

Chapter 6

Intermediate-Code Generation

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j . This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends.

This chapter deals with intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as in Fig. 6.1, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. We shall use the syntax-directed formalisms of Chapters 2 and 5 to specify checking and translation. Many of the translation schemes can be implemented during either bottom-up or top-down parsing, using the techniques of Chapter 5. All schemes can be implemented by creating a syntax tree and then walking the tree.

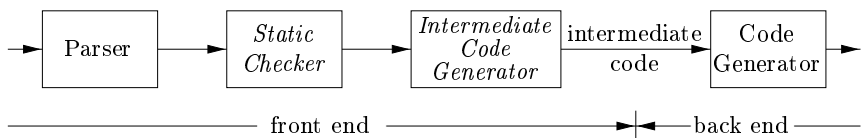


Figure 6.1: Logical structure of a compiler front end

Static checking includes *type checking*, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain

after parsing. For example, static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.

The approach in this chapter can be used for a wide range of intermediate representations, including syntax trees and three-address code, both of which were introduced in Section 2.8. The term “three-address code” comes from instructions of the general form $x = y \text{ op } z$ with three addresses: two for the operands y and z and one for the result x .

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

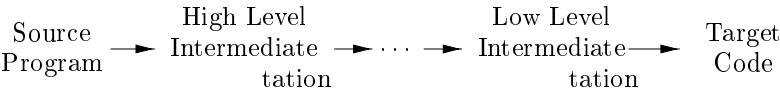


Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. For looping statements, for example, a syntax tree contains the components of a statement, whereas three-address code contains labels and jump instructions to control the flow of control, as in machine language.

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler. C is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available. The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end.

6.1 Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a *DAG*) for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression. As we shall see in this section, DAG’s can be constructed by using the same techniques that construct syntax trees.

6.1.1 Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

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

The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression $b - c$ are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions $a * (b - c)$ and $(b - c) * d$. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b - c$. $\square$

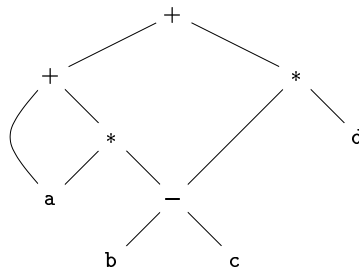


Figure 6.3: Dag for the expression $a + a * (b - c) + (b - c) * d$

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, *Node*(*op*, *left*, *right*), we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new} \text{ Node}(' + ', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new} \text{ Node}(' - ', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new} \text{ Leaf}(\mathbf{id}, \mathbf{id}.entry)$
6) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new} \text{ Leaf}(\mathbf{num}, \mathbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

- 1) $p_1 = \text{Leaf}(\mathbf{id}, \text{entry-}a)$
- 2) $p_2 = \text{Leaf}(\mathbf{id}, \text{entry-}a) = p_1$
- 3) $p_3 = \text{Leaf}(\mathbf{id}, \text{entry-}b)$
- 4) $p_4 = \text{Leaf}(\mathbf{id}, \text{entry-}c)$
- 5) $p_5 = \text{Node}(' - ', p_3, p_4)$
- 6) $p_6 = \text{Node}(' * ', p_1, p_5)$
- 7) $p_7 = \text{Node}(' + ', p_1, p_6)$
- 8) $p_8 = \text{Leaf}(\mathbf{id}, \text{entry-}b) = p_3$
- 9) $p_9 = \text{Leaf}(\mathbf{id}, \text{entry-}c) = p_4$
- 10) $p_{10} = \text{Node}(' - ', p_3, p_4) = p_5$
- 11) $p_{11} = \text{Leaf}(\mathbf{id}, \text{entry-}d)$
- 12) $p_{12} = \text{Node}(' * ', p_5, p_{11})$
- 13) $p_{13} = \text{Node}(' + ', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

discussed above. We assume that *entry-a* points to the symbol-table entry for *a*, and similarly for the other identifiers.

When the call to *Leaf(id, entry-a)* is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same as that returned at step 5; i.e., $p_{10} = p_5$. $\square$

6.1.2 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and

interior nodes have two additional fields indicating the left and right children.

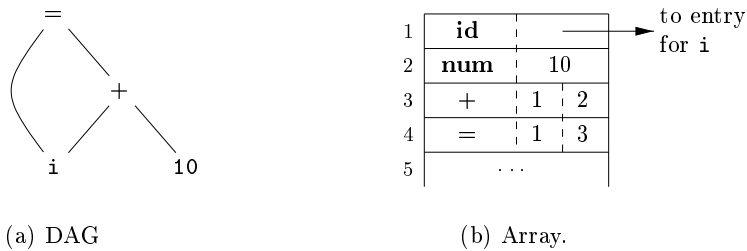


Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its “value number.” If stored in an appropriate data structure, value numbers help us construct expression DAG’s efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple $\langle op, l, r \rangle$, where *op* is the label, *l* its left child’s value number, and *r* its right child’s value number. A unary operator may be assumed to have $r = 0$.

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label *op*, node *l*, and node *r*.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: Search the array for a node *M* with label *op*, left child *l*, and right child *r*. If there is such a node, return the value number of *M*. If not, create in the array a new node *N* with label *op*, left child *l*, and right child *r*, and return its value number. □

While Algorithm 6.3 yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especially if the array holds expressions from an entire program. A more efficient approach is to use a hash table, in which the nodes are put into “buckets,” each of which typically will have only a few nodes. The hash table is one of several data structures that support *dictionaries* efficiently.¹ A dictionary is an abstract data type that

¹See Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a discussion of data structures supporting dictionaries.

allows us to insert and delete elements of a set, and to determine whether a given element is currently in the set. A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

To construct a hash table for the nodes of a DAG, we need a *hash function* h that computes the index of the bucket for a signature $\langle op, l, r \rangle$, in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes. The bucket index $h(op, l, r)$ is computed deterministically from op , l , and r , so that we may repeat the calculation and always get to the same bucket index for node $\langle op, l, r \rangle$.

The buckets can be implemented as linked lists, as in Fig. 6.7. An array, indexed by hash value, holds the *bucket headers*, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node $\langle op, l, r \rangle$ can be found on the list whose header is at index $h(op, l, r)$ of the array.

