!! **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. $merge(p_1, p_2)$ 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 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:

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

The translation scheme is in Fig. 6.43.

1) $B \rightarrow B_1 \mid\mid M\, B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1 \,\&\&\, M\, B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \rightarrow !\, B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow (\, B_1\,)$ { $B.truelist = B_1.truelist$;
$B.falselist = B_1.falselist$; }

5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto _`'$)$;
$emit($'`goto _`'$)$; }

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

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

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

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production $B \rightarrow B_1 \mid\mid M\, B_2$. If B_1 is true, then B is also true, so the jumps on $B_1.truelist$ become part of $B.truelist$. 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.

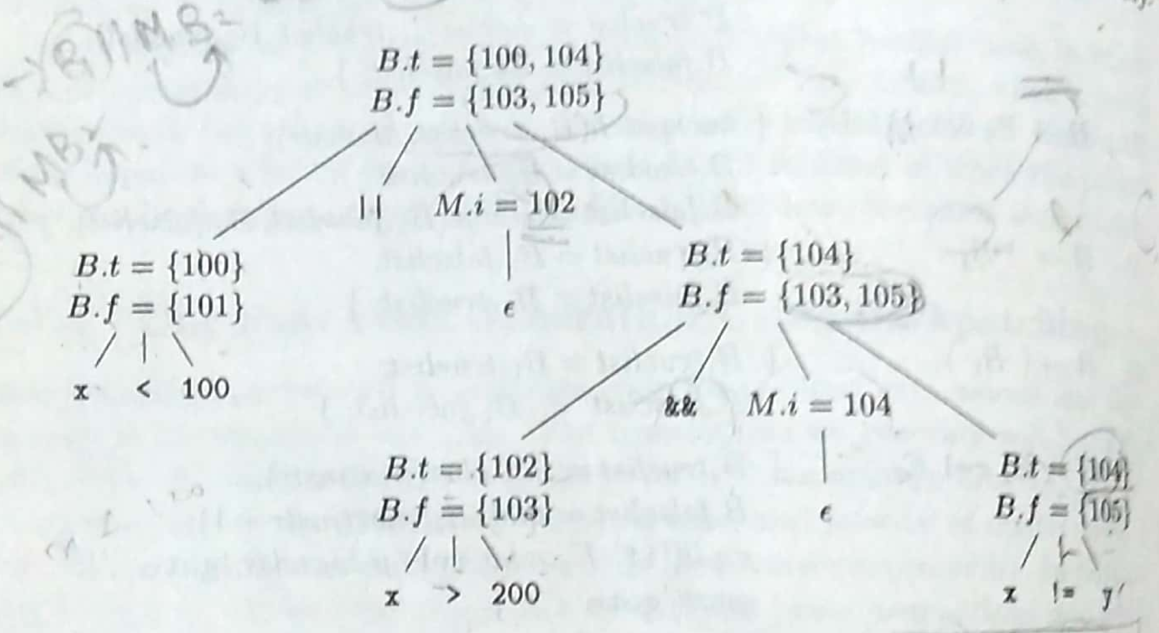

Figure 6.44: Annotated parse tree for $x < 100 \ || \ x > 200 \ \&\& \ x \ != \ y$

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

$$B \rightarrow B_1 \mid\mid M\, B_2$$

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 $x \mathrel{!}= y$ 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 $B \rightarrow B_1 \mid\mid M\, B_2$ 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