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.

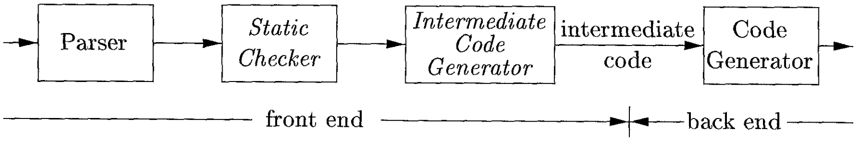


Figure 6.1: Logical structure of a compiler front end

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 $x = y \ op \ z$ 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.

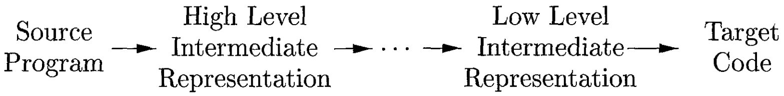

Source Program
High Level Intermediate Representation
· · ·
Low Level Intermediate Representation
Target Code

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 represents the components of a statement, whereas three-address code contains labels and jump instructions to represent 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 codes 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`. □

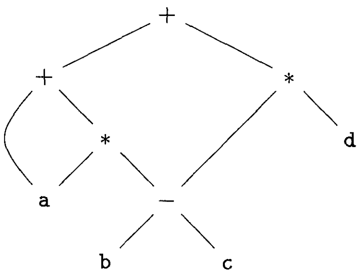


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

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

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

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

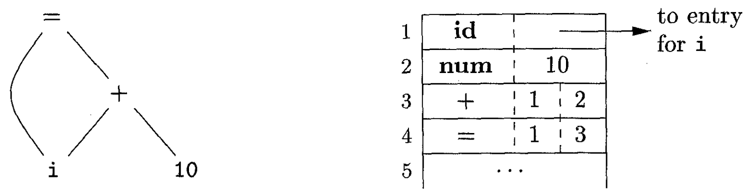
Figure 6.5: Steps for constructing the DAG of Fig. 6.3

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 at that returned at step 5; i.e., $p_{10} = p_5$. □

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.



1	**id**	→	
2	**num**	10	
3	+	1	2
4	=	1	3
5	. . .		

(a) DAG (b) Array.

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

[1] See 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.

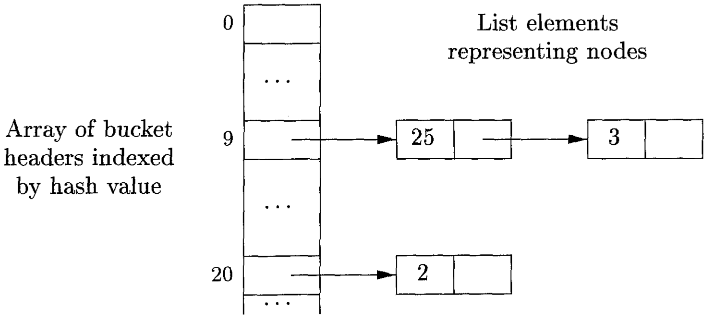


Figure 6.7: Data structure for searching buckets

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) $a + a + \big((a + a + a + (a + a + a + a)\big)$.

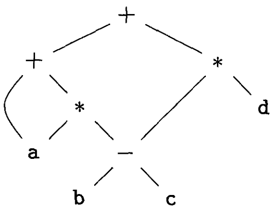
6.2 Three-Address Code

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

```
t1 = y * z
t2 = x + t1
```

where $\mathtt{t}_1$ and $\mathtt{t}_2$ 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. □



(a) DAG

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

(b) Three-address code

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 $x = y\ op\ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op\ y$, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, 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 x1
   param x2
   ...
   param xn
   call p, n
   ```

   generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. 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 uses a symbolic label `L`, attached to the first instruction. The

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

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. □

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(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 = 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 = t`$_5$.

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

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t$_1$
1	*	b	t$_1$	t$_2$
2	minus	c		t$_3$
3	*	b	t$_3$	t$_4$
4	+	t$_2$	t$_4$	t$_5$
5	=	t$_5$		a
		. . .		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

For readability, we use actual identifiers like a, b, and c in the fields *arg*$_1$, *arg*$_2$, and *result* in Fig. 6.10(b), instead of pointers to their symbol-table entries. Temporary names can either by 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*, *arg*$_1$, and *arg*$_2$. 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 t$_1$ in Fig. 6.10(b), a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. In Section 6.1.2, positions or pointers to positions were called value numbers.

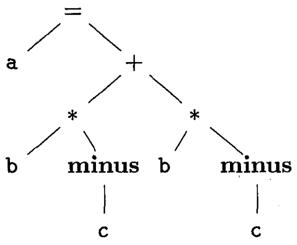
Triples are equivalent to signatures in Algorithm 6.3. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

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 = t$_5$ is encoded in the triple representation by placing a in the *arg*$_1$ field and (4) in the *arg*$_2$ 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.



(a) Syntax tree

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

$t = y[i]$ 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
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
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-assigment.* 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
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$\begin{aligned} p_1 &= a + b \\ q_1 &= p_1 - c \\ p_2 &= q_1 * d \\ p_3 &= e - p_2 \\ q_2 &= p_3 + q_1 \end{aligned}$$

(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:

$$\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1\texttt{;}$$
$$x_3 = \phi(x_1, x_2)\texttt{;}$$

Here, $\phi(\mathtt{x}_1, \mathtt{x}_2)$ has the value $\mathtt{x}_1$ if the control flow passes through the true part of the conditional and the value $\mathtt{x}_2$ 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. □

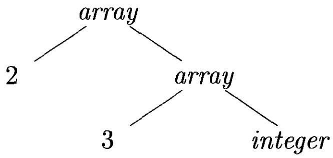


Figure 6.14: Type expression for `int[2][3]`

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 → 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 × associates to the left and that it has higher precedence than →.

- 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.

[3] Since 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

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

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 $B \rightarrow$ **int** then $B.type$ is set to *integer* and $B.width$ is set to 4, the width of an integer. Similarly, if $B \rightarrow$ **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 $C \rightarrow$ [**num**] C_1 forms $C.type$ by applying the type constructor *array* to the operands **num**.*value* and $C_1.type.$ For instance, the result of applying *array* might be a tree structure such as Fig. 6.14.

[4] Storage 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.

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

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. □

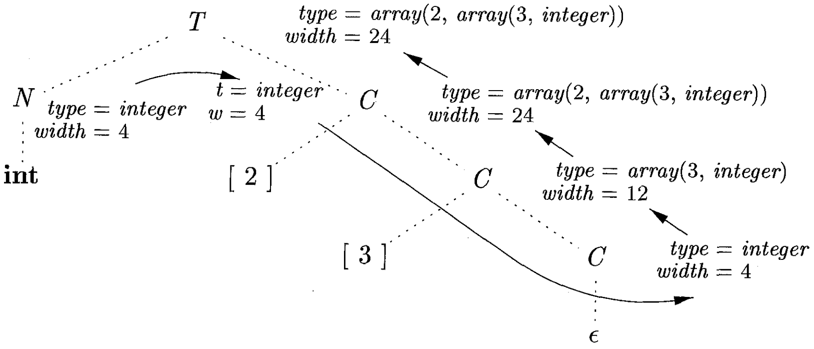


Figure 6.16: Syntax-directed translation of array types

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 **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.

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

Figure 6.17: Computing the relative addresses of declared names

The semantic action within the production $D \rightarrow T\,\mathbf{id}\ ;\ D_1$ creates a symbol-table entry by executing *top.put*(**id**.*lexeme*, *T.type*, *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:

$$P \ \rightarrow\ \{\ \mathit{offset} = 0;\ \}\ D \tag{6.1}$$

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:

$$\begin{array}{llll}
P & \rightarrow & M\ D & \\
M & \rightarrow & \epsilon & \{\ \mathit{offset} = 0;\ \}
\end{array}$$

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

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

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. □

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.

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

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.

6.4 Translation of Expressions

The rest of this chapter explores issues that arise during the translation of expressions and statements. We begin in this section with the translation of expressions into three-address code. An expression with more than one operator, like $a + b * c$, will translate into instructions with at most one operator per instruction. An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference. We shall consider type checking of expressions in Section 6.5 and the use of boolean expressions to direct the flow of control through a program in Section 6.6.

6.4.1 Operations Within Expressions

The syntax-directed definition in Fig. 6.19 builds up the three-address code for an assignment statement S using attribute *code* for S and attributes *addr* and *code* for an expression E. Attributes $S.code$ and $E.code$ denote the three-address code for S and E, respectively. Attribute $E.addr$ denotes the address that will

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \Vert$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \ \Vert\ E_2.code \ \Vert$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -\ E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \ \Vert$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = '\ '$

Figure 6.19: Three-address code for expressions

hold the value of E. Recall from Section 6.2.1 that an address can be a name, a constant, or a compiler-generated temporary.

Consider the last production, $E \rightarrow \textbf{id}$, in the syntax-directed definition in Fig. 6.19. When an expression is a single identifier, say x, then x itself holds the value of the expression. The semantic rules for this production define $E.addr$ to point to the symbol-table entry for this instance of **id**. Let *top* denote the current symbol table. Function *top.get* retrieves the entry when it is applied to the string representation **id**.*lexeme* of this instance of **id**. $E.code$ is set to the empty string.

When $E \rightarrow (\ E_1\ )$, the translation of E is the same as that of the subexpression E_1. Hence, $E.addr$ equals $E_1.addr$, and $E.code$ equals $E_1.code$.

The operators + and unary - in Fig. 6.19 are representative of the operators in a typical language. The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** *Temp*().

For convenience, we use the notation $gen(x\ '='\ y\ '+'\ z)$ to represent the three-address instruction $x = y + z$. Expressions appearing in place of variables like x, y, and z are evaluated when passed to *gen*, and quoted strings like $'='$ are taken literally.[5] Other three-address instructions will be built up similarly

[5] In syntax-directed definitions, *gen* builds an instruction and returns it. In translation schemes, *gen* builds an instruction and incrementally emits it by putting it into the stream

by applying *gen* to a combination of expressions and strings.

When we translate the production $E \rightarrow E_1 + E_2$, the semantic rules in Fig. 6.19 build up $E.code$ by concatenating $E_1.code$, $E_2.code$, and an instruction that adds the values of E_1 and E_2. The instruction puts the result of the addition into a new temporary name for E, denoted by $E.addr$.

The translation of $E \rightarrow - E_1$ is similar. The rules create a new temporary for E and generate an instruction to perform the unary minus operation.

Finally, the production $S \rightarrow \textbf{id} = E$; generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \textbf{id}$. *S.code* consists of the instructions to compute the value of E into an address given by $E.addr$, followed by an assignment to the address *top.get*(**id**.*lexeme*) for this instance of **id**.

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

