Control Flow

Here we discuss the uses of **Boolean expressions** to alter the flow of control. We will learn about a new non-terminal **B** for the same.

# Boolean Expressions:

 Boolean expressions are composed of Boolean operators like &&,!!,! for AND, OR , NOT respectively. Here we consider the Boolean expressions generated by the following grammar:

 B -> B || B | B && B | ! B | ( B ) | E rel E | true | false

We use rel.op to represent which of the six comparison operators <, <=, =, !=, >, or >= is represented by rel. As is customary, we assume that || and && are left –associative, and that || has the lowest precedence, then &&, then !.

# Short Circuit Code:

 In this type of code the Boolean operators like &&, !! or ! are translated into jumps. Instead of appearing as code, they are shown by a position in the code sequence.

 Example: Consider the expression given below.

 If ( x < 100 || x > 200 && x != y ) x = 0;

 Translation:

 If x < 100 goto L2

If False x > 200 goto L1

If false x != y goto L1

 L2: x = 0

 L1:

# Flow of control statements:

 Now we consider the conversion of Boolean expression into **three-address** code in the context of statement such as those generated by the following grammar:

 S -> if ( B ) S1

 S -> if ( B ) S1 else S2

 S -> while ( B ) S1

In the above expressions, non-terminal B represents a Boolean expression and S represents a statement. Now let us try to translate the above code.

 The translation of *if ( B ) S1* consists of B.code followed by S1.code, as illustrated in Fig. 6.35(a). Within B.code are jumps based on the value of B. If B is true, control follows to the first instruction of S1.code, and if B is false, control flows to the instruction immediately following S1.code

 For **if** statement

 B.code -> to B.true

 -> to B.false

B.true: S1.code

 B.false: …..

 For **if-else** statement

 B.code -> to B.true

 -> to B.false

B.true: S1.code

 Goto S.next

B.false: S2.code

 S.next: ……

 For **while** statement

 begin : B.code -> to B.true

 -> to B.false

B.true: S1.code

 Goto begin

B.false: S2.code

The labels for the jumps in B.code and S.code are managed using inherited attributes

Type Checking

To do type checking a compiler must assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.

1. Assign type with individual construct { stmt if-else while }
2. Rule -> if validation -> Error

Associated type information with expressions:

 E -> E1 + E2 { Type } E-> id { E.Type = lookup(id)Type }

 E -> E1 + E2 {

E->Type = if ( E1.Type = int & E2.type = int)

 => E.Type = int

 Else Type error

}

 S -> id = E S.type =

 If (id.Type matches with E.Type)

 S.type = null

 S -> if ( B ) S1 if ( B.Type = Boolean )

 S.type = S1.type