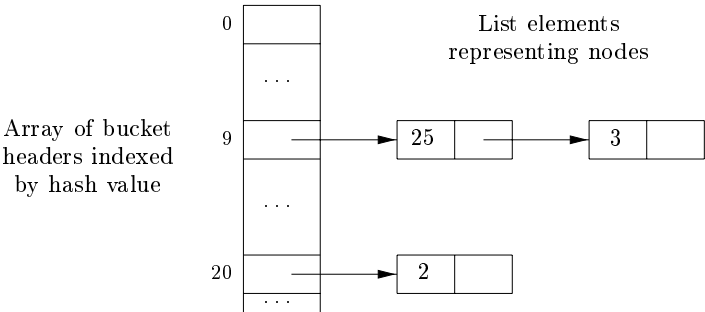


Figure 6.7: Data structure for searching buckets

Thus, given the input node op , l , and r , we compute the bucket index $h(op, l, r)$ and search the list of cells in this bucket for the given input node. Typically, there are enough buckets so that no list has more than a few cells. We may need to look at all the cells within a bucket, however, and for each value number v found in a cell, we must check whether the signature $\langle op, l, r \rangle$ of the input node matches the node with value number v in the list of cells (as in Fig. 6.7). If we find a match, we return v . If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index $h(op, l, r)$, and return the value number in that new cell.

6.1.3 Exercises for Section 6.1

Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

Exercise 6.1.2: Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

- a) $a + b + (a + b)$.
- b) $a + b + a + b$.
- c) $a + a + (a + a + a + (a + a + a + a))$.

6.2 Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like $x+y*z$ might be translated into the sequence of three-address instructions

$$\begin{aligned}t_1 &= y * z \\t_2 &= x + t_1\end{aligned}$$

where t_1 and t_2 are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization, as discussed in Chapters 8 and 9. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence. $\square$

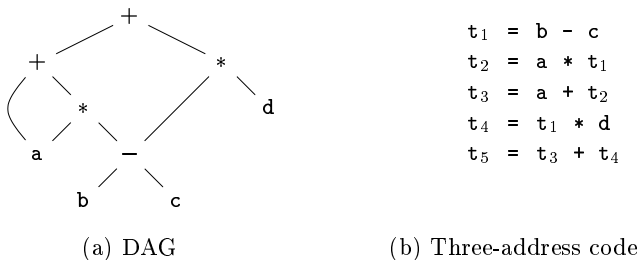


Figure 6.8: A DAG and its corresponding three-address code

6.2.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in Section 6.2.2.

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in Section 6.5.2.
- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

We now consider the common three-address instructions used in the rest of this book. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by “backpatching,” discussed in Section 6.7. Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form $x = y \text{ op } z$, where *op* is a binary arithmetic or logical operation, and x , y , and z are addresses.
2. Assignments of the form $x = \text{op } y$, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.
3. *Copy instructions* of the form $x = y$, where x is assigned the value of y .
4. An unconditional jump `goto  $L$` . The three-address instruction with label L is the next to be executed.
5. Conditional jumps of the form `if  $x$  goto  $L$` and `ifFalse  $x$  goto  $L$` . These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if  $x$  relop  $y$  goto  $L$` , which apply a relational operator (`<`, `=`, `>`, etc.) to x and y , and execute the instruction with label L next if x stands in relation *relop* to y . If not, the three-address instruction following `if  $x$  relop  $y$  goto  $L$` is executed next, in sequence.
7. Procedure calls and returns are implemented using the following instructions: `param  $x$` for parameters; `call  $p, n$` and `$y$  = call  $p, n$` for procedure and function calls, respectively; and `return  $y$` , where y , representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```

param  $x_1$ 
param  $x_2$ 
...
param  $x_n$ 
call  $p, n$ 

```

generated as part of a call of the procedure $p(x_1, x_2, \dots, x_n)$. The integer n , indicating the number of actual parameters in “call p, n ,” is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form `$x$  =  $y[i]$` and `$x[i]$  =  $y$` . The instruction `$x$  =  $y[i]$` sets x to the value in the location i memory units beyond location y . The instruction `$x[i]$  =  $y$` sets the contents of the location i units beyond x to the value of y .
9. Address and pointer assignments of the form `$x$  = & $y$` , `$x$  = * $y$` , and `* $x$  =  $y$` . The instruction `$x$  = & $y$` sets the r -value of x to be the location (l -value) of y .² Presumably y is a name, perhaps a temporary, that denotes an expression with an l -value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction `$x$  = * $y$` , presumably y is a pointer or a temporary whose r -value is a location. The r -value of x is made equal to the contents of that location. Finally, `* $x$  =  $y$` sets the r -value of the object pointed to by x to the r -value of y .

Example 6.5: Consider the statement

