8.6 BACKPATCHING

The easiest way to implement the syntax-directed definitions in Section 8.4 is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations given in the definition. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. We can get around this problem by generating a series of branching statements with the targets of the jumps temporarily left unspecified. Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels *backpatching*.

In this section, we show how 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 8.4, except for the manner in which we generate labels. For specificity, we generate quadruples into a quadruple array. Labels will be indices into this array. To manipulate lists of labels, we use three functions:

1. *makelist*(i) creates a new list containing only i, an index into the array of quadruples; *makelist* returns a pointer to the list it has made.
2. *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. *backpatch*(p, i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. We insert a marker nonterminal M into the grammar to cause a semantic action to pick up, at appropriate times, the index of the next quadruple to be generated. The grammar we use is the following:

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) $\quad | \ E_1$ **and** $M\ E_2$
(3) $\quad | \ $ **not** E_1
(4) $\quad | \ (\ E_1\ )$
(5) $\quad | \ \mathbf{id}_1$ **relop** $\mathbf{id}_2$
(6) $\quad | \ $ **true**
(7) $\quad | \ $ **false**
(8) $M \rightarrow \epsilon$

Synthesized attributes *truelist* and *falselist* of nonterminal E are used to generate jumping code for boolean expressions. As code is generated for E, 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 *E.truelist* and *E.falselist*, as appropriate.

The semantic actions reflect the considerations mentioned above. Consider the production $E \rightarrow E_1$ **and** $M\ E_2$. If E_1 is false, then E is also false, so the statements on E_1.*falselist* become part of *E.falselist*. If E_1 is true, however, we must next test E_2, so the target for the statements E_1.*truelist* must be the beginning of the code generated for E_2. This target is obtained using the marker nonterminal M. Attribute *M.quad* records the number of the first statement of E_2.*code*. With the production $M \rightarrow \epsilon$ we associate the semantic action

$$\{\ M.quad := nextquad\ \}$$

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the E_1.*truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** $M\ E_2$. The translation scheme is as follows.

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* (E_1.*falselist*, *M.quad*);
E.truelist := *merge*(E_1.*truelist*, E_2.*truelist*);
E.falselist := E_2.*falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* (E_1.*truelist*, *M.quad*);
E.truelist := E_2.*truelist*;
E.falselist := *merge*(E_1.*falselist*, E_2.*falselist*) }

(3) $E \rightarrow$ **not** E_1 { *E.truelist* := E_1.*falselist*;
E.falselist := E_1.*truelist* }

(4) $E \rightarrow$ (E_1) { *E.truelist* := E_1.*truelist*;
E.falselist := E_1.*falselist* }

(5) $E \rightarrow$ **id**$_1$ **relop** **id**$_2$ { *E.truelist* := *makelist*(*nextquad*);
E.falselist := *makelist*(*nextquad* + 1);
emit('`if`' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* '`goto _`')
emit('`goto _`') }

(6) $E \rightarrow$ **true** { *E.truelist* := *makelist*(*nextquad*);
emit('`goto _`') }

(7) $E \rightarrow$ **false** { *E.falselist* := *makelist*(*nextquad*);
emit('`goto _`') }

(8) $M \rightarrow \epsilon$ { *M.quad* := *nextquad* }

For simplicity, semantic action (5) generates two statements, a conditional goto and an unconditional one. Neither has its target filled in. The index of the first generated statement is made into a list, and *E.truelist* is given a pointer to that list. The second generated statement `goto _` is also made into a list and given to *E.falselist*.

Example 8.6. Consider again the expression `a<b or c<d and e<f`. An annotated parse tree is shown in Fig. 8.29. 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 `a < b` to *E* by production (5), the two quadruples

```
100: if a < b goto _
101: goto _
```

are generated. (We again arbitrarily start statement numbers at 100.) The marker nonterminal *M* in the production $E \rightarrow E_1$ **or** $M\ E_2$ records the value of *nextquad*, which at this time is 102. The reduction of `c < d` to *E* by production (5) generates the quadruples

```
102: if c < d goto _
103: goto _
```

We have now seen E_1 in the production $E \rightarrow E_1$ **and** $M\ E_2$. The marker nonterminal in this production records the current value of *nextquad*, which is now 104. Reducing `e < f` into *E* by production (5) generates

```
104: if e < f goto _
105: goto _
```

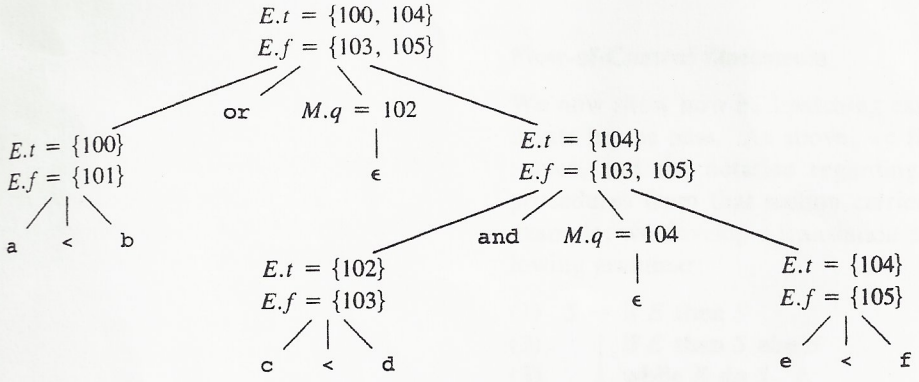

Fig. 8.29. Annotated parse tree for `a<b or c<d and e<f`.

