Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

Type Conversions

- **Example**
  - Suppose that integers are converted to floats when necessary, using a unary operator (float)

$$2*3.14 \quad \Longrightarrow \quad \begin{array}{l} t_1 = \text{(float) } 2 \\ t_2 = t_1 * 3.14 \end{array}$$

- Type conversion rules vary from language to language
- Conversion from one type to another is said to be implicit if it is done automatically by the compiler
- Conversion is said to be explicit if the programmer must write something to cause the conversion

Type Conversions

- **Example:** Introducing type conversions into expression evaluation

$$
\boxed{E \rightarrow E_1 + E_2} \quad \begin{array}{l} \{\ E.type \;=\; max(E_1.type, E_2.type); \\ \quad a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\ \quad a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\ \quad E.addr = \textbf{new}\ Temp\,(); \\ \quad gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \} \end{array}
$$

  - $\boldsymbol{max(t_1, t_2)}$ takes two types t_1 and t_2 and returns the maximum
  - $\boldsymbol{widen(a, t, w)}$ generates type conversions if needed to widen the contents of an address a of type t into a value of type w

Control Flow

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

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

- **Example** $\boxed{B \;\rightarrow\; B \;||\; B \;\mid\; B \;\&\&\; B \;\mid\; !\;B \;\mid\; (\;B\;) \;\mid\; E\;\textbf{rel}\;E \;\mid\; \textbf{true} \;\mid\; \textbf{false}}$
  - We use the attribute $rel.op$ to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by rel

- Given the expression $B_1||B_2$, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2. Similarly, given $B_1\&\&B_2$, if B_1 is false, then the entire expression is false.

Control Flow

- **Short-Circuit Code**
  - In short-circuit (or jumping) code, the boolean operators `&&`, `||`, and `!` translate into jumps

  - **Example**

```
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:
```

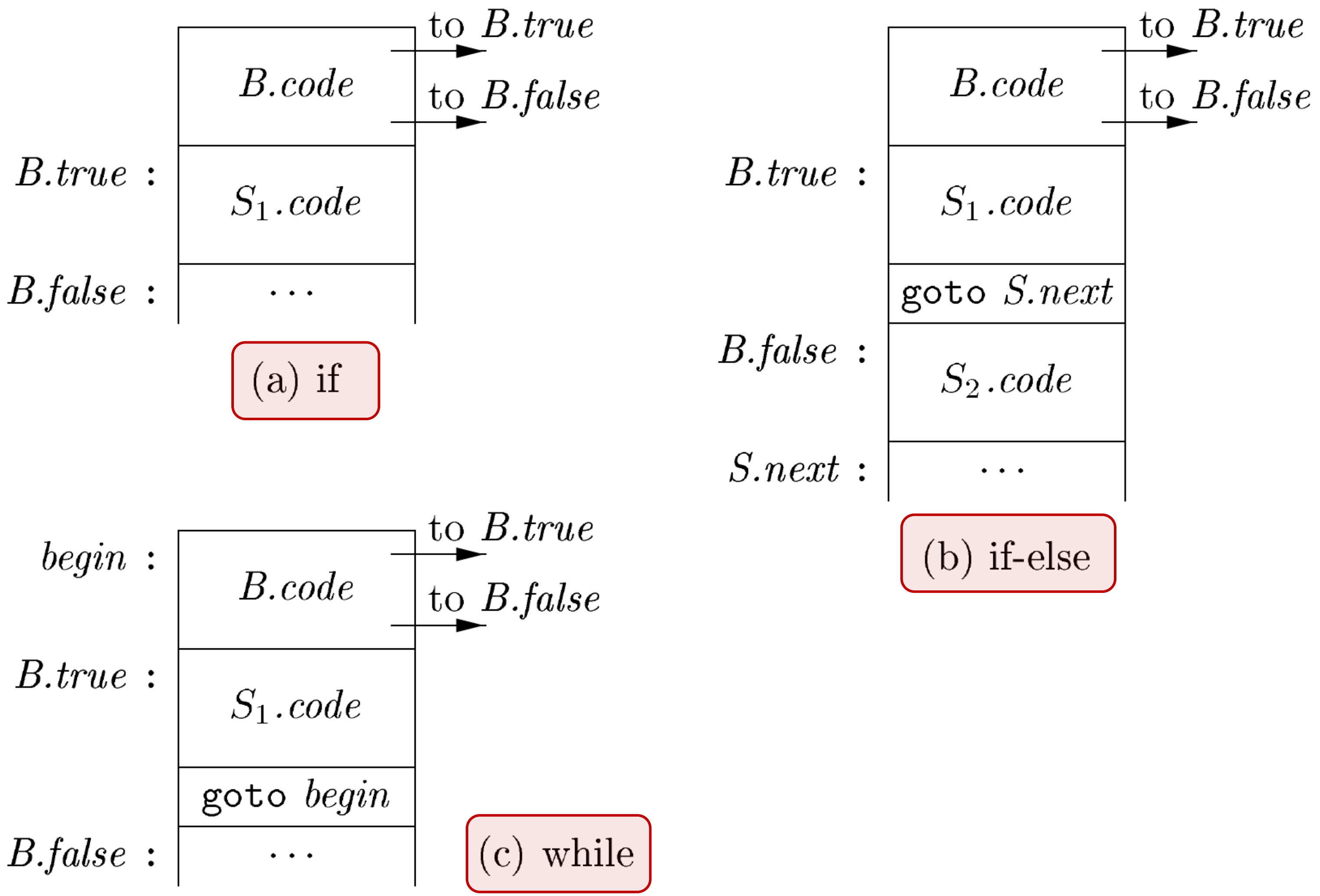
Control Flow

- **Flow-of-Control Statements**
  - **Example**

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

  - Non-terminal ***B*** represents a boolean expression and non-terminal ***S*** represents a statement

Control Flow



Control Flow

- **Example:** Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

  - ***newlabel***() creates a new label each time it is called
  - ***label***(***L***) attaches label L to the next three-address instruction to be generated
  - Token **assign** in the production $S \rightarrow$ **assign** is a placeholder for assignment statements

Control Flow

- **Example (cont.):** Syntax-directed definition for flow-of-control statements

$S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad\quad \| \, label(B.true) \, \| \, S_1.code$ $\quad\quad \| \, gen('\texttt{goto}' \; S.next)$ $\quad\quad \| \, label(B.false) \, \| \, S_2.code$

Control Flow

- **Example (cont.):** Syntax-directed definition for flow-of-control statements

$S \rightarrow \textbf{while} \; (\; B \;) \; S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \; \vert\vert \; B.code \; \vert\vert \; label(B.true) \; \vert\vert \; S_1.code \; \vert\vert \; gen('\texttt{goto}' \; begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \; \vert\vert \; label(S_1.next) \; \vert\vert \; S_2.code$

Control Flow

- If B1 is true, then we immediately know that B itself is true, so B1.true is the same as B.true
- If B1 is false, then B2 must be evaluated, so we make B1.false be the label of the first instruction in the code for B2
- The true and false exits of B2 are the same as the true and false exits of B

- **Example:** Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$

Control Flow

- **Example (cont.):** Generating three-address code for booleans

$B \rightarrow \, ! \, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \Vert \ E_2.code$ $\Vert \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\Vert \ gen('\texttt{goto}' \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