```
do i = i+1; while (a[i] < v);
```

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9(a) uses a symbolic label L , attached to the first instruction.

²From Section 2.8.3, l - and r -values are appropriate on the left and right sides of assignments, respectively.

The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication $i * 8$ is appropriate for an array of elements that each take 8 units of space. $\square$

L:	$t_1 = i + 1$	100:	$t_1 = i + 1$
	$i = t_1$	101:	$i = t_1$
	$t_2 = i * 8$	102:	$t_2 = i * 8$
	$t_3 = a [t_2]$	103:	$t_3 = a [t_2]$
	if $t_3 < v$ goto L	104:	if $t_3 < v$ goto 100

(a) Symbolic labels.

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

6.2.2 **Quadruples**

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure. In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called “quadruples,” “triples,” and “indirect triples.”

A *quadruple* (or just “*quad*”) has four fields, which we call *op*, *arg*₁, *arg*₂, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing $+$ in *op*, y in *arg*₁, z in *arg*₂, and x in *result*. The following are some exceptions to this rule:

- 1. Instructions with unary operators like $x = \text{minus } y$ or $x = y$ do not use *arg*₂. Note that for a copy statement like $x = y$, *op* is $=$, while for most other operations, the assignment operator is implied.
- 2. Operators like *param* use neither *arg*₂ nor *result*.
- 3. Conditional and unconditional jumps put the target label in *result*.

Example 6.6: Three-address code for the assignment $a = b * -c + b * -c$; appears in Fig. 6.10(a). The special operator *minus* is used to distinguish the

unary minus operator, as in `-c`, from the binary minus operator, as in `b - c`. Note that the unary-minus “three-address” statement has only two addresses, as does the copy statement `a = t5`.

The quadruples in Fig. 6.10(b) implement the three-address code in (a). □

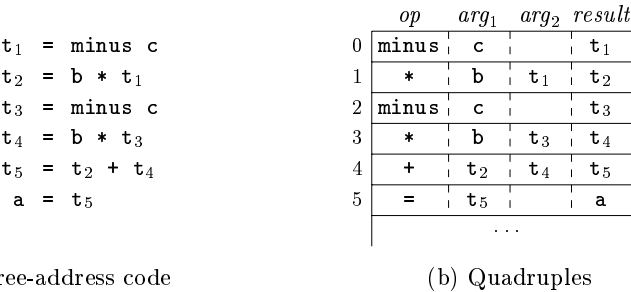


Figure 6.10: Three-address code and its quadruple representation

For readability, we use actual identifiers like `a`, `b`, and `c` in the fields `arg1`, `arg2`, and `result` in Fig. 6.10(b), instead of pointers to their symbol-table entries. Temporary names can either be entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class *Temp* with its own methods.

6.2.3 Triples

A *triple* has only three fields, which we call `op`, `arg1`, and `arg2`. Note that the `result` field in Fig. 6.10(b) is used primarily for temporary names. Using triples, we refer to the result of an operation `x op y` by its position, rather than by an explicit temporary name. Thus, instead of the temporary `t1` in Fig. 6.10(b), a triple representation would refer to position (0). Parenthesized numbers `t` pointers into the triple structure itself. In Section 6.1.2, positions or pointers to positions were called value numbers.

Triples are equivalent to signatures in Algorithm 6.3. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

Example 6.7: The syntax tree and triples in Fig. 6.11 correspond to the three-address code and quadruples in Fig. 6.10. In the triple representation in Fig. 6.11(b), the copy statement `a = t5` is encoded in the triple representation by placing `a` in the `arg1` field and (4) in the `arg2` field. □

A ternary operation like `x[i] = y` requires two entries in the triple structure; for example, we can put `x` and `i` in one triple and `y` in the next. Similarly, `x = y[i]` can implemented by treating it as if it were the two instructions

Why Do We Need Copy Instructions?

A simple algorithm for translating expressions generates copy instructions for assignments, as in Fig. 6.10(a), where we copy t_5 into a rather than assigning $t_2 + t_4$ to a directly. Each subexpression typically gets its own, new temporary to hold its result, and only when the assignment operator $=$ is processed do we learn where to put the value of the complete expression. A code-optimization pass, perhaps using the DAG of Section 6.1.1 as an intermediate form, can discover that t_5 can be replaced by a .

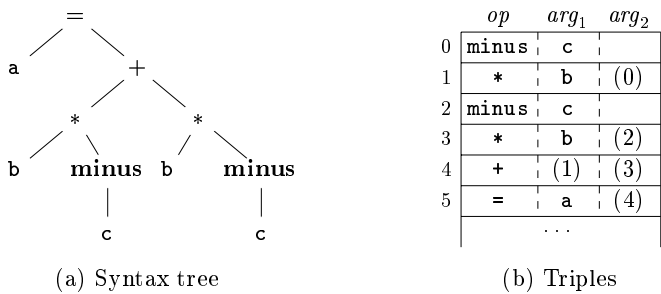


Figure 6.11: Representations of $a = b * - c + b * - c$;

$t = y[i]$ and $x = t$, where t is a compiler-generated temporary. Note that the temporary t does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t , then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples, which we consider next.

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array *instruction* to list pointers to triples in the desired order. Then, the triples in Fig. 6.11(b) might be represented as in Fig. 6.12.

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

instruction		op	arg ₁	arg ₂
35	(0)	minus	c	
36	(1)	*	b	(0)
37	(2)	minus	c	
38	(3)	*	b	(2)
39	(4)	+	(1)	(3)
40	(5)	=	a	(4)
	...			

Figure 6.12: Indirect triples representation of three-address code

6.2.4 Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term *static single-assignment*. Figure 6.13 shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables *p* and *q* in the SSA representation.

p = a + b	p ₁ = a + b
q = p - c	q ₁ = p ₁ - c
p = q * d	p ₂ = q ₁ * d
p = e - p	p ₃ = e - p ₂
q = p + q	q ₂ = p ₃ + q ₁

(a) Three-address code. (b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

has two control-flow paths in which the variable *x* gets defined. If we use different names for *x* in the true part and the false part of the conditional statement, then which name should we use in the assignment *y = x * a*? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the *φ*-function to combine the two definitions of *x*:

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

Here, $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional and the value x_2 if the control flow passes through the false part. That is to say, the ϕ -function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ -function.

6.2.5 Exercises for Section 6.2

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

- a) A syntax tree.
- b) Quadruples.
- c) Triples.
- d) Indirect triples.

Exercise 6.2.2: Repeat Exercise 6.2.1 for the following assignment statements:

- i. $a = b[i] + c[j]$.
- ii. $a[i] = b * c - b * d$.
- iii. $x = f(y+1) + 2$.
- iv. $x = *p + \&y$.

! Exercise 6.2.3: Show how to transform a three-address code sequence into one in which each defined variable gets a unique variable name.

6.3 Types and Declarations

The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the `&&` operator in Java expects its two operands to be booleans; the result is also of type boolean.
- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

In this section, we examine types and storage layout for names declared within a procedure or a class. The actual storage for a procedure call or an object is allocated at run time, when the procedure is called or the object is created. As we examine local declarations at compile time, we can, however, lay out *relative addresses*, where the relative address of a name or a component of a data structure is an offset from the start of a data area.

6.3.1 Type Expressions

Types have structure, which we shall represent using *type expressions*: a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.

Example 6.8: The array type `int[2][3]` can be read as “array of 2 arrays of 3 integers each” and written as a type expression `array(2, array(3, integer))`. This type is represented by the tree in Fig. 6.14. The operator *array* takes two parameters, a number and a type. $\square$

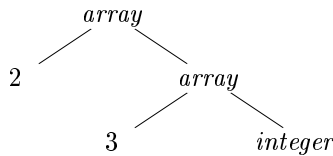


Figure 6.14: Type expression for `int[2][3]`

We shall use the following definition of type expressions:

- A basic type is a type expression. Typical basic types for a language include *boolean*, *char*, *integer*, *float*, and *void*; the latter denotes “the absence of a value.”
- A type name is a type expression.
- A type expression can be formed by applying the *array* type constructor to a number and a type expression.
- A record is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types. Record types will be implemented in Section 6.3.6 by applying the constructor *record* to a symbol table containing entries for the fields.
- A type expression can be formed by using the type constructor $\rightarrow$ for function types. We write $s \rightarrow t$ for “function from type s to type t .” Function types will be useful when type checking is discussed in Section 6.5.

Type Names and Recursive Types

Once a class is defined, its name can be used as a type name in C++ or Java; for example, consider `Node` in the program fragment

```
public class Node { ... }
...
public Node n;
```

Names can be used to define recursive types, which are needed for data structures such as linked lists. The pseudocode for a list element

