Intermediate-Code Generation

Syntax-directed Techniques for Constructing Compiler Front-End

The starting point for a syntax-directed translator is a *grammar* for the source language.

The result of syntax-analysis is a representation of the source program, called *intermediate code*

Two primary forms of intermediate code are illustrated in Figure 2.46

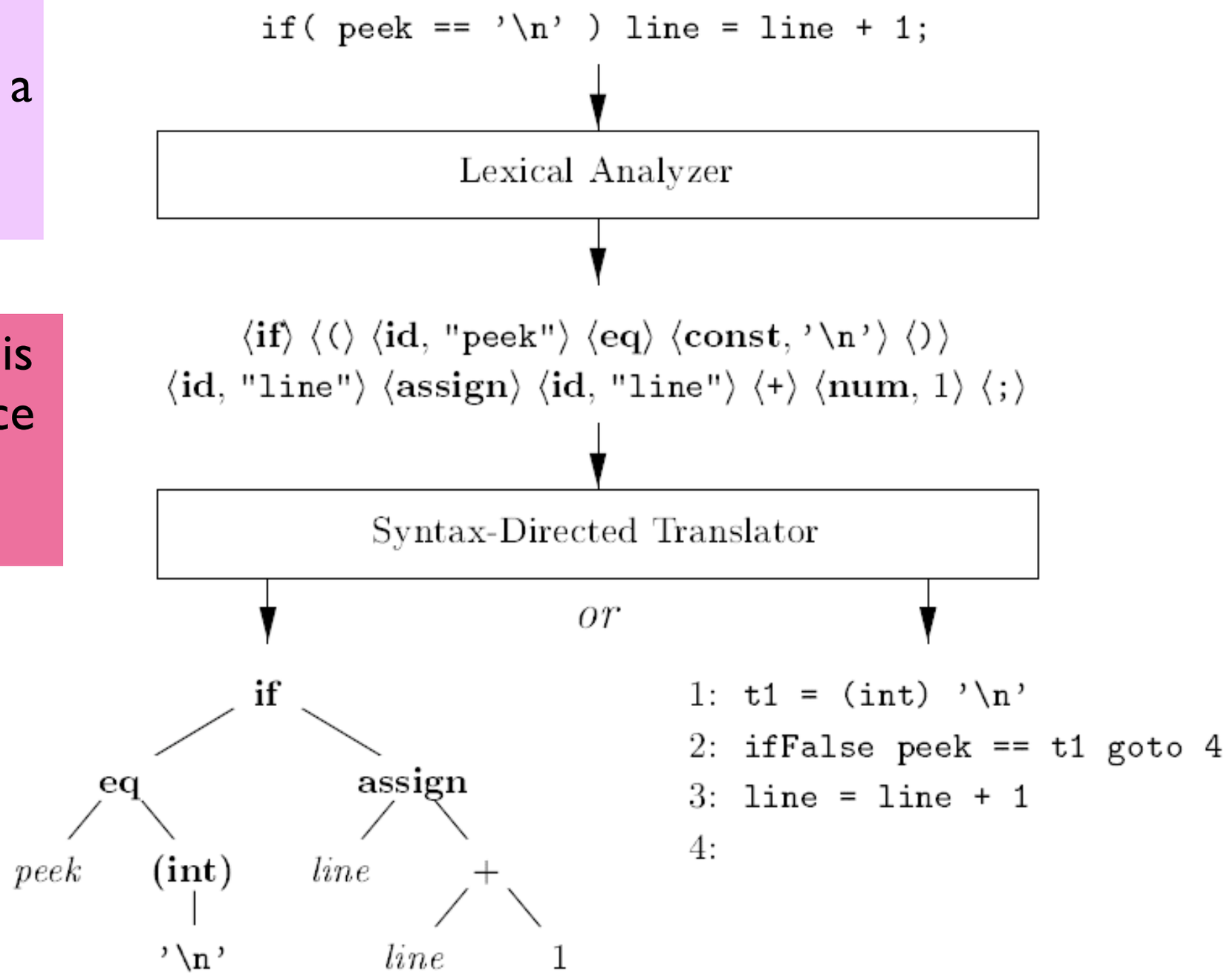


Figure 2.46: Two possible translations of a statement

Symbol Tables

- ***Symbol tables*** hold information about identifiers
- This information is inserted when declarations are analyzed
- A semantic action gets information from the symbol table when the identifier is subsequently used (e.g. as a factor in an expression)
- Symbol tables must support *multiple declarations* of the same identifier within a program
- The notion of *scope* is crucial: set up a separate symbol table for each scope.

