Module 25 – Control Flow statements and Boolean Expressions

In this module we learn to generate three address code for control flow statements. We will also try to incorporate short circuit information in Boolean expression's 3-address code generation. As the flow in control flow's statements involve Boolean expressions, we will look at the semantic rules for generating three-address code involving Boolean expressions with Control flow.

25.1 Control Flow statements

We have seen semantic rules that would help generate three-address code for declarations, array accesses, arithmetic expressions, Boolean expressions and to keep track of scope information. Thus, a context free grammar is used to define every programming construct and we write semantic rules based on this context free grammar to generate three-address code.

Control flow statements are essential components of programming languages that allows executing code based on a decision. The typically available control flow statements are if-then, if-then-else, while-do statement and do – while statements. The control flow statements have an expression that needs to be evaluated and based on the true or false value of the expression the appropriate branching is taken.

The context free grammar for defining control flow statements can be defined as

$$S \rightarrow \text{if } E \text{ then } S1 \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S1 \mid \text{do } S1 \text{ while } E$$

The LHS symbol 'S' stands for statement. The production defines a statement could be a simple "if Expression then Statement" or "if Expression then statement else alternate statement", "while Expression is true do the statements repeatedly" or do a statement repeatedly while the expression is true. In all these statements "E" corresponds to a Boolean expression or sometimes it could be an assignment statement. Thus to generate three-address code for a control flow statement, the first step is to generate code for the expression. This expression could be a sequence of expressions combined with relational and logical operators. Let us discuss each type of control flow statements and the semantic rules used for generating three-address code in the subsequent sections.

25.1.1 Three-address code for if-then statements

The schematic for generating three-address code for if-then statements is given in figure 25.1. As discussed already the expression E needs to be executed first and hence the code corresponding to E is generated first. The value of the expression is evaluated and if it is found to be "true" the statement corresponding to the body of the if-then is executed followed by the statement following the if-then statement. If the expression is false, the body of the if-then is skipped and directly we go to the statement that is following the if-then statement's body.

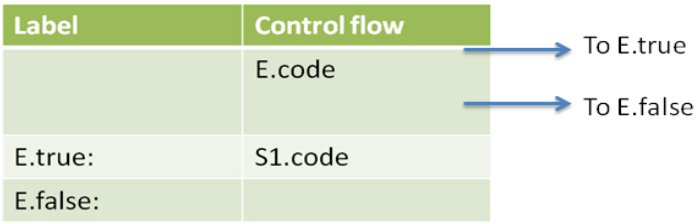


Figure 25.1 Schematic for three-address code for if-then statements

Figure 25.1 explains this schematic flow. The body of the if-then statement is given a label and is labeled as E.true and these statements need to be evaluated if the expression is evaluated to be true. The semantic rules to implement this sequence are given in Table 25.1. The expression E has attributes code, true and false labels. The attributes of the statement S are code and next. S1 in turn could have a if-then or if-then else etc and hence we simply leave it as S1.code rather than going into the details of S1.

Table 25.1 Semantic rules for three-address code for if-then statement

Production	**Semantic Rules**	**Inference**
S → if E then S1	E.true:= newlabel E.false := S.next S1.next := S.next S.code := E.code || gen (E.true':') || S1.code	We first generate a new address location E.true. If the expression E is false then control should go to the statement following the body of the if-then. Hence, we assign E.false as S.next. If the expression E is true the body of S, the statements following if-then block need to be evaluated and this is ensured by setting S1.next and S.next as same. Finally, the code corresponding to S is evaluated as E.code followed by the generation of E.true label followed by S1.code.

25.1.2 Three-address code for if-then-else statements

The schematic for generating three-address code for if-then statements is given in figure 25.2. As discussed already the expression E needs to be executed first and hence the code corresponding to E is generated first. The value of the expression is evaluated and if it is found to be "true" the

statement S1 corresponding to the body of the if-then is executed; the body of the else is skipped followed by the statement following the if-then-else statement. If the expression is false, the body of the else S2 is executed followed by the statement following the if-then-else statement's body.