```
class Cell { int info; Cell next; ... }
```

defines the recursive type `Cell` as a class that contains a field `info` and a field `next` of type `Cell`. Similar recursive types can be defined in C using records and pointers. The techniques in this chapter carry over to recursive types.

- If s and t are type expressions, then their Cartesian product $s \times t$ is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$.
- Type expressions may contain variables whose values are type expressions. Compiler-generated type variables will be used in Section 6.5.4.

A convenient way to represent a type expression is to use a graph. The value-number method of Section 6.1.2, can be adapted to construct a dag for a type expression, with interior nodes for type constructors and leaves for basic types, type names, and type variables; for example, see the tree in Fig. 6.14.³

6.3.2 Type Equivalence

When are two type expressions equivalent? Many type-checking rules have the form, “**if** two type expressions are equal **then** return a certain type **else** error.” Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions. The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

³Since type names denote type expressions, they can set up implicit cycles; see the box on “Type Names and Recursive Types.” If edges to type names are redirected to the type expressions denoted by the names, then the resulting graph can have cycles due to recursive types.

When type expressions are represented by graphs, two types are *structurally equivalent* if and only if one of the following conditions is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to *name equivalence* of type expressions.

Name-equivalent expressions are assigned the same value number, if we use Algorithm 6.3. Structural equivalence can be tested using the unification algorithm in Section 6.5.5.

6.3.3 Declarations

We shall study types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can be handled as discussed in Example 5.10. The grammar is

$$\begin{aligned}
 D &\rightarrow T \text{ id} ; D \mid \epsilon \\
 T &\rightarrow B C \mid \text{record } \{ ' D ' \} \\
 B &\rightarrow \text{int} \mid \text{float} \\
 C &\rightarrow \epsilon \mid [\text{num}] C
 \end{aligned}$$

The fragment of the above grammar that deals with basic and array types was used to illustrate inherited attributes in Section 5.3.2. The difference in this section is that we consider storage layout as well as types.

Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types. Nonterminal B generates one of the basic types **int** and **float**. Nonterminal C , for “component,” generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B , followed by array components specified by nonterminal C . A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

6.3.4 Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data. Run-time storage management is discussed in Chapter 7.

Address Alignment

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine. For example, instructions to add integers may expect integers to be *aligned*, that is, placed at certain positions in memory such as an address divisible by 4. Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes — the next multiple of 4 — leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as *padding*. When space is at a premium, a compiler may *pack* data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. Typically, a byte is eight bits, and some number of bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of the first byte.

The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.⁴

The translation scheme (SDT) in Fig. 6.15 computes types and their widths for basic and array types; record types will be discussed in Section 6.3.6. The SDT uses synthesized attributes *type* and *width* for each nonterminal and two variables *t* and *w* to pass type and width information from a *B* node in a parse tree to the node for the production $C \rightarrow \epsilon$. In a syntax-directed definition, *t* and *w* would be inherited attributes for *C*.

The body of the *T*-production consists of nonterminal *B*, an action, and nonterminal *C*, which appears on the next line. The action between *B* and *C* sets *t* to *B.type* and *w* to *B.width*. If $B \rightarrow \mathbf{int}$ then *B.type* is set to *integer* and *B.width* is set to 4, the width of an integer. Similarly, if $B \rightarrow \mathbf{float}$ then *B.type* is *float* and *B.width* is 8, the width of a float.

The productions for *C* determine whether *T* generates a basic type or an array type. If $C \rightarrow \epsilon$, then *t* becomes *C.type* and *w* becomes *C.width*.

Otherwise, *C* specifies an array component. The action for $C \rightarrow [\mathbf{num}] C_1$ forms *C.type* by applying the type constructor *array* to the operands *num.value* and *C₁.type*. For instance, the result of applying *array* might be a tree structure such as Fig. 6.14.

⁴Storage allocation for pointers in C and C++ is simpler if all pointers have the same width. The reason is that the storage for a pointer may need to be allocated before we learn the type of the objects it can point to.

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
C	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \mathbf{int}$	$\{ B.type = integer; B.width = 4; \}$
$B \rightarrow \mathbf{float}$	$\{ B.type = float; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\mathbf{num}] C_1$	$\{ C.type = array(\mathbf{num.value}, C_1.type);$ $C.width = \mathbf{num.value} \times C_1.width; \}$

Figure 6.15: Computing types and their widths

The width of an array is obtained by multiplying the width of an element by the number of elements in the array. If addresses of consecutive integers differ by 4, then address calculations for an array of integers will include multiplications by 4. Such multiplications provide opportunities for optimization, so it is helpful for the front end to make them explicit. In this chapter, we ignore other machine dependencies such as the alignment of data objects on word boundaries.

Example 6.9: The parse tree for the type `int[2][3]` is shown by dotted lines in Fig. 6.16. The solid lines show how the type and width are passed from B , down the chain of C 's through variables t and w , and then back up the chain as synthesized attributes $type$ and $width$. The variables t and w are assigned the values of $B.type$ and $B.width$, respectively, before the subtree with the C nodes is examined. The values of t and w are used at the node for $C \rightarrow \epsilon$ to start the evaluation of the synthesized attributes up the chain of C nodes. $\square$

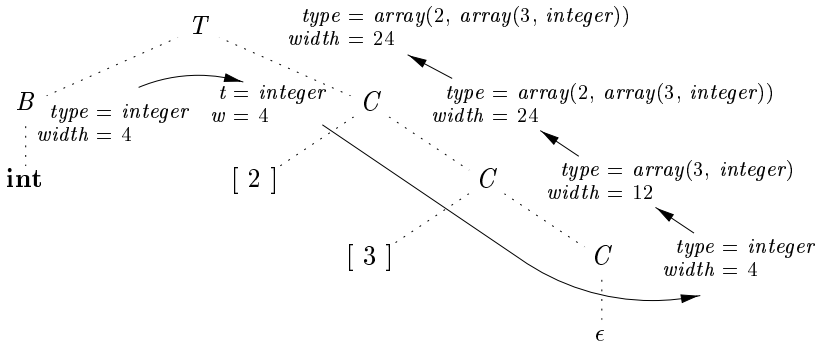


Figure 6.16: Syntax-directed translation of array types

6.3.5 Sequences of Declarations

Languages such as C and Java allow all the declarations in a single procedure to be processed as a group. The declarations may be distributed within a Java procedure, but they can still be processed when the procedure is analyzed. Therefore, we can use a variable, say *offset*, to keep track of the next available relative address.

The translation scheme of Fig. 6.17 deals with a sequence of declarations of the form $T \text{ id}$, where T generates a type as in Fig. 6.15. Before the first declaration is considered, *offset* is set to 0. As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of *offset*, which is then incremented by the width of the type of x .

$$\begin{array}{ll}
 P \rightarrow & \{ \text{offset} = 0; \} \\
 & D \\
 D \rightarrow T \text{ id} ; & \{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset}); \\
 & \text{offset} = \text{offset} + T.\text{width}; \} \\
 & D_1 \\
 D \rightarrow & \epsilon
 \end{array}$$

Figure 6.17: Computing the relative addresses of declared names

The semantic action within the production $D \rightarrow T \text{ id} ; D_1$ creates a symbol-table entry by executing $\text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset})$. Here *top* denotes the current symbol table. The method *top.put* creates a symbol-table entry for *id.lexeme*, with type *T.type* and relative address *offset* in its data area.

The initialization of *offset* in Fig. 6.17 is more evident if the first production appears on one line as:

$$P \rightarrow \{ \text{offset} = 0; \} D \quad (6.1)$$

Nonterminals generating ϵ , called marker nonterminals, can be used to rewrite productions so that all actions appear at the ends of right sides; see Section 5.5.4. Using a marker nonterminal M , (6.1) can be restated as:

$$\begin{array}{ll}
 P \rightarrow & M D \\
 M \rightarrow & \epsilon \quad \{ \text{offset} = 0; \}
 \end{array}$$

6.3.6 Fields in Records and Classes

The translation of declarations in Fig. 6.17 carries over to fields in records and classes. Record types can be added to the grammar in Fig. 6.15 by adding the following production

$$T \rightarrow \text{record } \{ ' D ' \}$$

The fields in this record type are specified by the sequence of declarations generated by D . The approach of Fig. 6.17 can be used to determine the types and relative addresses of fields, provided we are careful about two things:

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D .
- The offset or relative address for a field name is relative to the data area for that record.

Example 6.10: The use of a name x for a field within a record does not conflict with other uses of the name outside the record. Thus, the three uses of x in the following declarations are distinct and do not conflict with each other:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

A subsequent assignment $x = p.x + q.x$; sets variable x to the sum of the fields named x in the records p and q . Note that the relative address of x in p differs from the relative address of x in q . $\square$

For convenience, record types will encode both the types and relative addresses of their fields, using a symbol table for the record type. A record type has the form $record(t)$, where $record$ is the type constructor, and t is a symbol-table object that holds information about the fields of this record type.

The translation scheme in Fig. 6.18 consists of a single production to be added to the productions for T in Fig. 6.15. This production has two semantic actions. The embedded action before D saves the existing symbol table, denoted by top and sets top to a fresh symbol table. It also saves the current $offset$, and sets $offset$ to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using top , before restoring the saved symbol table and offset.

$$\begin{array}{ll}
 T \rightarrow \mathbf{record} \text{ '}' & \{ Env.push(top); top = \mathbf{new} Env(); \\
 & \quad Stack.push(offset); offset = 0; \} \\
 D \text{ '}' & \{ T.type = record(top); T.width = offset; \\
 & \quad top = Env.pop(); offset = Stack.pop(); \}
 \end{array}$$

Figure 6.18: Handling of field names in records

For concreteness, the actions in Fig. 6.18 give pseudocode for a specific implementation. Let class *Env* implement symbol tables. The call *Env.push(top)* pushes the current symbol table denoted by top onto a stack. Variable top is then set to a new symbol table. Similarly, $offset$ is pushed onto a stack called *Stack*. Variable $offset$ is then set to 0.

After the declarations in D have been translated, the symbol table top holds the types and relative addresses of the fields in this record. Further, $offset$ gives the storage needed for all the fields. The second action sets $T.type$ to $record(top)$ and $T.width$ to $offset$. Variables top and $offset$ are then restored to their pushed values to complete the translation of this record type.

This discussion of storage for record types carries over to classes, since no storage is reserved for methods. See Exercise 6.3.2.

6.3.7 Exercises for Section 6.3

Exercise 6.3.1: Determine the types and relative addresses for the identifiers in the following sequence of declarations:

