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

CSD 3202-COMPILER DESIGN

MODULE III

PART-3

Outline

- 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.*

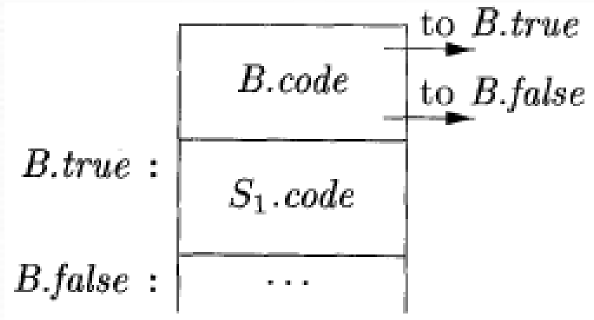
Short-Circuit Code

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

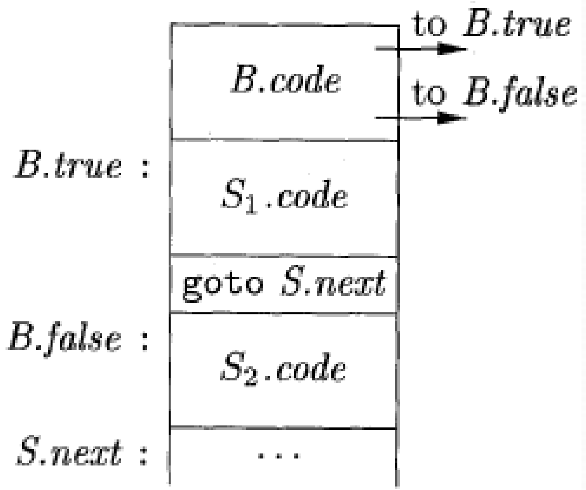
-
  ```
          if x < 100 goto L2
          ifFalse x > 200 goto L1
          ifFalse x != y goto L1
  L2:     x = 0
  L1:
  ```

Flow-of-Control Statements

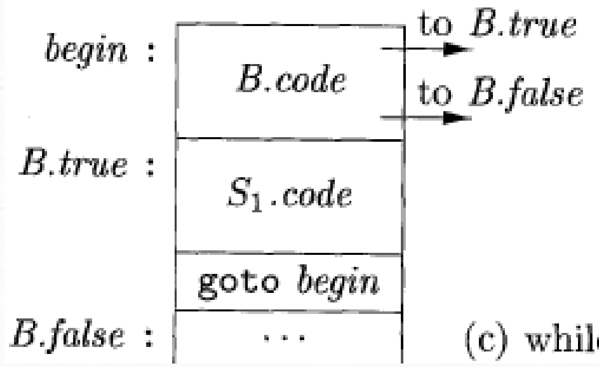
$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if} \ (\ B\ )\ S_1 \\
S & \rightarrow & \textbf{if} \ (\ B\ )\ S_1 \ \textbf{else} \ S_2 \\
S & \rightarrow & \textbf{while} \ (\ B\ )\ S_1
\end{array}
$$