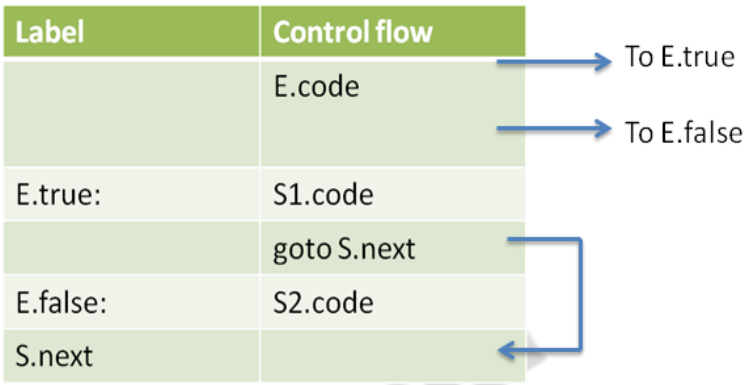


Figure 25.2 Schematic for three-address code for if-then-else statements

As in the earlier situation, S has two attributes, code and next. The body of S1 and S2 could in-turn contain multiple statements, so we leave it as S1.code and S2.code respectively. The semantic rules to implement the flow as given in figure 25.2 is discussed in Table 25.2

Table 25.2 Semantic rules for generating three-address code for if-then-else

Production	**Semantic Rules**	**Inference**
S → if E then S1 else S2	E.true := newlabel E.false := newlabel S1.next := S.next S2.next := S.next S.code := E.code || gen (E.true':') || S1.code || gen ('goto' S.next) || gen (E.false ':') || S2.code	Here we generate two new labels E.true and E.false which will be the labels for the statements beginning of S1 and S2 respectively. After executing S1, S2 need to be skipped and if S1 is skipped after executing S2, the statement corresponding to the next of S need to be executed. This is done as assigning S1 and S2's next as S.next. The code corresponding to S is E.code followed by E.true label generation, then S1.code then generating a goto(s.next), which is to skip S2's code and generating the E.false label followed by S2.code

25.1.3 Three-address code for while loops

The schematic for generating three-address code for statements involving 'while' loop is given in figure 25.3. As discussed already, even in this scenario, the expression E needs to be executed first and hence the code corresponding to E is generated first. The value of the expression is evaluated and if it is found to be "true" the statement S1 corresponding to the body of the 'while' is executed. The body of the 'while' will have options to change the variable involved in the expression and once again the expression is evaluated. If the expression is false, the statement following the while block is executed.

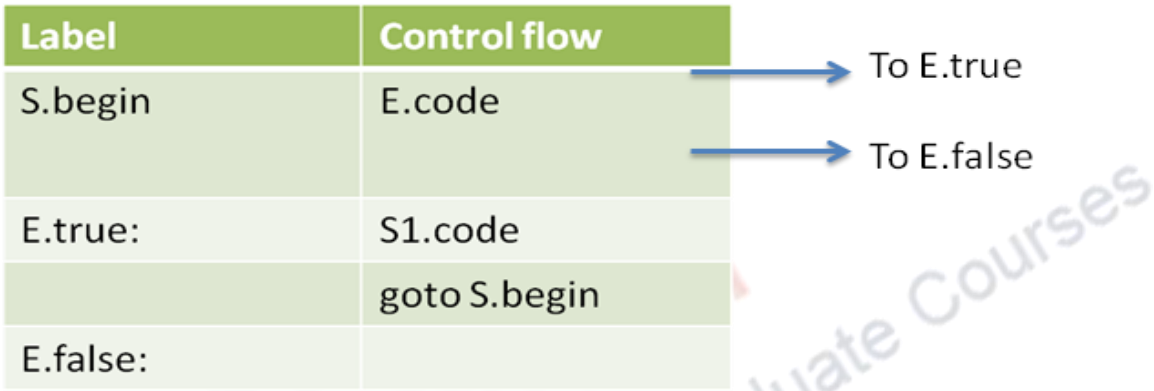


Figure 25.3 Schematic for while loop's control flow

Here one more attribute is used for the statement S – 'begin'. This begin points to the first line of the expression E which is part of the while block and is mandatory as the expression needs to be evaluated multiple times to iterate through the while loop. Hence, S.begin and E.true are two new labels and E.false is assigned as S.next as in the if-then scenario. The semantic rules are given in Table 25.3.

Table 25.3 Schematic rules for while loop

Production	Semantic Rules	Inference
S → while E do S1	S.begin := newlabel E.true := newlabel E.false := S.next S1.next := S.begin S.code := gen (S.begin ':') || E.code || gen (E.true':') || S1.code || gen ('goto' S.begin)	Two new labels S.begin which points to the expression's first line and E.true which points to the beginning of the statement S1 is generated. Expression's false should skip the body of the while and should go to the statement following the statement S and hence E.false is assigned S.next. The next of the statement S1 is assigned as S.begin as the exit from the while block if from

		S.begin only. Thus we first generate the label S.begin and then generate code for E, followed by the E.true label, then code for S1, then goto to S.begin.

25.1.4 Three-address code for do-while loops

The schematic for generating three-address code for statements involving 'do-while' loop is given in figure 25.4. In this scenario, the body of the statement S1 is executed first without considering the expression's true or false values. After running the body of the loop once, we check and execute the expression E. True code corresponding to E is generated then and if the value of the expression is found to be "true" the statement S1 corresponding to the body of the 'do-while' is executed. The body of the 'do-while' will have options to change the variable involved in the expression and once again the expression is evaluated. If the expression is false, the statement following the do-while block is executed.