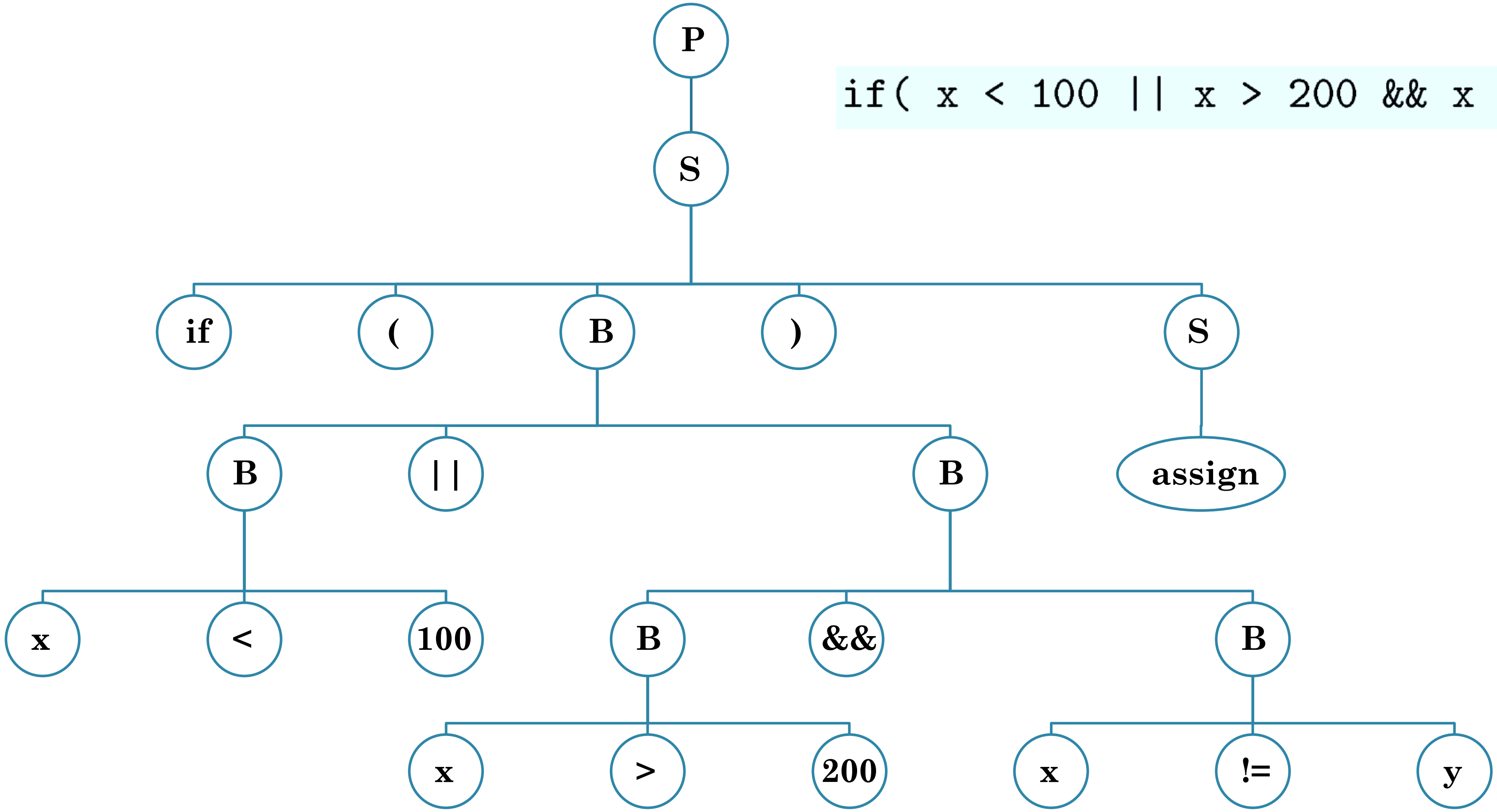
Control Flow

- **Example:** Obtain the code of the following statement

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

  - The grammar is:
    - $S \rightarrow \textbf{if}\ (\ B\ )\ S \mid \textbf{if}\ (\ B\ )\ S\ \textbf{else}\ S \mid \textbf{while}\ (\ B\ )\ S \mid SS \mid \textbf{assign}$
    - $B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$

