Control-Flow Translation of Boolean Expressions

- A boolean expression B is translated into three-address instructions 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
$B \rightarrow B_1 \,\|\|\, B_2$	$B_1.true = B.true$
	$B_1.false = newlabel()$
	$B_2.true = B.true$
	$B_2.false = B.false$
	$B.code = B_1.code \,\|\|\, label(B_1.false) \,\|\|\, B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$
	$B_1.false = B.false$
	$B_2.true = B.true$
	$B_2.false = B.false$
	$B.code = B_1.code \,\|\|\, label(B_1.true) \,\|\|\, B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$
	$B_1.false = B.true$
	$B.code = B_1.code$
$B \rightarrow E_1 \; rel \; E_2$	$B.code = E_1.code \,\|\|\, E_2.code$
	$\|\| \; gen('if' \; E_1.addr \; rel.op \; E_2.addr \; 'goto' \; B.true)$
	$\|\| \; gen('goto' \; B.false)$
$B \rightarrow true$	$B.code = gen('goto' \; B.true)$
$B \rightarrow false$	$B.code = gen('goto' \; B.false)$

Fig: Generating three-address code for booleans

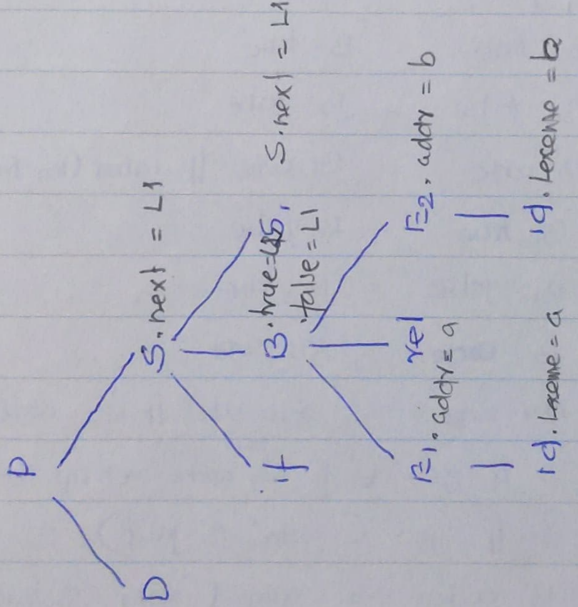
- For example, B of the form $a < b$ translates into:

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

$B \rightarrow id$ to be added? For considering !id.

• Examples :