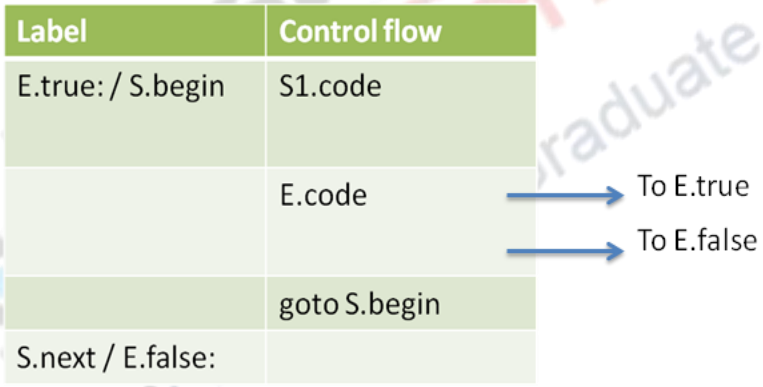


Figure 25.4 Schematic for do-while loop's three address code

In this scenario, we again generate the S.begin label to indicate the first line corresponding to S1.begin which is also the same as the E.true label. Table 25.4 indicates the semantic rules for the three-address code generation based on the schematic of figure 25.4

Production	**Semantic Rules**	**Inference**
S → do S1 while E	S.begin := newlabel E.true := S.begin E.false := S.next S.code := S1.code || E.code || gen (E.true ':') || gen ('goto' S.begin)	We first generate a new label as S.begin. E.true is also assigned as this S.begin. E.false is assigned as S.next. The code for S is given as generating S.begin label followed by code for S1, followed by code for E and generating goto to S.begin

The sections 25.1.1 to 25.1.4 discussed the various semantic rules for, if-then, if-then-else, while, do-while loops. A 'for' loop construct may be considered as a 'while' construct and this can be used to generate 3-address code

25.2 Control flow with Boolean expression

In the previous module we discussed about how to generate three-address code for Boolean expressions. We had generated two new variable and one was set to value '0' if the expression is false and other set to '1' if the expression is true. However, we could avoid generating code for the Boolean expression based on the logical operators if they are connected by one. Avoiding generation of code for some expression in a Boolean expression is referred to as short circuit based code generation.

Short circuit avoids computing the full expression involving logical operators. Consider the example where the expression is given by "a > b" . The corresponding code for this expression would be

100 If a > b goto E.true

101 goto E.false

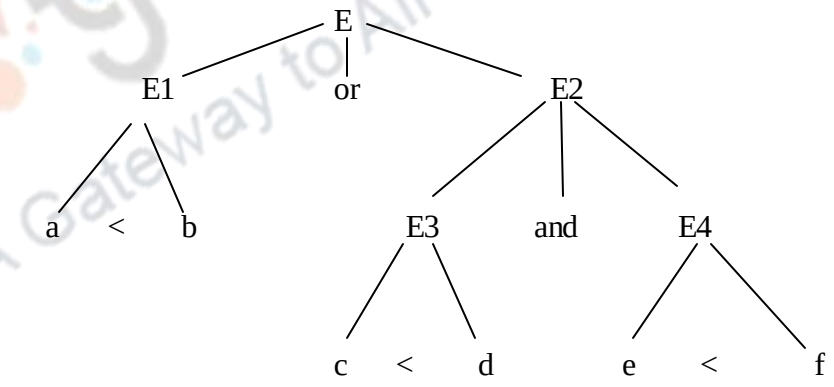
This is different from creating a temporary variable and assigning it 0 / 1. In other words if an expression E is given as E1 or E2 where 'or' is a logical operator, then if E1 is true, then this can directly be associated to E.true and we could skip generating code for E2. However, if E1 is false then E2 need to be evaluated. The table 25.5 summarizes the various semantic rules associated by incorporating this short circuit information to generate three-address code.

Table 25.5 Short-circuit based three-address code generation

Production	**Semantic Rule**	**Inference**
E → E1 or E2	E1.true : = E. true E1.false := newlabel E2.true := E.true E2.false := E.false E.code := E1.code || gen (E1.false ':') ||E2.code	Instead of creating newlabel, we directly assign E1.true to E.true. This avoids generating code for E2. If E1 is false we generate a new label and generate code for E2 and if it is true we assign E.true to E2.true. If E2 is false, we assign false for the entire expression. Thus the code corresponding to E is E1.code followed by generating a label for E1's false followed by E2.code
E → E1 and E2	E1.true : = newlabel E1.false := E.false E2.true := E.true	As the operator here is 'and' if E1 is false we don't evaluate E2 and assign the expression E as false.