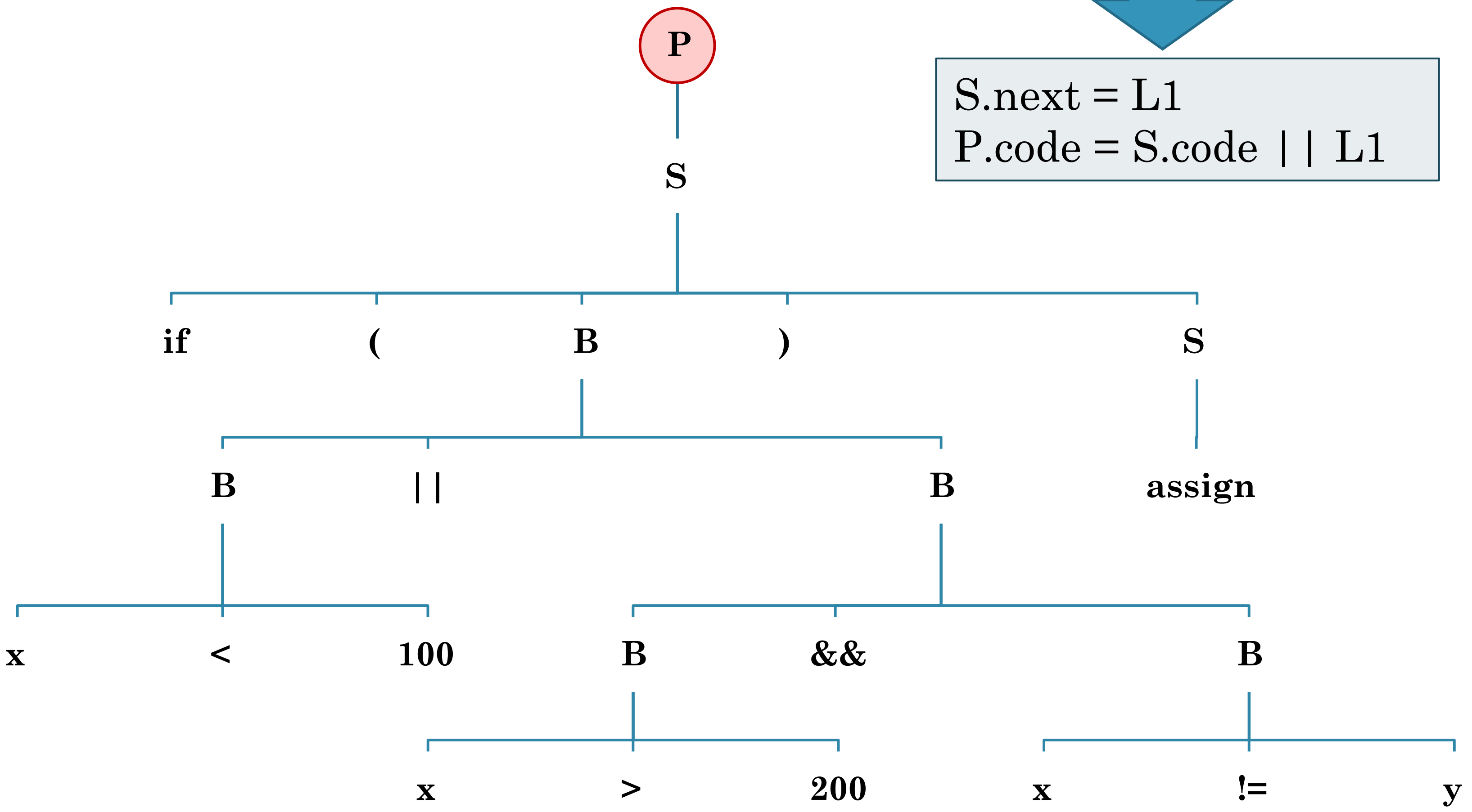
Control Flow




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

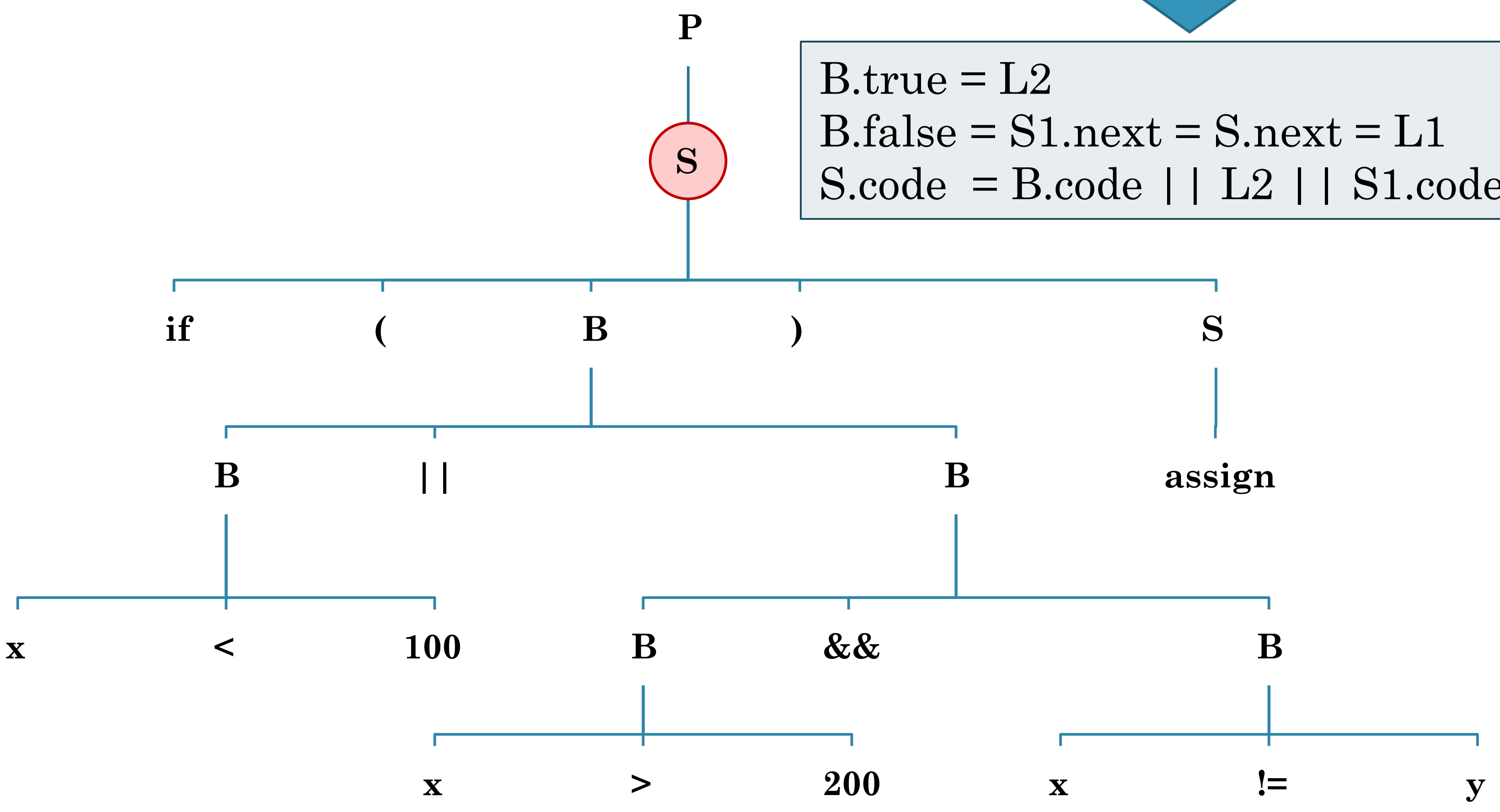
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$

S.next = L1
P.code = S.code | | L1



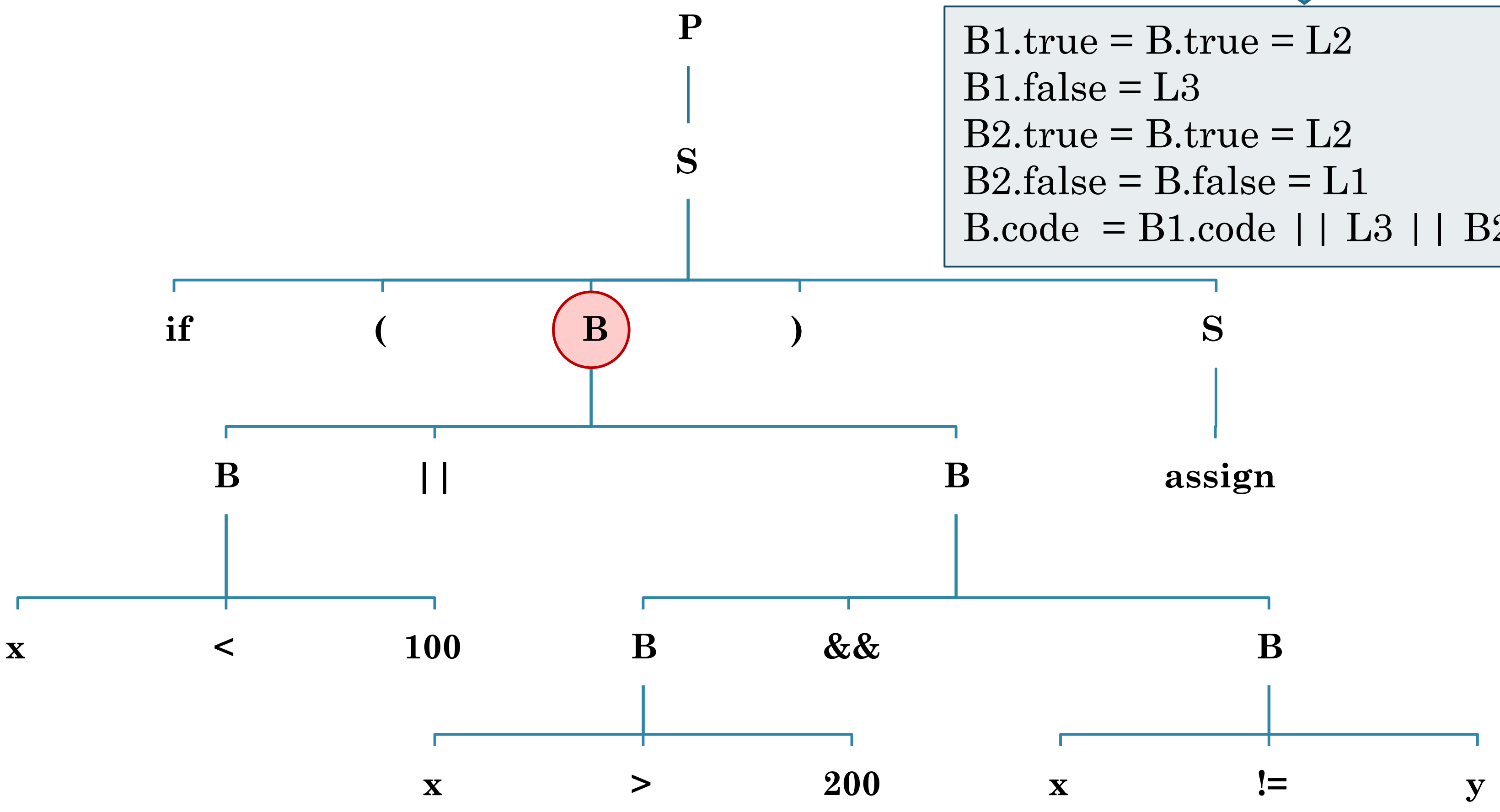
$$S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \qquad \begin{array}{l} B.true = newlabel() \\ B.false = S_1.next = S.next \\ S.code = B.code \; || \; label(B.true) \; || \; S_1.code \end{array}$$

