6.6 Control Flow

Translation of statements such as if-else statements and while-statements is tied to the translation of boolean expressions.

Boolean Expression are often used to ⇒

1. Alter the flow of control

Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expression is implicit in a position reached in a program

for eg ⇒ if (E) S

the expression E must be true if statement S is reached.

2. Compute logical values

A boolean expression can represent true or false as values. Such boolean expression can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

* → The intended use of Boolean Exp. is determined by its syntatic context.

for example ⇒

→ an expression following the keyword if is used to alter the flow of control.

→ exp. on right side of an assignment is used to denote a logical value.

Such Syntatic context can be specified in a no. of ways

→ We may use two diff. Non-Terminals.

→ use inherited attributes

→ set a flag during Parsing

→ build a syntax tree and invoke diff. procedures for two different use of boolean expression.

6.6.1 Boolean Expressions

Boolean Expressions are composed of boolean operators (which we denote && , || and ! using C convention for the operators AND, OR. and NOT respectively) applied to elements that are boolean variables or relational expressions.

Relational Expressions are of the form E_1 rel E_2 where E_1 and E_2 are arithmetic expressions.

Consider boolean exp. generated by following grammar:

$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$

rel-op is used to indicate which of the six comparison operators $<, <=, ==, !=, >$ or $>=$ is represented by rel

* Assume || and && → left associative
  || → lowest precedence. then && then !

Given the exp. $B_1 || B_2 \Rightarrow$

- If we determine B_1 is true, then entire exp. is true without having to evaluate B_2
- Why. $B_1 \&\& B_2$ =)
  if B_1 is false, then entire exp. is false.

Therefore
the semantic def. of the programming language determines whether all parts of Boolean exp. must be evaluated.

If the language def. permits portions of Boolean exp. to go unevaluated, then the compiler can optimize evaluation of exp. by computing only enough of exp. to determine value.

6.6.2 Short-Circuit Code

In Short-circuit code, the boolean exp. operators &&, || and ! translate into jumps. The operators do not appear on the code; instead the value of boolean exp. is represented by a position in a code sequence.

Eg → 6.21

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

This statement is translated into the code. In this translation the boolean exp. is true if control reaches L_2

If exp. is false, control goes to L_1, skipping L_2 and the assignment x = 0

```
    if x < 100 goto L2
    if false x > 200 goto L1
    if false x != y goto L1
L2: x2 = 0
L1:
```

→ jumping code.

6.6.3 → Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in context of statements.

Suppose we have the following grammer :-

$S \rightarrow$ if $(B)\ S_1$
$S \rightarrow$ if $(B)\ S_1$ else S_2
$S \rightarrow$ while $(B)\ S_1$

In these productions, non-terminal B supresents a boolean expression and non-terminal S represents a statement.

Code for the above grammer i.e. if-, if-else, and while-statements

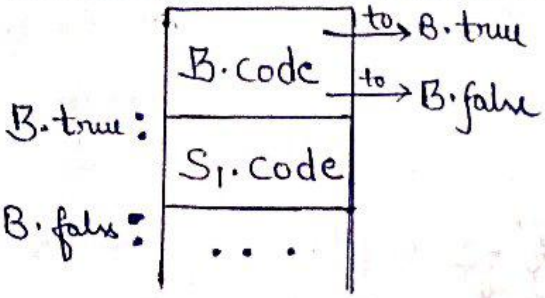

fig. (a) if

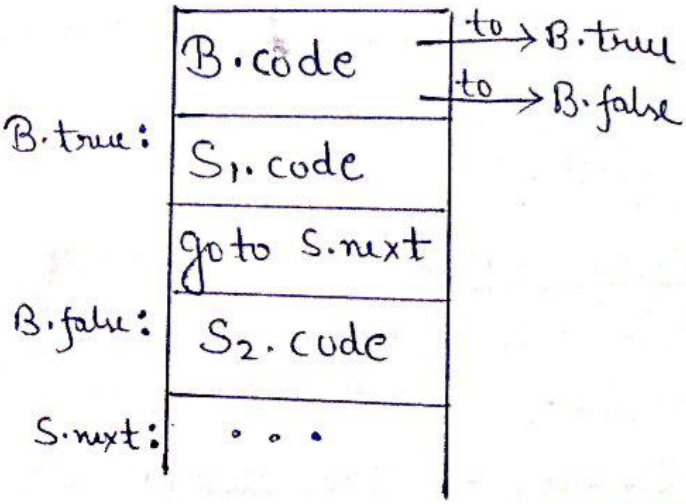

fig. (b) if-else

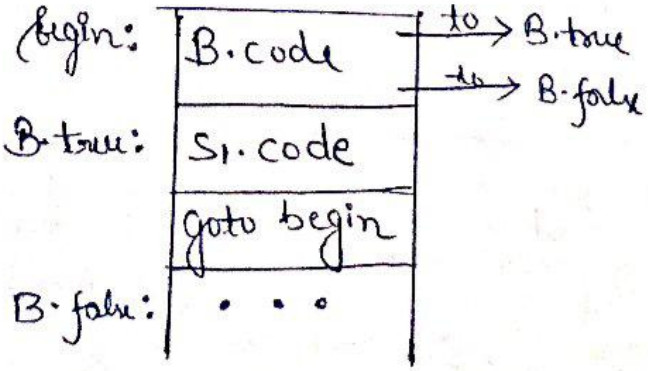

fig. (C) while

There are some points related to this 3 code :-

Here, S.code and B.code are synthesized attribute code which gives translation into three-address instructions.

