follows:

1. Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.

2. Recursively generate code for the big child, using base $b = 1$. The result of this evaluation will appear in register R_r.

3. Generate the machine instruction `ST` t_k, R_r, where t_k is a temporary variable used for temporary results used to help evaluate nodes with label k.

4. Generate code for the little child as follows. If the little child has label r or greater, pick base $b = 1$. If the label of the little child is $j < r$, then pick $b = r - j$. Then recursively apply this algorithm to the little child; the result appears in R_r.

5. Generate the instruction `LD` R_{r-1}, t_k.

6. If the big child is the right child of N, then generate the instruction `OP` R_r, R_r, R_{r-1}. If the big child is the left child, generate `OP` R_r, R_{r-1}, R_r.

□

Example 8.27: Let us revisit the expression represented by Fig. 8.23, but now assume that $r = 2$; that is, only registers `R1` and `R2` are available to hold temporaries used in the evaluation of expressions. When we apply Algorithm 8.26 to Fig. 8.23, we see that the root, with label 3, has a label that is larger than $r = 2$. Thus, we need to identify one of the children as the "big" child. Since they have equal labels, either would do. Suppose we pick the right child as the big child.

Since the label of the big child of the root is 2, there are enough registers. We thus apply Algorithm 8.24 to this subtree, with $b = 1$ and two registers. The result looks very much like the code we generated in Fig. 8.24, but with registers `R1` and `R2` in place of `R2` and `R3`. This code is

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
```

Now, since we need both registers for the left child of the root, we need to generate the instruction

```
ST  t3, R2
```

Next, the left child of the root is handled. Again, the number of registers is sufficient for this child, and the code is

```
LD  R2, b
LD  R1, a
SUB R2, R1, R2
```

Finally, we reload the temporary that holds the right child of the root with the instruction

```
LD  R1, t3
```

and execute the operation at the root of the tree with the instruction

```
ADD R2, R2, R1
```

The complete sequence of instructions is shown in Fig. 8.25. □

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
ST  t3, R2
LD  R2, b
LD  R1, a
SUB R2, R1, R2
LD  R1, t3
ADD R2, R2, R1
```

Figure 8.25: Optimal three-register code for the tree of Fig. 8.23, using only two registers

8.10.4 Exercises for Section 8.10

Exercise 8.10.1 : Compute Ershov numbers for the following expressions:

a) $a/(b+c) - d * (e+f)$.

b) $a + b * \big(c * (d+e)\big)$.

c) $(-a + *p) * \big((b - *q)/(-c + *r)\big)$.

Exercise 8.10.2: Generate optimal code using two registers for each of the expressions of Exercise 8.10.1.

Exercise 8.10.3: Generate optimal code using three registers for each of the expressions of Exercise 8.10.1.

! **Exercise 8.10.4:** Generalize the computation of Ershov numbers to expression trees with interior nodes with three or more children.

! **Exercise 8.10.5:** An assignment to an array element, such as `a[i] = x`, appears to be an operator with three operands: a, i, and x. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

! **Exercise 8.10.6:** The original Ershov numbers were used for a machine that allowed the right operand of an expression to be in memory, rather than a register. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

! **Exercise 8.10.7:** Some machines require two registers for certain single-precision values. Suppose that the result of a multiplication of single-register quantities requires two consecutive registers, and when we divide a/b, the value of a must be held in two consecutive registers. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

8.11 Dynamic Programming Code-Generation

Algorithm 8.26 in Section 8.10 produces optimal code from an expression tree using an amount of time that is a linear function of the size of the tree. This procedure works for machines in which all computation is done in registers and in which instructions consist of an operator applied to two registers or to a register and a memory location.

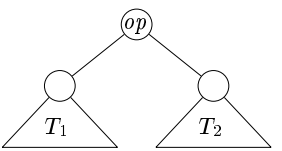
An algorithm based on the principle of dynamic programming can be used to extend the class of machines for which optimal code can be generated from expression trees in linear time. The dynamic programming algorithm applies to a broad class of register machines with complex instruction sets.

The dynamic programming algorithm can be used to generate code for any machine with r interchangeable registers `R0`, `R1`, ... , `R`$r-1$ and load, store, and operation instructions. For simplicity, we assume every instruction costs one unit, although the dynamic programming algorithm can easily be modified to work even if each instruction has its own cost.

8.11.1 Contiguous Evaluation

The dynamic programming algorithm partitions the problem of generating optimal code for an expression into the subproblems of generating optimal code for the subexpressions of the given expression. As a simple example, consider an expression E of the form $E_1 + E_2$. An optimal program for E is formed by combining optimal programs for E_1 and E_2, in one or the other order, followed by code to evaluate the operator $+$. The subproblems of generating optimal code for E_1 and E_2 are solved similarly.