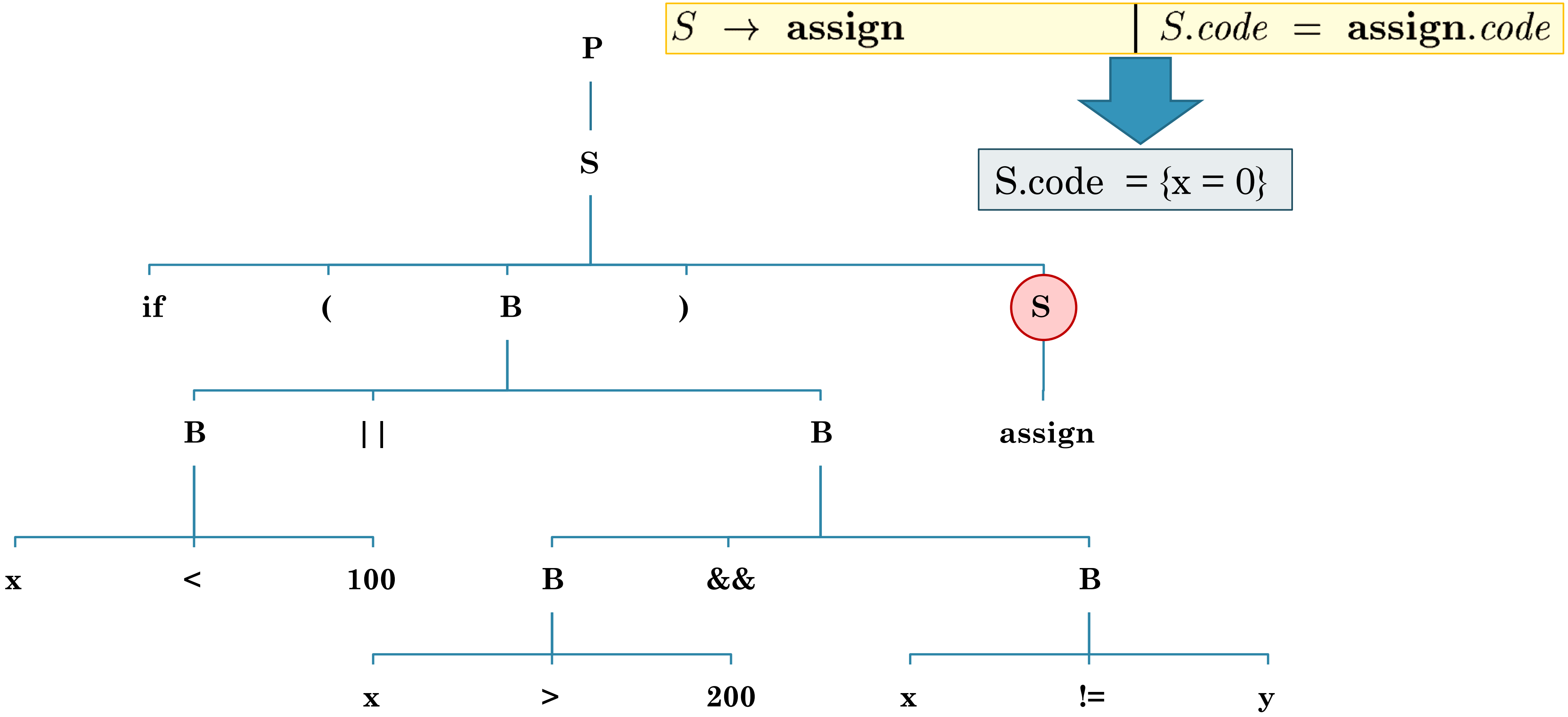
B.true = L2
B.false = S1.next = S.next = L1
S.code = B.code | | L2 | | S1.code



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

B1.true = B.true = L2
B1.false = L3
B2.true = B.true = L2
B2.false = B.false = L1
B.code = B1.code | | L3 | | B2.code

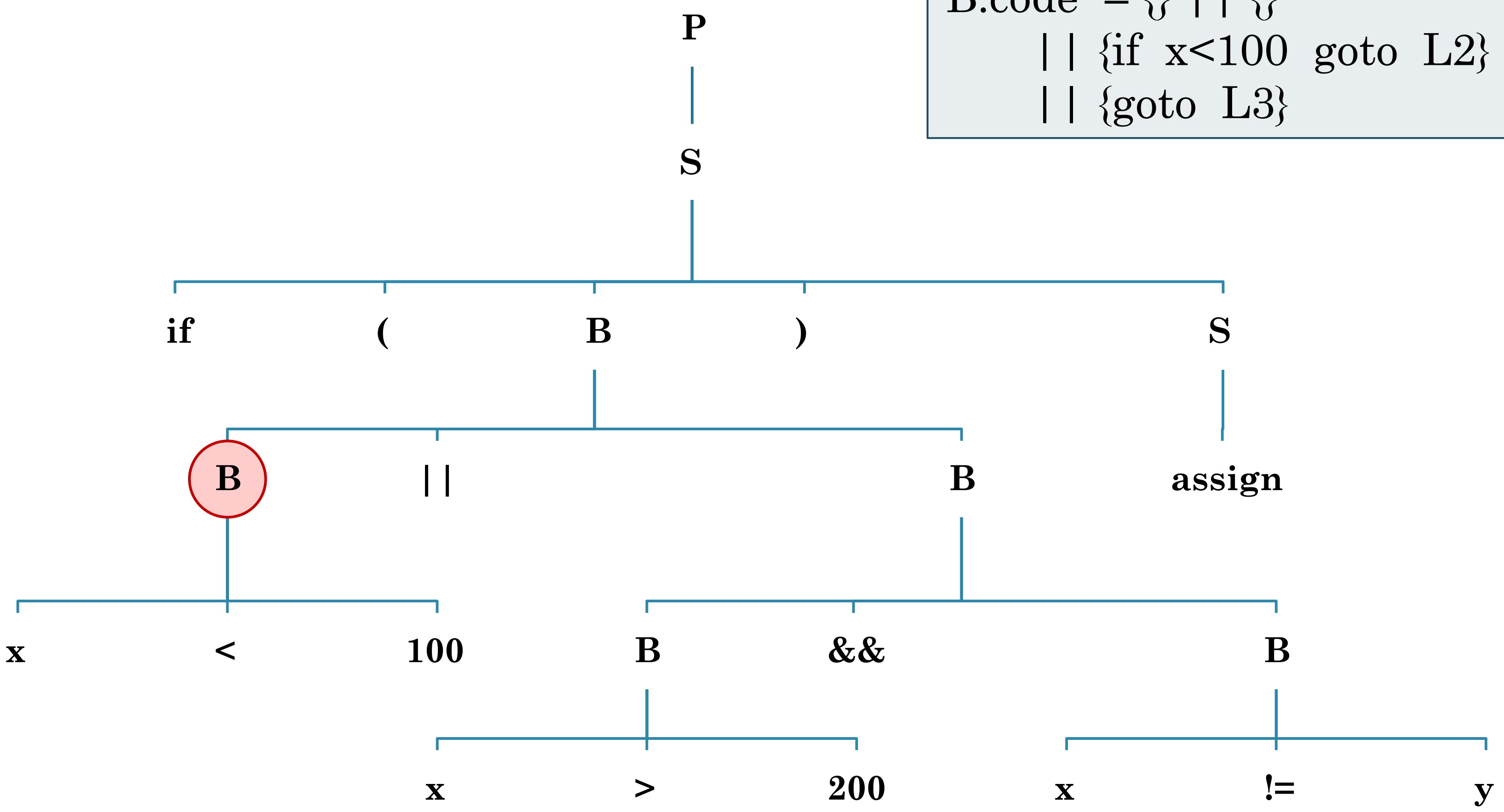




S → assign | S.code = assign.code
S.code = {x = 0}
P
S
if
(
B
)
S
B
| |
B
assign
x
<
100
B
&&
B
x
>
200
x
!=
y