- S.code is sequence of intermediate-code steps which implements S and ends with jump to S.next.
- B.code are jumps based on B value. If B is true, control flows to first instruction of S1.code, & if B is false, it flows to instruction immediately following S1.code.
- Labels for jumps in B.code and S.code are managed using inherited attributes. With B boolean expression, we associate 2 labels: B.true, label to which control flows if B is true and B.false, label to which cntrl flows if B is false.
- With S statement, we associate an inherited attribute S.next which denotes a label for instruction immediately after code for S. A jump to a jump to L from within S.code is avoided using S.next.

Now, Syntax-directed definition OR Semantic Rules for if-, if-else and while statements which produces three-address code for boolean expressions in context of these statements :-

$S \rightarrow$ if (B) S_1

B.true = newlabel()

B.false = S_1.next = S.next

S.code = B.code || label(B.true) || S_1.code

- Here, we assume newlabel() creates a new label each time it is called, and that label(L) attaches label L to next three-address instruction to be generated.
- In translating $S \rightarrow$ if (B) S_1, semantic rules create a new label B.true and attach it to first three-address instruction generated for statement S_1, as we write in fig (a). Thus, jumps to B.true within code for B will go to the code for S_1. Further by setting B.false to S.next, we ensure that cntrl will skip code for S_1 if B is false.

$S \rightarrow$ if (B) S_1 else S_2

B.true = newlabel()

B.false = newlabel()

S_1.next = S_2.next = S.next

S.code = B.code || label(B.true) || S_1.code || gen('goto' S.next)
|| label(B.false) || S_2.code

- Here, in translating this, code for B has jumps out of it to first instruction of code for S_1 if B is true and for S_2 if B is false, as we write in fig (b). Further, control flows from both S_1 and S_2 to three

address instructions immediately following the code for S. An explicit goto S.next appears after code for S_1 to skip over the code for S_2. No goto is needed after S_2, since S_1.next is same as S.next.

$S \rightarrow$ while (B) S_1

begin = newlabel()
B.true = newlabel()
B.false = S.next
S_1.next = begin
S.code = label (begin) || B.code || label (B.true) || S_1.code
|| gen('goto' begin)

- In this we use a local variable begin to hold a new label attached to first instruction, which is also the first statement for B. Here, we use a variable rather than an attribute because begin is local to semantic rules for this production. Here, B.false is set to S.next. The code for B generates a jump to B.true if B is true. Now, we place instruction goto begin, which causes a jump back to beginning of code for B.

Here are some productions with their Semantic Rules -

$P \rightarrow S$

S.next = newlabel()
P.code = S.code || label (S.next)

$S \rightarrow assign$

$S.code = assign.code$

Here, assign is a placeholder for assignment statements.

$S \rightarrow S_1 S_2$

$S_1.next = newlabel()$

$S_2.next = S.next$

$S.code = S_1.code \,||\, label(S_1.next) \,||\, S_2.code$

6.6.4 CONTROL FLOW TRANSLATION OF BOOLEAN EXPRESSIONS

→ A boolean expression B is translated into three-address code that evaluate B using conditional and unconditional jumps to one of two labels: B.true if B is true and B.false if B is false.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \text{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code \mid\mid gen('if' E_1.addr \text{ rel.op } E_2.addr \text{ 'goto' } B.true) \mid\mid gen('goto' B.false)$

Date : ___/___/___

$B \rightarrow$ true	B.code = gen('goto' B.true)
$B \rightarrow$ false	B.code = gen('goto' B.false)

$B \rightarrow E_1$ rel E_2 is translated directly into a comparison three-address instruction with jumps to the appropriate places. For instance, B of the form $a < b$ translates into :

if $a < b$ goto B.true
goto B.false

Remaining productions for B are translated as follows :-

1. B is of form $B_1 \,||\, B_2$. If B_1 is true, B is also true automatically. If B_1 is false, B_2 must be evaluated, so we make B_1.false be the label of the first instruction in the code for B_2.

2. Translation of $B_1 \,\&\&\, B_2$ is similar.
3. No code needed for an expression B of the form $!B_1$: just interchange the true and false exits of B to get the true and false exits of B_1.
4. True and false translate into jumps to B.true and B.false respectively.

Example → if ($x < 100 \,||\, x > 200 \,\&\&\, x != y$) $x = 0$;

if $x < 100$ goto L_2
goto L_3
L_3 : if $x > 200$ goto L_4
goto L_1

Date : ___/___/___

L_4 : if $x != y$ goto L_2
goto L_1
L_2 : $x = 0$
L_1 = ...