```
float x;  
record { float x; float y; } p;  
record { int tag; float x; float y; } q;
```

! Exercise 6.3.2: Extend the handling of field names in Fig. 6.18 to classes and single-inheritance class hierarchies.

- a) Give an implementation of class *Env* that allows linked symbol tables, so that a subclass can either redefine a field name or refer directly to a field name in a superclass.
- b) Give a translation scheme that allocates a contiguous data area for the fields in a class, including inherited fields. Inherited fields must maintain the relative addresses they were assigned in the layout for the superclass.

6.6 Control Flow

The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions. In programming languages, boolean expressions are often used to

1. *Alter the flow of control.* Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in **if** (E) S , the expression E must be true if statement S is reached.
2. *Compute logical values.* A boolean expression can represent *true* or *false* as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

The intended use of boolean expressions is determined by its syntactic context. For example, an expression following the keyword **if** is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value. Such syntactic contexts can be specified in a number of ways: we may use two different nonterminals, use inherited attributes, or set a flag during parsing. Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

This section concentrates on the use of boolean expressions to alter the flow of control. For clarity, we introduce a new nonterminal B for this purpose. In Section 6.6.6, we consider how a compiler can allow boolean expressions to represent logical values.

6.6.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote $\&\&$, $||$, and $!$, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **rel** E_2 , where E_1 and

E_2 are arithmetic expressions. In this section, we consider boolean expressions generated by the following grammar:

$$B \rightarrow B \mid\mid B \mid B \&\& B \mid ! B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

We use the attribute **rel.op** to indicate which of the six comparison operators $<$, $<=$, $=$, $!=$, $>$, or $>=$ is represented by **rel**. As is customary, we assume that $\mid\mid$ and $\&\&$ are left-associative, and that $\mid\mid$ has lowest precedence, then $\&\&$, then $!$.

Given the expression $B_1 \mid\mid B_2$, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2 . Similarly, given $B_1 \&\& B_2$, if B_1 is false, then the entire expression is false.

The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated. If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as $B_1 \mid\mid B_2$, neither B_1 nor B_2 is necessarily evaluated fully. If either B_1 or B_2 is an expression with side effects (e.g., it contains a function that changes a global variable), then an unexpected answer may be obtained.

6.6.2 Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators $\&\&$, $\mid\mid$, and $!$ translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21 : The statement

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

might be translated into the code of Fig. 6.34. In this translation, the boolean expression is true if control reaches label L_2 . If the expression is false, control goes immediately to L_1 , skipping L_2 and the assignment $x = 0$. $\square$

