BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flowof control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,andnot**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the formE_1 **relop**E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E→E**or**E | E**and**E |**not**E | (E) |**id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false***numerically***and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for

  a**or**b**and not**c

  is the three-address sequence

  t_1 : =**not**c
  t_2 : = b**and**t_1
  t_3 : = a**or**t_2

- A relational expression such as a < b is equivalent to the conditional statement

  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103
101 :   t : = 0
102 :   goto 104
103 :   t : = 1
104 :
```

Translation scheme using a numerical representation for booleans

E □ E_1 **or** E_2 { *E.place : = newtemp;*
emit(E.place ': =' E_1.place **'or'** *E_2.place) }*

E □ E_1 **and** E_2 { *E.place : = newtemp;*
emit(E.place ': =' E_1.place **'and'** *E_2.place) }*

E □ **not** E_1 { *E.place : = newtemp;*
emit(E.place ': =' **'not'** *E_1.place) }*

E □ (E_1)   { *E.place : = E_1.place* }

E □ **id_1 relop id_2**   { *E.place : = newtemp;*
emit('if' **id** *$_1$.place***relop**.*opid $_2$.place* **'goto'** *nextstat* +**3**);
emit(E.place ': =' **'0'**);
*emit(***'goto'***nextstat* +**2**);
emit(E.place ': =' **'1'**) }

E □ **true** { *E.place : = newtemp;*
emit(E.place ': =' **'1'**) }

E □ **false** { *E.place : = newtemp;*
emit(E.place ': =' **'0'**) }

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called **"short-circuit"** or **"jumping"** code. It is possible to evaluate boolean expressions without generating code for the boolean operators**and, or,**and**not**if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto 103	107 : t_2 : = 1
101 : t_1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t_3 : = 0
103 : t_1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t_3 : = 1
105 : t_2 : = 0	112 : t_4 : = t_2 and t_3
106 : goto 108	113 : t_5 : = t_1 or t_4

Flow-of-Control Statements

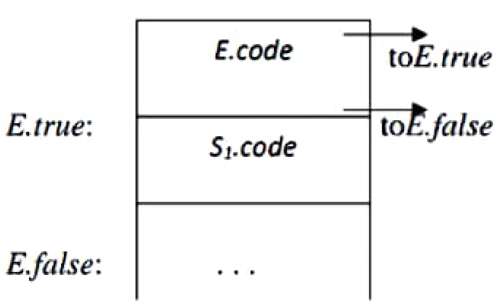
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S→ifEthenS $_1$
|ifEthenS $_1$ **elseS** $_2$
| whileEdoS $_1$

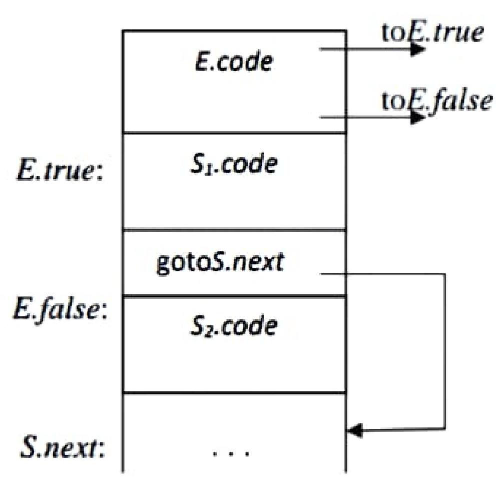
In each of these productions,*E*is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel*returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

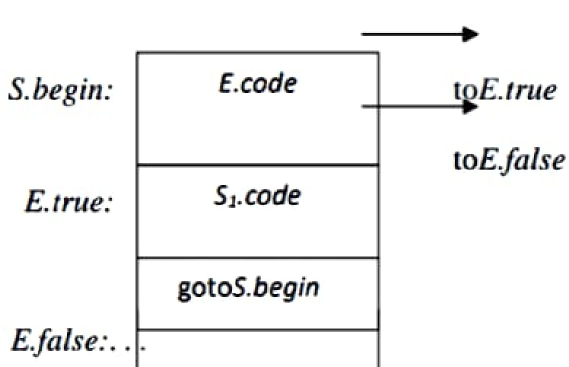
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \mid\mid gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen($'**goto**'$\ S.next) \mid\mid$ $gen(E.false\ ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \mid\mid E.code \mid\mid$ $gen(E.true\ ':') \mid\mid S_1.code \mid\mid$ $gen($'**goto**'$\ S.begin)$

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.false\ ':') \mid\mid E_2.code$
$E \rightarrow E_1$ **and** E_2	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \mid\mid gen(E_1.true\ ':') \mid\mid E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (\ E1\ )$	$E_1.true := E.true;$

	$E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow$ **id** $_1$ **relop id** $_2$	$E.code := gen($'**if**' **id** $_1$$.place$ **relop.op** **id** $_2$$.place$ '**goto**' $E.true) \,\|\|\, gen($'**goto**' $E.false)$
$E \rightarrow$ **true**	$E.code := gen($'**goto**' $E.true)$
$E \rightarrow$ **false**	$E.code := gen($'**goto**' $E.false)$