This was the control-flow translation of a simple if-statement.

6.6.5
AVOIDING REDUNDANT GOTOS

In the example in 6.6.4, comparison $x > 200$ translates into the code:

if $x > 200$ goto L_1
goto L_4
L_4 : ...

Instead, consider this:

ifFalse $x > 200$ goto L_1
L_4 : ...

ifFalse takes advantage of the natural flow from one instruction to the next in sequence, so control simply 'falls through' to label L_4 if $x > 200$, thereby avoiding a jump

New rules for $S \rightarrow$ if (B) S_1 set B.true to fall.

B.true = fall
B.false = S_1.next = S.next
S.code = B.code || S_1.code

Similarly, rules for if-else and while statements also set B.true to fall.

New rules for $B \rightarrow E_1$ rel E_2 generate 2 instructions if both B.true and B.false are explicit labels; that is, neither equals fall.

Date: ___/___/___

Otherwise if B.true is explicit label, then B.false must be fall, so they generate an if instruction that lets control fall through if the condition is false. Conversely if B.false is is explicit, then they generate ifFalse instruction. In the remaining case, both B.true and B.false are fall, so no jump is generated.

New rules for $B \rightarrow B_1 \| B_2$. Suppose B.true is fall, i.e, control falls through B, if B is true. Although B evaluates to true if B_1 does, B_1.true must ensure that control jumps over the code for B_2 to get to the next instruction after B.

On other hand, if B_1 evaluates to false, the truth-value of B is determined by the value of B_2, so rules in next example ensure that B_1.false corresponds to control falling through from B_1 to the code for B_2.

Semantic rules for $B \rightarrow E_1$ rel E_2

↓

test = E_1.addr rel.op E_2.addr

s = if B.true ≠ fall and B.false ≠ fall then gen('if' test 'goto' B.true) || gen('goto' B.false)
else if B.true ≠ fall then gen('if' test 'goto' B.true)
else if B.false ≠ fall then gen('ifFalse' test 'goto' B.false)
else ''

B.code = E_1.code || E_2.code || s

Rules for $B \rightarrow B_1 \,||\, B_2$

B_1.true = if B.true ≠ fall then B.true else newlabel()

B_1.false = fall

B_2.true = B.true

B_2.false = B.false

B.code = if B.true ≠ fall then B_1.code || B_2.code

else B_1.code || B_2.code || label(B_1.true)

~~Rule for B → B₁ ||~~

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

translate into ↴

if $x < 100$ goto L_2

iffalse $x > 200$ goto L_1

iffalse $x != y$ goto L_1

L_2 : $x = 0$

L_1 : ...

This was if-statement translated using fall-through technique

Here, new label L_2 is created to allow a jump over the code for B_2 if B_1 evaluates to true. Thus, B_1.true is L_2 and B_1.false is fall, since B_2 must be evaluated if B_1 is false.

Production $B \rightarrow E_1$ rel E_2 that generates $x < 100$ is therefore reached with B.true = L_2 and B.false = fall. With these inherited labels, rules in $B \rightarrow E_1$ rel E_2 therefore generate a single instruction if $x < 100$ goto L_2.

6.6.6 Boolean Values and Jumping Code

A Boolean exp. may also be evaluated for its value as in assignment statements such as $x = true$ or $x = a < b$

A clean way of handling both roles of Boolean exp. is to first build a syntax tree for expressions, using either

(1) Use two passes
- → Construct complete syntax tree for input
- → then walk the tree in depth-first order
- → computing the translations specified by semantic rules.

(2) Use one pass for statements, but two passes for exp.
- → we would translate E in while(E) S_1 before S_1 is examined.
- → translation of E by building its syntax tree
- → then walking the tree.

The following grammar has a single N.T E for exp ⇒

$S \rightarrow id = E ; \mid if(E)S \mid while(E)S \mid SS$

$E \rightarrow E \| E \mid E \&\& E \mid E == E \mid E + E \mid (E) \mid id \mid true \mid False$

* Non-terminal E governs the flow of control in $S \rightarrow while(E) S_1$
* Same Non-Terminal E denotes a value in $S \rightarrow id = E ;$ and $E \rightarrow E + E$

We can handle these two roles of Expressions by using separate code generation functions.

Suppose attribute E.n denotes the syntax tree node for exp. E and that nodes are objects.

Let method jump generate jumping code at an exp. node
let method rvalue generate code to compute the value of node into a temporary.

(1) When E appears in $S \rightarrow \text{while}(E)\, S_1$, method jump is called at node E.n.

Jumping code is generated by calling E.n.jump(t, f), where t is a new label for the first instruction of S_1.code and f is the label S.next.

(2) When E appears in $S \rightarrow id = E$; method rvalue is called at node E.n.

If E has form $E_1 + E_2$, the method call E.n.rvalue() generates code as 6.4.

If E has the form $E_1 \&\& E_2$, we first generate jumping code for E and then assign true or false to a new temporary t at true or false exits from jumping code.