- Control Flow
- Back patching
- Switch Statements
- Intermediate Code for Procedures

Control Flow

boolean expressions are often used to:

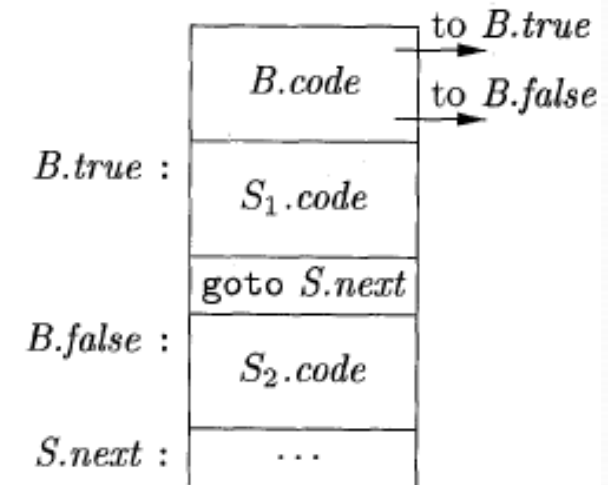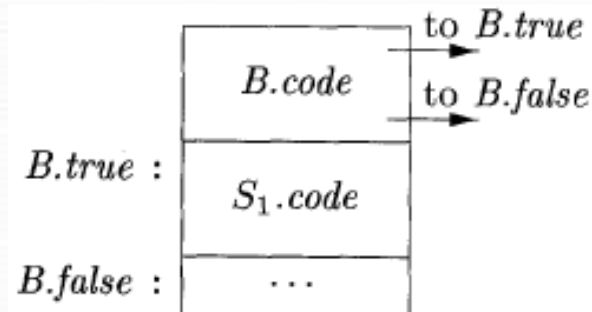- *Alter the flow of control.*
- *Compute logical values.*

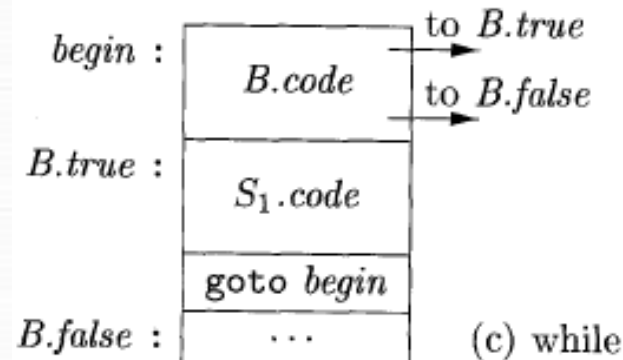# Short-Circuit Code

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

-
```
        if x < 100 goto L₂
        ifFalse x > 200 goto L₁
        ifFalse x != y goto L₁
L₂:    x = 0
L₁:
```

Flow-of-Control Statements

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

(a) if

(b) if-else

(c) while

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\| \; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} \; (\; B \;) \; S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\| \; label(B.true) \;\|\| \; S_1.code$
$S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\qquad \;\|\| \; label(B.true) \;\|\| \; S_1.code$ $\qquad \;\|\| \; gen('\textbf{goto}' \; S.next)$ $\qquad \;\|\| \; label(B.false) \;\|\| \; S_2.code$
$S \rightarrow \textbf{while} \; (\; B \;) \; S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\| \; B.code$ $\qquad \;\|\| \; label(B.true) \;\|\| \; S_1.code$ $\qquad \;\|\| \; gen('\textbf{goto}' \; begin)$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code$

$B \rightarrow B_1 \;\&\&\; B_2$

$B_1.true = newlabel()$

$B_1.false = B.false$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \;||\; label(B_1.true) \;||\; B_2.code$

$B \rightarrow \;!\; B_1$

$B_1.true = B.false$

$B_1.false = B.true$

$B.code = B_1.code$

$B \rightarrow E_1 \; \mathbf{rel} \; E_2$

$B.code = E_1.code \;||\; E_2.code$

$\qquad ||\; gen('\mathtt{if}'\; E_1.addr\; \mathbf{rel}.op\; E_2.addr\; '\mathtt{goto}'\; B.true)$

$\qquad ||\; gen('\mathtt{goto}'\; B.false)$

$B \rightarrow \mathbf{true}$

$B.code = gen('\mathtt{goto}'\; B.true)$

$B \rightarrow \mathbf{false}$

$B.code = gen('\mathtt{goto}'\; B.false)$

translation of a simple if-statement

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

-
```
        if x < 100 goto L₂
        goto L₃
L₃:     if x > 200 goto L₄
        goto L₁
L₄:     if x != y goto L₂
        goto L₁
L₂:     x = 0
L₁:
```