Symbol Tables

```
1)  {    int x1; int y1;
2)       {    int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)         ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

Example 2.15

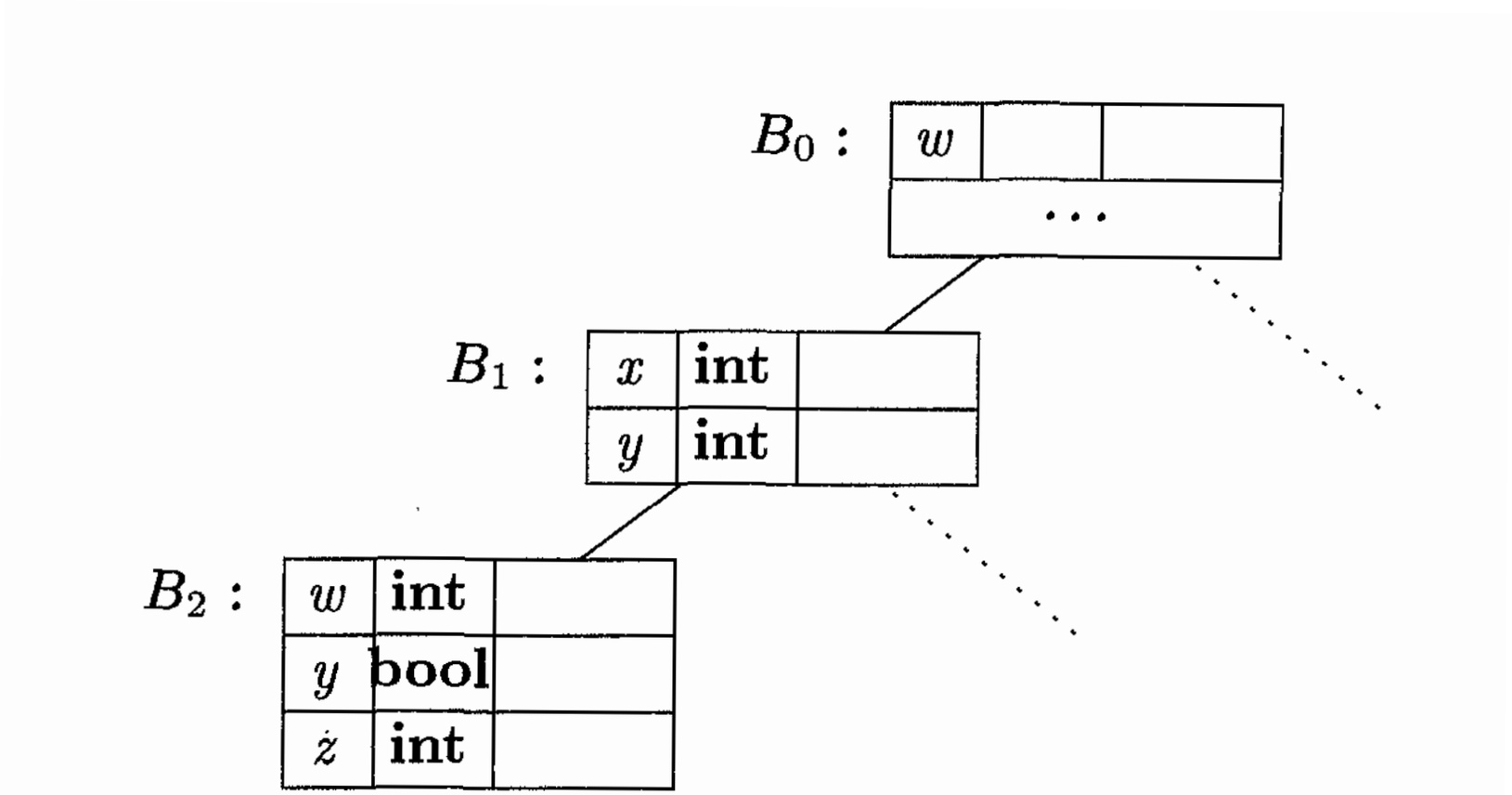
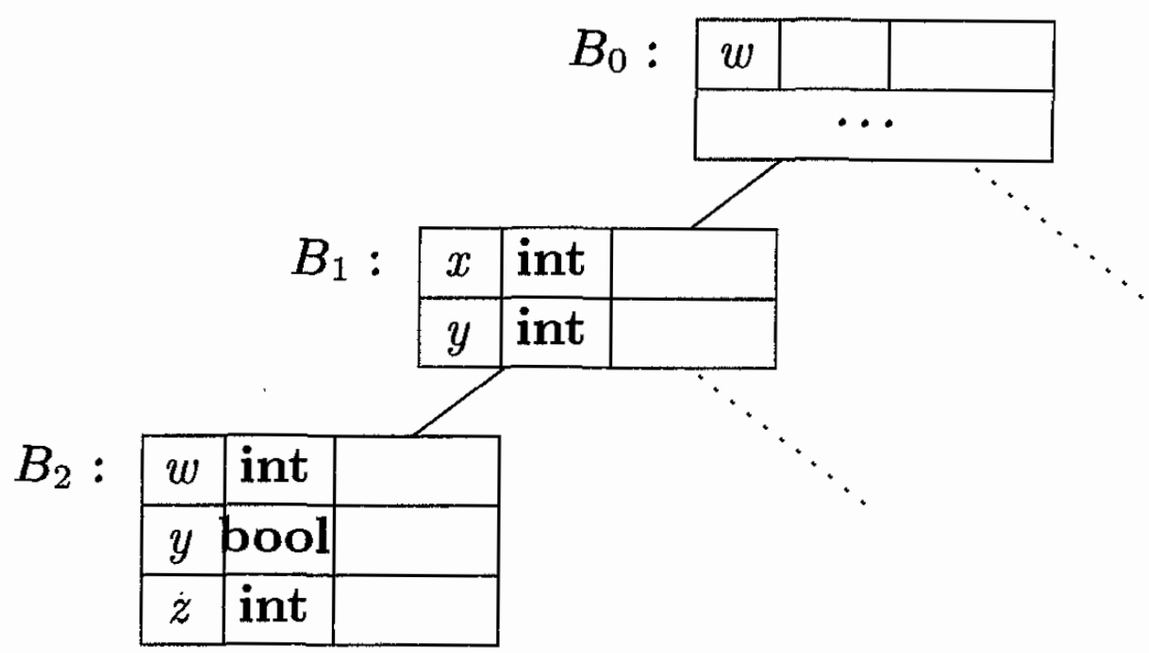
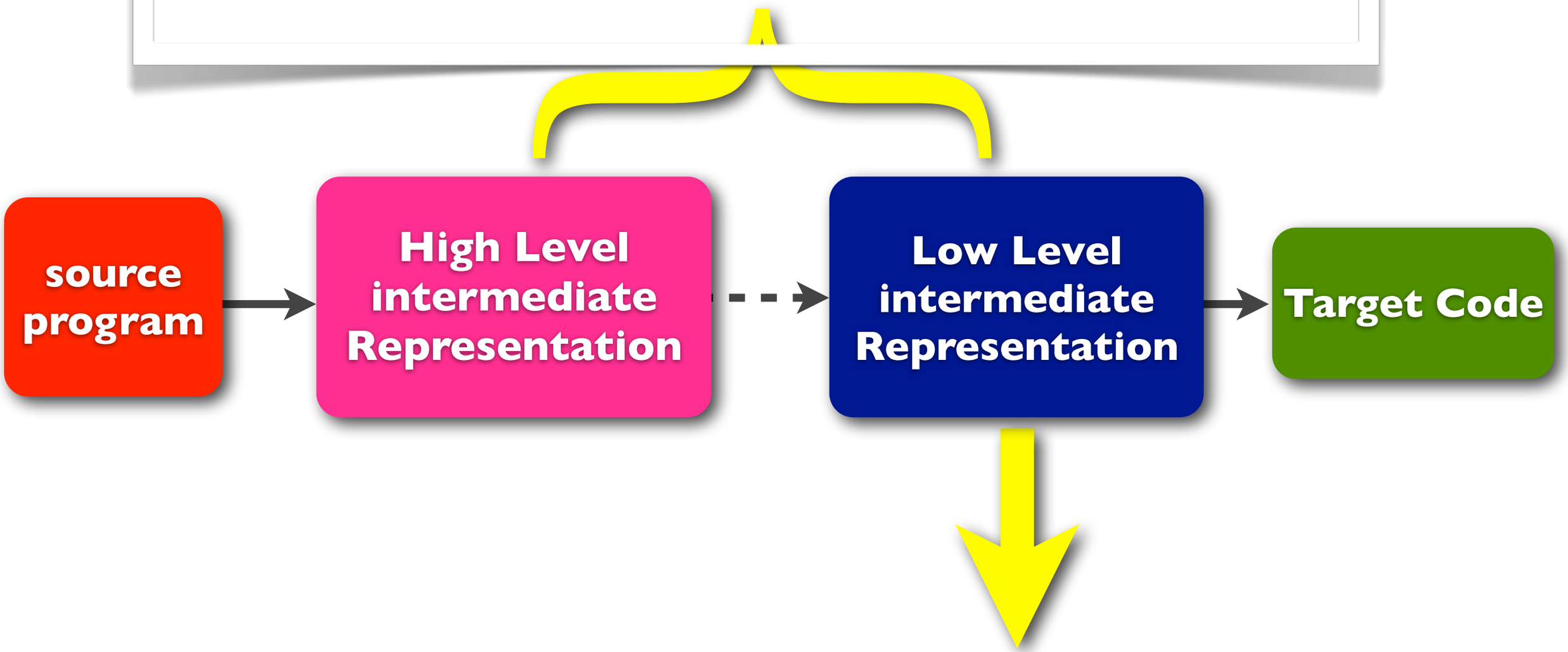


Figure 2.36: Chained symbol tables for Example 2.15