```

        if x < 100 goto L2
        iffFalse x > 200 goto L1
        iffFalse x != y goto L1
L2:    x = 0
L1:
```

Figure 6.34: Jumping code

6.6.3 Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

$$\begin{aligned} S &\rightarrow \text{if } (B) S_1 \\ S &\rightarrow \text{if } (B) S_1 \text{ else } S_2 \\ S &\rightarrow \text{while } (B) S_1 \end{aligned}$$

In these productions, nonterminal B represents a boolean expression and non-terminal S represents a statement.

This grammar generalizes the running example of while expressions that we introduced in Example 5.19. As in that example, both B and S have a synthesized attribute *code*, which gives the translation into three-address instructions. For simplicity, we build up the translations $B.code$ and $S.code$ as strings, using syntax-directed definitions. The semantic rules defining the *code* attributes could be implemented instead by building up syntax trees and then emitting code during a tree traversal, or by any of the approaches outlined in Section 5.5.

The translation of $\text{if } (B) S_1$ consists of $B.code$ followed by $S_1.code$, as illustrated in Fig. 6.35(a). Within $B.code$ are jumps based on the value of B . If B is true, control flows to the first instruction of $S_1.code$, and if B is false, control flows to the instruction immediately following $S_1.code$.

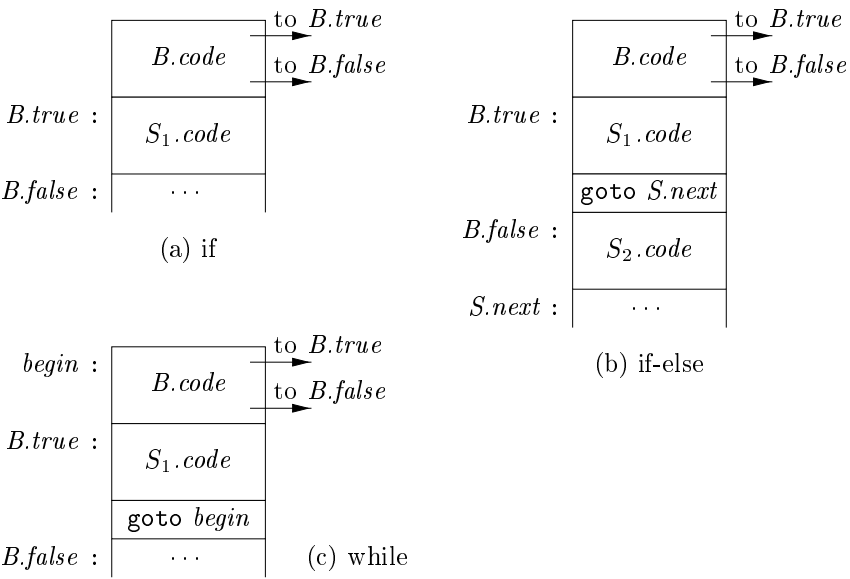


Figure 6.35: Code for if-, if-else-, and while-statements

The labels for the jumps in $B.code$ and $S.code$ are managed using inherited attributes. With a boolean expression B , we associate two labels: $B.true$, the

label to which control flows if B is true, and $B.false$, the label to which control flows if B is false. With a statement S , we associate an inherited attribute $S.next$ denoting a label for the instruction immediately after the code for S . In some cases, the instruction immediately following $S.code$ is a jump to some label L . A jump to a jump to L from within $S.code$ is avoided using $S.next$.

The syntax-directed definition in Fig. 6.36-6.37 produces three-address code for boolean expressions in the context of if-, if-else-, and while-statements.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

We assume that $newlabel()$ creates a new label each time it is called, and that $label(L)$ attaches label L to the next three-address instruction to be generated.⁸

⁸If implemented literally, the semantic rules will generate lots of labels and may attach more than one label to a three-address instruction. The backpatching approach of Section 6.7

A program consists of a statement generated by $P \rightarrow S$. The semantic rules associated with this production initialize $S.next$ to a new label. $P.code$ consists of $S.code$ followed by the new label $S.next$. Token **assign** in the production $S \rightarrow \mathbf{assign}$ is a placeholder for assignment statements. The translation of assignments is as discussed in Section 6.4; for this discussion of control flow, $S.code$ is simply **assign.code**.

In translating $S \rightarrow \mathbf{if} (B) S_1$, the semantic rules in Fig. 6.36 create a new label $B.true$ and attach it to the first three-address instruction generated for the statement S_1 , as illustrated in Fig. 6.35(a). Thus, jumps to $B.true$ within the code for B will go to the code for S_1 . Further, by setting $B.false$ to $S.next$, we ensure that control will skip the code for S_1 if B evaluates to false.

In translating the if-else-statement $S \rightarrow \mathbf{if} (B) S_1 \mathbf{else} S_2$, the code for the boolean expression B has jumps out of it to the first instruction of the code for S_1 if B is true, and to the first instruction of the code for S_2 if B is false, as illustrated in Fig. 6.35(b). Further, control flows from both S_1 and S_2 to the three-address instruction immediately following the code for S — its label is given by the inherited attribute $S.next$. An explicit **goto** $S.next$ appears after the code for S_1 to skip over the code for S_2 . No **goto** is needed after S_2 , since $S_2.next$ is the same as $S.next$.

The code for $S \rightarrow \mathbf{while} (B) S_1$ is formed from $B.code$ and $S_1.code$ as shown in Fig. 6.35(c). We use a local variable *begin* to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for B . We use a variable rather than an attribute, because *begin* is local to the semantic rules for this production. The inherited label $S.next$ marks the instruction that control must flow to if B is false; hence, $B.false$ is set to be $S.next$. A new label $B.true$ is attached to the first instruction for S_1 ; the code for B generates a jump to this label if B is true. After the code for S_1 we place the instruction **goto** *begin*, which causes a jump back to the beginning of the code for the boolean expression. Note that $S_1.next$ is set to this label *begin*, so jumps from within $S_1.code$ can go directly to *begin*.

The code for $S \rightarrow S_1 S_2$ consists of the code for S_1 followed by the code for S_2 . The semantic rules manage the labels; the first instruction after the code for S_1 is the beginning of the code for S_2 ; and the instruction after the code for S_2 is also the instruction after the code for S .

We discuss the translation of flow-of-control statements further in Section 6.7. There we shall see an alternative method, called “backpatching,” which emits code for statements in one pass.

6.6.4 Control-Flow Translation of Boolean Expressions

The semantic rules for boolean expressions in Fig. 6.37 complement the semantic rules for statements in Fig. 6.36. As in the code layout of Fig. 6.35, a boolean expression B is translated into three-address instructions that evaluate B using

creates labels only when they are needed. Alternatively, unnecessary labels can be eliminated during a subsequent optimization phase.

conditional and unconditional jumps to one of two labels: $B.true$ if B is true, and $B.false$ if B is false.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \ \ B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \ label(B_1.false) \ \ B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \ label(B_1.true) \ \ B_2.code$
$B \rightarrow ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \ E_2.code$ $\quad \ \ gen('if' \ E_1.addr \ \mathbf{rel.op} \ E_2.addr \ 'goto' \ B.true)$ $\quad \ \ gen('goto' \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('goto' \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('goto' \ B.false)$

Figure 6.37: Generating three-address code for booleans

The fourth production in Fig. 6.37, $B \rightarrow E_1 \ \mathbf{rel} \ E_2$, is translated directly into a comparison three-address instruction with jumps to the appropriate places. For instance, B of the form $a < b$ translates into:

```

if a < b goto B.true
goto B.false

```

The remaining productions for B are translated as follows:

1. Suppose B is of the form $B_1 \ || \ B_2$. If B_1 is true, then we immediately know that B itself is true, so $B_1.true$ is the same as $B.true$. If B_1 is false, then B_2 must be evaluated, so we make $B_1.false$ be the label of the first instruction in the code for B_2 . The true and false exits of B_2 are the same as the true and false exits of B , respectively.

2. The translation of $B_1 \ \&\& \ B_2$ is similar.
3. No code is needed for an expression B of the form $!B_1$: just interchange the true and false exits of B to get the true and false exits of B_1 .
4. The constants **true** and **false** translate into jumps to $B.true$ and $B.false$, respectively.

Example 6.22: Consider again the following statement from Example 6.21:

`if( x < 100 || x > 200 && x != y ) x = 0;` (6.13)

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```

        if x < 100 goto L2
        goto L3
L3:    if x > 200 goto L4
        goto L1
L4:    if x != y goto L2
        goto L1
L2:    x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

The statement (6.13) constitutes a program generated by $P \rightarrow S$ from Fig. 6.36. The semantic rules for the production generate a new label L_1 for the instruction after the code for S . Statement S has the form **if** (B) S_1 , where S_1 is $x = 0$; , so the rules in Fig. 6.36 generate a new label L_2 and attach it to the first (and only, in this case) instruction in $S_1.code$, which is $x = 0$.

Since $||$ has lower precedence than $\&\&$, the boolean expression in (6.13) has the form $B_1 || B_2$, where B_1 is $x < 100$. Following the rules in Fig. 6.37, $B_1.true$ is L_2 , the label of the assignment $x = 0$; . $B_1.false$ is a new label L_3 , attached to the first instruction in the code for B_2 .

Note that the code generated is not optimal, in that the translation has three more instructions (goto's) than the code in Example 6.21. The instruction `goto L3` is redundant, since L_3 is the label of the very next instruction. The two `goto L1` instructions can be eliminated by using `ifFalse` instead of `if` instructions, as in Example 6.21. $\square$

6.6.5 Avoiding Redundant Gotos

In Example 6.22, the comparison $x > 200$ translates into the code fragment:

```

        if x > 200 goto L4
        goto L1
L4:    ...

```

Instead, consider the instruction:

```

        ifFalse x > 200 goto L1
L4:    ...

```

This `ifFalse` instruction takes advantage of the natural flow from one instruction to the next in sequence, so control simply “falls through” to label L_4 if $x > 200$, thereby avoiding a jump.

In the code layouts for `if`- and `while`-statements in Fig. 6.35, the code for statement S_1 immediately follows the code for the boolean expression B . By using a special label *fall* (i.e., “don’t generate any jump”), we can adapt the semantic rules in Fig. 6.36 and 6.37 to allow control to fall through from the code for B to the code for S_1 . The new rules for $S \rightarrow \text{if } (B) S_1$ in Fig. 6.36 set $B.\text{true}$ to *fall*:

$$\begin{aligned}
 B.\text{true} &= \text{fall} \\
 B.\text{false} &= S_1.\text{next} = S.\text{next} \\
 S.\text{code} &= B.\text{code} \parallel S_1.\text{code}
 \end{aligned}$$

Similarly, the rules for `if-else`- and `while`-statements also set $B.\text{true}$ to *fall*.

We now adapt the semantic rules for boolean expressions to allow control to fall through whenever possible. The new rules for $B \rightarrow E_1 \text{ rel } E_2$ in Fig. 6.39 generate two instructions, as in Fig. 6.37, if both $B.\text{true}$ and $B.\text{false}$ are explicit labels; that is, neither equals *fall*. Otherwise, if $B.\text{true}$ is an explicit label, then $B.\text{false}$ must be *fall*, so they generate an `if` instruction that lets control fall through if the condition is false. Conversely, if $B.\text{false}$ is an explicit label, then they generate an `ifFalse` instruction. In the remaining case, both $B.\text{true}$ and $B.\text{false}$ are *fall*, so no jump is generated.⁹

In the new rules for $B \rightarrow B_1 \parallel B_2$ in Fig. 6.40, note that the meaning of label *fall* for B is different from its meaning for B_1 . Suppose $B.\text{true}$ is *fall*; i.e., control falls through B , if B evaluates to true. Although B evaluates to true if B_1 does, $B_1.\text{true}$ must ensure that control jumps over the code for B_2 to get to the next instruction after B .

On the other hand, if B_1 evaluates to false, the truth-value of B is determined by the value of B_2 , so the rules in Fig. 6.40 ensure that $B_1.\text{false}$ corresponds to control falling through from B_1 to the code for B_2 .

The semantic rules for $B \rightarrow B_1 \&\& B_2$ are similar to those in Fig. 6.40. We leave them as an exercise.

Example 6.23: With the new rules using the special label *fall*, the program (6.13) from Example 6.21

⁹In C and Java, expressions may contain assignments within them, so code must be generated for the subexpressions E_1 and E_2 , even if both $B.\text{true}$ and $B.\text{false}$ are *fall*. If desired, dead code can be eliminated during an optimization phase.

```

test = E1.addr rel op E2.addr

s = if B.true ≠ fall and B.false ≠ fall then
    gen('if' test 'goto' B.true) || gen('goto' B.false)
else if B.true ≠ fall then gen('if' test 'goto' B.true)
else if B.false ≠ fall then gen('ifFalse' test 'goto' B.false)
else ''

B.code = E1.code || E2.code || s

```

Figure 6.39: Semantic rules for $B \rightarrow E_1 \text{ rel } E_2$

```

B1.true = if B.true ≠ fall then B.true else newlabel()
B1.false = fall
B2.true = B.true
B2.false = B.false
B.code = if B.true ≠ fall then B1.code || B2.code
        else B1.code || B2.code || label(B1.true)

```

Figure 6.40: Semantic rules for $B \rightarrow B_1 \mid \mid B_2$

```

if( x < 100 || x > 200 && x != y ) x = 0;

```

translates into the code of Fig. 6.41.