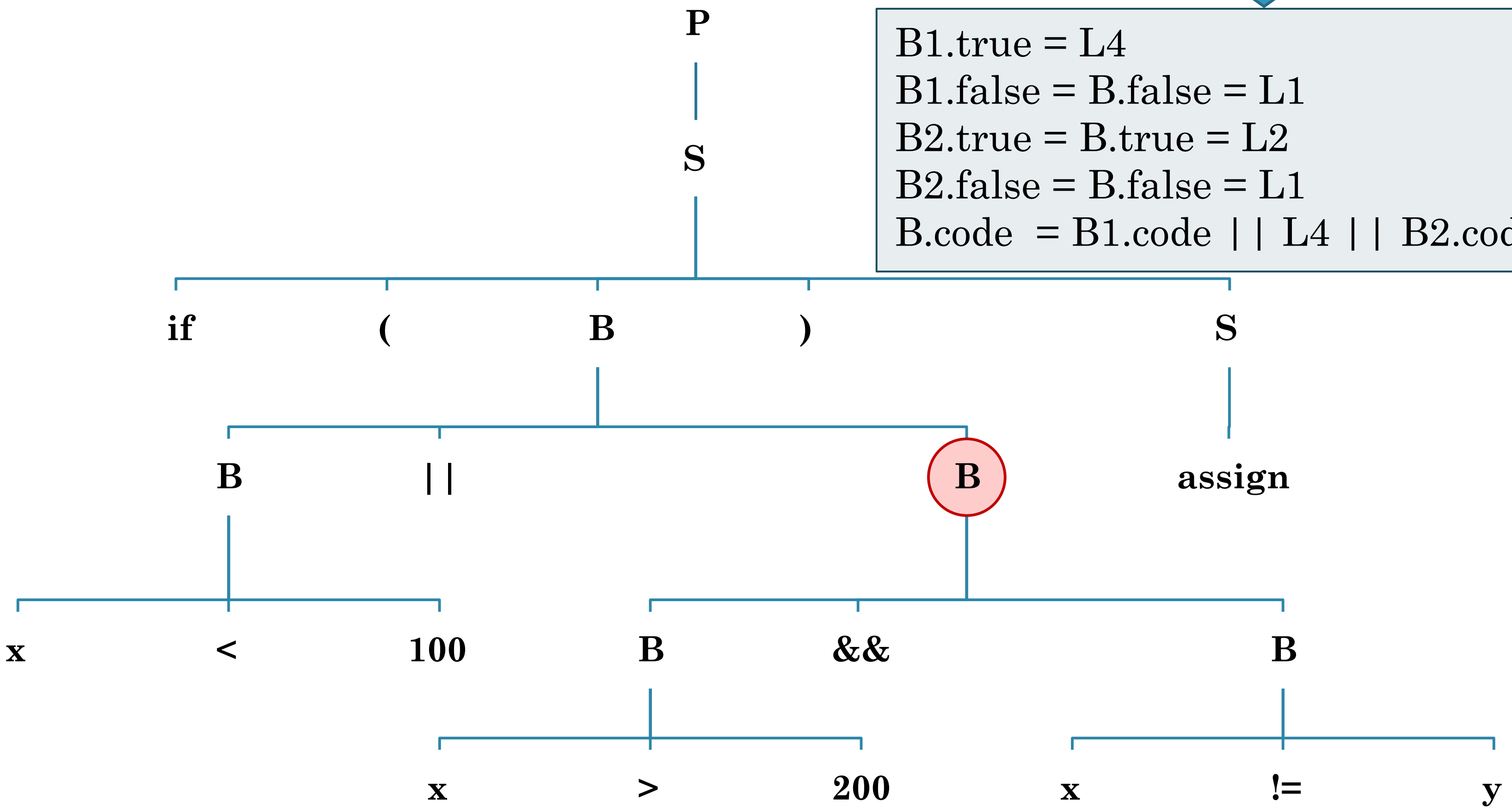
$B \rightarrow E_1 \; \textbf{rel} \; E_2$	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\| \; gen(\texttt{'if'} \; E_1.addr \; \textbf{rel}.op \; E_2.addr \; \texttt{'goto'} \; B.true)$ $\|\| \; gen(\texttt{'goto'} \; B.false)$

B.code = {} | | {}
| | {if x<100 goto L2}
| | {goto L3}



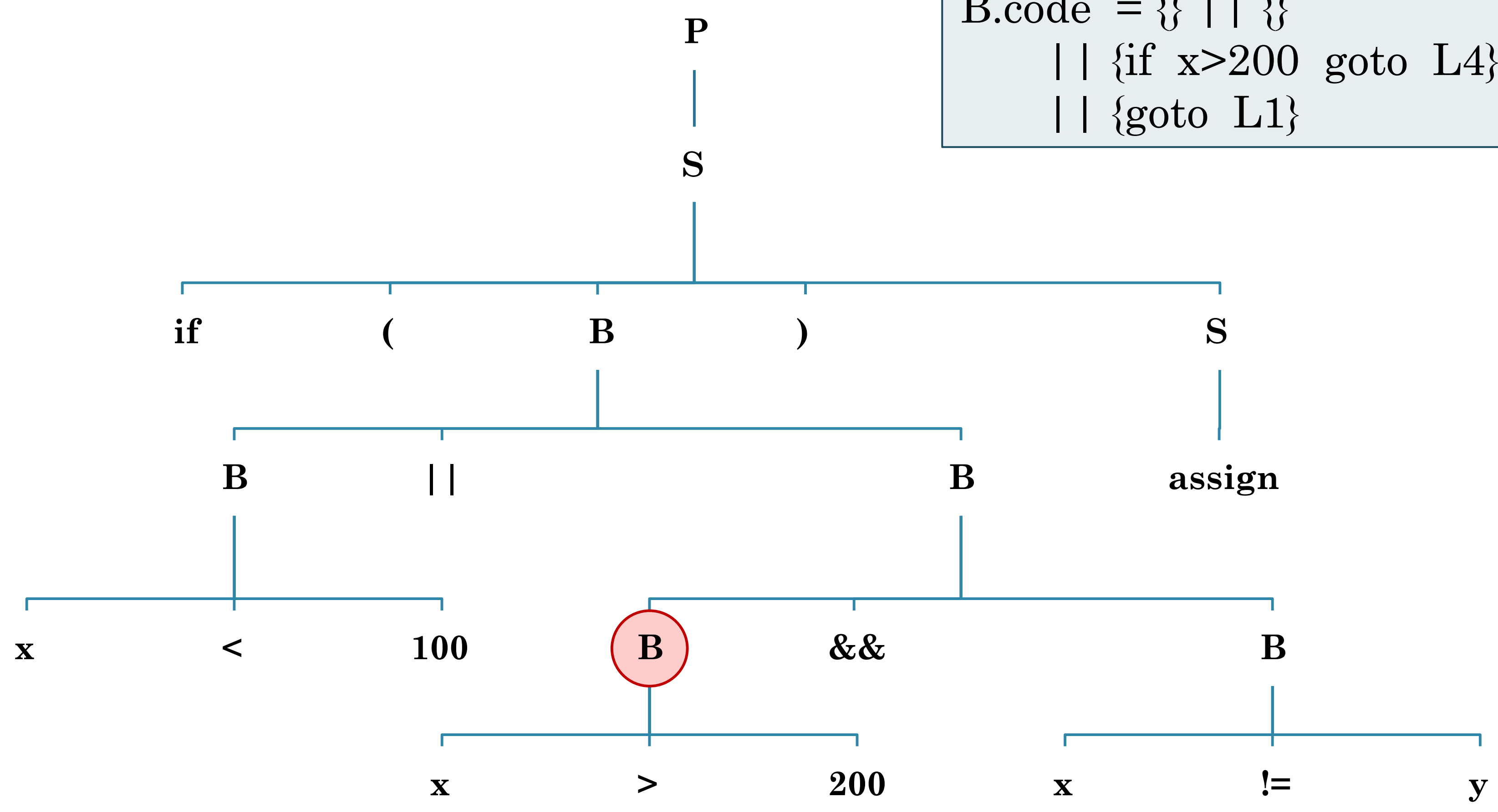
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \Vert \ label(B_1.true) \ \Vert \ B_2.code$

B1.true = L4
B1.false = B.false = L1
B2.true = B.true = L2
B2.false = B.false = L1
B.code = B1.code | | L4 | | B2.code