Backpatching

- Previous codes for Boolean expressions insert symbolic labels for Jumps.
- It therefore needs a separate pass to set them to appropriate addresses
- We can use a technique named backpatching to avoid this.
- Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels *backpatching.*
- We assume we save instructions into an array and labels will be indices in the array.
- For nonterminal B we use two attributes B.truelist and B.falselist together with following functions:
  - makelist(i): create a new list containing only i, an index into the array of instructions
  - Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a
    pointer to the concatenated list
  - Backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

Backpatching for Boolean Expressions

1) $B \rightarrow B_1 \mid\mid M\ B_2$ $\{$ $backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist;$ $\}$

2) $B \rightarrow B_1\ \&\&\ M\ B_2$ $\{$ $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist);$ $\}$

3) $B \rightarrow\ !\ B_1$ $\{$ $B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist;$ $\}$

4) $B \rightarrow (\ B_1\ )$ $\{$ $B.truelist = B_1.truelist;$
$B.falselist\ =\ B_1.falselist;$ $\}$

5) $B \rightarrow E_1\ \mathbf{rel}\ E_2$ $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$emit('\mathbf{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ 'goto\ \_');$
$emit('goto\ \_');$ $\}$

Backpatching for Boolean Expressions

- Annotated parse tree for $x < 100 \;||\; x > 200 \;\&\&\; x \,!=\, y$
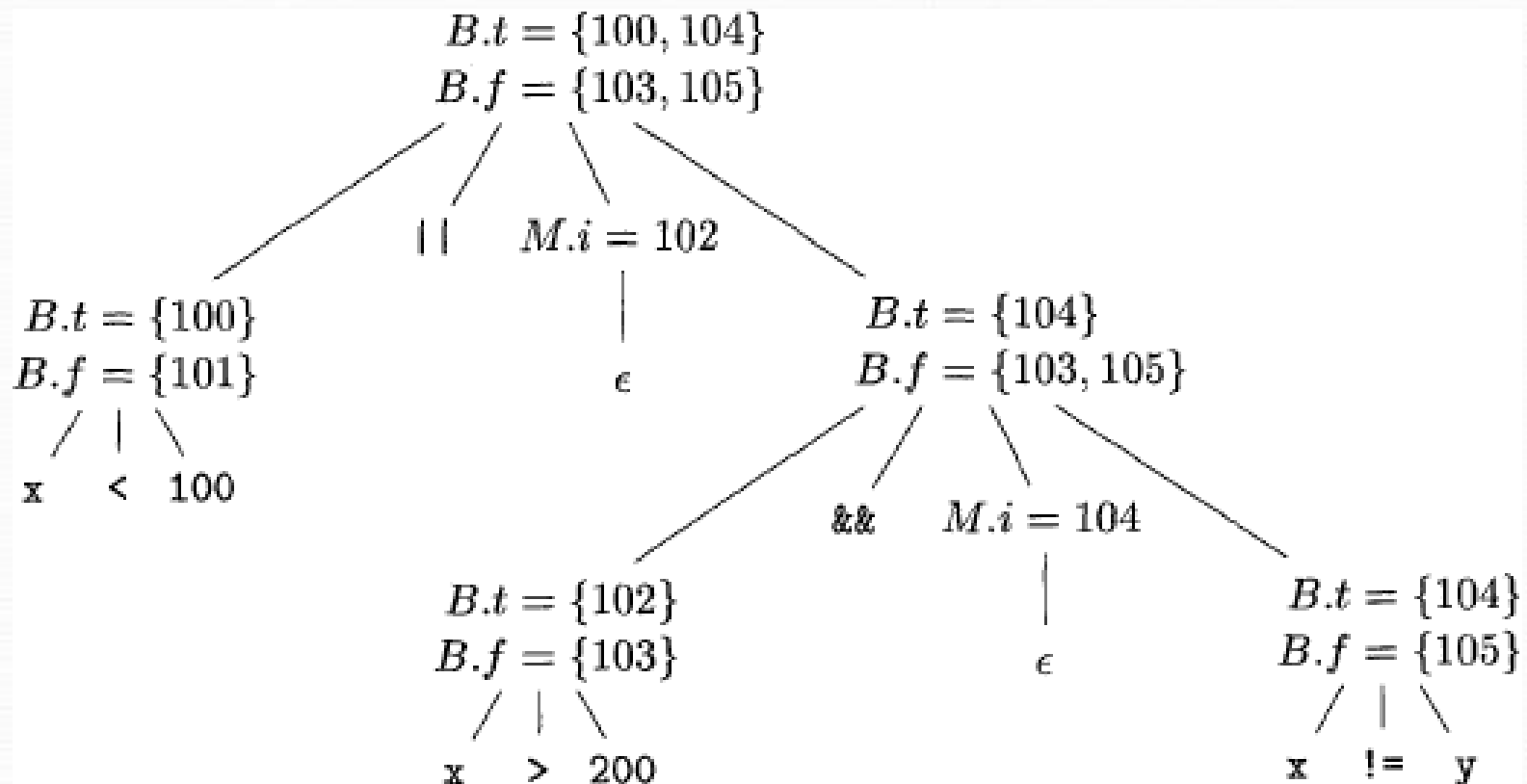
1) $S \rightarrow \textbf{if} \, (B) \, M \, S_1$ { $backpatch(B.truelist, \, M.instr);$
$\qquad\qquad\qquad\qquad S.nextlist \, = \, merge(B.falselist, \, S_1.nextlist);$ }

