Backpatching

The syntax directed definition to generate three-address code is typically done in two passes. In the first pass the syntax tree is constructed and annotated with rules. In the second pass a depth first search of the tree is carried out to perform syntax directed translation. If a statement consists of multiple lines of instructions then the labels to branch may not be known in advance if SDD needs to be done in a single pass. To address this issue we introduce a technique called backpatching.

The fundamental behind the technique of backpatching is to generate a series of branching statements with the target of jumps unspecified in the first pass. In the second pass we put each statement in a list and fill them with appropriate true and false labels.

Functions to incorporate backpatching :

Backpatching technique is incorporated using three functions. Makelist(), merge() and backpatch() are the three functions carried out in two passes to generate code using backpatching.

- *makelist*(*i*) – This is used to create a new list containing three-address location *i*, and it returns a pointer to the list. This is the first function which is created to form a true / false list.
- *merge*(*p*1, *p*2) – This function concatenates lists pointed to by *p*1 and *p*2, returns a pointer to the concatenated list. This is used to assign the same true / false labels to more than one address.
- *backpatch*(*p*, *i*) – This function is used to insert '*i'* as the target label for each of the statements in the list pointed to by *p*. Using the information provided by this function labels are attached to all the statements.

Consider the Boolean expression "a < b or c < d and e < f". To generate three-address code for this, we have already incorporated semantic rules in the previous module. In backpatching the same code is generated in two passes. In the first pass, the following would be generated:

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

In the second pass, the same code is re-run to generate the true, false labels by incorporating short circuit information.

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

In this module, we will write semantic rules to generate three-address code based in two passes using the backpatching functions discussed already.

Boolean Expressions and Backpatching

The productions for the Boolean expressions are the same. To just generate line numbers and to incorporate backpatching we add new non-terminals as part of the production. This non-terminal just produces 'ε' and doesn't alter the string of the grammar. The semantic rules that incorporate backpatching are given in Table 26.1. A function nextquad() is used to generate the next line number for generating three-address code and that is the reason behind introducing the new non-terminal M.

Table 26.1 Semantic rules for incorporating Backpatching

Production	**Semantic Rule**	**Inference**
M → ε	{ M.quad := nextquad() }	The semantic rule associated with this variable helps in generating the next line number to generate three address code
E → E1 or M E2	{ backpatch(E_1.falselist, M.quad); E.truelist := merge(E_1.truelist, E_2.truelist); E.falselist := E_2.falselist }	Merge function concatenates the truelist of E1 and E2. If E1 is false we need to associate the false list of E1 with the next line number using M.quad. This line will contain the first instruction corresponding to E2 as this will be evaluated only if E1 is false. The expression E's false list will be E2's false list after incorporating short circuit

E → E1 and M E2	{ backpatch(E_1.truelist, M.quad); E.truelist := E_2.truelist; E.falselist := merge(E_1.falselist, E_2.falselist); }	Here as the operator is 'and', we merge the false list of E1 and E2's and assign as E's false list. The true list of E is E2's true list as we will be executing E2 only if E1 is true. To execute E2, we backpatch E1's true to the line number corresponding to E2's first instruction which is given by M.quad
E → not E1	{ E.truelist := E_1.falselist; E.falselist := E_1.truelist }	The false and true lists of E and E1 are reversed.
E → (E1)	{ E.truelist := E_1.truelist; E.falselist := E_1.falselist }	The false and true lists of E and E1 are the same as the parenthesis is just to prioritize the expression E1
E → id1 relop id2	{ E.truelist := makelist(nextquad()); E.falselist := makelist(nextquad() + 1); emit('if' id_1.place relop.op id_2.place 'goto _'); emit('goto _') }	The line numbers of truelist and falselist for E is considered as the next line number and its following line number. The code is generated using "emit" in a similar fashion as explained in the previous modules, with the only difference being the goto is left blank which will be backpatched later.
E→ true	{ E.truelist := makelist(nextquad()); E.falselist := nil; emit('goto _') }	Basic terminating production which will generate a goto blank which will be backpatched with truelist's number
E → false	{ E.falselist := makelist(nextquad()); E.truelist := nil; emit('goto _') }	Basic terminating production which will generate a goto blank which will be backpatched with falselist's number

Example 26.1 Consider the same example Boolean expression "a < b or c < d and e < f". The corresponding derivation tree would be as shown in figure 26.1

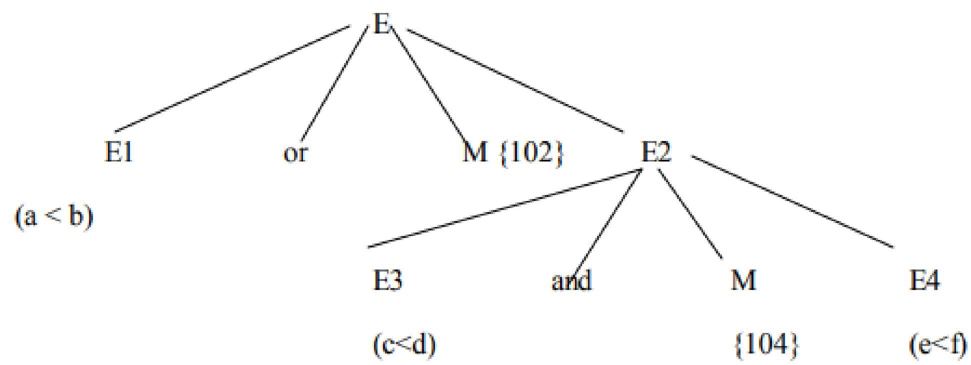

Figure 26.1 Example derivation tree

Consider the first instruction is to start at line number 100. The sequence of three-address code is given in Table 26.2