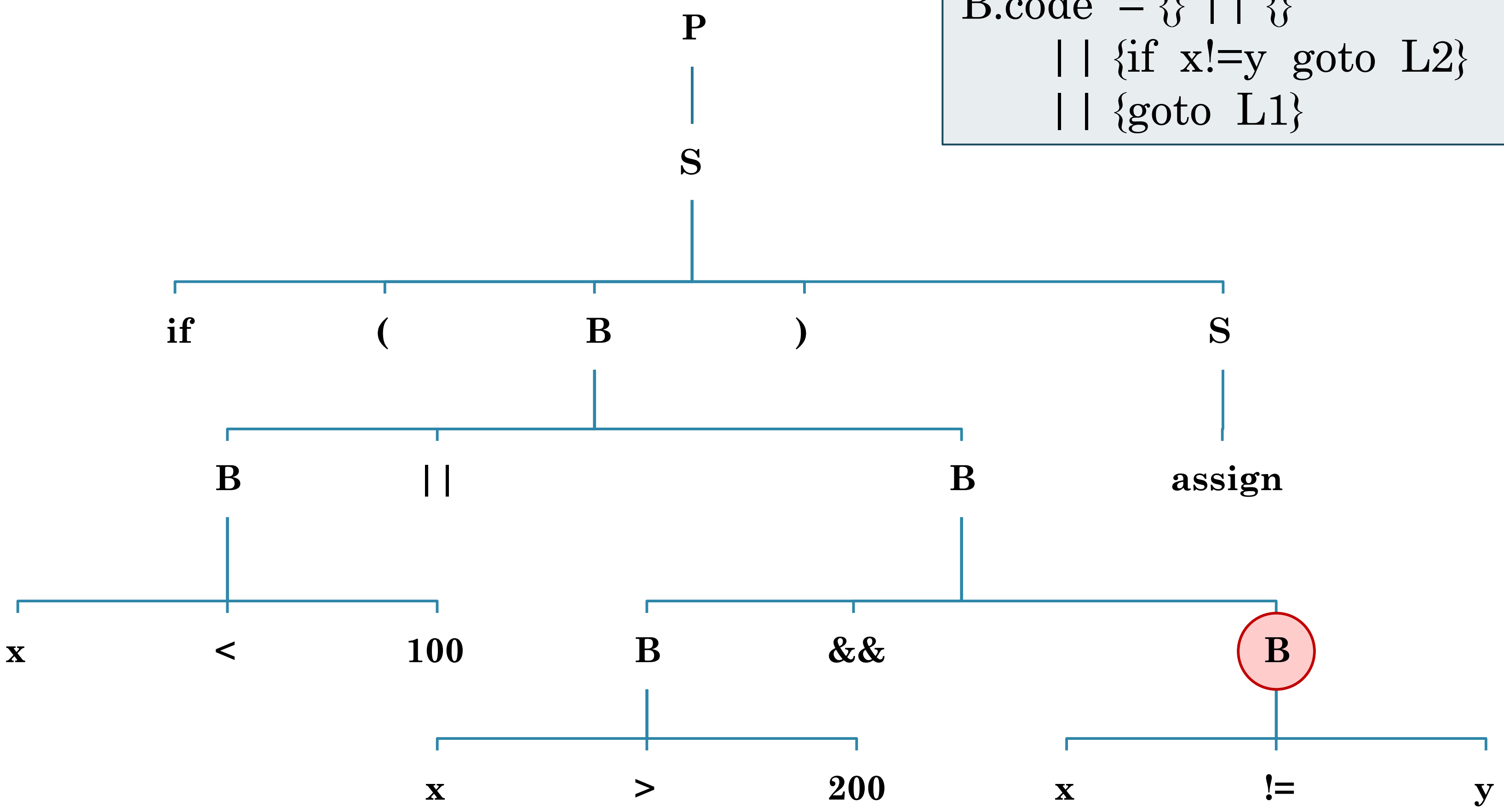
$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\|\; gen('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\|\|\; gen('\texttt{goto}'\ B.false)$

B.code = {} | | {}
| | {if x>200 goto L4}
| | {goto L1}

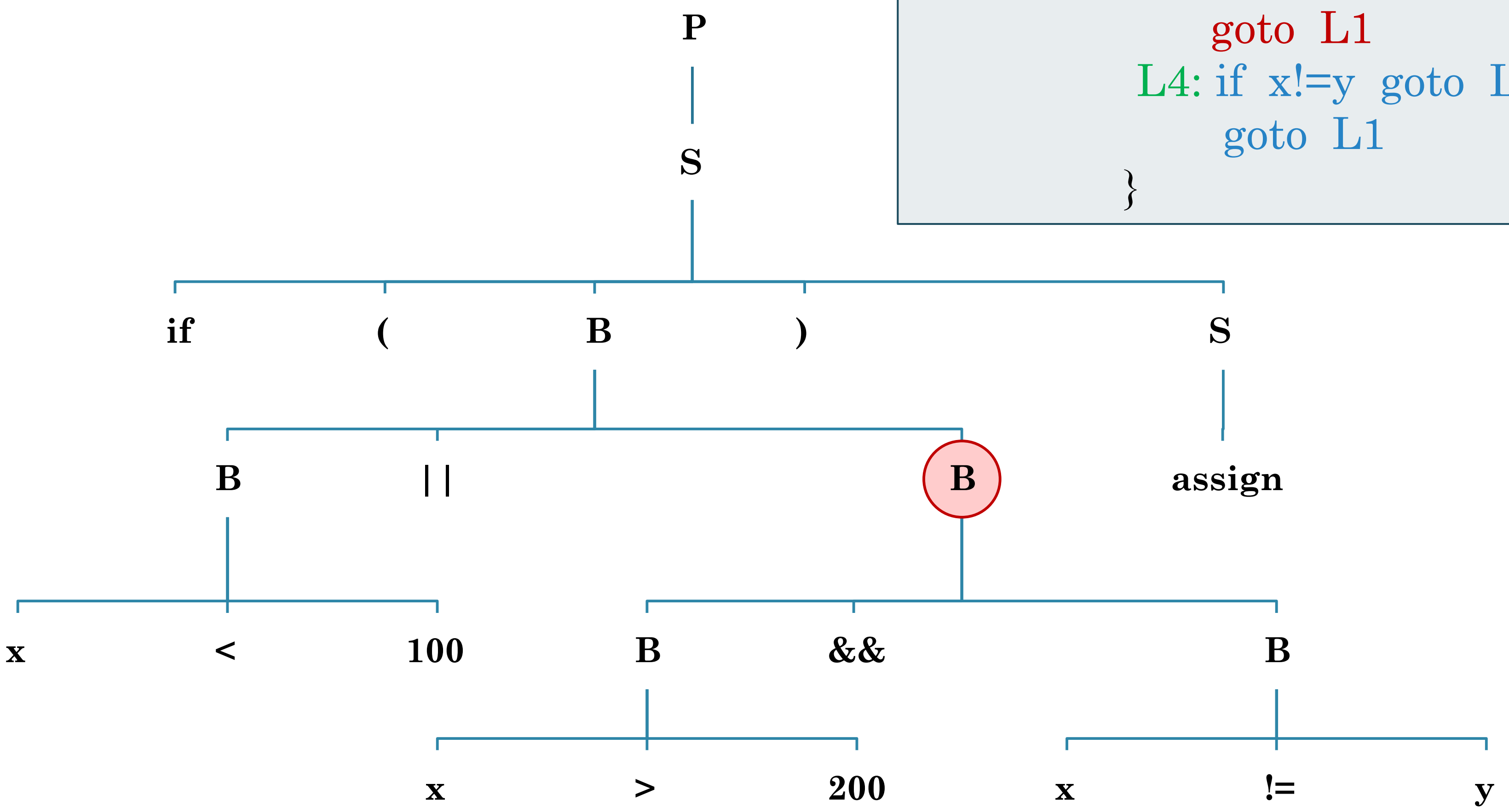


$$B \rightarrow E_1 \ \mathbf{rel} \ E_2 \qquad \begin{array}{l} B.code = E_1.code \ || \ E_2.code \\ \quad || \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true) \\ \quad || \ gen('\texttt{goto}' \ B.false) \end{array}$$