An optimal program produced by the dynamic programming algorithm has an important property. It evaluates an expression $E = E_1$ **op** E_2 "contiguously." We can appreciate what this means by looking at the syntax tree T for E:



Here T_1 and T_2 are trees for E_1 and E_2, respectively.

We say a program P evaluates a tree T *contiguously* if it first evaluates those subtrees of T that need to be computed into memory. Then, it evaluates the remainder of T either in the order T_1, T_2, and then the root, or in the order T_2, T_1, and then the root, in either case using the previously computed values from memory whenever necessary. As an example of noncontiguous evaluation, P might first evaluate part of T_1 leaving the value in a register (instead of memory), next evaluate T_2, and then return to evaluate the rest of T_1.

For the register machine in this section, we can prove that given any machine-language program P to evaluate an expression tree T, we can find an equivalent program P' such that

1. P' is of no higher cost than P,
2. P' uses no more registers than P, and
3. P' evaluates the tree contiguously.

This result implies that every expression tree can be evaluated optimally by a contiguous program.

By way of contrast, machines with even-odd register pairs do not always have optimal contiguous evaluations; the x86 architecture uses register pairs for multiplication and division. For such machines, we can give examples of expression trees in which an optimal machine language program must first evaluate into a register a portion of the left subtree of the root, then a portion of the right subtree, then another part of the left subtree, then another part of the right, and so on. This type of oscillation is unnecessary for an optimal evaluation of any expression tree using the machine in this section.

The contiguous evaluation property defined above ensures that for any expression tree T there always exists an optimal program that consists of optimal programs for subtrees of the root, followed by an instruction to evaluate the root. This property allows us to use a dynamic programming algorithm to generate an optimal program for T.

8.11.2 The Dynamic Programming Algorithm

The dynamic programming algorithm proceeds in three phases (suppose the target machine has r registers):

1. Compute bottom-up for each node n of the expression tree T an array C of costs, in which the ith component $C[i]$ is the optimal cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for $1 \leq i \leq r$.

2. Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory.

3. Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

Each of these phases can be implemented to run in time linearly proportional to the size of the expression tree.

The cost of computing a node n includes whatever loads and stores are necessary to evaluate S in the given number of registers. It also includes the cost of computing the operator at the root of S. The zeroth component of the cost vector is the optimal cost of computing the subtree S into memory. The contiguous evaluation property ensures that an optimal program for S can be generated by considering combinations of optimal programs only for the subtrees of the root of S. This restriction reduces the number of cases that need to be considered.

In order to compute the costs $C[i]$ at node n, we view the instructions as tree-rewriting rules, as in Section 8.9. Consider each template E that matches the input tree at node n. By examining the cost vectors at the corresponding descendants of n, determine the costs of evaluating the operands at the leaves of E. For those operands of E that are registers, consider all possible orders in which the corresponding subtrees of T can be evaluated into registers. In each ordering, the first subtree corresponding to a register operand can be evaluated using i available registers, the second using $i-1$ registers, and so on. To account for node n, add in the cost of the instruction associated with the template E. The value $C[i]$ is then the minimum cost over all possible orders.

The cost vectors for the entire tree T can be computed bottom up in time linearly proportional to the number of nodes in T. It is convenient to store at each node the instruction used to achieve the best cost for $C[i]$ for each value

of i. The smallest cost in the vector for the root of T gives the minimum cost of evaluating T.

Example 8.28: Consider a machine having two registers `R0` and `R1`, and the following instructions, each of unit cost:

```
LD   Ri, Mj        // Ri = Mj
op   Ri, Ri, Rj    // Ri = Ri op Rj
op   Ri, Ri, Mj    // Ri = Ri op Mj
LD   Ri, Rj        // Ri = Rj
ST   Mi, Rj        // Mi = Rj
```

In these instructions, `R`i is either `R0` or `R1`, and `M`j is a memory location. The operator op represents any arithmetic operator.

Let us apply the dynamic programming algorithm to generate optimal code for the syntax tree in Fig. 8.26. In the first phase, we compute the cost vectors shown at each node. To illustrate this cost computation, consider the cost vector at the leaf `a`. $C[0]$, the cost of computing `a` into memory, is 0 since it is already there. $C[1]$, the cost of computing `a` into a register, is 1 since we can load it into a register with the instruction `LD R0, a`. $C[2]$, the cost of loading `a` into a register with two registers available, is the same as that with one register available. The cost vector at leaf `a` is therefore $(0, 1, 1)$.