```
t1 = minus c
t2 = b + t1
a = t2
```

□

6.4.2 Incremental Translation

Code attributes can be long strings, so they are usually generated incrementally, as discussed in Section 5.5.2. Thus, instead of building up $E.code$ as in Fig. 6.19, we can arrange to generate only the new three-address instructions, as in the translation scheme of Fig. 6.20. In the incremental approach, *gen* not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally.

The translation scheme in Fig. 6.20 generates the same code as the syntax-directed definition in Fig. 6.19. With the incremental approach, the *code* attribute is not used, since there is a single sequence of instructions that is created by successive calls to *gen*. For example, the semantic rule for $E \rightarrow E_1 + E_2$ in Fig. 6.20 simply calls *gen* to generate an add instruction; the instructions to compute E_1 into $E_1.addr$ and E_2 into $E_2.addr$ have already been generated.

The approach of Fig. 6.20 can also be used to build a syntax tree. The new semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$$E \rightarrow E_1 + E_2 \quad \{ \; E.addr \; = \; \textbf{new} \; Node('+', E_1.addr, E_2.addr); \; \}$$

Here, attribute *addr* represents the address of a node rather than a variable or constant.

of generated instructions.

S	$\rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' *E.addr*); }

E	$\rightarrow$	E_1 + E_2	{ *E.addr* = **new** *Temp* ();
			gen(*E.addr* '=' E_1.*addr* '+' E_2.*addr*); }

	|	- E_1	{ *E.addr* = **new** *Temp* ();
			gen(*E.addr* '=' '**minus**' E_1.*addr*); }

	|	(E_1)	{ *E.addr* = E_1.*addr*; }
	|	**id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }

Figure 6.20: Generating three-address code for expressions incrementally

6.4.3 Addressing Array Elements

Array elements can be accessed quickly if they are stored in a block of consecutive locations. In C and Java, array elements are numbered $0, 1, \ldots, n-1$, for an array with n elements. If the width of each array element is w, then the ith element of array A begins in location

$$base + i \times w \tag{6.2}$$

where *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of $A[0]$.

The formula (6.2) generalizes to two or more dimensions. In two dimensions, we write $A[i_1][i_2]$ in C and Java for element i_2 in row i_1. Let w_1 be the width of a row and let w_2 be the width of an element in a row. The relative address of $A[i_1][i_2]$ can then be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

where w_j, for $1 \leq j \leq k$, is the generalization of w_1 and w_2 in (6.3).

Alternatively, the relative address of an array reference can be calculated in terms of the numbers of elements n_j along dimension j of the array and the width $w = w_k$ of a single element of the array. In two dimensions (i.e., $k = 2$ and $w = w_2$), the location for $A[i_1][i_2]$ is given by

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

In k dimensions, the following formula calculates the same address as (6.4):

$$base + ((\cdots(i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w \tag{6.6}$$

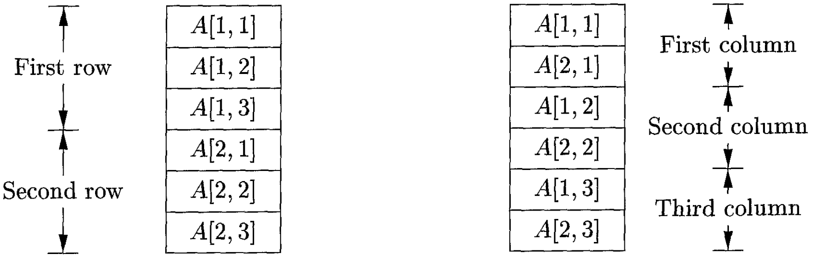
More generally, array elements need not be numbered starting at 0. In a one-dimensional array, the array elements are numbered *low*, *low* + 1, ... , *high* and *base* is the relative address of $A[low]$. Formula (6.2) for the address of $A[i]$ is replaced by:

$$base + (i - low) \times w \tag{6.7}$$

The expressions (6.2) and (6.7) can be both be rewritten as $i \times w + c$, where the subexpression $c = base - low \times w$ can be precalculated at compile time. Note that $c = base$ when *low* is 0. We assume that c is saved in the symbol table entry for A, so the relative address of $A[i]$ is obtained by simply adding $i \times w$ to c.

Compile-time precalculation can also be applied to address calculations for elements of multidimensional arrays; see Exercise 6.4.5. However, there is one situation where we cannot use compile-time precalculation: when the array's size is dynamic. If we do not know the values of *low* and *high* (or their generalizations in many dimensions) at compile time, then we cannot compute constants such as c. Then, formulas like (6.7) must be evaluated as they are written, when the program executes.

The above address calculations are based on row-major layout for arrays, which is used in C and Java. A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column). Figure 6.21 shows the layout of a 2 × 3 array A in (a) row-major form and (b) column-major form. Column-major form is used in the Fortran family of languages.



(a) Row Major (b) Column Major

Figure 6.21: Layouts for a two-dimensional array.

We can generalize row- or column-major form to many dimensions. The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest, like the numbers on an odometer. Column-major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest.

6.4.4 Translation of Array References

The chief problem in generating code for array references is to relate the address-calculation formulas in Section 6.4.3 to a grammar for array references. Let nonterminal L generate an array name followed by a sequence of index expressions:

$$L \;\rightarrow\; L\;[\;E\;]\;\mid\;\mathbf{id}\;[\;E\;]$$

As in C and Java, assume that the lowest-numbered array element is 0. Let us calculate addresses based on widths, using the formula (6.4), rather than on numbers of elements, as in (6.6). The translation scheme in Fig. 6.22 generates three-address code for expressions with array references. It consists of the productions and semantic actions from Fig. 6.20, together with productions involving nonterminal L.

```
S → id = E ;     { gen( top.get(id.lexeme) '=' E.addr); }

  | L = E ;      { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

E → E1 + E2      { E.addr = new Temp();
                   gen(E.addr '=' E1.addr '+' E2.addr); }

  | id           { E.addr = top.get(id.lexeme); }

  | L            { E.addr = new Temp();
                   gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id [ E ]     { L.array = top.get(id.lexeme);
                   L.type = L.array.type.elem;
                   L.addr = new Temp();
                   gen(L.addr '=' E.addr '*' L.type.width); }

  | L1 [ E ]     { L.array = L1.array;
                   L.type = L1.type.elem;
                   t = new Temp();
                   L.addr = new Temp();
                   gen(t '=' E.addr '*' L.type.width); }
                   gen(L.addr '=' L1.addr '+' t); }
```

Figure 6.22: Semantic actions for array references

Nonterminal L has three synthesized attributes:

1. *L.addr* denotes a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$ in (6.4).

2. *L.array* is a pointer to the symbol-table entry for the array name. The base address of the array, say, *L.array.base* is used to determine the actual *l*-value of an array reference after all the index expressions are analyzed.

3. *L.type* is the type of the subarray generated by L. For any type t, we assume that its width is given by *t.width*. We use types as attributes, rather than widths, since types are needed anyway for type checking. For any array type t, suppose that *t.elem* gives the element type.

The production $S \rightarrow \textbf{id} = E$; represents an assignment to a nonarray variable, which is handled as usual. The semantic action for $S \rightarrow L = E$; generates an indexed copy instruction to assign the value denoted by expression E to the location denoted by the array reference L. Recall that attribute *L.array* gives the symbol-table entry for the array. The array's base address — the address of its 0th element — is given by *L.array.base*. Attribute *L.addr* denotes the temporary that holds the offset for the array reference generated by L. The location for the array reference is therefore *L.array.base*[*L.addr*]. The generated instruction copies the *r*-value from address *E.addr* into the location for L.

Productions $E \rightarrow E_1 + E_2$ and $E \rightarrow \textbf{id}$ are the same as before. The semantic action for the new production $E \rightarrow L$ generates code to copy the value from the location denoted by L into a new temporary. This location is *L.array.base*[*L.addr*], as discussed above for the production $S \rightarrow L = E$;. Again, attribute *L.array* gives the array name, and *L.array.base* gives its base address. Attribute *L.addr* denotes the temporary that holds the offset. The code for the array reference places the *r*-value at the location designated by the base and offset into a new temporary denoted by *E.addr*.

Example 6.12: Let `a` denote a 2×3 array of integers, and let `c`, `i`, and `j` all denote integers. Then, the type of `a` is *array*(2, *array*(3, *integer*)). Its width w is 24, assuming that the width of an integer is 4. The type of `a[i]` is *array*(3, *integer*), of width $w_1 = 12$. The type of `a[i][j]` is *integer*.

An annotated parse tree for the expression `c + a[i][j]` is shown in Fig. 6.23. The expression is translated into the sequence of three-address instructions in Fig. 6.24. As usual, we have used the name of each identifier to refer to its symbol-table entry. □

6.4.5 Exercises for Section 6.4

Exercise 6.4.1: Add to the translation of Fig. 6.19 rules for the following productions:

a) $E \rightarrow E_1 * E_2$.

b) $E \rightarrow + E_1$ (unary plus).

Exercise 6.4.2: Repeat Exercise 6.4.1 for the incremental translation of Fig. 6.20.

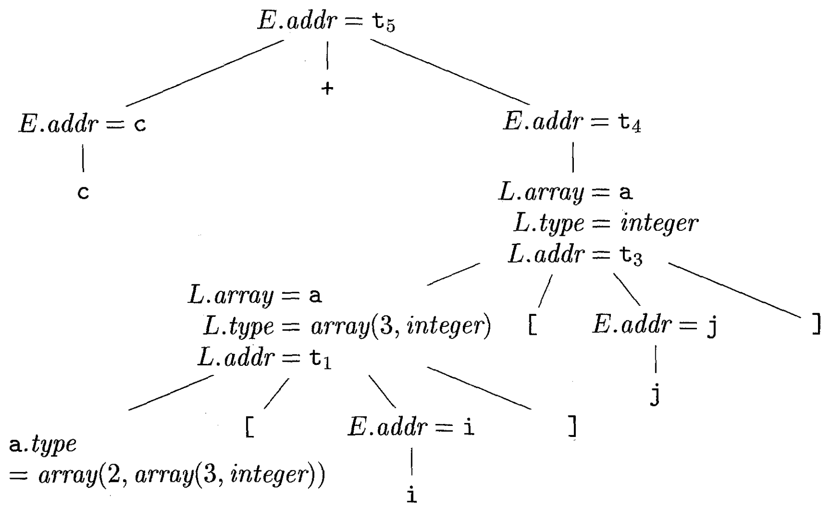


Figure 6.23: Annotated parse tree for `c + a[i][j]`

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Figure 6.24: Three-address code for expression `c + a[i][j]`

Exercise 6.4.3: Use the translation of Fig. 6.22 to translate the following assignments:

a) `x = a[i] + b[j].`

b) `x = a[i][j] + b[i][j].`

! c) `x = a[b[i][j]][c[k]].`

! **Exercise 6.4.4:** Revise the translation of Fig. 6.22 for array references of the Fortran style, that is, **id**$[E_1, E_2, \ldots, E_n]$ for an n-dimensional array.

Exercise 6.4.5: Generalize formula (6.7) to multidimensional arrays, and indicate what values can be stored in the symbol table and used to compute offsets. Consider the following cases:

a) An array A of two dimensions, in row-major form. The first dimension has indexes running from l_1 to h_1, and the second dimension has indexes from l_2 to h_2. The width of a single array element is w.

Symbolic Type Widths

The intermediate code should be relatively independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine. However, as we have described the calculation of type widths, an assumption regarding how basic types is built into the translation scheme. For instance, Example 6.12 assumes that each element of an integer array takes four bytes. Some intermediate codes, e.g., P-code for Pascal, leave it to the code generator to fill in the size of array elements, so the intermediate code is independent of the size of a machine word. We could have done the same in our translation scheme if we replaced 4 (as the width of an integer) by a symbolic constant.

b) The same as (a), but with the array stored in column-major form.

! c) An array A of k dimensions, stored in row-major form, with elements of size w. The jth dimension has indexes running from l_j to h_j.

! d) The same as (c) but with the array stored in column-major form.

Exercise 6.4.6: An integer array $A[i,j]$ has index i ranging from 1 to 10 and index j ranging from 1 to 20. Integers take 4 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[4,5]$ b)$A[10,8]$ c) $A[3,17]$.

Exercise 6.4.7: Repeat Exercise 6.4.6 if A is stored in column-major order.

Exercise 6.4.8: A real array $A[i,j,k]$ has index i ranging from 1 to 4, index j ranging from 0 to 4, and index k ranging from 5 to 10. Reals take 8 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[3,4,5]$ b)$A[1,2,7]$ c) $A[4,3,9]$.

Exercise 6.4.9: Repeat Exercise 6.4.8 if A is stored in column-major order.

6.5 Type Checking

To do *type checking* a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element. A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

Besides their use for compiling, ideas from type checking have been used to improve the security of systems that allow software modules to be imported and executed. Java programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

6.5.1 Rules for Type Checking

Type checking can take on two forms: synthesis and inference. *Type synthesis* builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form

$$\begin{array}{l}\textbf{if } f \text{ has type } s \rightarrow t \textbf{ and } x \text{ has type } s,\\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \tag{6.8}$$

Here, f and x denote expressions, and $s \rightarrow t$ denotes a function from s to t. This rule for functions with one argument carries over to functions with several arguments. The rule (6.8) can be adapted for $E_1 + E_2$ by viewing it as a function application $add(E_1, E_2)$.[6]

Type inference determines the type of a language construct from the way it is used. Looking ahead to the examples in Section 6.5.4, let *null* be a function that tests whether a list is empty. Then, from the usage $null(x)$, we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters $\alpha, \beta, \cdots$ for type variables in type expressions.

A typical rule for type inference has the form

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,}\\ \textbf{then} \text{ for some } \alpha \text{ and } \beta,\ f \text{ has type } \alpha \rightarrow \beta \textbf{ and } x \text{ has type } \alpha\end{array} \tag{6.9}$$

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

[6] We shall use the term "synthesis" even if some context information is used to determine types. With overloaded functions, where the same name is given to more than one function, the context of $E_1 + E_2$ may also need to be considered in some languages.

In this section, we consider type checking of expressions. The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement "**if** (E) S;" as if it were the application of a function *if* to E and S. Let the special type *void* denote the absence of a value. Then function *if* expects to be applied to a *boolean* and a *void*; the result of the application is a *void*.

6.5.2 Type Conversions

Consider expressions like $x + i$, where x is of type float and i is of type integer. Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

Suppose that integers are converted to floats when necessary, using a unary operator `(float)`. For example, the integer 2 is converted to a float in the code for the expression `2 * 3.14`:

```
t1 = (float) 2
t2 = t1 * 3.14
```

We can extend such examples to consider integer and float versions of the operators; for example, `int*` for integer operands and `float*` for floats.

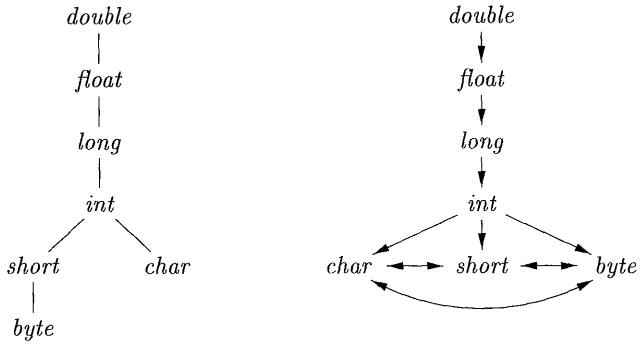
Type synthesis will be illustrated by extending the scheme in Section 6.4.2 for translating expressions. We introduce another attribute $E.type$, whose value is either *integer* or *float*. The rule associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode

```
if ( $E_1.type = integer$ and $E_2.type = integer$ ) $E.type = integer$;
else if ( $E_1.type = float$ and $E_2.type = integer$ ) ...
...
```

As the number of types subject to conversion increases, the number of cases increases rapidly. Therefore with large numbers of types, careful organization of the semantic actions becomes important.

Type conversion rules vary from language to language. The rules for Java in Fig. 6.25 distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information. The widening rules are given by the hierarchy in Fig. 6.25(a): any type lower in the hierarchy can be widened to a higher type. Thus, a *char* can be widened to an *int* or to a *float*, but a *char* cannot be widened to a *short*. The narrowing rules are illustrated by the graph in Fig. 6.25(b): a type s can be narrowed to a type t if there is a path from s to t. Note that *char*, *short*, and *byte* are pairwise convertible to each other.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions*,



(a) Widening conversions

(b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

are limited in many languages to widening conversions. Conversion is said to be *explicit* if the programmer must write something to cause the conversion. Explicit conversions are also called *casts*.

The semantic action for checking $E \to E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*, appears in Fig. 6.26.

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

The semantic action for $E \rightarrow E_1 + E_2$ in Fig. 6.27 illustrates how type conversions can be added to the scheme in Fig. 6.20 for translating expressions. In the semantic action, temporary variable a_1 is either $E_1.addr$, if the type of E_1 does not need to be converted to the type of E, or a new temporary variable returned by *widen* if this conversion is necessary. Similarly, a_2 is either $E_2.addr$ or a new temporary holding the type-converted value of E_2. Neither conversion is needed if both types are *integer* or both are *float*. In general, however, we could find that the only way to add values of two different types is to convert them both to a third type.

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type \;=\; max(E_1.type, E_2.type); \\
 & & a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \textbf{new}\ Temp\,(); \\
 & & gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{array}
$$

Figure 6.27: Introducing type conversions into expression evaluation

6.5.3 Overloading of Functions and Operators

An *overloaded* symbol has different meanings depending on its context. Overloading is *resolved* when a unique meaning is determined for each occurrence of a name. In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example 6.13: The + operator in Java denotes either string concatenation or addition, depending on the types of its operands. User-defined functions can be overloaded as well, as in

```
void err() { ··· }
void err(String s) { ··· }
```

Note that we can choose between these two versions of a function `err` by looking at their arguments. □

The following is a type-synthesis rule for overloaded functions:

$$
\begin{array}{l}
\textbf{if } f \text{ can have type } s_i \rightarrow t_i, \text{ for } 1 \le i \le n, \text{ where } s_i \ne s_j \text{ for } i \ne j \\
\textbf{and } x \text{ has type } s_k, \text{ for some } 1 \le k \le n \\
\textbf{then } \text{expression } f(x) \text{ has type } t_k
\end{array}
\tag{6.10}
$$

The value-number method of Section 6.1.2 can be applied to type expressions to resolve overloading based on argument types, efficiently. In a DAG representing a type expression, we assign an integer index, called a value number, to each node. Using Algorithm 6.3, we construct a signature for a node, consisting of its label and the value numbers of its children, in order from left to right. The signature for a function consists of the function name and the types of its arguments. The assumption that we can resolve overloading based on the types of arguments is equivalent to saying that we can resolve overloading based on signatures.

It is not always possible to resolve overloading by looking only at the arguments of a function. In Ada, instead of a single type, a subexpression standing alone may have a set of possible types for which the context must provide sufficient information to narrow the choice down to a single type (see Exercise 6.5.2).

6.5.4 Type Inference and Polymorphic Functions

Type inference is useful for a language like ML, which is strongly typed, but does not require names to be declared before they are used. Type inference ensures that names are used consistently.

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types. In this section, we consider *parametric polymorphism*, where the polymorphism is characterized by parameters or type variables. The running example is the ML program in Fig. 6.28, which defines a function *length*. The type of *length* can be described as, "for any type α, *length* maps a list of elements of type α to an integer."