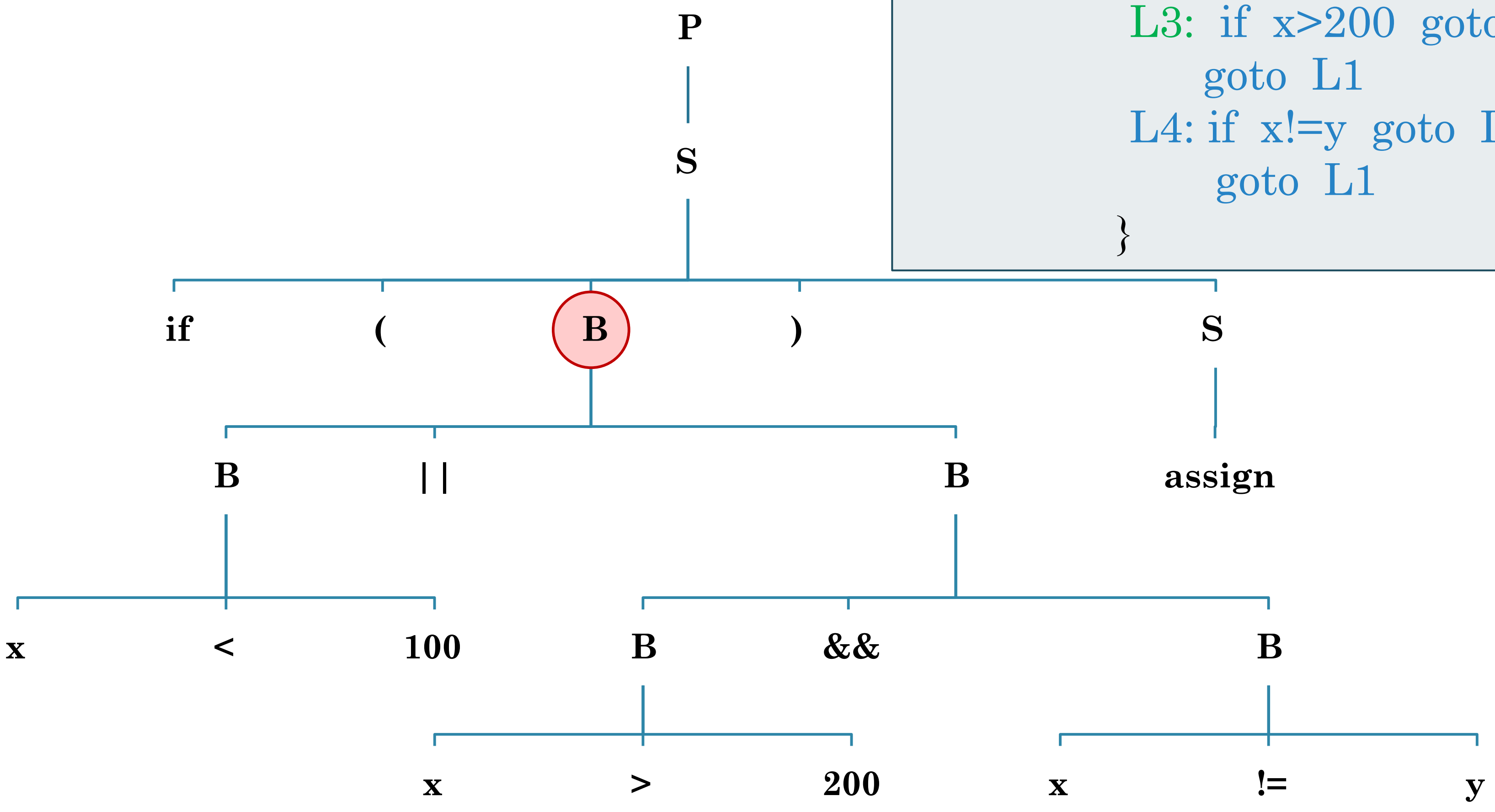
B.code = {} | | {}
| | {if x!=y goto L2}
| | {goto L1}



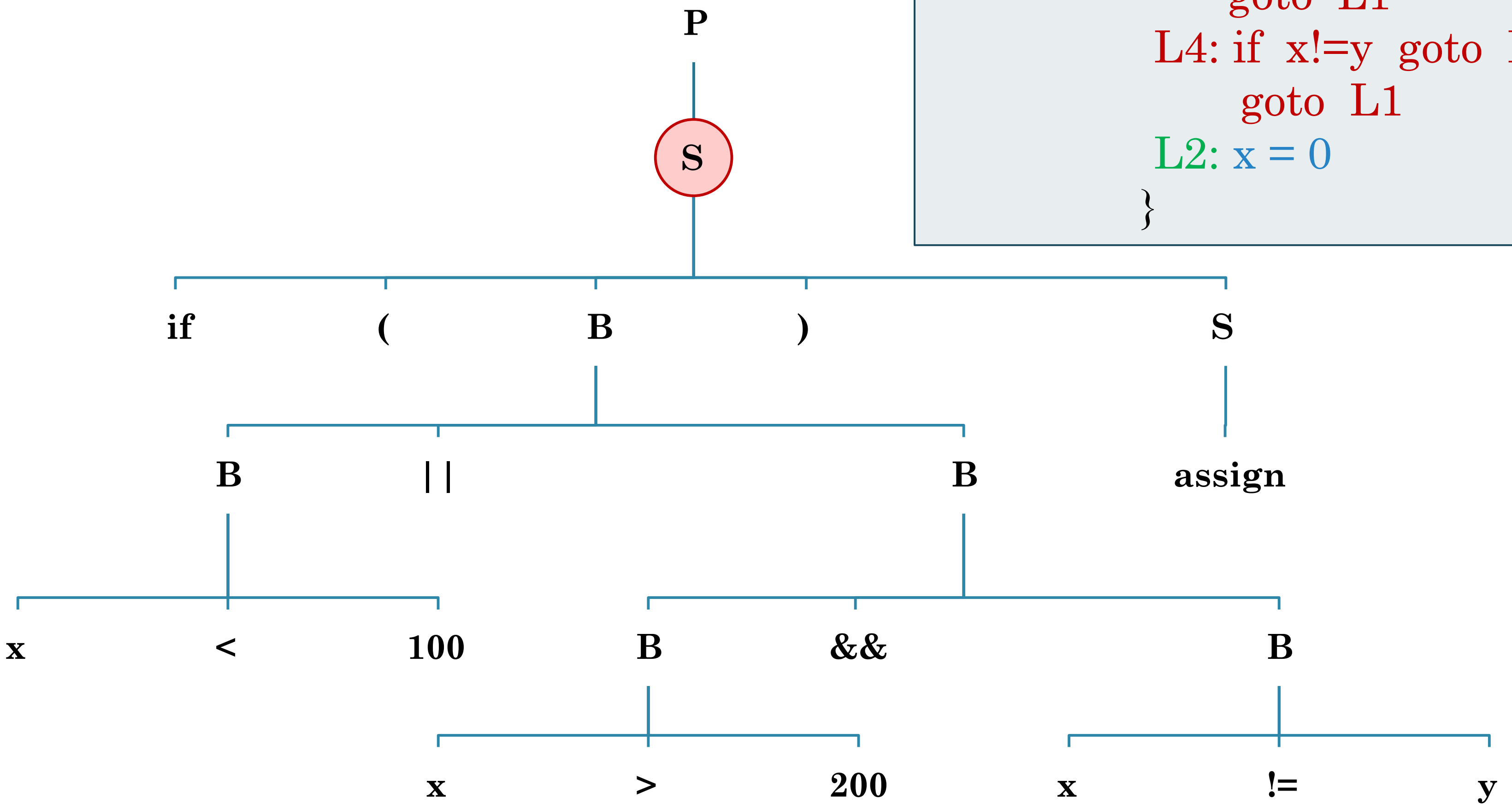
```
B.code  = B1.code || L4 || B2.code
        = {
              if x>200 goto L4
              goto L1
          L4: if x!=y goto L2
              goto L1
          }
```