```
1)package symbols;
2)import java.util.*; import lexer.*; import inter.*;
3)public class Env {
4)    private Hashtable table;
5)    protected Env prev;
6)    public Env(Env p) {
7)           table = new Hashtable(); prev = p;
8)    }
9)    public void put(Token w, id i) {
10)      table.put(w, i);
11)   }
12)   public id get(Token w) {
13)      for( Env e = this; e != null; e = e.prev ) {
14)         id found = (id)(e.table.get(w));
15)         if( found != null ) return found;
16)      }
17)      return null;
18)   }
19)}
```

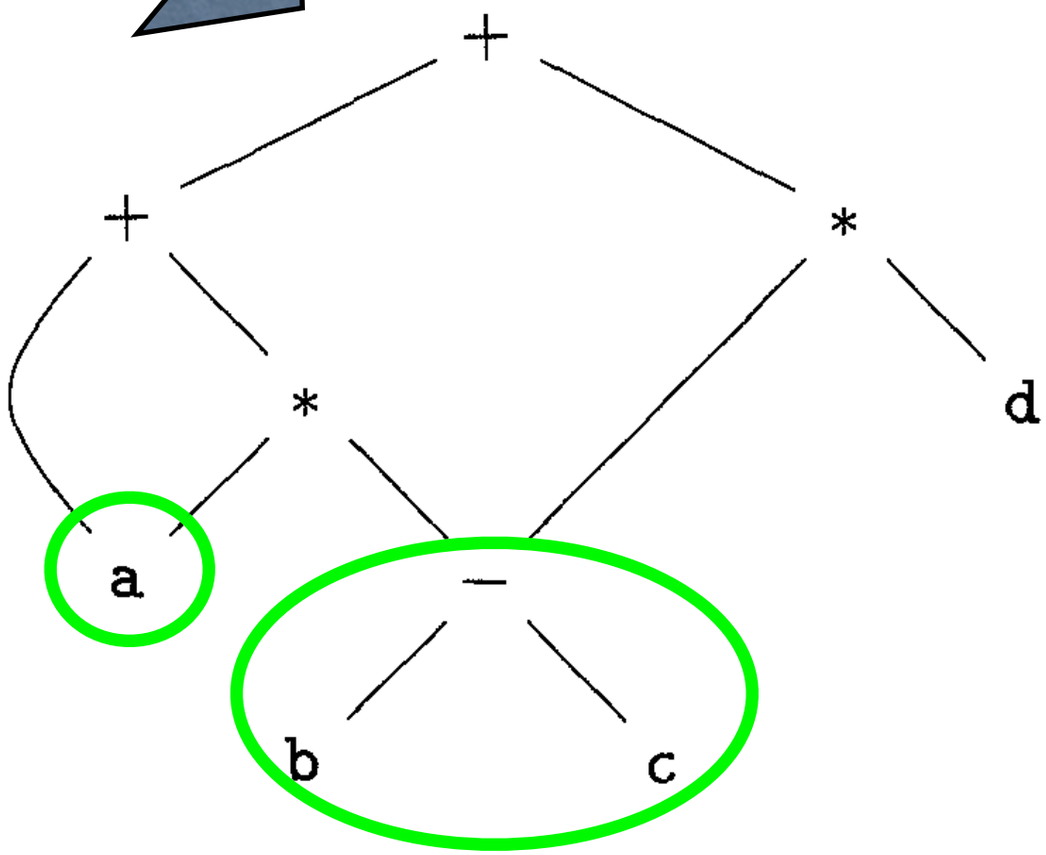


Sequence of intermediate representations



DAG (variants of syntax trees)