```
fun length(x) =
        if null(x) then 0 else length(tl(x)) + 1;
```

Figure 6.28: ML program for the length of a list

Example 6.14: In Fig. 6.28, the keyword **fun** introduces a function definition; functions can be recursive. The program fragment defines function *length* with one parameter x. The body of the function consists of a conditional expression. The predefined function *null* tests whether a list is empty, and the predefined function *tl* (short for "tail") returns the remainder of a list after the first element is removed.

The function *length* determines the length or number of elements of a list x. All elements of a list must have the same type, but *length* can be applied to lists whose elements are of any one type. In the following expression, *length* is applied to two different types of lists (list elements are enclosed within "[" and "]"):

$$length([\texttt{"sun"}, \texttt{"mon"}, \texttt{"tue"}]) + length([10, 9, 8, 7]) \tag{6.11}$$

The list of strings has length 3 and the list of integers has length 4, so expression (6.11) evaluates to 7. □

Using the symbol ∀ (read as "for any type") and the type constructor *list*, the type of *length* can be written as

$$\forall \alpha.\ list(\alpha) \rightarrow integer \tag{6.12}$$

The ∀ symbol is the *universal quantifier*, and the type variable to which it is applied is said to be *bound* by it. Bound variables can be renamed at will, provided all occurrences of the variable are renamed. Thus, the type expression

$$\forall \beta.\ list(\beta) \rightarrow integer$$

is equivalent to (6.12). A type expression with a ∀ symbol in it will be referred to informally as a "polymorphic type."

Each time a polymorphic function is applied, its bound type variables can denote a different type. During type checking, at each use of a polymorphic type we replace the bound variables by fresh variables and remove the universal quantifiers.

The next example informally infers a type for *length*, implicitly using type inference rules like (6.9), which is repeated here:

if $f(x)$ is an expression,
then for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

Example 6.15: The abstract syntax tree in Fig. 6.29 represents the definition of *length* in Fig. 6.28. The root of the tree, labeled **fun**, represents the function definition. The remaining nonleaf nodes can be viewed as function applications. The node labeled + represents the application of the operator + to a pair of children. Similarly, the node labeled **if** represents the application of an operator **if** to a triple formed by its children (for type checking, it does not matter that either the **then** or the **else** part will be evaluated, but not both).

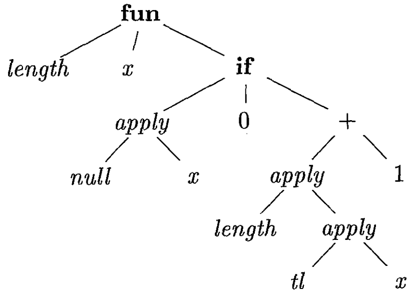


Figure 6.29: Abstract syntax tree for the function definition in Fig. 6.28

From the body of function *length*, we can infer its type. Consider the children of the node labeled **if**, from left to right. Since *null* expects to be applied to lists, x must be a list. Let us use variable α as a placeholder for the type of the list elements; that is, x has type "list of α."

Substitutions, Instances, and Unification

If t is a type expression and S is a substitution (a mapping from type variables to type expressions), then we write $S(t)$ for the result of consistently replacing all occurrences of each type variable α in t by $S(\alpha)$. $S(t)$ is called an *instance* of t. For example, *list*(*integer*) is an instance of *list*(α), since it is the result of substituting *integer* for α in *list*(α). Note, however, that *integer* → *float* is not an instance of $\alpha \rightarrow \alpha$, since a substitution must replace all occurrences of α by the same type expression.

Substitution S is a *unifier* of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$. S is the *most general unifier* of t_1 and t_2 if for any other unifier of t_1 and t_2, say S', it is the case that for any t, $S'(t)$ is an instance of $S(t)$. In words, S' imposes more constraints on t than S does.

If *null*(x) is true, then *length*(x) is 0. Thus, the type of *length* must be "function from list of α to integer." This inferred type is consistent with the usage of *length* in the else part, *length*(*tl*(x)) + 1. □

Since variables can appear in type expressions, we have to re-examine the notion of equivalence of types. Suppose E_1 of type $s \rightarrow s'$ is applied to E_2 of type t. Instead of simply determining the equality of s and t, we must "unify" them. Informally, we determine whether s and t can be made structurally equivalent by replacing the type variables in s and t by type expressions.

A *substitution* is a mapping from type variables to type expressions. We write $S(t)$ for the result of applying the substitution S to the variables in type expression t; see the box on "Substitutions, Instances, and Unification." Two type expressions t_1 and t_2 *unify* if there exists some substitution S such that $S(t_1) = S(t_2)$. In practice, we are interested in the most general unifier, which is a substitution that imposes the fewest constraints on the variables in the expressions. See Section 6.5.5 for a unification algorithm.

Algorithm 6.16: Type inference for polymorphic functions.

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \rightarrow t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

Check the function definitions and the expression in the input sequence. Use the inferred type of a function if it is subsequently used in an expression.

- For a function definition **fun** $\mathbf{id}_1(\mathbf{id}_2) = E$, create fresh type variables α and β. Associate the type $\alpha \rightarrow \beta$ with the function $\mathbf{id}_1$, and the type α with the parameter $\mathbf{id}_2$. Then, infer a type for expression E. Suppose α denotes type s and β denotes type t after type inference for E. The inferred type of function $\mathbf{id}_1$ is $s \rightarrow t$. Bind any type variables that remain unconstrained in $s \rightarrow t$ by $\forall$ quantifiers.

- For a function application $E_1(E_2)$, infer types for E_1 and E_2. Since E_1 is used as a function, its type must have the form $s \rightarrow s'$. (Technically, the type of E_1 must unify with $\beta \rightarrow \gamma$, where β and γ are new type variables). Let t be the inferred type of E_1. Unify s and t. If unification fails, the expression has a type error. Otherwise, the inferred type of $E_1(E_2)$ is s'.

- For each occurrence of a polymorphic function, replace the bound variables in its type by distinct fresh variables and remove the $\forall$ quantifiers. The resulting type expression is the inferred type of this occurrence.

- For a name that is encountered for the first time, introduce a fresh variable for its type.

□

Example 6.17: In Fig. 6.30, we infer a type for function *length*. The root of the syntax tree in Fig. 6.29 is for a function definition, so we introduce variables β and γ, associate the type $\beta \rightarrow \gamma$ with function *length*, and the type β with x; see lines 1-2 of Fig. 6.30.

At the right child of the root, we view **if** as a polymorphic function that is applied to a triple, consisting of a boolean and two expressions that represent the **then** and **else** parts. Its type is $\forall \alpha.\ \mathit{boolean} \times \alpha \times \alpha \rightarrow \alpha$.

Each application of a polymorphic function can be to a different type, so we make up a fresh variable α_i (where i is from "if") and remove the $\forall$; see line 3 of Fig. 6.30. The type of the left child of **if** must unify with *boolean*, and the types of its other two children must unify with α_i.

The predefined function *null* has type $\forall \alpha.\ \mathit{list}(\alpha) \rightarrow \mathit{boolean}$. We use a fresh type variable α_n (where n is for "null") in place of the bound variable α; see line 4. From the application of *null* to x, we infer that the type β of x must match $\mathit{list}(\alpha_n)$; see line 5.

At the first child of **if**, the type *boolean* for $\mathit{null}(x)$ matches the type expected by **if**. At the second child, the type α_i unifies with *integer*; see line 6.

Now, consider the subexpression $\mathit{length}(\mathit{tl}(x)) + 1$. We make up a fresh variable α_t (where t is for "tail") for the bound variable α in the type of *tl*; see line 8. From the application $\mathit{tl}(x)$, we infer $\mathit{list}(\alpha_t) = \beta = \mathit{list}(\alpha_n)$; see line 9.

Since $\mathit{length}(\mathit{tl}(x))$ is an operand of $+$, its type γ must unify with *integer*; see line 10. It follows that the type of *length* is $\mathit{list}(\alpha_n) \rightarrow \mathit{integer}$. After the

LINE	EXPRESSION : TYPE	UNIFY
1)	$\mathit{length} : \beta \to \gamma$	
2)	$x : \beta$	
3)	$\mathbf{if} : \mathit{boolean} \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$\mathit{null} : \mathit{list}(\alpha_n) \to \mathit{boolean}$	
5)	$\mathit{null}(x) : \mathit{boolean}$	$\mathit{list}(\alpha_n) = \beta$
6)	$0 : \mathit{integer}$	$\alpha_i = \mathit{integer}$
7)	$+ : \mathit{integer} \times \mathit{integer} \to \mathit{integer}$	
8)	$\mathit{tl} : \mathit{list}(\alpha_t) \to \mathit{list}(\alpha_t)$	
9)	$\mathit{tl}(x) : \mathit{list}(\alpha_t)$	$\mathit{list}(\alpha_t) = \mathit{list}(\alpha_n)$
10)	$\mathit{length}(\mathit{tl}(x)) : \gamma$	$\gamma = \mathit{integer}$
11)	$1 : \mathit{integer}$	
12)	$\mathit{length}(\mathit{tl}(x)) + 1 : \mathit{integer}$	
13)	$\mathbf{if}(\ \cdots\ ) : \mathit{integer}$	

Figure 6.30: Inferring a type for the function *length* of Fig. 6.28

function definition is checked, the type variable α_n remains in the type of *length*. Since no assumptions were made about α_n, any type can be substituted for it when the function is used. We therefore make it a bound variable and write

$$\forall \alpha_n .\ \mathit{list}(\alpha_n) \to \mathit{integer}$$

for the type of *length*. □

6.5.5 An Algorithm for Unification

Informally, unification is the problem of determining whether two expressions s and t can be made identical by substituting expressions for the variables in s and t. Testing equality of expressions is a special case of unification; if s and t have constants but no variables, then s and t unify if and only if they are identical. The unification algorithm in this section extends to graphs with cycles, so it can be used to test structural equivalence of circular types.[7]

We shall implement a graph-theoretic formulation of unification, where types are represented by graphs. Type variables are represented by leaves and type constructors are represented by interior nodes. Nodes are grouped into equivalence classes; if two nodes are in the same equivalence class, then the type expressions they represent must unify. Thus, all interior nodes in the same class must be for the same type constructor, and their corresponding children must be equivalent.

Example 6.18: Consider the two type expressions

[7] In some applications, it is an error to unify a variable with an expression containing that variable. Algorithm 6.19 permits such substitutions.