if (a < b) x = 0



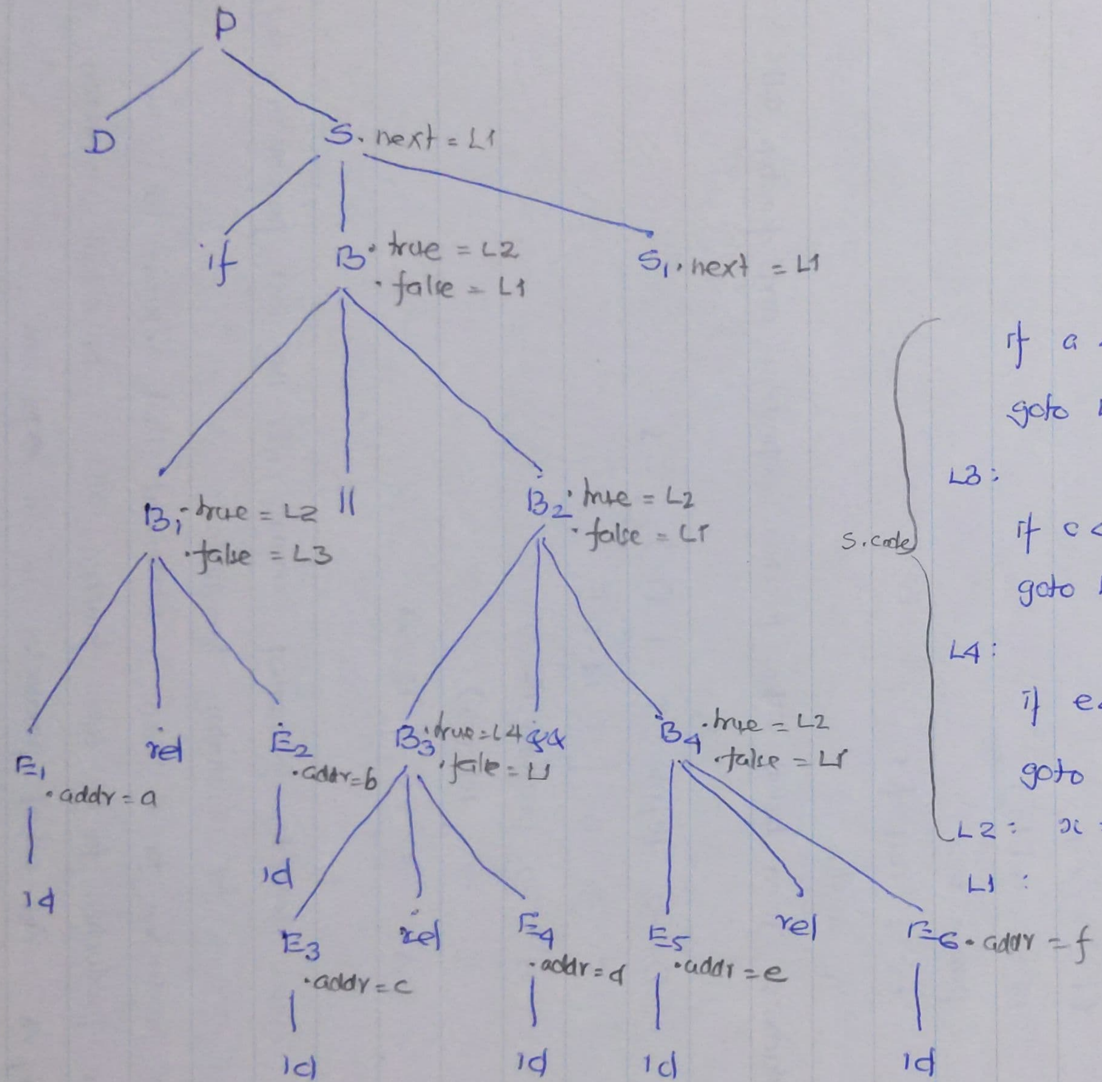
if a < b goto L2

goto L1

L2 : x = 0

L1 :

- Example 2: if ((a < b) || (c < d) && (e < f)) x = 0




```
    if a < b goto L2   } B1.code
    goto L3
L3:
    if c < d goto L4   } B3.code
    goto L1
L4:
    if e < f goto L2   } B4.code
    goto L1
L2: x = 0
L1:
```

S.code, B2.code, B.code, P.code

Intermediate Code for Procedures

- We discuss function declarations and three-address code for function calls. In three-address code, function call is unravelled into the evaluation of parameters in preparation for a call, followed by the call itself.
- For simplicity, we assume that parameters are passed by value.
- Example: Suppose that 'a' is an array of integers, and 'f' is a function from integer to integers. Then the assignment

  $n = f(a[i])$

  might translate into the following three-address code:

  $t_1 = i \times 4$
  $t_2 = a[t_1]$
  param t_2
  t_3 = call f,1
  $n = t_3$

- The productions given below allow function definitions and function calls.

  $D \rightarrow$ define T id (F) { S }
  $F \rightarrow \epsilon \mid T$ id ; F
  $S \rightarrow \ldots \mid$ return E
  $E \rightarrow \ldots \mid$ id (A)
  $A \rightarrow \epsilon \mid E, A$

- The syntax generates unwanted commas after the last parameter, but is good enough for illustrating translation.
- The production for S adds a statement that returns the value of an expression.
- The production for E adds function calls, with actual parameters generated by A. An actual parameter is an expression.

- Function definitions and function calls can be translated using concepts that have already been introduced.
  - Function types : Includes return type and types of formal parameters. Let 'void' be the type that represents no return type.
  - Symbol table : The function name is entered into the symbol table along with the formal parameters.
  - Type checking : Within expressions, a function is treated like any other operator.
  - Function calls : When generating three-address code for a function call "id (E, E, ..., E)" it is sufficient to generate the three-address code for evaluating the arguments to addresses, followed by a 'param' instruction for each parameter. If we do not want to mix the parameter-evaluating instructions with the 'param' instructions, the attribute 'E.addr' for each expression E can be saved in a data structure such as a queue. Once all the expressions are translated, the 'param' instructions can be generated as the queue is emptied.
- Example :

```
A → E A        { Add E.addr into the Queue }
A → ϵ          { Initialize Queue to empty }
E → id (A)     { for each item p in Queue do
                     emit ('param' p);
                 emit ('call' id.lexeme); }
```

Switch Statements

- The syntax of the switch statement is given below:

```
switch (E)
{
    case V1 : S1
    case V2 : S2
        ...
    case Vn : Sn
    default : S
}
```

- The intermediate code for the above switch statement is given below:

```
    code to evaluate E into t
    if t != V1 goto L1
    S1.code
    goto next
L1: if t != V2 goto L2
    S2.code
    goto next
L2:
        ...
Ln-1: if t != Vn goto Ln
    Sn.code
    goto next
Ln  : Sn code
next :
```