We now reduce by $E \rightarrow E_1$ **and** $M\ E_2$. The corresponding semantic action calls *backpatch*({102},104) where {102} as argument denotes a pointer to the list containing only 102, that list being the one pointed to by E_1.*truelist*. This call to *backpatch* fills in 104 in statement 102. The six statements generated so far are thus:

```
100:  if a < b goto _
101:  goto _
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

The semantic action associated with the final reduction by $E \rightarrow E_1$ **or** $M\ E_2$ calls *backpatch*({101},102) which leaves the statements looking like:

```
100:  if a < b goto _
101:  goto 102
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

The entire expression is true if and only if the goto's of statements 100 or 104 are reached, and is false if and only if the goto's of statements 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. □

Flow-of-Control Statements

We now show how backpatching can be used to translate flow-of-control statements in one pass. As above, we fix our attention on the generation of quadruples, and the notation regarding translation field names and list-handling procedures from that section carries over to this section as well. As a larger example, we develop a translation scheme for statements generated by the following grammar:

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L$; S
(7) | S

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. Note that there must be other productions, such as those for assignment statements. The productions given, however, will be sufficient to illustrate the techniques used to translate flow-of-control statements.

We use the same structure of code for if-then, if-then-else, and while-do statements as in Section 8.4. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. If that is not true, an explicit jump must be provided.

Our general approach will be to fill in the jumps out of statements when their targets are found. Not only do boolean expressions need two lists of jumps that occur when the expression is true and when it is false, but statements also need lists of jumps (given by attribute *nextlist*) to the code that follows them in the execution sequence.

Scheme to Implement the Translation

We now describe a syntax-directed translation scheme to generate translations for the flow-of-control constructs given above. The nonterminal E has two attributes *E.truelist* and *E.falselist*, as above. L and S each also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L.nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

In the code layout for $S \rightarrow$ **while** E **do** S_1 in Fig. 8.22(c), there are labels *S.begin* and *E.true* marking the beginning of the code for the complete statement S and the body S_1. The two occurrences of the marker nonterminal M in the following production record the quadruple numbers of these positions:

$S \rightarrow$ **while** M_1 E **do** M_2 S_1

Again, the only production for M is $M \rightarrow \epsilon$ with an action setting attribute *M.quad* to the number of the next quadruple. After the body S_1 of the while statement is executed, control flows to the beginning. Therefore, when we reduce **while** M_1 E **do** M_2 S_1 to S, we backpatch S_1.*nextlist* to make all targets on that list be M_1.*quad*. An explicit jump to the beginning of the code for E is appended after the code for S_1 because control may also "fall out the bottom." *E.truelist* is backpatched to go to the beginning of S_1 by making jumps on *E.truelist* go to M_2.*quad*.

A more compelling argument for using *S.nextlist* and *L.nextlist* comes when code is generated for the conditional statement **if** E **then** 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 introduce 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 quadruple number of the statement `goto _` that is generated by the semantic rule for N. We now give the semantic rules for the revised grammar.

(1) $S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2

{ *backpatch*(*E.truelist*, M_1.*quad*);
backpatch(*E.falselist*, M_2.*quad*);
S.nextlist := *merge*(*S*$_1$.*nextlist*, *merge*(*N.nextlist*, S_2.*nextlist*)) }

We backpatch the jumps when E is true to the quadruple M_1.*quad*, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E 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.

(2) $N \rightarrow \epsilon$ { *N.nextlist* := *makelist*(*nextquad*);
emit('`goto _`') }

(3) $M \rightarrow \epsilon$ { *M.quad* := *nextquad* }

(4) $S \rightarrow$ **if** E **then** M S_1 { *backpatch*(*E.truelist*, *M.quad*);
S.nextlist := *merge*(*E.falselist*, *S*$_1$.*nextlist*) }

(5) $S \rightarrow$ **while** M_1 E **do** M_2 S_1 { *backpatch*(*S*$_1$.*nextlist*, M_1.*quad*);
backpatch(*E.truelist*, M_2.*quad*);
S.nextlist := *E.falselist*
emit('`goto`' M_1.*quad*) }

(6) $S \rightarrow$ **begin** L **end** { *S.nextlist* := *L.nextlist* }

(7) $S \rightarrow A$ { *S.nextlist* := **nil** }

The assignment *S.nextlist* := **nil** initializes *S.nextlist* to an empty list.

(8) $L \rightarrow L_1$; M S { *backpatch*(*L*$_1$.*nextlist*, *M.quad*);
L.nextlist := *S.nextlist* }

The statement following 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 $M.quad$.

(9) $L \rightarrow S$ { $L.nextlist := S.nextlist$ }

Note that no new quadruples are generated anywhere in these semantic rules except for rules (2) and (5). All other code is generated by the semantic actions associated with assignment statements and expressions. What the flow of control does is cause the proper backpatching so that the assignments and boolean expression evaluations will connect properly.

Labels and Gotos

The most elementary programming language construct for changing the flow of control in a program is the label and goto. When a compiler encounters a statement like `goto L`, it must check that there is exactly one statement with label `L` in the scope of this `goto` statement. If the label has already appeared, either in a label-declaration statement or as the label of some source statement, then the symbol table will have an entry giving the compiler-generated label for the first three-address instruction associated with the source statement labeled `L`. For the translation we generate a `goto` three-address statement with that compiler-generated label as target.

When a label `L` is encountered for the first time in the source program, either in a declaration or as the target of a forward goto, we enter `L` into the symbol table and generate a symbolic label for `L`.