CHAPTER 9

Code Generation

The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program, as indicated in Fig. 9.1. The code generation techniques presented in this chapter can be used whether or not an optimization phase occurs before code generation, as in some so-called "optimizing" compilers. Such a phase tries to transform the intermediate code into a form from which more efficient target code can be produced. We shall talk about code optimization in detail in the next chapter.

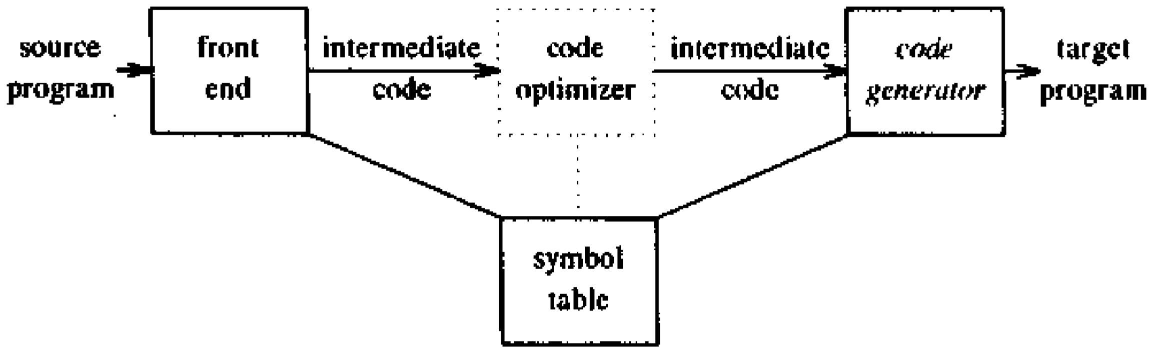

Fig. 9.1. Position of code generator.

The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

Mathematically, the problem of generating optimal code is undecidable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. The choice of heuristics is important, in that a carefully designed code generation algorithm can easily produce code that is several times faster than that produced by a hastily conceived algorithm.

9.1 ISSUES IN THE DESIGN OF A CODE GENERATOR

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code-generation problems. In this section, we shall examine the generic issues in the design of code generators.

Input to the Code Generator

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together 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 intermediate representation.

As we noted in the previous chapter, there are several choices for the intermediate language, including: linear representations such as postfix notation, three-address representations such as quadruples, virtual machine representations such as stack machine code, and graphical representations such as syntax trees and dags. Although the algorithms in this chapter are couched in terms of three-address code, trees, and dags, many of the techniques also apply to the other intermediate representations.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has taken place, so type-conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating-point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

Target Programs

The output of the code generator is the target program. Like the intermediate code, this output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

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. A small program can be compiled and executed quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine-language program (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 segments.

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. Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must use several passes. In this chapter, we use assembly code as the target language for readability. However, we should emphasize that as long as addresses can be calculated from the 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.

Memory Management

Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator. In the last chapter, we assumed that a name in a three-address statement refers to a symbol-table entry for the name. In Section 8.2, symbol-table entries were created as the declarations in a procedure were examined. The type in a declaration determines the width, i.e., the amount of storage, needed for the declared name. From the symbol-table information, a relative address can be determined for the name in a data area for the procedure. In Section 9.3, we outline implementations of static and stack allocation of data areas, and show how names in the intermediate representation can be converted into addresses in the target code.

If machine code is being generated, labels in three-address statements have to be converted to addresses of instructions. This process is analogous to the "backpatching" technique in Section 8.6. Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as *j*: `goto` *i* is encountered, and *i* is less than *j*, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple *i*. If, however, the jump is forward, so *i* exceeds *j*, we must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. Then, when we process quadruple *i*, we fill in the proper machine location for all instructions that are forward jumps to *i*.

Instruction Selection

The nature of the instruction set of the target machine determines the difficulty of instruction selection. 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.

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 outlines 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

```
MOV y,R0   /* load y into register R0 */
ADD z,R0   /* add z to R0 */
MOV R0,x   /* store R0 into x */
```

Unfortunately, this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements

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

would be translated into

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

Here the fourth statement is redundant, and so is the third if `a` is not subsequently used.

The quality of the generated code is determined by its speed and size. A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may 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 in `a`:

```
MOV   a, R0
ADD   #1, R0
MOV   R0, a
```

Instruction speeds are needed to design good code sequences but, unfortunately, accurate timing 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. Tools for constructing instruction selectors are discussed in Section 9.12.

Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code. The use of registers is often subdivided into two subproblems:

1. During *register allocation*, we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent *register assignment* phase, 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 values. 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.

Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, in the IBM System/370 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 even register of an even/odd register pair. The multiplicand value is taken from the odd register of the pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form

```
D    x, y
```

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in Fig. 9.2(a) and (b), in which the only difference is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 9.3.

R*i* stands for register *i*. (SRDA[1] 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

[1] Shift Right Double Arithmetic.

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

   (a)                     (b)
```

Fig. 9.2. 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)
```

Fig. 9.3. Optimal machine-code sequences.

a is to be loaded depends on what will ultimately happen to t. Strategies for register allocation are discussed in Section 9.7.

Choice of Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others, as we shall see. Picking a best order is another 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.

Approaches to Code Generation

Undoubtedly 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.

Section 9.6 contains a straightforward code-generation algorithm that uses information about subsequent uses of an operand to generate code for a register machine. It considers each statement in turn, keeping operands in registers as long as possible. The output of such a code generator can be improved by peephole optimization techniques such as those discussed in Section 9.9.

Section 9.7 presents techniques to make better use of registers by considering the flow of control in the intermediate code. The emphasis is on