$$((\alpha_1 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$
$$((\alpha_3 \to \alpha_4) \times list(\alpha_3)) \to \alpha_5$$

The following substitution S is the most general unifier for these expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

This substitution maps the two type expressions to the following expression

$$((\alpha_1 \to \alpha_2) \ \times \ list(\alpha_1)) \to list(\alpha_2)$$

The two expressions are represented by the two nodes labeled →: 1 in Fig. 6.31. The integers at the nodes indicate the equivalence classes that the nodes belong to after the nodes numbered 1 are unified. □

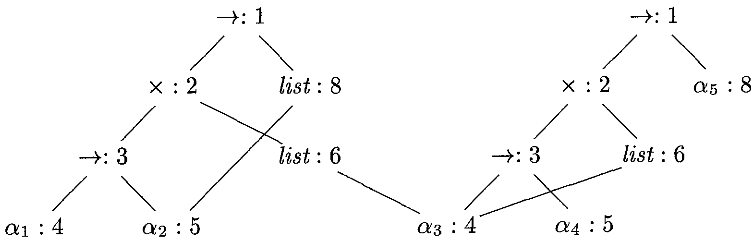


Figure 6.31: Equivalence classes after unification

Algorithm 6.19: Unification of a pair of nodes in a type graph.

INPUT: A graph representing a type and a pair of nodes m and n to be unified.

OUTPUT: Boolean value true if the expressions represented by the nodes m and n unify; false, otherwise.

METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contain a null pointer. The *set* fields of the remaining nodes in the equivalence class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

The unification algorithm, shown in Fig. 6.32, uses the following two operations on nodes:

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s_1 and s_2 and
                        t is an op-node with children t_1 and t_2) {
                union(s, t);
                return unify(s_1, t_1) and unify(s_2, t_2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```

Figure 6.32: Unification algorithm.

- *find*(n) returns the representative node of the equivalence class currently containing node n.

- *union*(m, n) merges the equivalence classes containing nodes m and n. If one of the representatives for the equivalence classes of m and n is a nonvariable node, *union* makes that nonvariable node be the representative for the merged equivalence class; otherwise, *union* makes one or the other of the original representatives be the new representative. This asymmetry in the specification of *union* is important because a variable cannot be used as the representative for an equivalence class for an expression containing a type constructor or basic type. Otherwise, two inequivalent expressions may be unified through that variable.

The *union* operation on sets is implemented by simply changing the *set* field of the representative of one equivalence class so that it points to the representative of the other. To find the equivalence class that a node belongs to, we follow the *set* pointers of nodes until the representative (the node with a null pointer in the set field) is reached.

Note that the algorithm in Fig. 6.32 uses $s = find(m)$ and $t = find(n)$ rather than m and n, respectively. The representative nodes s and t are equal if m and n are in the same equivalence class. If s and t represent the same basic type, the call *unify*(m, n) returns true. If s and t are both interior nodes for a binary type constructor, we merge their equivalence classes on speculation and recursively check that their respective children are equivalent. By merging first, we decrease the number of equivalence classes before recursively checking the children, so the algorithm terminates.

The substitution of an expression for a variable is implemented by adding the leaf for the variable to the equivalence class containing the node for that expression. Suppose either m or n is a leaf for a variable. Suppose also that this leaf has been put into an equivalence class with a node representing an expression with a type constructor or a basic type. Then *find* will return a representative that reflects that type constructor or basic type, so that a variable cannot be unified with two different expressions. □

Example 6.20: Suppose that the two expressions in Example 6.18 are represented by the initial graph in Fig. 6.33, where each node is in its own equivalence class. When Algorithm 6.19 is applied to compute *unify*(1, 9), it notes that nodes 1 and 9 both represent the same operator. It therefore merges 1 and 9 into the same equivalence class and calls *unify*(2, 10) and *unify*(8, 14). The result of computing *unify*(1, 9) is the graph previously shown in Fig. 6.31. □

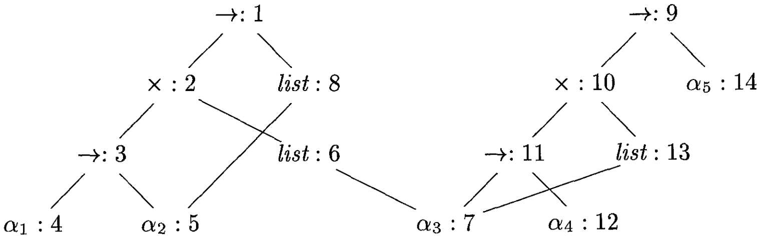


Figure 6.33: Initial graph with each node in its own equivalence class

If Algorithm 6.19 returns true, we can construct a substitution S that acts as the unifier, as follows. For each variable α, *find*(α) gives the node n that is the representative of the equivalence class of α. The expression represented by n is $S(\alpha)$. For example, in Fig. 6.31, we see that the representative for α_3 is node 4, which represents α_1. The representative for α_5 is node 8, which represents $list(\alpha_2)$. The resulting substitution S is as in Example 6.18.

6.5.6 Exercises for Section 6.5

Exercise 6.5.1: Assuming that function *widen* in Fig. 6.26 can handle any of the types in the hierarchy of Fig. 6.25(a), translate the expressions below. Assume that c and d are characters, s and t are short integers, i and j are integers, and x is a float.

a) `x = s + c.`

b) `i = s + c.`

c) `x = (s + c) * (t + d).`

Exercise 6.5.2: As in Ada, suppose that each expression must have a unique type, but that from a subexpression, by itself, all we can deduce is a set of possible types. That is, the application of function E_1 to argument E_2, represented by $E \rightarrow E_1 \,(\, E_2 \,)$, has the associated rule

$$E.type \;=\; \{\; t \mid \text{ for some } s \text{ in } E_2.type,\; s \rightarrow t \text{ is in } E_1.type \;\}$$

Describe an SDD that determines a unique type for each subexpression by using an attribute *type* to synthesize a set of possible types bottom-up, and, once the unique type of the overall expression is determined, proceeds top-down to determine attribute *unique* for the type of each subexpression.

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:

$$B \rightarrow B \;||\; B \mid B \;\&\&\; B \mid \;!\,B \mid (\,B\,) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}$$

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 || and && are left-associative, and that || has lowest precedence, then &&, then !.

Given the expression B_1 || 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 || 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 &&, ||, 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. □

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

Figure 6.34: Jumping code

6.6.3 Flow-of-Control Statements

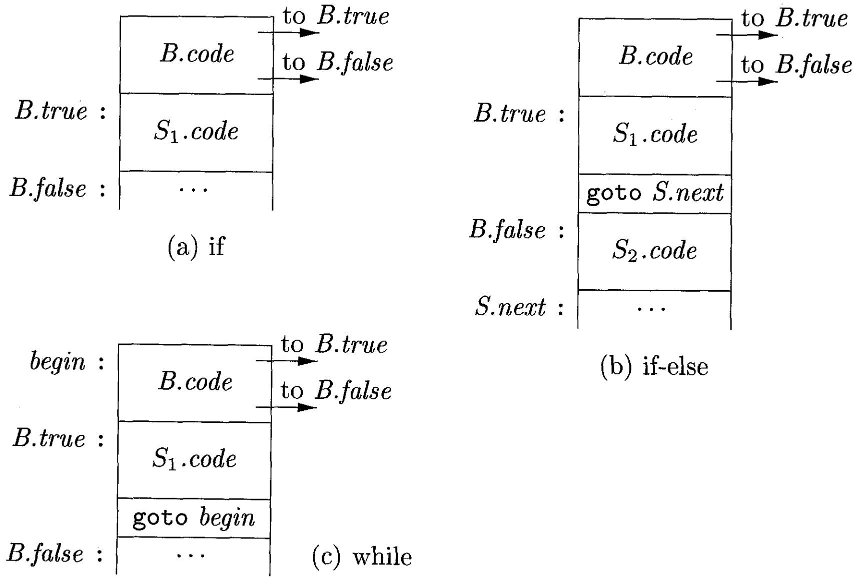
We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

$$\begin{array}{lll} S & \rightarrow & \textbf{if } (B) \; S_1 \\ S & \rightarrow & \textbf{if } (B) \; S_1 \textbf{ else } S_2 \\ S & \rightarrow & \textbf{while } (B) \; S_1 \end{array}$$

In these productions, nonterminal B represents a boolean expression and nonterminal 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 **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 $S_1.code$.



(a) if

(b) if-else

(c) while

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

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
$P \rightarrow S$	$S.\mathit{next} = \mathit{newlabel}()$ $P.\mathit{code} = S.\mathit{code} \;\|\|\; \mathit{label}(S.\mathit{next})$
$S \rightarrow$ **assign**	$S.\mathit{code} = \textbf{assign}.\mathit{code}$
$S \rightarrow$ **if** (B) S_1	$B.\mathit{true} = \mathit{newlabel}()$ $B.\mathit{false} = S_1.\mathit{next} = S.\mathit{next}$ $S.\mathit{code} = B.\mathit{code} \;\|\|\; \mathit{label}(B.\mathit{true}) \;\|\|\; S_1.\mathit{code}$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.\mathit{true} = \mathit{newlabel}()$ $B.\mathit{false} = \mathit{newlabel}()$ $S_1.\mathit{next} = S_2.\mathit{next} = S.\mathit{next}$ $S.\mathit{code} = B.\mathit{code}$ $\|\| \; \mathit{label}(B.\mathit{true}) \;\|\|\; S_1.\mathit{code}$ $\|\| \; \mathit{gen}(\texttt{'goto'}\; S.\mathit{next})$ $\|\| \; \mathit{label}(B.\mathit{false}) \;\|\|\; S_2.\mathit{code}$
$S \rightarrow$ **while** (B) S_1	$\mathit{begin} = \mathit{newlabel}()$ $B.\mathit{true} = \mathit{newlabel}()$ $B.\mathit{false} = S.\mathit{next}$ $S_1.\mathit{next} = \mathit{begin}$ $S.\mathit{code} = \mathit{label}(\mathit{begin}) \;\|\|\; B.\mathit{code}$ $\|\| \; \mathit{label}(B.\mathit{true}) \;\|\|\; S_1.\mathit{code}$ $\|\| \; \mathit{gen}(\texttt{'goto'}\; \mathit{begin})$
$S \rightarrow S_1 \; S_2$	$S_1.\mathit{next} = \mathit{newlabel}()$ $S_2.\mathit{next} = S.\mathit{next}$ $S.\mathit{code} = S_1.\mathit{code} \;\|\|\; \mathit{label}(S_1.\mathit{next}) \;\|\|\; S_2.\mathit{code}$

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

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

[8] If 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 $S \rightarrow$ **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 $S \rightarrow$ **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 $S \rightarrow$ **if** (B) S_1 **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 $S \rightarrow$ **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	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \,\|\|\, label(B_1.true) \,\|\|\, B_2.code$
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \,\|\|\, E_2.code$ $\|\|\ gen($'`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto`' $B.true)$ $\|\|\ gen($'`goto`' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'`goto`' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'`goto`' $B.false)$

Figure 6.37: Generating three-address code for booleans

The fourth production in Fig. 6.37, $B \rightarrow E_1$ **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 $\mathtt{L}_3$, 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` $\mathtt{L}_3$ is redundant, since L_3 is the label of the very next instruction. The two `goto` $\mathtt{L}_1$ instructions can be eliminated by using `ifFalse` instead of `if` instructions, as in Example 6.21. □

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 $x > 200$ is false, 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 $S \rightarrow$ **if** (B) S_1 in Fig. 6.36 set $B.\mathit{true}$ to *fall*:

$$\begin{aligned} B.\mathit{true} &= \mathit{fall} \\ B.\mathit{false} &= S_1.\mathit{next} \;=\; S.\mathit{next} \\ S.\mathit{code} &= B.\mathit{code} \;||\; S_1.\mathit{code} \end{aligned}$$

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

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

In the new rules for $B \rightarrow B_1$ || B_2 in Fig. 6.40, note that the meaning of label *fall* for B is different from its meaning for B_1. Suppose $B.\mathit{true}$ is *fall*; i.e, control falls through B, if B evaluates to true. Although B evaluates to true if B_1 does, $B_1.\mathit{true}$ 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 $B_1.\mathit{false}$ corresponds to control falling through from B_1 to the code for B_2.

The semantic rules are 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

[9] In 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 $B.\mathit{true}$ and $B.\mathit{false}$ are *fall*. If desired, dead code can be eliminated during an optimization phase.