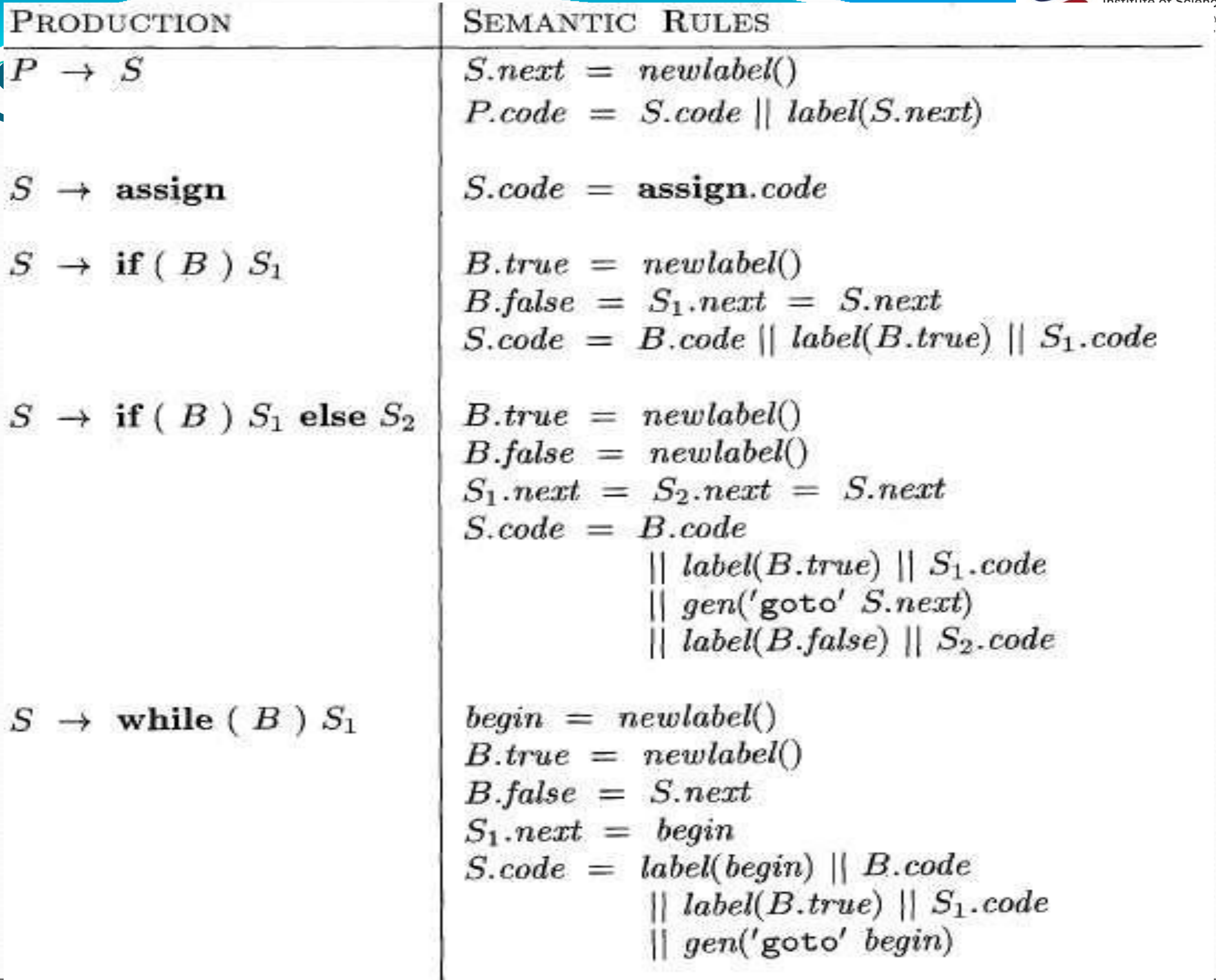
(a) if



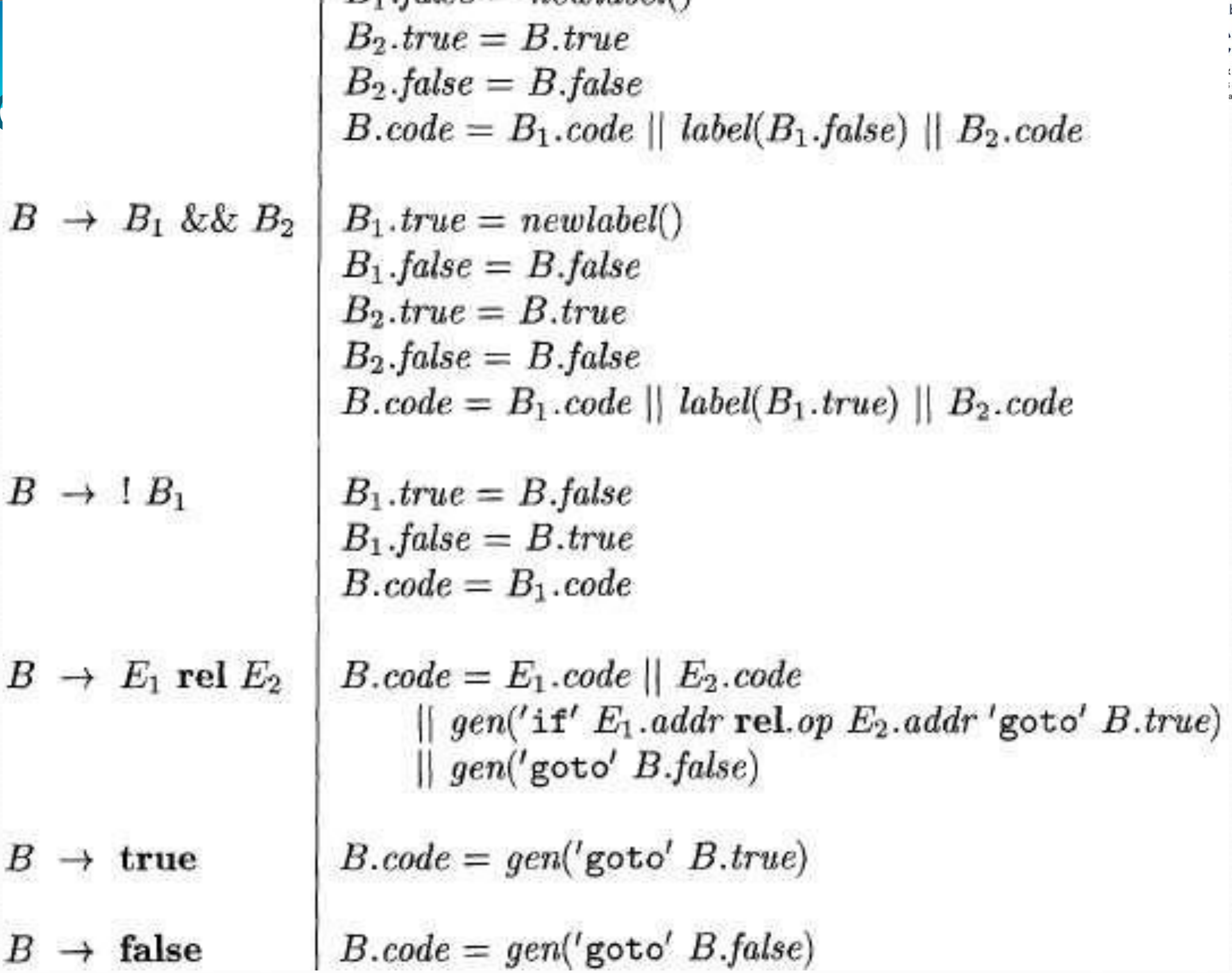
(b) if-else



(c) while



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



	$B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$
$B \rightarrow \;! \; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \;\mathbf{rel}\; E_2$	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\| \; gen('\texttt{if}' \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; '\texttt{goto}' \; B.true)$ $\|\| \; gen('\texttt{goto}' \; B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}' \; B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}' \; B.false)$