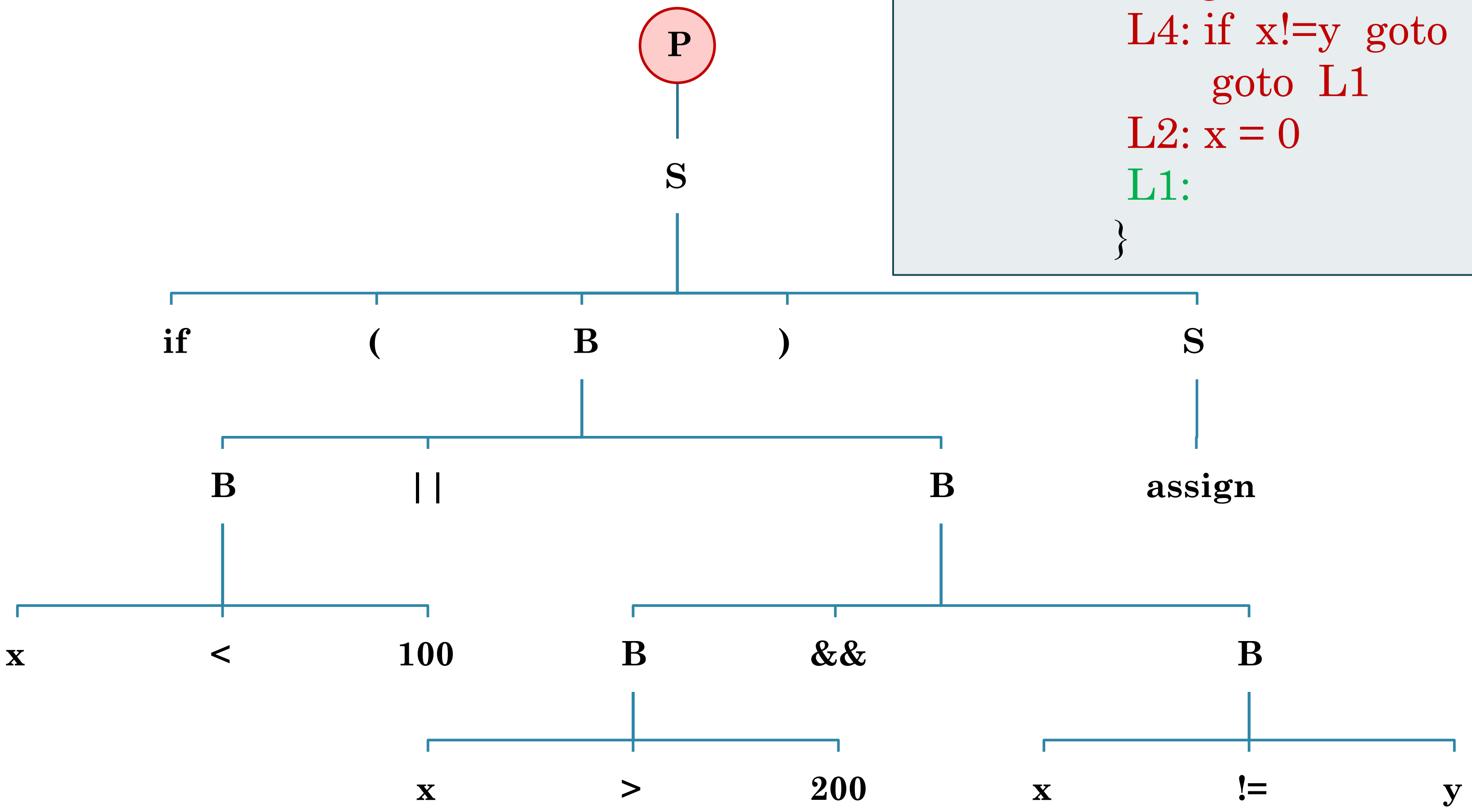
```
B.code  = B1.code | | L3 | | B2.code
        = {
              if x<100 goto L2
              goto L3
          L3: if x>200 goto L4
              goto L1
          L4: if x!=y goto L2
              goto L1
          }
```



```
S.code = B.code || L2 || S1.code
       = {
               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
         }
```



```
P.code  = S.code | | L1
        = {
              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:
          }
```



Translation of Switch-Statements

switch (E) {
    case V_1: S_1
    case V_2: S_2
    $\dots$
    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`
	$\dots$
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
	$\dots$
	`if t =` V_{n-1} `goto` L_{n-1}
	`goto` L_n
`next:`	

Translation of Switch-Statements

switch (E) {
    case V_1: S_1
    case V_2: S_2
    $\ldots$
    case V_{n-1}: S_{n-1}
    default: S_n
}

	code to evaluate E into `t`
	`if t !=` V_1 `goto` $\mathtt{L}_1$
	code for S_1
	`goto next`
$\mathtt{L}_1$:	`if t !=` V_2 `goto` $\mathtt{L}_2$
	code for S_2
	`goto next`
$\mathtt{L}_2$:	
	$\ldots$
$\mathtt{L}_{n-2}$:	`if t !=` V_{n-1} `goto` $\mathtt{L}_{n-1}$
	code for S_{n-1}
	`goto next`
$\mathtt{L}_{n-1}$:	code for S_n
`next:`	

Code Optimization

Code Optimization

- ***Elimination of unnecessary instructions in object code***, or ***the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing*** is usually called "code improvement" or "code optimization“
- **Local code optimization**
  - Code improvement within a basic block
- **Global code optimization**
  - Code improvement across basic blocks

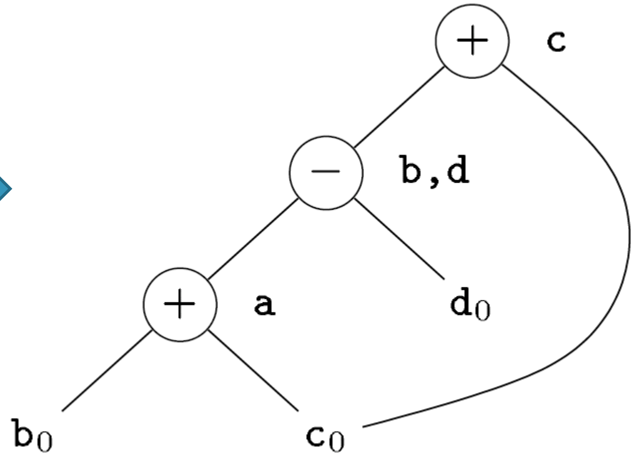
Local Optimization

- **DAG Representation of Basic Blocks**
  - Many important techniques for local optimization begin by transforming a basic block into a DAG
  - We construct a DAG for a basic block as follows:
    1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block
    2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s
    3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block
    4. Certain nodes are designated **output nodes**
       - Output nodes are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph

Local Optimization

- **Finding Local Common Subexpressions**
  - Common subexpressions can be detected by noticing, as a new node ***M*** is about to be added, whether there is an existing node ***N*** with the same children, in the same order, and with the same operator
  - **Example:** A DAG for the following block

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Local Optimization

- **Finding Local Common Subexpressions**
  - Since there are only three non-leaf nodes in the DAG of the previous example, the basic block can be replaced by a block **with only three statements**
  - In fact, if ***b*** is not live on exit from the block, then we do not need to compute that variable, and can use ***d***
  - The block then becomes

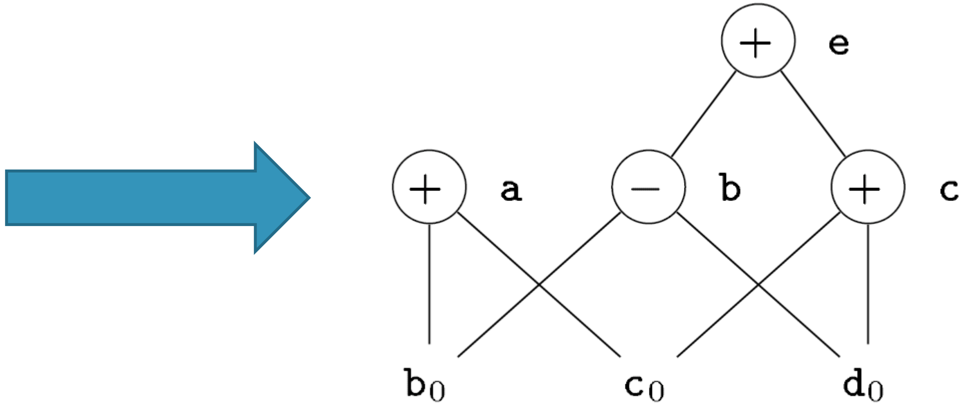
```
a = b + c
d = a - d
c = d + c
```

  - However, if both ***b*** and ***d*** are live on exit, then a fourth statement must be used to copy the value from one to the other

Local Optimization

- **Finding Local Common Subexpressions**
  - **Example**
    - The following DAG does not exhibit any common subexpressions

```
a = b + c
b = b - d
c = c + d
e = b + c
```



Local Optimization

- **Dead Code Elimination**
  - The operation on DAGs that corresponds to dead-code elimination can be implemented as follows
    - Delete from a DAG any root that has no live variables attached
    - Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code
  - **Example**
    - In the previous example, if ***a*** and ***b*** are live but ***c*** and ***e*** are not, we can immediately remove the root labeled ***e***
    - Then, the node labeled ***c*** becomes a root and can be removed
    - The roots labeled ***a*** and ***b*** remain, since they each have live variables attached