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

3. $\text{backpatch}(p, i)$ 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:

$$B \rightarrow B_1 \mid \mid M \mid B_1 \& \& M \mid \mid B_2 \mid ! B_1 \mid (B_1) \mid E_1 \text{ rel } E_2 \mid \text{true} \mid \text{false}$$

$$M \rightarrow \epsilon$$

The translation scheme is in Fig. 6.43.

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

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

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

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

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

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

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

8) $M \rightarrow \epsilon$  
   \{ $M.\text{Instr} = \text{nextinstr}$ \}

Figure 6.43: Translation scheme for boolean expressions

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

To obtain that instruction index, we associate with the production M → ε the semantic action

\[
\{ \text{M.instr = nextinstr; } \}
\]

The variable nextinstr holds the index of the next instruction to follow. This value will be backpatched onto the B1.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.

Semantic action (2) for B → B1 && M B2 is similar to (1). Action (3) for B → !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.

![Annotated parse tree for x < 100 || x > 200 && x != y](image)

**Figure 6.44:** Annotated parse tree for \( x < 100 \) || \( x > 200 \) && \( x \neq y \)

**Example 6.24:** Consider again the expression

\[ x < 100 \text{ || } x > 200 \text{ && } x \neq y \]

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

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

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

$$B \rightarrow B_1 \land M B_2$$

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 $B \rightarrow B_1 \land M B_2$. 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 $B \rightarrow B_1 \mid M B_2$ 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:

\[
S \rightarrow \text{if } (B) S \mid \text{if } (B) S \text{ else } S \mid \text{while } (B) S \mid \{ L \} \mid A ; \\
L \rightarrow L S \mid S
\]

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.

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 \textbf{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 \textit{begin}
and B.true, respectively.
Figure 6.46: Translation of statements
marker with production $N \rightarrow \epsilon$. $N$ has attribute $N\.nextlist$, which will be a list consisting of the instruction number of the jump $\text{goto -}$ 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

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

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

$$L \rightarrow L_1 \text{ } M \text{ } S$$

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 $\text{goto } L$ 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:

```java
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, i1 through i8 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 i1 through i8) that eventually replaces each place on each of the following lists:

(a) E3.false  (b) S2.next  (c) E4.false  (d) S1.next  (e) E2.true
while \((E_1)\) {
    \text{\textbf{if} } (E_2) \text{ \textbf{while} } (E_3) \text{ \textbf{else} } \{
        \text{\textbf{if} } (E_4) \text{ \textbf{else} } S_2;
        S_3
    \}
}

(a) \hspace{2cm} (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 \(S_.\text{next}\) 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:

\(S_4:\) \text{ while } (E_3) S_1.

\(S_5:\) \text{ if } (E_4) S_2.

\(S_6:\) The block consisting of \(S_5\) and \(S_3\).

\(S_7:\) The statement \text{ if } \text{ \textbf{S}4 \textbf{else} } S_6.

\(S_8:\) The entire program.

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

(a) \(S_4\_\text{next}\) (b) \(S_5\_\text{next}\) (c) \(S_6\_\text{next}\) (d) \(S_7\_\text{next}\) (e) \(S_8\_\text{next}\)

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 \(V_1, V_2, \ldots, V_n\) that the expression might take, perhaps including a \textit{default} "value," which always matches the expression if no other value does.