translation of a simple if-statement

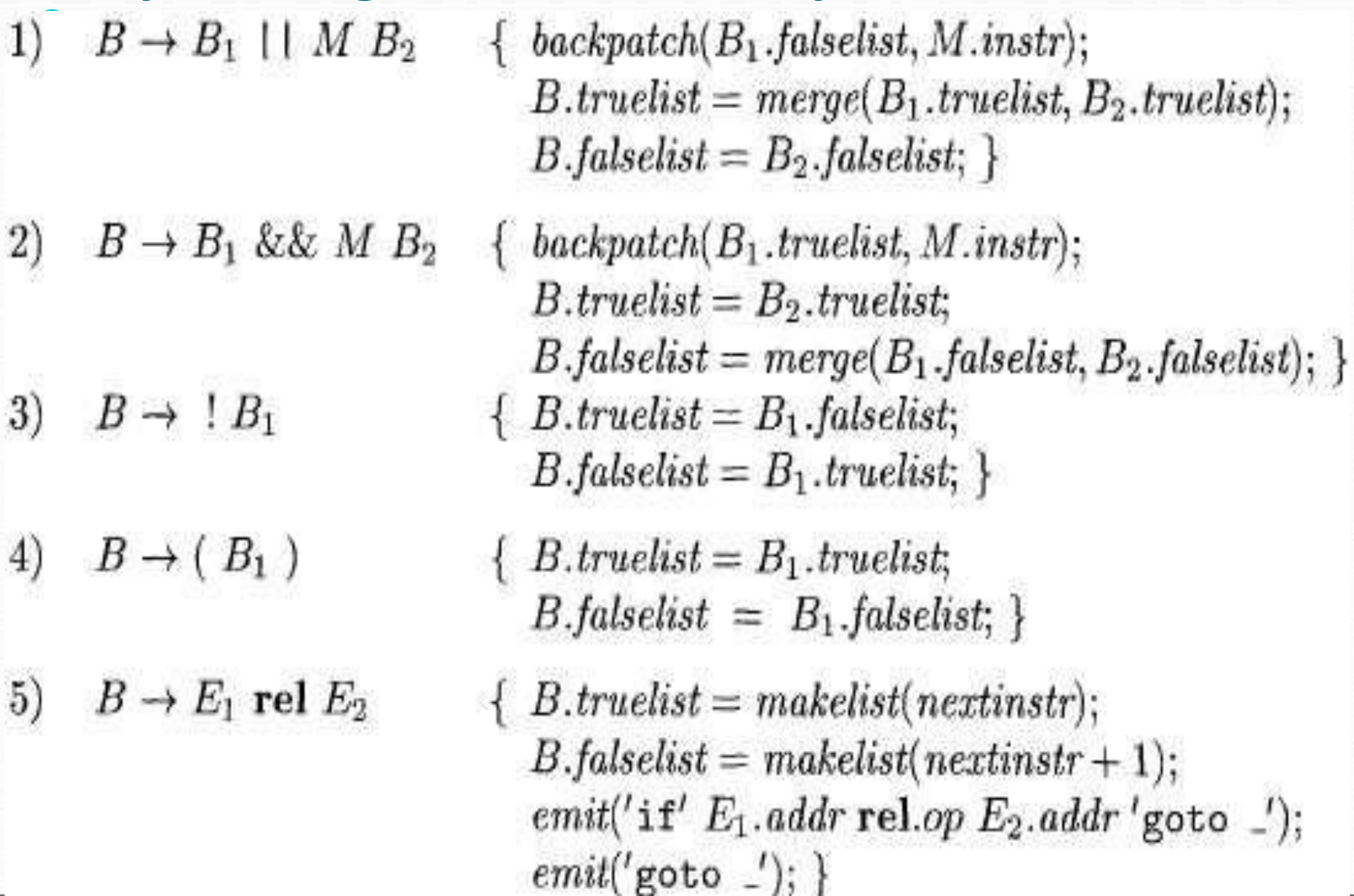
-
  ```
  if( x < 100 || x > 200 && x != y ) x = 0;
  ```

-
  ```
          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

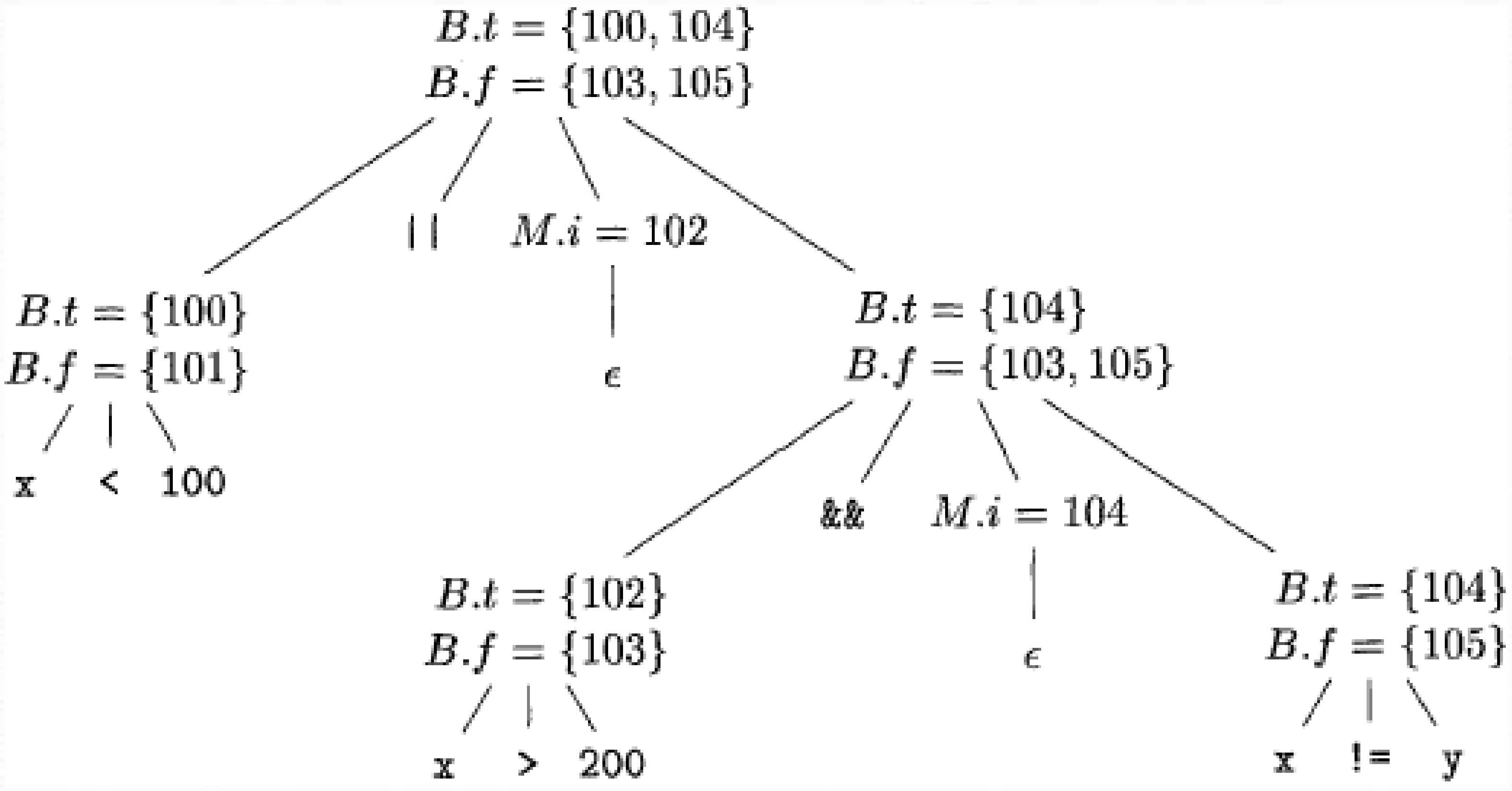
Backpatching for Boolean Expressions



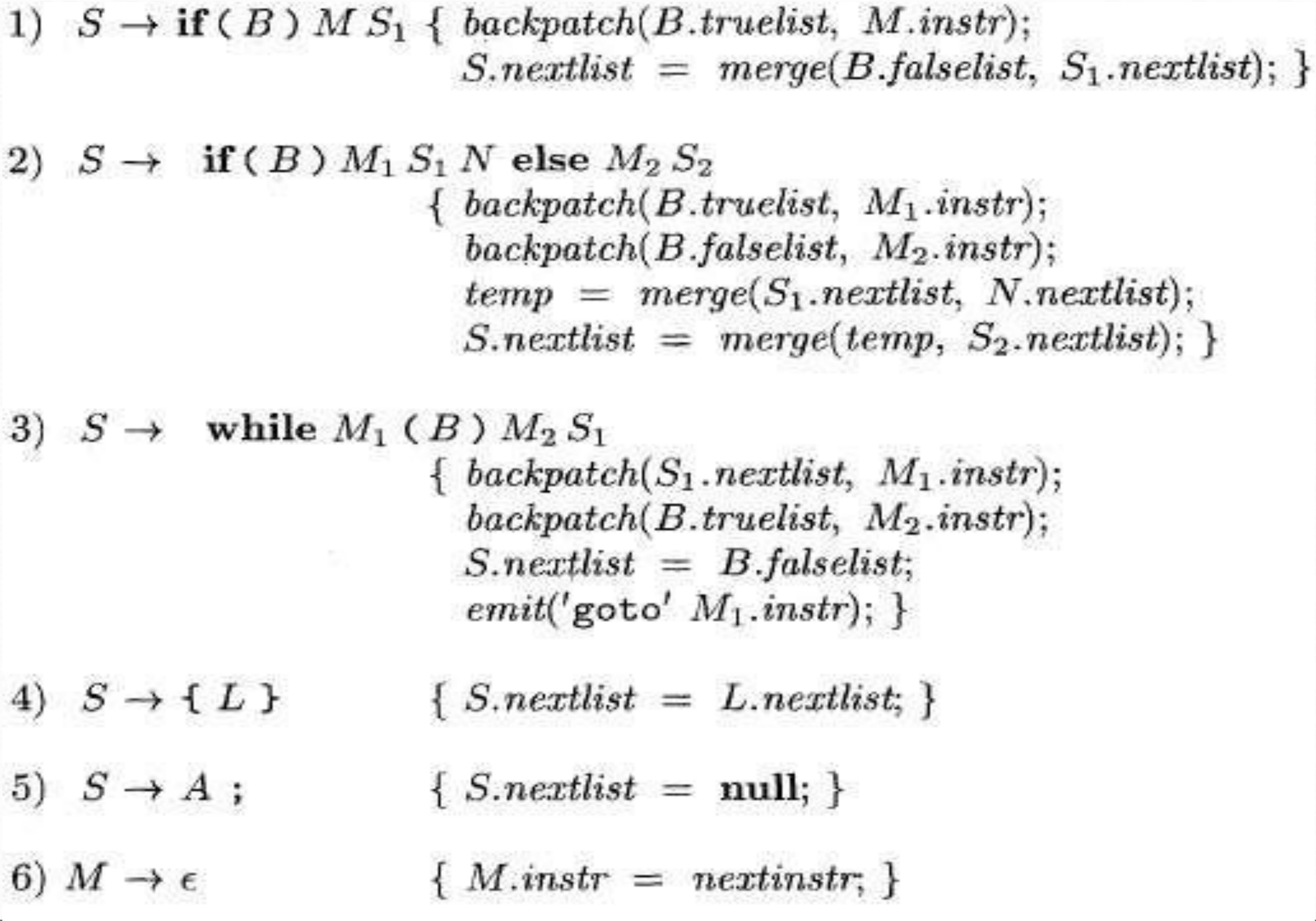
1)	$B \rightarrow B_1 \;\|\|\; M \; B_2$	$\{$ *backpatch*(B_1.*falselist*, M.*instr*); B.*truelist* = *merge*(B_1.*truelist*, B_2.*truelist*); B.*falselist* = B_2.*falselist*; $\}$