Table 26.2 Three-address code for the tree of figure 26.1

Line number	**Code**	**Truelist**	**Falselist**	**Inference**
100	if a<b goto ---	E1 – {100}	E1 – {101}	From the semantic rules for a Boolean expression E1 as given in Table 26.1, the two instructions are generated and the corresponding truelist and falselist is given as the line number and the following line number
101	goto -----			
102	if c < d goto -------	E3 – {102}	E3 – {103}	Using M.quad, 102 line number is generated. Using the same semantic rule, E3's truelist and false list are also generated
103	goto ---------			
104	if e < f goto ------	E4 – {104}	E4 – {105}	Using the M of the 'and' expression we generate 104 as the line number. Using the same semantic rule, E4's truelist and falselist is generated
105	goto -------------			
		E2 – {104}	E2 – {103, 105}	E2 is the 'and' of E3 and E4. Using the semantic rule, the false list of E2 is the merger of falselist of E3 and E4. The

Control flow statements and Backpatching

Control flow statements have been discussed in the previous module along with their semantic rules. As Boolean expressions are part of control flow statements, backpatching can be applied to control flow statements also. Consider the following grammar with productions for control flow statements.

S → if E the n S | if E then S else S | while E do S | begin L end| A

L → L ; S | S

Example of the statements could be a sequences of statements separated by ';'. S1; S2; S3; S4 ; S5; etc. The attributes of S is S.nextlist which will backpatch list for jumps to the next statement after *S* (or nil). Similarly the attributes of L is also *L*.nextlist which backpatch list for jumps to the next statement after *L* (or nil).

For example of the sequence of statements S1, S2, S3, S4, S5, etc.. we will have the code for S1 to S5 followed by the backpatch of each statement, to the statement following it.

100: Code for S1
200: Code for S2
300: Code for S3
400: Code for S4
500: Code for S5

The following backpatch will ensure that the sequence of statements are executed in the order.

backpatch(*S*1.nextlist, 200)
backpatch(*S*2.nextlist, 300)
backpatch(*S*3.nextlist, 400)
backpatch(*S*4.nextlist, 500)

Our aim would be to add semantic rules to handle such a scenario and other control flow statements. The semantic rules are given in Table 26.3. In this case also we use a dummy variable M to generate the next line number.

Production	**Semantic Rules**	Inference
$S \rightarrow A$	{ S.nextlist := nil }	This production is a termination production and hence there is no need for a backpatch
$S \rightarrow$ **begin** L **end**	{ S.nextlist := L.nextlist }	Both S and L has a nextlist attribute and they are set to the same. The statements between 'begin' and 'end' are run only once.
$S \rightarrow$ **if** E **then** M S_1	{ backpatch(E.truelist, M.quad); S.nextlist := merge(E.falselist, S_1.nextlist) }	The variable M produces ε and it indicates the same semantic rule as discussed in the table 26.1. If the expression is false, then the statement S1 need to be skipped. If Expression is true, then S1 should be executed. In both the scenarios, the statement that is available outside S1 need to be continued. To carry out this, the falselist of E and the nextlist of S1 are merged and that is assigned as S's nextlist. If expression is true then the statement S1 is to be

		executed. To incorporate this we backpatch the truelist of the expression to S1 which is done with the help of M.quad
$L \rightarrow L_1$; M S	{ backpatch(L_1.nextlist, M.quad); L.nextlist := S.nextlist; }	After executing L1, we need to execute S. To incorporate this we backpatch L1's nextlist with M.quad which corresponds to the statement comprising S. The next of L and S are same
$L \rightarrow S$	{ L.nextlist := S.nextlist; }	There is no backpatching and we simply say the nextlist of L and S are same
$S \rightarrow$ if E then M 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)) }	The expression is evaluated and if it is true S1 is to be executed and S2 if the statement is false. This is implemented by backpatching the truelist and falselist to M1.quad and M2.quad which is the beginning of statements S1, S2 respectively. After executing S1, S2 need to be skipped and the statement which is available after S needs to be executed. After executing S2 by skipping S1, we need to execute the statement outside the body of S. To incorporate this we use the symbol N to skip S2. We assign the nextlist of S as S1's next and S2's next.
$S \rightarrow$ 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 _') }	The variable M2 helps to go to go to statement S1 if the expression is true. If the expression is false then we need to go to the statement following S1 which is done as the nextlist of S the same as E's falselist. To incorporate continuation of the loop, M1 is used which is to come back after finishing

Generate the intermediate code for the following:

```
x = y + 2;
if x < y then x = x + y;
repeat
    y = y * 2;
    while x > 10 do x = x/2;
until x < y;
```