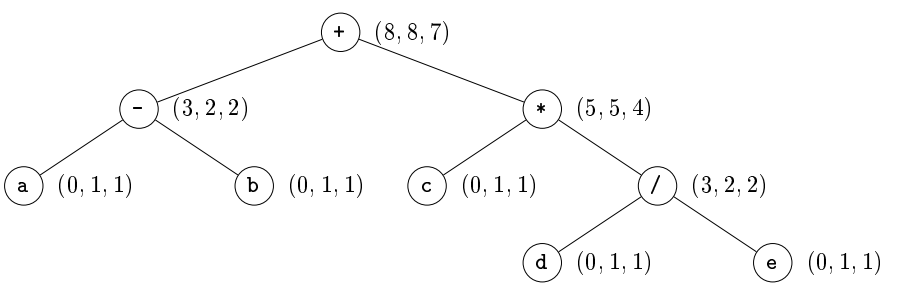


Figure 8.26: Syntax tree for `(a-b)+c*(d/e)` with cost vector at each node

Consider the cost vector at the root. We first determine the minimum cost of computing the root with one and two registers available. The machine instruction `ADD R0, R0, M` matches the root, because the root is labeled with the operator +. Using this instruction, the minimum cost of evaluating the root with one register available is the minimum cost of computing its right subtree into memory, plus the minimum cost of computing its left subtree into the register, plus 1 for the instruction. No other way exists. The cost vectors at the right and left children of the root show that the minimum cost of computing the root with one register available is $5 + 2 + 1 = 8$.

Now consider the minimum cost of evaluating the root with two registers available. Three cases arise depending on which instruction is used to compute the root and in what order the left and right subtrees of the root are evaluated.

1. Compute the left subtree with two registers available into register `R0`, compute the right subtree with one register available into register `R1`, and use the instruction `ADD R0, R0, R1` to compute the root. This sequence has cost $2 + 5 + 1 = 8$.

2. Compute the right subtree with two registers available into `R1`, compute the left subtree with one register available into `R0`, and use the instruction `ADD R0, R0, R1`. This sequence has cost $4 + 2 + 1 = 7$.

3. Compute the right subtree into memory location `M`, compute the left subtree with two registers available into register `R0`, and use the instruction `ADD R0, R0, M`. This sequence has cost $5 + 2 + 1 = 8$.

The second choice gives the minimum cost 7.

The minimum cost of computing the root into memory is determined by adding one to the minimum cost of computing the root with all registers available; that is, we compute the root into a register and then store the result. The cost vector at the root is therefore $(8, 8, 7)$.

From the cost vectors we can easily construct the code sequence by making a traversal of the tree. From the tree in Fig. 8.26, assuming two registers are available, an optimal code sequence is

```
LD  R0, c          // R0 = c
LD  R1, d          // R1 = d
DIV R1, R1, e      // R1 = R1 / e
MUL R0, R0, R1     // R0 = R0 * R1
LD  R1, a          // R1 = a
SUB R1, R1, b      // R1 = R1 - b
ADD R1, R1, R0     // R1 = R1 + R0
```

□

Dynamic programming techniques have been used in a number of compilers, including the second version of the portable C compiler, PCC2. The technique facilitates retargeting because of the applicability of the dynamic programming technique to a broad class of machines.

8.11.3 Exercises for Section 8.11

Exercise 8.11.1: Augment the tree-rewriting scheme in Fig. 8.20 with costs, and use dynamic programming and tree matching to generate code for the statements in Exercise 8.9.1.

!! **Exercise 8.11.2:** How would you extend dynamic programming to do optimal code generation on DAG's?

8.12 Summary of Chapter 8

- *Code generation* is the final phase of a compiler. The code generator maps the intermediate representation produced by the front end, or if there is a code optimization phase by the code optimizer, into the target program.
- *Instruction selection* is the process of choosing target-language instructions for each IR statement.
- *Register allocation* is the process of deciding which IR values to keep in registers. Graph coloring is an effective technique for doing register allocation in compilers.
- *Register assignment* is the process of deciding which register should hold a given IR value.
- A *retargetable compiler* is one that can generate code for multiple instruction sets.
- A *virtual machine* is an interpreter for a bytecode intermediate language produced for languages such as Java and C#.
- A *CISC machine* is typically a two-address machine with relatively few registers, several register classes, and variable-length instructions with complex addressing modes.
- A *RISC machine* is typically a three-address machine with many registers in which operations are done in registers.
- A *basic block* is a maximal sequence of consecutive three-address statements in which flow of control can only enter at the first statement of the block and leave at the last statement without halting or branching except possibly at the last statement in the basic block.
- A *flow graph* is a graphical representation of a program in which the nodes of the graph are basic blocks and the edges of the graph show how control can flow among the blocks.
- A *loop* in a flow graph is a strongly connected region with a single entry point called the loop entry.
- A *DAG* representation of a basic block is a directed acyclic graph in which the nodes of the DAG represent the statements within the block and each child of a node corresponds to the statement that is the last definition of an operand used in the statement.
- *Peephole optimizations* are local code-improving transformations that can be applied to a program, usually through a sliding window.