2)	$B \rightarrow B_1 \;\&\&\; M \; B_2$	$\{$ *backpatch*(B_1.*truelist*, M.*instr*); B.*truelist* = B_2.*truelist*; B.*falselist* = *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)	$B \rightarrow E_1$ **rel** E_2	$\{$ B.*truelist* = *makelist*(*nextinstr*); B.*falselist* = *makelist*(*nextinstr* + 1); *emit*('`if`' E_1.*addr* **rel**.*op* E_2.*addr* '`goto _`'); *emit*('`goto _`'); $\}$

Backpatching for Boolean Expressions

- Annotated parse tree for x < 100 || x > 200 && x ! = y



Flow-of-Control Statements



1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ $\{\ backpatch(B.truelist,\ M.instr);$
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist);\ \}$

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

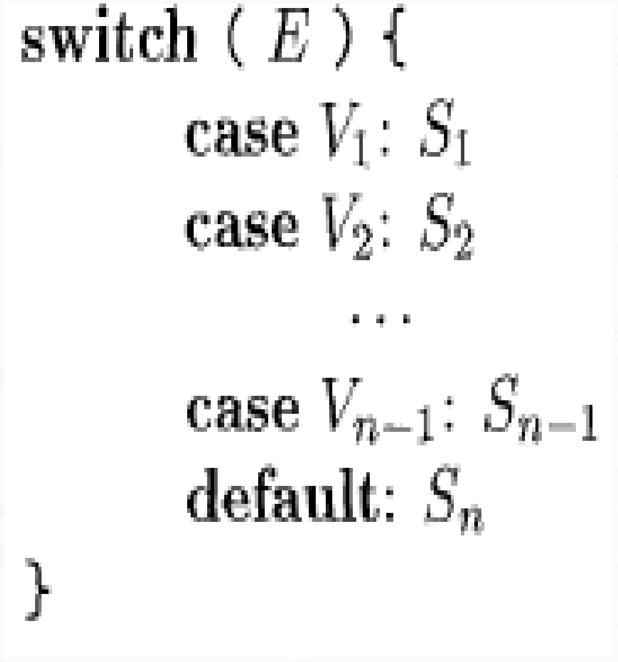
3) $S \rightarrow\ \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$
$\{\ backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist\ =\ B.falselist;$