```
test = $E_1.addr$ rel.op $E_2.addr$

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

B.code = $E_1.code$ || $E_2.code$ || s
```

Figure 6.39: Semantic rules for $B \rightarrow E_1$ **rel** E_2

```
 $B_1.true$ = if $B.true \neq fall$ then $B.true$ else newlabel()
$B_1.false$ = fall
 $B_2.true$ = $B.true$
$B_2.false$ = $B.false$
   B.code = if $B.true \neq fall$ then $B_1.code$ || $B_2.code$
            else $B_1.code$ || $B_2.code$ || label($B_1.true$)
```

Figure 6.40: Semantic rules for $B \rightarrow B_1$ || 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 $B \rightarrow B_1$ || 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 $B \rightarrow E_1$ **rel** E_2 that generates $x < 100$ is therefore reached with $B.true = L_2$ and $B.false = fall$. With these inherited labels, the rules in Fig. 6.39 therefore generate a single instruction `if x < 100 goto L2`. □

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 = 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:

$$S \rightarrow \textbf{id} = E \; ; \;\mid\; \textbf{if} \,(\, E \,)\, S \;\mid\; \textbf{while} \,(\, E \,)\, S \;\mid\; S \; S$$
$$E \rightarrow E \,||\, E \;\mid\; E \,\&\&\, E \mid E \;\textbf{rel}\; E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false}$$

Nonterminal E governs the flow of control in $S \rightarrow$ **while** (E) S_1. The same nonterminal E denotes a value in $S \rightarrow \textbf{id} = E$; 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 $S \rightarrow$ **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 $E.n.jump(t, f)$, where t is a new label for the first instruction of $S_1.code$ and f is the label $S.next$.

When E appears in $S \rightarrow \textbf{id} = E$;, method *rvalue* is called at node $E.n$. If E has the form $E_1 + E_2$, the method call $E.n.rvalue()$ 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-statment **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 interation: 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.

!! **Exercise 6.6.8:** Section 6.6.5 talks about using fall-through code to minimize the number of jumps in the generated intermediate code. However, it does not take advantage of the option to replace a condition by its complement, e.g., replace `if a < b goto` L_1`;` `goto` L_2 by `if b >= a goto` L_2`;` `goto` L_1. Develop a SDD that does take advantage of this option when needed.

6.7 Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression B in `if (` B `)` S contains a jump, for when B is false, to the instruction following the code for S. In a one-pass translation, B must be translated before S is examined. What then is the target of the `goto` that jumps over the code for S? In Section 6.6 we addressed this problem by passing labels as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.

This section takes a complementary approach, called *backpatching*, in which lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

6.7.1 One-Pass Code Generation Using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass. The translations we generate will be of the same form as those in Section 6.6, except for how we manage labels.

In this section, synthesized attributes *truelist* and *falselist* of nonterminal B are used to manage labels in jumping code for boolean expressions. In particular, B.*truelist* will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true. B.*falselist* likewise is the list of instructions that eventually get the label to which control goes when B is false. As code is generated for B, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by B.*truelist* and B.*falselist*, as appropriate. Similarly, a statement S has a synthesized attribute S.*nextlist*, denoting a list of jumps to the instruction immediately following the code for S.

For specificity, we generate instructions into an instruction array, and labels will be indices into this array. To manipulate lists of jumps, we use three functions:

1. *makelist*(i) creates a new list containing only i, an index into the array of instructions; *makelist* returns a pointer to the newly created list.

2. $\mathit{merge}(p_1, p_2)$ concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.

3. $\mathit{backpatch}(p, i)$ inserts i as the target label for each of the instructions on the list pointed to by p.

6.7.2 Backpatching for Boolean Expressions

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

$$B \rightarrow B_1 \;||\; M\, B_2 \mid B_1 \;\&\&\; M\; B_2 \mid \;!\; B_1 \mid (\, B_1\,) \mid E_1 \;\mathbf{rel}\; E_2 \mid \mathbf{true} \mid \mathbf{false}$$
$$M \rightarrow \epsilon$$

The translation scheme is in Fig. 6.43.

1)	$B \rightarrow B_1 \;\vert\vert\; M\; B_2$	$\{\ \mathit{backpatch}(B_1.\mathit{falselist}, M.\mathit{instr});$ $B.\mathit{truelist} = \mathit{merge}(B_1.\mathit{truelist}, B_2.\mathit{truelist});$ $B.\mathit{falselist} = B_2.\mathit{falselist};\ \}$
2)	$B \rightarrow B_1 \;\&\&\; M\; B_2$	$\{\ \mathit{backpatch}(B_1.\mathit{truelist}, M.\mathit{instr});$ $B.\mathit{truelist} = B_2.\mathit{truelist};$ $B.\mathit{falselist} = \mathit{merge}(B_1.\mathit{falselist}, B_2.\mathit{falselist});\ \}$
3)	$B \rightarrow \;!\; B_1$	$\{\ B.\mathit{truelist} = B_1.\mathit{falselist};$ $B.\mathit{falselist} = B_1.\mathit{truelist};\ \}$
4)	$B \rightarrow (\ B_1\ )$	$\{\ B.\mathit{truelist} = B_1.\mathit{truelist};$ $B.\mathit{falselist}\ =\ B_1.\mathit{falselist};\ \}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{\ B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr});$ $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr} + 1);$ $\mathit{emit}(\texttt{'if'}\ E_1.\mathit{addr}\ \mathbf{rel}.\mathit{op}\ E_2.\mathit{addr}\ \texttt{'goto \_'});$ $\mathit{emit}(\texttt{'goto \_'});\ \}$
6)	$B \rightarrow \mathbf{true}$	$\{\ B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr});$ $\mathit{emit}(\texttt{'goto \_'});\ \}$
7)	$B \rightarrow \mathbf{false}$	$\{\ B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr});$ $\mathit{emit}(\texttt{'goto \_'});\ \}$
8)	$M \rightarrow \epsilon$	$\{\ M.\mathit{instr} = \mathit{nextinstr};\ \}$

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production $B \rightarrow B_1 \;||\; M\, B_2$. If B_1 is true, then B is also true, so the jumps on $B_1.\mathit{truelist}$ become part of $B.\mathit{truelist}$. If B_1 is false, however, we must next test B_2, so the target for the jumps

B_1.*falselist* must be the beginning of the code generated for B_2. This target is obtained using the marker nonterminal M. That nonterminal produces, as a synthesized attribute $M.instr$, the index of the next instruction, just before B_2 code starts being generated.

To obtain that instruction index, we associate with the production $M \rightarrow \epsilon$ the semantic action

$$\{ \ M.instr = nextinstr; \ \}$$

The variable *nextinstr* holds the index of the next instruction to follow. This value will be backpatched onto the B_1.*falselist* (i.e., each instruction on the list B_1.*falselist* will receive $M.instr$ as its target label) when we have seen the remainder of the production $B \rightarrow B_1$ || $M \ B_2$.

Semantic action (2) for $B \rightarrow B_1$ && $M \ B_2$ is similar to (1). Action (3) for $B \rightarrow$! B swaps the true and false lists. Action (4) ignores parentheses.

For simplicity, semantic action (5) generates two instructions, a conditional goto and an unconditional one. Neither has its target filled in. These instructions are put on new lists, pointed to by B.*truelist* and B.*falselist*, respectively.

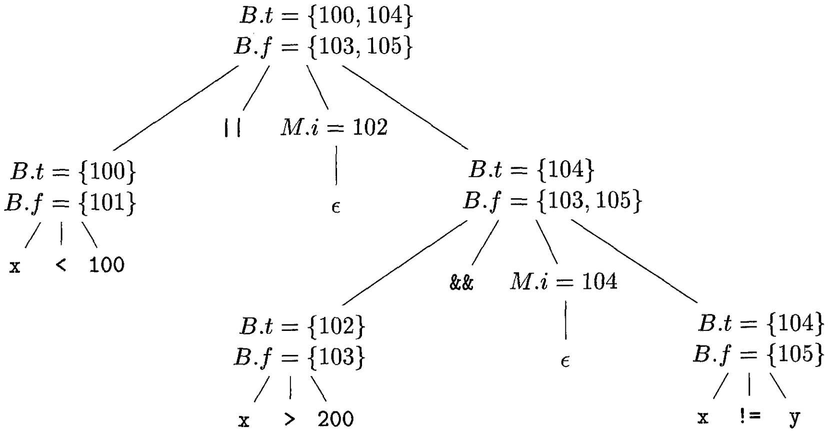


Figure 6.44: Annotated parse tree for $x < 100$ || $x > 200$ && x ! = y

Example 6.24: Consider again the expression

$$x < 100 \ || \ x > 200 \ \&\& \ x \ ! = \ y$$

An annotated parse tree is shown in Fig. 6.44; for readability, attributes *truelist*, *falselist*, and *instr* are represented by their initial letters. The actions are performed during a depth-first traversal of the tree. Since all actions appear at the ends of right sides, they can be performed in conjunction with reductions during a bottom-up parse. In response to the reduction of $x < 100$ to B by production (5), the two instructions

```
100:  if x < 100 goto _
101:  goto _
```

are generated. (We arbitrarily start instruction numbers at 100.) The marker nonterminal M in the production

$$B \rightarrow B_1 \;\texttt{||}\; M \, B_2$$

records the value of *nextinstr*, which at this time is 102. The reduction of $x > 200$ to B by production (5) generates the instructions

```
102:  if x > 200 goto _
103:  goto _
```

The subexpression $x > 200$ corresponds to B_1 in the production

$$B \rightarrow B_1 \;\&\&\; M \; B_2$$

The marker nonterminal M records the current value of *nextinstr*, which is now 104. Reducing $x \; != \; y$ into B by production (5) generates

```
104:  if x != y goto _
105:  goto _
```

We now reduce by $B \rightarrow B_1 \;\&\&\; M \; B_2$. The corresponding semantic action calls *backpatch*(B_1.*truelist*, M.*instr*) to bind the true exit of B_1 to the first instruction of B_2. Since B_1.*truelist* is $\{102\}$ and M.*instr* is 104, this call to *backpatch* fills in 104 in instruction 102. The six instructions generated so far are thus as shown in Fig. 6.45(a).

The semantic action associated with the final reduction by $B \rightarrow B_1 \;\texttt{||}\; M \, B_2$ calls *backpatch*($\{101\}$,102) which leaves the instructions as in Fig. 6.45(b).

The entire expression is true if and only if the gotos of instructions 100 or 104 are reached, and is false if and only if the gotos of instructions 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression. □

6.7.3 Flow-of-Control Statements

We now use backpatching to translate flow-of-control statements in one pass. Consider statements generated by the following grammar:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if}\,(\,B\,)\,S \;\mid\; \textbf{if}\,(\,B\,)\,S\;\textbf{else}\;S \;\mid\; \textbf{while}\,(\,B\,)\,S \;\mid\; \{\,L\,\} \mid A\;; \\ L & \rightarrow & L\,S \;\mid\; S \end{array}$$

Here S denotes a statement, L a statement list, A an assignment-statement, and B a boolean expression. Note that there must be other productions, such as

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instruction 101.

Figure 6.45: Steps in the backpatch process

those for assignment-statements. The productions given, however, are sufficient to illustrate the techniques used to translate flow-of-control statements.

The code layout for if-, if-else-, and while-statements is the same as in Section 6.6. We make the tacit assumption that the code sequence in the instruction array reflects the natural flow of control from one instruction to the next. If not, then explicit jumps must be inserted to implement the natural sequential flow of control.

The translation scheme in Fig. 6.46 maintains lists of jumps that are filled in when their targets are found. As in Fig. 6.43, boolean expressions generated by nonterminal B have two lists of jumps, B.*truelist* and B.*falselist*, corresponding to the true and false exits from the code for B, respectively. Statements generated by nonterminals S and L have a list of unfilled jumps, given by attribute *nextlist*, that must eventually be completed by backpatching. S.*nextlist* is a list of all conditional and unconditional jumps to the instruction following the code for statement S in execution order. L.*nextlist* is defined similarly.

Consider the semantic action (3) in Fig. 6.46. The code layout for production $S \rightarrow$ **while** (B) S_1 is as in Fig. 6.35(c). The two occurrences of the marker nonterminal M in the production

$$S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$$

record the instruction numbers of the beginning of the code for B and the beginning of the code for S_1. The corresponding labels in Fig. 6.35(c) are *begin* and B.*true*, respectively.

1) $S \rightarrow \textbf{if}\,(\,B\,)\; M\; S_1$ $\{\ \mathit{backpatch}(B.\mathit{truelist},\ M.\mathit{instr});$
$S.\mathit{nextlist}\ =\ \mathit{merge}(B.\mathit{falselist},\ S_1.\mathit{nextlist});\ \}$

2) $S \rightarrow \textbf{if}\,(\,B\,)\; M_1\; S_1\; N\; \textbf{else}\; M_2\; S_2$
$\{\ \mathit{backpatch}(B.\mathit{truelist},\ M_1.\mathit{instr});$
$\mathit{backpatch}(B.\mathit{falselist},\ M_2.\mathit{instr});$
$\mathit{temp}\ =\ \mathit{merge}(S_1.\mathit{nextlist},\ N.\mathit{nextlist});$
$S.\mathit{nextlist}\ =\ \mathit{merge}(\mathit{temp},\ S_2.\mathit{nextlist});\ \}$

3) $S \rightarrow \textbf{while}\; M_1\; (\,B\,)\; M_2\; S_1$
$\{\ \mathit{backpatch}(S_1.\mathit{nextlist},\ M_1.\mathit{instr});$
$\mathit{backpatch}(B.\mathit{truelist},\ M_2.\mathit{instr});$
$S.\mathit{nextlist}\ =\ B.\mathit{falselist};$
$\mathit{emit}(\texttt{'goto'}\ M_1.\mathit{instr});\ \}$

4) $S \rightarrow \{\ L\ \}$ $\{\ S.\mathit{nextlist}\ =\ L.\mathit{nextlist};\ \}$

5) $S \rightarrow A\ ;$ $\{\ S.\mathit{nextlist}\ =\ \textbf{null};\ \}$

6) $M \rightarrow \epsilon$ $\{\ M.\mathit{instr}\ =\ \mathit{nextinstr};\ \}$

7) $N \rightarrow \epsilon$ $\{\ N.\mathit{nextlist}\ =\ \mathit{makelist}(\mathit{nextinstr});$
$\mathit{emit}(\texttt{'goto \_'});\ \}$

8) $L \rightarrow L_1\ M\ S$ $\{\ \mathit{backpatch}(L_1.\mathit{nextlist},\ M.\mathit{instr});$
$L.\mathit{nextlist}\ =\ S.\mathit{nextlist};\ \}$

9) $L \rightarrow S$ $\{\ L.\mathit{nextlist}\ =\ S.\mathit{nextlist};\ \}$

Figure 6.46: Translation of statements

Again, the only production for M is $M \rightarrow \epsilon$. Action (6) in Fig. 6.46 sets attribute $M.\mathit{instr}$ to the number of the next instruction. After the body S_1 of the while-statement is executed, control flows to the beginning. Therefore, when we reduce **while** M_1 (B) $M_2\, S_1$ to S, we backpatch $S_1.\mathit{nextlist}$ to make all targets on that list be $M_1.\mathit{instr}$. An explicit jump to the beginning of the code for B is appended after the code for S_1 because control may also "fall out the bottom." $B.\mathit{truelist}$ is backpatched to go to the beginning of S_1 by making jumps on $B.\mathit{truelist}$ go to $M_2.\mathit{instr}$.

A more compelling argument for using $S.\mathit{nextlist}$ and $L.\mathit{nextlist}$ comes when code is generated for the conditional statement **if** (B) S_1 **else** S_2. If control "falls out the bottom" of S_1, as when S_1 is an assignment, we must include at the end of the code for S_1 a jump over the code for S_2. We use another marker nonterminal to generate this jump after S_1. Let nonterminal N be this marker with production $N \rightarrow \epsilon$. N has attribute *N.nextlist*, which will be a list consisting of the instruction number of the jump goto _ that is generated by the semantic action (7) for N.

Semantic action (2) in Fig. 6.46 deals with if-else-statements with the syntax

$$S \quad \rightarrow \quad \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$$

We backpatch the jumps when B is true to the instruction M_1*.instr*; the latter is the beginning of the code for S_1. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S_2. The list *S.nextlist* includes all jumps out of S_1 and S_2, as well as the jump generated by N. (Variable *temp* is a temporary that is used only for merging lists.)

Semantic actions (8) and (9) handle sequences of statements. In

$$L \rightarrow L_1\, M\, S$$

the instruction following the code for L_1 in order of execution is the beginning of S. Thus the L_1*.nextlist* list is backpatched to the beginning of the code for S, which is given by *M.instr*. In $L \rightarrow S$, *L.nextlist* is the same as *S.nextlist*.

Note that no new instructions are generated anywhere in these semantic rules, except for rules (3) and (7). All other code is generated by the semantic actions associated with assignment-statements and expressions. The flow of control causes the proper backpatching so that the assignments and boolean expression evaluations will connect properly.

6.7.4 Break-, Continue-, and Goto-Statements

The most elementary programming language construct for changing the flow of control in a program is the goto-statement. In C, a statement like goto L sends control to the statement labeled L — there must be precisely one statement with label L in this scope. Goto-statements can be implemented by maintaining a list of unfilled jumps for each label and then backpatching the target when it is known.

Java does away with goto-statements. However, Java does permit disciplined jumps called break-statements, which send control out of an enclosing construct, and continue-statements, which trigger the next iteration of an enclosing loop. The following excerpt from a lexical analyzer illustrates simple break- and continue-statements:

```
1)  for ( ; ; readch() ) {
2)      if( peek == ' ' || peek == '\t' ) continue;
3)      else if( peek == '\n' ) line = line + 1;
4)      else break;
5)  }
```

Control jumps from the break-statement on line 4 to the next statement after the enclosing for loop. Control jumps from the continue-statement on line 2 to code to evaluate *readch*() and then to the if-statement on line 2.

If S is the enclosing construct, then a break-statement is a jump to the first instruction after the code for S. We can generate code for the break by (1) keeping track of the enclosing statement S, (2) generating an unfilled jump for the break-statement, and (3) putting this unfilled jump on $S.nextlist$, where *nextlist* is as discussed in Section 6.7.3.

In a two-pass front end that builds syntax trees, $S.nextlist$ can be implemented as a field in the node for S. We can keep track of S by using the symbol table to map a special identifier **break** to the node for the enclosing statement S. This approach will also handle labeled break-statements in Java, since the symbol table can be used to map the label to the syntax-tree node for the enclosing construct.

Alternatively, instead of using the symbol table to access the node for S, we can put a pointer to $S.nextlist$ in the symbol table. Now, when a break-statement is reached, we generate an unfilled jump, look up *nextlist* through the symbol table, and add the jump to the list, where it will be backpatched as discussed in Section 6.7.3.

Continue-statements can be handled in a manner analogous to the break-statement. The main difference between the two is that the target of the generated jump is different.

6.7.5 Exercises for Section 6.7

Exercise 6.7.1: Using the translation of Fig. 6.43, translate each of the following expressions. Show the true and false lists for each subexpression. You may assume the address of the first instruction generated is 100.

a) `a==b && (c==d || e==f)`

b) `(a==b || c==d) || e==f`

c) `(a==b && c==d) && e==f`

Exercise 6.7.2: In Fig. 6.47(a) is the outline of a program, and Fig. 6.47(b) sketches the structure of the generated three-address code, using the backpatching translation of Fig. 6.46. Here, i_1 through i_8 are the labels of the generated instructions that begin each of the "Code" sections. When we implement this translation, we maintain, for each boolean expression E, two lists of places in the code for E, which we denote by $E.true$ and $E.false$. The places on list $E.true$ are those places where we eventually put the label of the statement to which control must flow whenever E is true; $E.false$ similarly lists the places where we put the label that control flows to when E is found to be false. Also, we maintain for each statement S, a list of places where we must put the label to which control flows when S is finished. Give the value (one of i_1 through i_8) that eventually replaces each place on each of the following lists:

(a) $E_3.false$ (b) $S_2.next$ (c) $E_4.false$ (d) $S_1.next$ (e) $E_2.true$

```
while ($E_1$) {
    if ($E_2$)
        while ($E_3$)
            $S_1$;
    else {
        if ($E_4$)
            $S_2$;
        $S_3$
    }
}
```

(a)

```
$i_1$: Code for $E_1$
$i_2$: Code for $E_2$
$i_3$: Code for $E_3$
$i_4$: Code for $S_1$
$i_5$: Code for $E_4$
$i_6$: Code for $S_2$
$i_7$: Code for $S_3$
$i_8$: ···
```

(b)

Figure 6.47: Control-flow structure of program for Exercise 6.7.2

Exercise 6.7.3: When performing the translation of Fig. 6.47 using the scheme of Fig. 6.46, we create lists *S.next* for each statement, starting with the assignment-statements S_1, S_2, and S_3, and proceeding to progressively larger if-statements, if-else-statements, while-statements, and statement blocks. There are five constructed statements of this type in Fig. 6.47:

S_4: **while** (E_3) S_1.

S_5: **if** (E_4) S_2.

S_6: The block consisting of S_5 and S_3.

S_7: The statement **if** S_4 **else** S_6.

S_8: The entire program.

For each of these constructed statements, there is a rule that allows us to construct $S_i.next$ in terms of other $S_j.next$ lists, and the lists $E_k.true$ and $E_k.false$ for the expressions in the program. Give the rules for

(a) $S_4.next$ (b) $S_5.next$ (c) $S_6.next$ (d) $S_7.next$ (e) $S_8.next$

6.8 Switch-Statements

The "switch" or "case" statement is available in a variety of languages. Our switch-statement syntax is shown in Fig. 6.48. There is a selector expression E, which is to be evaluated, followed by n constant values $V_1, V_2, \cdots, V_n$ that the expression might take, perhaps including a *default* "value," which always matches the expression if no other value does.

```
switch ( $E$ ) {
        case $V_1$: $S_1$
        case $V_2$: $S_2$
                ...
        case $V_{n-1}$: $S_{n-1}$
        default: $S_n$
}
```

Figure 6.48: Switch-statement syntax

6.8.1 Translation of Switch-Statements

The intended translation of a switch is code to:

1. Evaluate the expression E.

2. Find the value V_j in the list of cases that is the same as the value of the expression. Recall that the default value matches the expression if none of the values explicitly mentioned in cases does.

3. Execute the statement S_j associated with the value found.

Step (2) is an n-way branch, which can be implemented in one of several ways. If the number of cases is small, say 10 at most, then it is reasonable to use a sequence of conditional jumps, each of which tests for an individual value and transfers to the code for the corresponding statement.

A compact way to implement this sequence of conditional jumps is to create a table of pairs, each pair consisting of a value and a label for the corresponding statement's code. The value of the expression itself, paired with the label for the default statement is placed at the end of the table at run time. A simple loop generated by the compiler compares the value of the expression with each value in the table, being assured that if no other match is found, the last (default) entry is sure to match.

If the number of values exceeds 10 or so, it is more efficient to construct a hash table for the values, with the labels of the various statements as entries. If no entry for the value possessed by the switch expression is found, a jump to the default statement is generated.

There is a common special case that can be implemented even more efficiently than by an n-way branch. If the values all lie in some small range, say *min* to *max*, and the number of different values is a reasonable fraction of $max - min$, then we can construct an array of $max - min$ "buckets," where bucket $j - min$ contains the label of the statement with value j; any bucket that would otherwise remain unfilled contains the default label.

To perform the switch, evaluate the expression to obtain the value j; check that it is in the range *min* to *max* and transfer indirectly to the table entry at offset $j - min$. For example, if the expression is of type character, a table of, say, 128 entries (depending on the character set) may be created and transferred through with no range testing.

6.8.2 Syntax-Directed Translation of Switch-Statements

The intermediate code in Fig. 6.49 is a convenient translation of the switch-statement in Fig. 6.48. The tests all appear at the end so that a simple code generator can recognize the multiway branch and generate efficient code for it, using the most appropriate implementation suggested at the beginning of this section.

```
           code to evaluate $E$ into t
           goto test
$L_1$:     code for $S_1$
           goto next
$L_2$:     code for $S_2$
           goto next
           ...
$L_{n-1}$: code for $S_{n-1}$
           goto next
$L_n$:     code for $S_n$
           goto next
test:      if t = $V_1$ goto $L_1$
           if t = $V_2$ goto $L_2$
           ...
           if t = $V_{n-1}$ goto $L_{n-1}$
           goto $L_n$
next:
```

Figure 6.49: Translation of a switch-statement

The more straightforward sequence shown in Fig. 6.50 would require the compiler to do extensive analysis to find the most efficient implementation. Note that it is inconvenient in a one-pass compiler to place the branching statements at the beginning, because the compiler could not then emit code for each of the statements S_i as it saw them.

To translate into the form of Fig. 6.49, when we see the keyword **switch**, we generate two new labels `test` and `next`, and a new temporary t. Then, as we parse the expression E, we generate code to evaluate E into t. After processing E, we generate the jump `goto test`.

Then, as we see each **case** keyword, we create a new label L_i and enter it into the symbol table. We place in a queue, used only to store cases, a value-label pair consisting of the value V_i of the case constant and L_i (or a pointer to the symbol-table entry for L_i). We process each statement **case** V_i: S_i by emitting the label L_i attached to the code for S_i, followed by the jump `goto next`.