	E2.false := E.false E.code := E1.code || gen (E1.true ':') || E2.code	Thus if E1 is true, we generate a new label and if E1 is false we evaluate the entire expression as false. If E2 is true, we would have reached this step only if E1 is true and hence expression E is true.
E → not E1	E1.true : = E. false E1.false := E.true E.code := E1.code	There is no short-circuit here and is the same as the simple code generation. We swap the true and false of E and E1.
E → (E1)	E1.true : = E. true E1.false := E.false E.code := E1.code	There is no short-circuit here and the true and false of E1 and E are same
E → id1 relop id2	E.code := gen ('if' id1.place relop.op id2.place 'goto' E.true) || gen ('goto' E.false)	This is the basic expression. We generate the expression followed by two goto's, one for true and false
E→ true	E.code := gen ('goto' E.true)	Basic value of true expression E results in generating code as goto E.true
E → false	E.code := gen ('goto' E.false)	Basic value of false expression E results in generating code as goto E.false

Example 25.1 consider the expression a < b or c < d and e < f



The following is a sequence of code generation based on the understanding.

- E1.true = E.true
- false label for E1
- true label for E3
- E3's false is E2's false and E's false

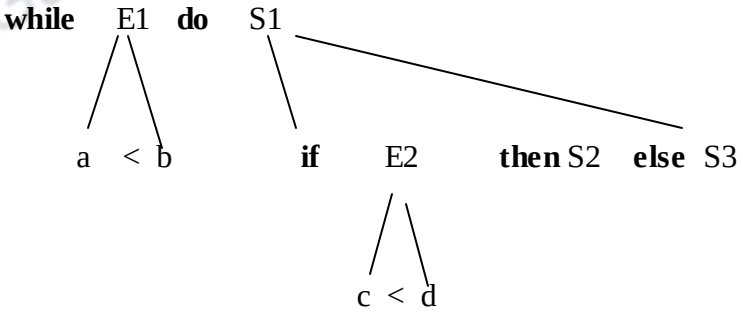
- E4's true is true for E
- E4's false is E's false

Based on the above sequence, we adopt a bottom up approach. Expression E1 is a < b. Hence we need to generate based on "gen ('if' id1.place relop.op id2.place 'goto' E.true) || gen ('goto' E.false)"

Code	**Comments**
if a < b goto Ltrue	Ltrue is the entire expression's true. If this is true, E2 need not be evaluated
goto L1	L1 is the label corresponding to E1's false
L1: if c < d goto L2	c < d is the expression of E3 and hence if it is true we need to check whether the expression E4's true. L2 is the place for E3's true
goto Lfalse	Lfalse is the entire expression's false. We have reached this point when E1 is false and E3 is false. E3 and E4 are connected using 'and' operator. Hence if E3 is false the entire expression is false
L2: if e < f goto Ltrue	e < f is the expression of E4 and if it is true, we have encountered truth of E3 'and' E4. Hence, the entire expression E2 is true and so we goto the expression E's truth Ltrue
goto Lfalse	We have reached this point after false of all expressions and so we conclude that the entire expression is false.

Example 25.2 Consider the following code segment

```
while a < b
      if c < d then
        x : = y + z
      else
        x : = y - z
```



The statement S2 corresponds to x:= y+z and that of S3 is x := y-z. The following is the sequence to generate code

- E1.code – if a < b goto true label

goto false label

- True label is the body of the while
- False label is the S.next
- E2.code – if c < d goto true label
  goto false label
- True label is x : = y + z and false label x:=y-z
- This should be done in the context of while and if-else

The following would be the code sequence:

Code	**Comments**
L1: if a < b goto L2	L1 corresponds to S.begin of the while. We generate label L2 as E.true and generate goto Lnext to follow with the next statement after the while if E is false
goto Lnext	If E is false we skip the body of the while and goto the statement following the while block
L2: if c < d goto L3	Here, we need to evaluate the if-then-else statement. L3 is the true of the expression c<d
goto L4	L4 is the false of the expression c < d
L3: t1 : = y+ z	In L3 we evaluate the body S2 of the if-then else block
x := t1	S2 is evaluated
goto L1	goto L1 corresponds to goto S.begin of the while to continue and skipping S3.
L4: t2 := y – z	In L4 we evaluate the body S3 of the if-then-else block
x := t2	
goto L1	Goto S.begin of the while to continue
Lnext:	The place to continue after the expression E of the while block fails

Summary: In this module we looked at generating three-address code for the control flow statements if-then, if-then-else, do-while and while. We also looked at incorporating short circuit code in generating 3-address code for Boolean expressions and integrating this with the control flow statements. The next module will discuss about another important task in Boolean expressions called back patching.