- *Instruction selection* can be done by a tree-rewriting process in which tree patterns corresponding to machine instructions are used to tile a syntax tree. We can associate costs with the tree-rewriting rules and apply dynamic programming to obtain an optimal tiling for useful classes of machines and expressions.
- An *Ershov number* tells how many registers are needed to evaluate an expression without storing any temporaries.
- *Spill code* is an instruction sequence that stores a value in a register into memory in order to make room to hold another value in that register.

8.13 References for Chapter 8

Many of the techniques covered in this chapter have their origins in the earliest compilers. Ershov's labeling algorithm appeared in 1958 [7]. Sethi and Ullman [16] used this labeling in an algorithm that they prove generated optimal code for arithmetic expressions. Aho and Johnson [1] used dynamic programming to generate optimal code for expression trees on CISC machines. Hennessy and Patterson [12] has a good discussion on the evolution of CISC and RISC machine architectures and the tradeoffs involved in designing a good instruction set.

RISC architectures became popular after 1990, although their origins go back to computers like the CDC 6600, first delivered in 1964. Many of the computers designed before 1990 were CISC machines, but most of the general-purpose computers installed after 1990 are still CISC machines because they are based on the Intel 80x86 architecture and its descendants, such as the Pentium. The Burroughs B5000 delivered in 1963 was an early stack-based machine.

Many of the heuristics for code generation proposed in this chapter have been used in various compilers. Our strategy of allocating a fixed number of registers to hold variables for the duration of a loop was used in the implementation of Fortran H by Lowry and Medlock [13].

Efficient register allocation techniques have also been studied from the time of the earliest compilers. Graph coloring as a register-allocation technique was proposed by Cocke, Ershov [8], and Schwartz [15]. Many variants of graph-coloring algorithms have been proposed for register allocation. Our treatment of graph coloring follows Chaitin [3] [4]. Chow and Hennessy describe their priority-based coloring algorithm for register allocation in [5]. See [6] for a discussion of more recent graph-splitting and rewriting techniques for register allocation.

Lexical analyzer and parser generators spurred the development of pattern-directed instruction selection. Glanville and Graham [11] used LR-parser generation techniques for automated instruction selection. Table-driven code generators evolved into a variety of tree-pattern matching code-generation tools [14]. Aho, Ganapathi, and Tjiang [2] combined efficient tree-pattern matching

techniques with dynamic programming in the code generation tool *twig*. Fraser, Hanson, and Proebsting [10] further refined these ideas in their simple efficient code-generator generator.

1. Aho, A. V. and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM* **23**:3, pp. 488–501.

2. Aho, A. V., M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Programming Languages and Systems* **11**:4 (1989), pp. 491–516.

3. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages* **6**:1 (1981), pp. 47–57.

4. Chaitin, G. J., "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17**:6 (1982), pp. 201–207.

5. Chow, F. and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Programming Languages and Systems* **12**:4 (1990), pp. 501–536.

6. Cooper, K. D. and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco CA, 2004.

7. Ershov, A. P., "On programming of arithmetic operations," *Comm. ACM* **1**:8 (1958), pp. 3–6. Also, *Comm. ACM* **1**:9 (1958), p. 16.

8. Ershov, A. P., *The Alpha Automatic Programming System*, Academic Press, New York, 1971.

9. Fischer, C. N. and R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Redwood City, CA, 1991.

10. Fraser, C. W., D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code generator generator," *ACM Letters on Programming Languages and Systems* **1**:3 (1992), pp. 213–226.

11. Glanville, R. S. and S. L. Graham, "A new method for compiler code generation," *Conf. Rec. Fifth ACM Symposium on Principles of Programming Languages* (1978), pp. 231–240.

12. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

13. Lowry, E. S. and C. W. Medlock, "Object code optimization," *Comm. ACM* **12**:1 (1969), pp. 13–22.

14. Pelegri-Llopart, E. and S. L. Graham, "Optimal code generation for expressions trees: an application of BURS theory," *Conf. Rec. Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 294–308.

15. Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Technical Report, Courant Institute of Mathematical Sciences, New York, 1973.

16. Sethi, R. and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM* **17**:4 (1970), pp. 715–728.

This page intentionally left blank