```
          code to evaluate $E$ into t
          if t != $V_1$ goto $L_1$
          code for $S_1$
          goto next
$L_1$:       if t != $V_2$ goto $L_2$
          code for $S_2$
          goto next
$L_2$:
          ...
$L_{n-2}$:   if t != $V_{n-1}$ goto $L_{n-1}$
          code for $S_{n-1}$
          goto next
$L_{n-1}$:   code for $S_n$
next:
```

Figure 6.50: Another translation of a switch statement

When the end of the switch is found, we are ready to generate the code for the n-way branch. Reading the queue of value-label pairs, we can generate a sequence of three-address statements of the form shown in Fig. 6.51. There, t is the temporary holding the value of the selector expression E, and L_n is the label for the default statement.

```
case t $V_1$ $L_1$
case t $V_2$ $L_2$
...
case t $V_{n-1}$ $L_{n-1}$
case t t $L_n$
label next
```

Figure 6.51: Case three-address-code instructions used to translate a switch-statement

The `case t` V_i L_i instruction is a synonym for `if t` $= V_i$ `goto` L_i in Fig. 6.49, but the `case` instruction is easier for the final code generator to detect as a candidate for special treatment. At the code-generation phase, these sequences of `case` statements can be translated into an n-way branch of the most efficient type, depending on how many there are and whether the values fall into a small range.

6.8.3 Exercises for Section 6.8

! **Exercise 6.8.1:** In order to translate a switch-statement into a sequence of `case`-statements as in Fig. 6.51, the translator needs to create the list of value-label pairs, as it processes the source code for the switch. We can do so, using an additional translation that accumulates just the pairs. Sketch a syntax-direction definition that produces the list of pairs, while also emitting code for the statements S_i that are the actions for each case.

6.9 Intermediate Code for Procedures

Procedures and their implementation will be discussed at length in Chapter 7, along with the run-time management of storage for names. We use the term function in this section for a procedure that returns a value. We briefly discuss function declarations and three-address code for function calls. In three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by the call itself. For simplicity, we assume that parameters are passed by value; parameter-passing methods are discussed in Section 1.6.6.

Example 6.25: Suppose that a is an array of integers, and that f is a function from integers to integers. Then, the assignment

```
n = f(a[i]);
```

might translate into the following three-address code:

```
1)  t1 = i * 4
2)  t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

The first two lines compute the value of the expression a[i] into temporary t_2, as discussed in Section 6.4. Line 3 makes t_2 an actual parameter for the call on line 4 of f with one parameter. Line 5 assigns the value returned by the function call to t_3. Line 6 assigns the returned value to n. □

The productions in Fig. 6.52 allow function definitions and function calls. (The syntax generates unwanted commas after the last parameter, but is good enough for illustrating translation.) Nonterminals D and T generate declarations and types, respectively, as in Section 6.3. A function definition generated by D consists of keyword **define**, a return type, the function name, formal parameters in parentheses and a function body consisting of a statement. Nonterminal F generates zero or more formal parameters, where a formal parameter consists of a type followed by an identifier. Nonterminals S and E generate statements and expressions, respectively. The production for S adds a statement that returns the value of an expression. The production for E adds function calls, with actual parameters generated by A. An actual parameter is an expression.

$$\begin{array}{lcl}
D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \} \\
F & \rightarrow & \epsilon \mid T\ \textbf{id}\ ,\ F \\
S & \rightarrow & \textbf{return}\ E\ ; \\
E & \rightarrow & \textbf{id}\ (\ A\ ) \\
A & \rightarrow & \epsilon \mid E\ ,\ A
\end{array}$$

Figure 6.52: Adding functions to the source language

Function definitions and function calls can be translated using concepts that have already been introduced in this chapter.

- *Function types.* The type of a function must encode the return type and the types of the formal parameters. Let *void* be a special type that represents no parameter or no return type. The type of a function *pop*() that returns an integer is therefore "function from *void* to *integer*." Function types can be represented by using a constructor *fun* applied to the return type and an ordered list of types for the parameters.

- *Symbol tables.* Let s be the top symbol table when the function definition is reached. The function name is entered into s for use in the rest of the program. The formal parameters of a function can be handled in analogy with field names in a record (see Fig. 6.18. In the production for D, after seeing **define** and the function name, we push s and set up a new symbol table

  $$\mathit{Env.push}(\mathit{top});\quad \mathit{top} = \textbf{new}\ \mathit{Env}(\mathit{top});$$

  Call the new symbol table, t. Note that *top* is passed as a parameter in **new** *Env*(*top*), so the new symbol table t can be linked to the previous one, s. The new table t is used to translate the function body. We revert to the previous symbol table s after the function body is translated.

- *Type checking.* Within expressions, a function is treated like any other operator. The discussion of type checking in Section 6.5.2 therefore carries over, including the rules for coercions. For example, if f is a function with a parameter of type real, then the integer 2 is coerced to a real in the call $f(2)$.

- *Function calls.* When generating three-address instructions for a function call $\textbf{id}(E, E, \ldots, E)$, it is sufficient to generate the three-address instructions for evaluating or reducing the parameters E to addresses, followed by a `param` instruction for each parameter. If we do not want to mix the parameter-evaluating instructions with the *param* instructions, the attribute $E.\mathit{addr}$ for each expression E can be saved in a data structure such as a queue. Once all the expressions are translated, the `param` instructions can be generated as the queue is emptied.

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to good code for procedure calls and returns. The run-time routines that handle procedure parameter passing, calls, and returns are part of the run-time support package. Mechanisms for run-time support are discussed in Chapter 7.

6.10 Summary of Chapter 6

The techniques in this chapter can be combined to build a simple compiler front end, like the one in Appendix A. The front end can be built incrementally:

- *Pick an intermediate representation*: An intermediate representation is typically some combination of a graphical notation and three-address code. As in syntax trees, a node in a graphical notation represents a construct; the children of a node represent its subconstructs. Three address code takes its name from instructions of the form $x = y$ **op** z, with at most one operator per instruction. There are additional instructions for control flow.

- *Translate expressions*: Expressions with built-up operations can be unwound into a sequence of individual operations by attaching actions to each production of the form $E \rightarrow E_1$ **op** E_2. The action either creates a node for E with the nodes for E_1 and E_2 as children, or it generates a three-address instruction that applies **op** to the addresses for E_1 and E_2 and puts the result into a new temporary name, which becomes the address for E.

- *Check types*: The type of an expression E_1 **op** E_2 is determined by the operator **op** and the types of E_1 and E_2. A coercion is an implicit type conversion, such as from *integer* to *float*. Intermediate code contains explicit type conversions to ensure an exact match between operand types and the types expected by an operator.

- *Use a symbol table to implement declarations*: A declaration specifies the type of a name. The width of a type is the amount of storage needed for a name with that type. Using widths, the relative address of a name at run time can be computed as an offset from the start of a data area. The type and relative address of a name are put into the symbol table due to a declaration, so the translator can subsequently get them when the name appears in an expression.

- *Flatten arrays*: For quick access, array elements are stored in consecutive locations. Arrays of arrays are flattened so they can be treated as a one-dimensional array of individual elements. The type of an array is used to calculate the address of an array element relative to the base of the array.

- *Generate jumping code for boolean expressions*: In short-circuit or jumping code, the value of a boolean expression is implicit in the position reached in the code. Jumping code is useful because a boolean expression B is typically used for control flow, as in **if** (B) S. Boolean values can be computed by jumping to `t = true` or `t = false`, as appropriate, where `t` is a temporary name. Using labels for jumps, a boolean expression can be translated by inheriting labels corresponding to its true and false exits. The constants *true* and *false* translate into a jump to the true and false exits, respectively.

- *Implement statements using control flow*: Statements can be translated by inheriting a label *next*, where *next* marks the first instruction after the code for this statement. The conditional $S \rightarrow$ **if** (B) S_1 can be translated by attaching a new label marking the beginning of the code for S_1 and passing the new label and $S.next$ for the true and false exits, respectively, of B.

- *Alternatively, use backpatching*: Backpatching is a technique for generating code for boolean expressions and statements in one pass. The idea is to maintain lists of incomplete jumps, where all the jump instructions on a list have the same target. When the target becomes known, all the instructions on its list are completed by filling in the target.

- *Implement records*: Field names in a record or class can be treated as a sequence of declarations. A record type encodes the types and relative addresses of the fields. A symbol table object can be used for this purpose.

6.11 References for Chapter 6

Most of the techniques in this chapter stem from the flurry of design and implementation activity around Algol 60. Syntax-directed translation into intermediate code was well established by the time Pascal [11] and C [6, 9] were created.

UNCOL (for Universal Compiler Oriented Language) is a mythical universal intermediate language, sought since the mid 1950's. Given an UNCOL, compilers could be constructed by hooking a front end for a given source language with a back end for a given target language [10]. The bootstrapping techniques given in the report [10] are routinely used to retarget compilers.

The UNCOL ideal of mixing and matching front ends with back ends has been approached in a number of ways. A retargetable compiler consists of one front end that can be put together with several back ends to implement a given language on several machines. Neliac was an early example of a language with a retargetable compiler [5] written in its own language. Another approach is to retrofit a front end for a new language onto an existing compiler. Feldman [2] describes the addition of a Fortran 77 front end to the C compilers [6] and [9]. GCC, the GNU Compiler Collection [3], supports front ends for C, C++, Objective-C, Fortran, Java, and Ada.

Value numbers and their implementation by hashing are from Ershov [1].

The use of type information to improve the security of Java bytecodes is described by Gosling [4].

Type inference by using unification to solve sets of equations has been rediscovered several times; its application to ML is described by Milner [7]. See Pierce [8] for a comprehensive treatment of types.

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

2. Feldman, S. I., "Implementation of a portable Fortran 77 compiler using modern tools," *ACM SIGPLAN Notices* **14**:8 (1979), pp. 98–106

3. GCC home page `http://gcc.gnu.org/`, Free Software Foundation.

4. Gosling, J., "Java intermediate bytecodes," *Proc. ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111–118.

5. Huskey, H. D., M. H. Halstead, and R. McArthur, "Neliac — a dialect of Algol," *Comm. ACM* **3**:8 (1960), pp. 463–468.

6. Johnson, S. C., "A tour through the portable C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

7. Milner, R., "A theory of type polymorphism in programming," *J. Computer and System Sciences* **17**:3 (1978), pp. 348–375.

8. Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, Mass., 2002.

9. Ritchie, D. M., "A tour through the UNIX C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

10. Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1**:8 (1958), pp. 12–18. Part 2: **1**:9 (1958), pp. 9–15. Report of the Share Ad-Hoc committee on Universal Languages.

11. Wirth, N. "The design of a Pascal compiler," *Software—Practice and Experience* **1**:4 (1971), pp. 309–333.