2) $S \rightarrow \quad \textbf{if} \, (B) \, M_1 \, S_1 \, N \, \textbf{else} \, M_2 \, S_2$
$\qquad\qquad\qquad$ { $backpatch(B.truelist, \, M_1.instr);$
$\qquad\qquad\qquad backpatch(B.falselist, \, M_2.instr);$
$\qquad\qquad\qquad temp \, = \, merge(S_1.nextlist, \, N.nextlist);$
$\qquad\qquad\qquad S.nextlist \, = \, merge(temp, \, S_2.nextlist);$ }

3) $S \rightarrow \quad \textbf{while} \, M_1 \, (B) \, M_2 \, S_1$
$\qquad\qquad\qquad$ { $backpatch(S_1.nextlist, \, M_1.instr);$
$\qquad\qquad\qquad backpatch(B.truelist, \, M_2.instr);$
$\qquad\qquad\qquad S.nextlist \, = \, B.falselist;$
$\qquad\qquad\qquad emit('goto' \, M_1.instr);$ }

4) $S \rightarrow \{ L \}$ $\qquad\qquad$ { $S.nextlist \, = \, L.nextlist;$ }

5) $S \rightarrow A \, ;$ $\qquad\qquad$ { $S.nextlist \, = \, \textbf{null};$ }

6) $M \rightarrow \epsilon$ $\qquad\qquad$ { $M.instr \, = \, nextinstr;$ }

```
switch ( E ) {
        case V_1: S_1
        case V_2: S_2
              ...
        case V_{n-1}: S_{n-1}
        default: S_n
}
```

```
                    code to evaluate E into t
                    goto test
L_1:                code for S_1
                    goto next
L_2:                code for S_2
                    goto next
                    ...
L_{n-1}:            code for S_{n-1}
                    goto next
L_n:                code for S_n
                    goto next
test:               if t = V_1 goto L_1
                    if t = V_2 goto L_2
                    ...
                    if t = V_{n-1} goto L_{n-1}
                    goto L_n
next:
```

Procedures

- A **procedure call** is a simple statement that includes the procedure name, parentheses with actual parameter names or values, and a semicolon at the end.

- The types of the actual parameters must match the types of the formal parameters created when the procedure was first declared (if any). The compiler will refuse to compile the compilation unit in which the call is made if this is not done.

- **General Form**

  Procedure_Name(actual_parameter_list);
  -- where commas separate the parameters

B.S. Abdur Rahman

Crescent
Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956
GST Road, Vandalur, Chennai 600 048

- **Calling Sequence**

A call's translation provides a list of activities executed at the beginning and end of each operation. In a calling sequence, the following actions occur:

- Space is made available for the activation record when a procedure is called.
- Evaluate the called procedure's argument.
- To allow the called method to access data in enclosing blocks, set the environment pointers.
- The caller procedure's state is saved so that it can resume execution following the call.
- Save the return address as well. It is the location to which the called procedure must transfer after it has been completed.
- Finally, for the called procedure, generate a jump to the beginning of the code.

- Let there be a function **f(a1, a2, a3, a4)**, a function f with four parameters a1,a2,a3,a4.

- Three address code for the above procedure call(f(a1, a2, a3, a4)) is given below.

```
param a1
param a2
param a3
param a4
call  f, n        //Here n is 4
```

'call' is a calling function with f and n, here f represents name of the procedure and n represents number of parameters.

n= f(a[i])

- Here, f is a function containing an array of integers a[i]. This function will return some value and the value is stored in 'n' which is also an integer variable.

    A→array of integers

    F→ function from integer to an integer.

- Three address codes for the above function can be written as:

    t1= i*4

    t2=a[t1]

    param t2

    t3= call f,1

    n=t3

Example 2:

- **t1= i*4**

In this instruction, we are calculating the value of i which can be passed as index value for array a.

- **t2=a[t1]**

In this instruction, we are getting value at a particular index in array a. Since t1 contains an index, here t2 will contain a value. The above two expressions are used to compute the value of the expression(a[i]) and then store it in t2.

- **param t2**

The value t2 is passed as a parameter of function f(a[i])

- **t3= call f,1**

This instruction is a function call, where position 1 represents the number of parameters in the function call. It can vary for different function calls but here it is 1. The calling function will return some value and the value is stored in t3.

- **n=t3**

The returned value will be assigned to variable n.