```

        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:   x = 0
L1:

```

Figure 6.41: If-statement translated using the fall-through technique

As in Example 6.22, the rules for $P \rightarrow S$ create label L_1 . The difference from Example 6.22 is that the inherited attribute $B.true$ is *fall* when the semantic rules for $B \rightarrow B_1 \mid \mid B_2$ are applied ($B.false$ is L_1). The rules in Fig. 6.40 create a new label L_2 to allow a jump over the code for B_2 if B_1 evaluates to true. Thus, $B_1.true$ is L_2 and $B_1.false$ is *fall*, since B_2 must be evaluated if B_1 is false.

The production $B \rightarrow E_1 \text{ rel } E_2$ that generates $x < 100$ is therefore reached with $B.true = L_2$ and $B.false = \text{fall}$. With these inherited labels, the rules in Fig. 6.39 therefore generate a single instruction `if x < 100 goto L2`. $\square$

6.6.6 Boolean Values and Jumping Code

The focus in this section has been on the use of boolean expressions to alter the flow of control in statements. A boolean expression may also be evaluated for its value, as in assignment statements such as $x = \text{true}$; or $x = a < b$;

A clean way of handling both roles of boolean expressions is to first build a syntax tree for expressions, using either of the following approaches:

1. *Use two passes.* Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.
2. *Use one pass for statements, but two passes for expressions.* With this approach, we would translate E in **while** (E) S_1 before S_1 is examined. The translation of E , however, would be done by building its syntax tree and then walking the tree.

The following grammar has a single nonterminal E for expressions:

$$\begin{aligned} S &\rightarrow \text{id} = E ; \mid \text{if} (E) S \mid \text{while} (E) S \mid S S \\ E &\rightarrow E \mid E \mid E \&\& E \mid E \text{rel} E \mid E + E \mid (E) \mid \text{id} \mid \text{true} \mid \text{false} \end{aligned}$$

Nonterminal E governs the flow of control in $S \rightarrow \text{while} (E) S_1$. The same nonterminal E denotes a value in $S \rightarrow \text{id} = E$; and $E \rightarrow E + E$.

We can handle these two roles of expressions by using separate code-generation functions. Suppose that attribute $E.n$ denotes the syntax-tree node for an expression E and that nodes are objects. Let method *jump* generate jumping code at an expression node, and let method *rvalue* generate code to compute the value of the node into a temporary.

When E appears in $S \rightarrow \text{while} (E) S_1$, method *jump* is called at node $E.n$. The implementation of *jump* is based on the rules for boolean expressions in Fig. 6.37. Specifically, jumping code is generated by calling $E.n.\text{jump}(t, f)$, where t is a new label for the first instruction of $S_1.\text{code}$ and f is the label $S.\text{next}$.

When E appears in $S \rightarrow \text{id} = E$;, method *rvalue* is called at node $E.n$. If E has the form $E_1 + E_2$, the method call $E.n.\text{rvalue}()$ generates code as discussed in Section 6.4. If E has the form $E_1 \&\& E_2$, we first generate jumping code for E and then assign true or false to a new temporary t at the true and false exits, respectively, from the jumping code.

For example, the assignment $x = a < b \ \&\& \ c < d$ can be implemented by the code in Fig. 6.42.

6.6.7 Exercises for Section 6.6

Exercise 6.6.1: Add rules to the syntax-directed definition of Fig. 6.36 for the following control-flow constructs:

- a) A repeat-statement **repeat** S **while** B .

```

        ifFalse a < b goto L1
        ifFalse c < d goto L1
        t = true
        goto L2
L1:   t = false
L2:   x = t

```

Figure 6.42: Translating a boolean assignment by computing the value of a temporary

! b) A for-loop **for** (S_1 ; B ; S_2) S_3 .

Exercise 6.6.2: Modern machines try to execute many instructions at the same time, including branching instructions. Thus, there is a severe cost if the machine speculatively follows one branch, when control actually goes another way (all the speculative work is thrown away). It is therefore desirable to minimize the number of branches. Notice that the implementation of a while-loop in Fig. 6.35(c) has two branches per iteration: one to enter the body from the condition B and the other to jump back to the code for B . As a result, it is usually preferable to implement **while** (B) S as if it were **if** (B) { **repeat** S **until** $!(B)$ }. Show what the code layout looks like for this translation, and revise the rule for while-loops in Fig. 6.36.

! **Exercise 6.6.3:** Suppose that there were an “exclusive-or” operator (true if and only if exactly one of its two arguments is true) in C. Write the rule for this operator in the style of Fig. 6.37.

Exercise 6.6.4: Translate the following expressions using the goto-avoiding translation scheme of Section 6.6.5:

a) `if (a==b && c==d || e==f) x == 1;`

b) `if (a==b || c==d || e==f) x == 1;`

c) `if (a==b && c==d && e==f) x == 1;`

Exercise 6.6.5: Give a translation scheme based on the syntax-directed definition in Figs. 6.36 and 6.37.

Exercise 6.6.6: Adapt the semantic rules in Figs. 6.36 and 6.37 to allow control to fall through, using rules like the ones in Figs. 6.39 and 6.40.

! **Exercise 6.6.7:** The semantic rules for statements in Exercise 6.6.6 generate unnecessary labels. Modify the rules for statements in Fig. 6.36 to create labels as needed, using a special label *deferred* to mean that a label has not yet been created. Your rules must generate code similar to that in Example 6.21.

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

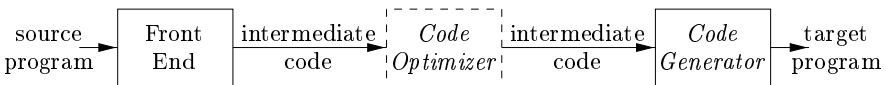


Figure 8.1: Position of code generator

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.

$t = a + b$ $t = t * c$ $t = t / d$	$t = a + b$ $t = t + c$ $t = t / d$
(a)	(b)

Figure 8.2: Two three-address code sequences

L R1, a A R1, b M R0, c D R0, d ST R1, t	L R0, a A R0, b A R0, c SRDA R0, 32 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 . $\square$

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, $R_0, R_1, \dots, R_{n-1}$. 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 r1, r2` 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, src1, src2`, 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 r1, r2, r3` 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 `Bcond  $r, L$` , 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 *i*th 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. $\square$

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