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?*

One-Pass Code Generation Using Backpatching

- Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass. 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.
- 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.

  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(pi,p2) concatenates the lists pointed to by p1 and p2, 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.

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 gram-mar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

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

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

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

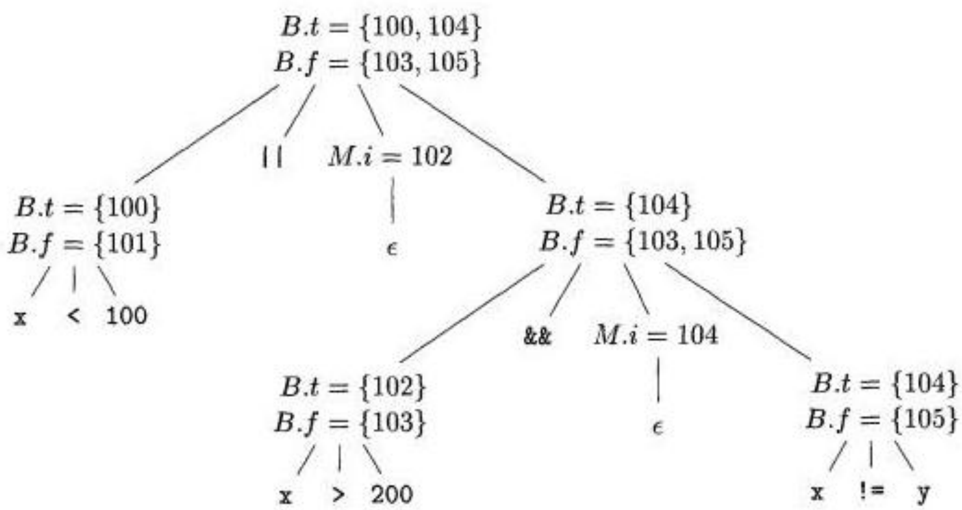
5) $B \rightarrow E_1\ \mathbf{rel}\ E_2$ $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$emit('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_');$
$emit('\texttt{goto}\ \_');$ $\}$

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

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

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

- Consider semantic action for the production B -> B1 || M B2. If Bx is true, then B is also true, so the jumps on Bi.truelist become part of B.truelist.
- If Bi is false, however, we must next test B2, so the target for the jumps B>i .falselist must be the beginning of the code generated for B2.
- 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 B2 code starts being generated.
- The variable nextinstr holds the index of the next instruction to follow. This value will be backpatched onto the Bi.falselist (i.e., each instruction on the list B1.falselist will receive M.instr as its target label) when we have seen the remainder of the production B ->• B1 || M B2.



Flow-of-Control Statements

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

$$S \rightarrow \textbf{if}\,(\,B\,)\,S \;\mid\; \textbf{if}\,(\,B\,)\,S\;\textbf{else}\;S \;\mid\; \textbf{while}\,(\,B\,)\,S \;\mid\; \{\,L\,\} \mid A\;;$$

$$L \rightarrow L\,S \;\mid\; S$$

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.

- 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 from one instruction to the next.