$emit('\texttt{goto}'\ M_1.instr);\ \}$

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

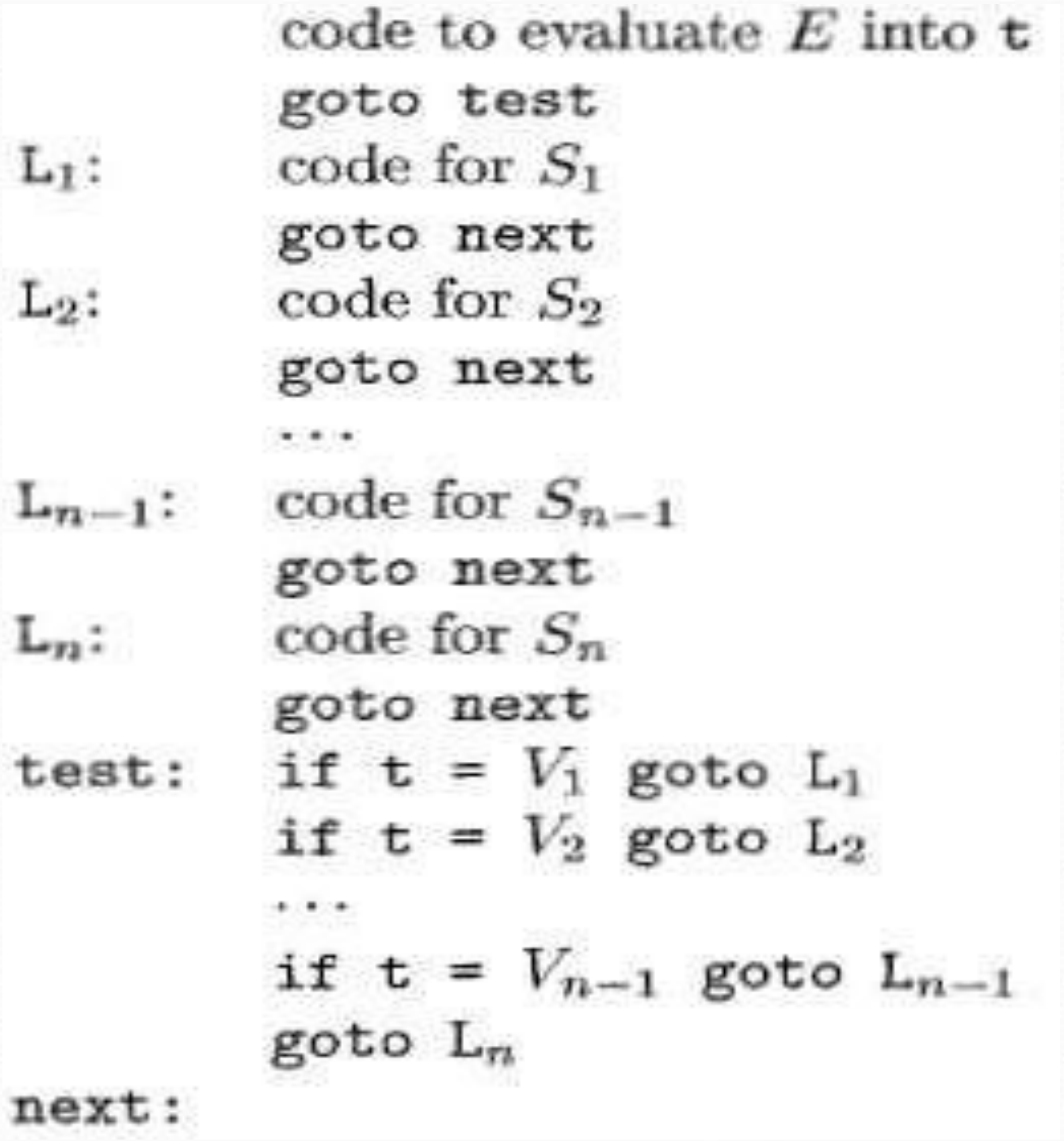
5) $S \rightarrow A\ ;$ $\{\ S.nextlist\ =\ \textbf{null};\ \}$

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

Translation of a switch-statement



```
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
```

- **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.

Example 2:

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.