```
a+a*(b-c)+(b-c)*d
```



PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow$ **id**	$T.node$ = **new** $Leaf($**id**, **id**$.entry)$
6) $T \rightarrow$ **num**	$T.node$ = **new** $Leaf$ (**num**, **num**$.val)$

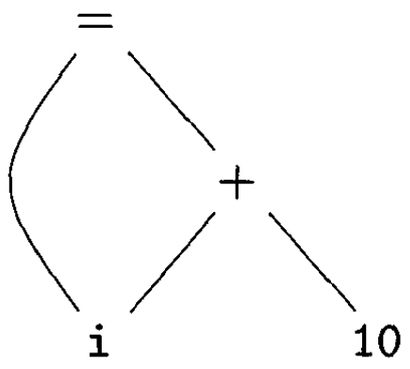
1. $P_1 = Leaf$ (**id**, entry-a)
2. $P_2 = Leaf$ (**id**, entry-a) $= P_1$
3. $P_3 = Leaf$ (**id**, entry-b)
4. $P_4 = Leaf$ (**id**, entry-c)
5. $P_5 = Node("-", P_3, P_4)$
6. $P_6 = Node("*", P_1, P_5)$
7. $P_7 = Node("+", P_1, P_6)$
8. $P_8 = Leaf$ (**id**, entry-b) $= P_3$
9. $P_9 = Leaf$ (**id**, entry-c) $= P_4$
10. $P_{10} = Node("-", P_3, P_4) = P_5$
11. $P_{11} = Leaf$ (**id**, entry-d)
12. $P_{12} = Node("*", P_5, P_{11})$
13. $P_{13} = Node("+", P_7, P_{12})$

Steps for constructing the DAG

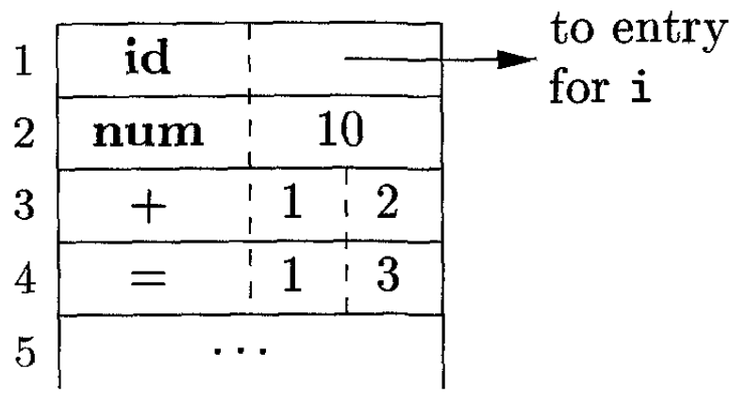
a+a*(b-c)+(b-c)*d

When the call to *Leaf* (**id**, *entry-a*) is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

The Value-Number Method for constructing DAGs



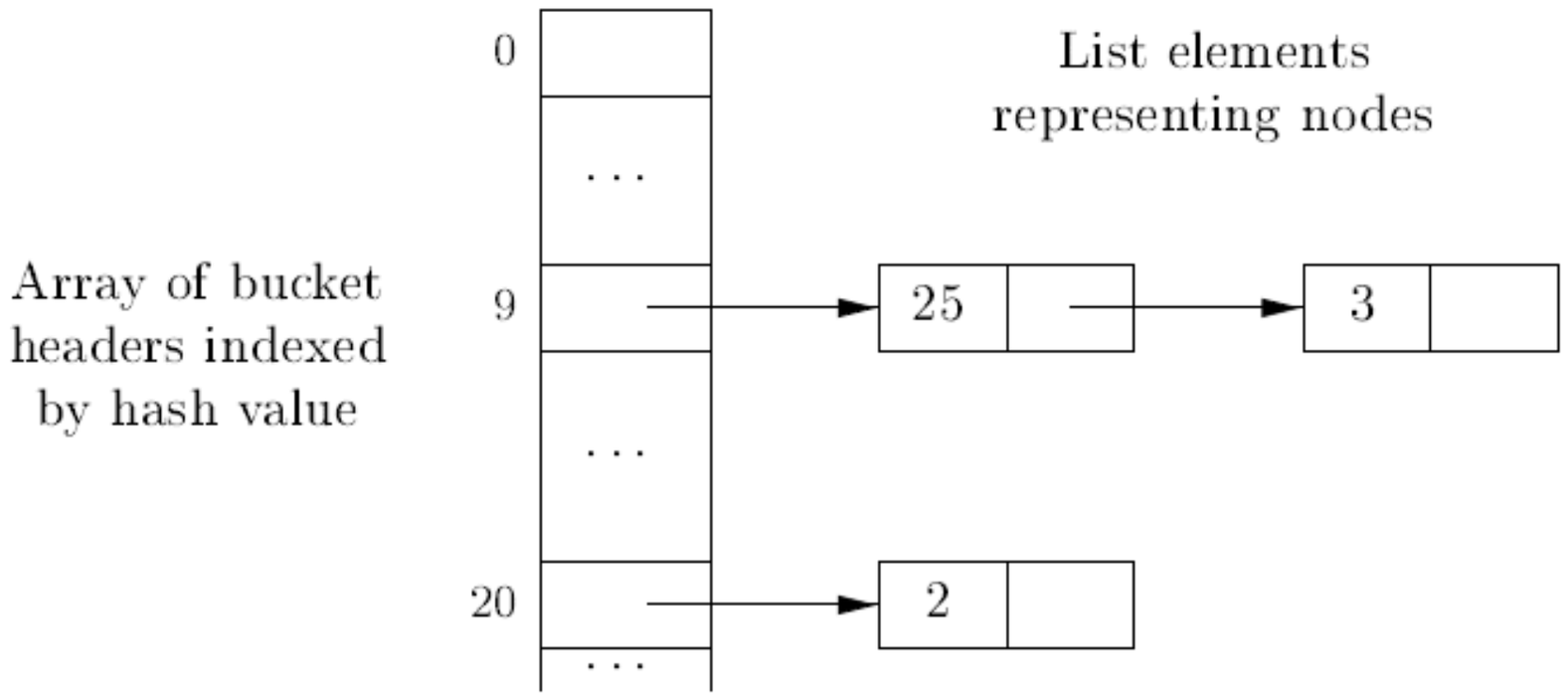
(a) DAG



(b) Array.

Nodes of a DAG for "**`i=i+10`**" allocated in an array

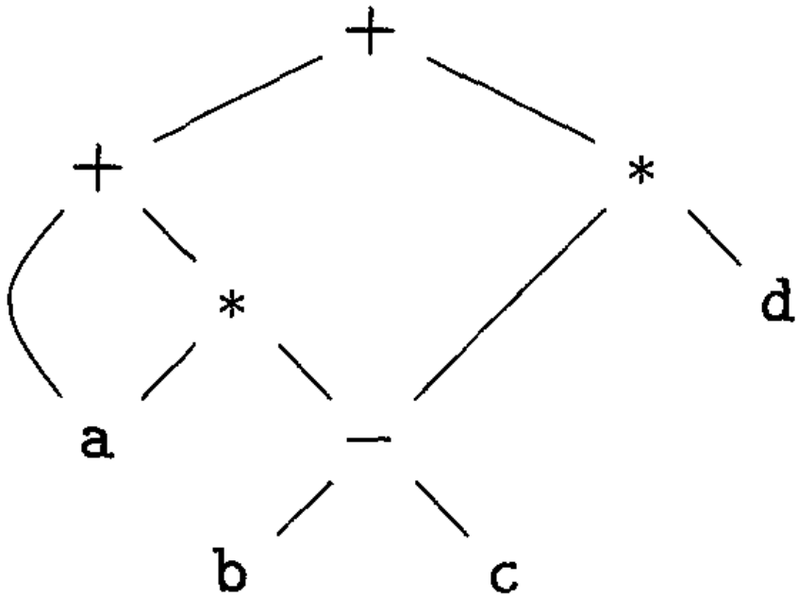
An efficient implementation is obtained by using *dictionaries* such as hash table



THREE-ADDRESS CODE

x+y*z

$$t_1 = y * z$$
$$t_2 = x + t_1$$



(a) DAG

$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

(b) Three-address code

a+a*(b-c)+(b-c)*d

Addresses

1. *A* ***name****.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

2. *A* ***constant****.* In practice, a compiler must deal with many different types of constants and variables.

3. *A* ***compiler-generated temporary****.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

Instructions

1)**Assignment:** **x = y op z**,

(*op* is a binary arithmetic operator)

2)**Assignment:** **x = op y**

(*op* is a unary operation: unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number).

3) **Copy:** **x = y**,

(*x* is assigned the value of *y)*

4) **Unconditional jump:** **`goto L`**.

5) **Conditional jump:** **`if x goto L`**, **`ifFalse x goto L`**.

6) **Conditional jump:** **`if x relop y goto L`**,

(this applies a relational operator (`<`, `==`, `>=`, etc.) to *x* and *y,* and execute the instruction with label *L* next if *x relop y* holds. Otherwise, the three-address instruction following `if *x relop y* goto *L'* is executed next, in sequence.)

7)**Procedure call**, **return:**

parameters: `param x` (for each parameter)

procedure call: `call p, n` (n is the number of parameters)

function call: `y = call p, n`

return: `return y`, (the returned-value *y* is optional).

example:

```
param x1
param x2
…
param xn
call   p, n
```

is* generated as part of a call of the procedure** ***p(x1, x2,..., xn).

8)**indexed copy instruction:** `x = y [i]` and `y [i] = x`.

(y[i] means i memory units beyond location *y)*

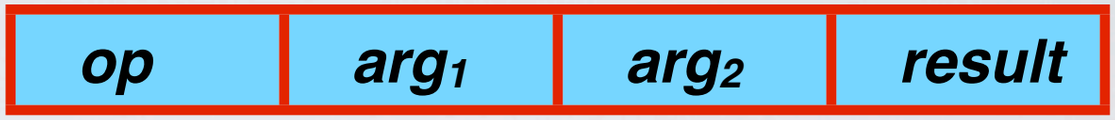
9)**Address and pointer assignment:** `x = & y`, `x = * y`, and `* x = y`.

```
do i=i+1; while (a[i]< v);
```

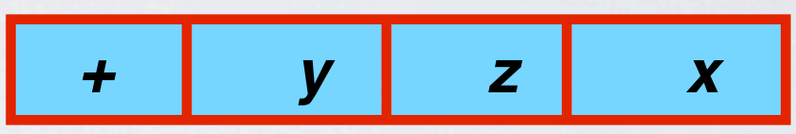
L: t_1=i+1	100: t_1=i+1
i=t_1	101: i=t_1
t_2=i*8	102: t_2=i*8
t_3=a[t_2]	103: t_3=a[t_2]
if t_3 < v goto L	104: if t_3 < v goto 100
(a) Symbolic labels.	(b) Position numbers.

Quadruples

quadruple (quad):



`x = y + z` →

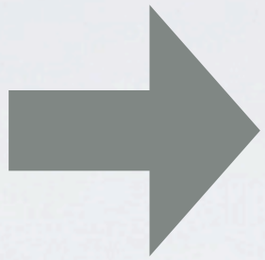


Some exceptions:

1. Instructions with unary operators like ***x* = minus *y*** or ***x* = *y*** do not use *arg2.*

   for a copy statement like ***x* = *y*,** *op* is =, while for most other operations, the assignment operator is implied.
2. Operators like **param** use neither *arg2* nor *result.*
3. Conditional and unconditional jumps put the target label in ***result.***

```
t1 =  minus c
t2 =  b*t1
t3 =  minus c
t4 =  b*t3
t5 =  t2+t4
a  =  t5
```



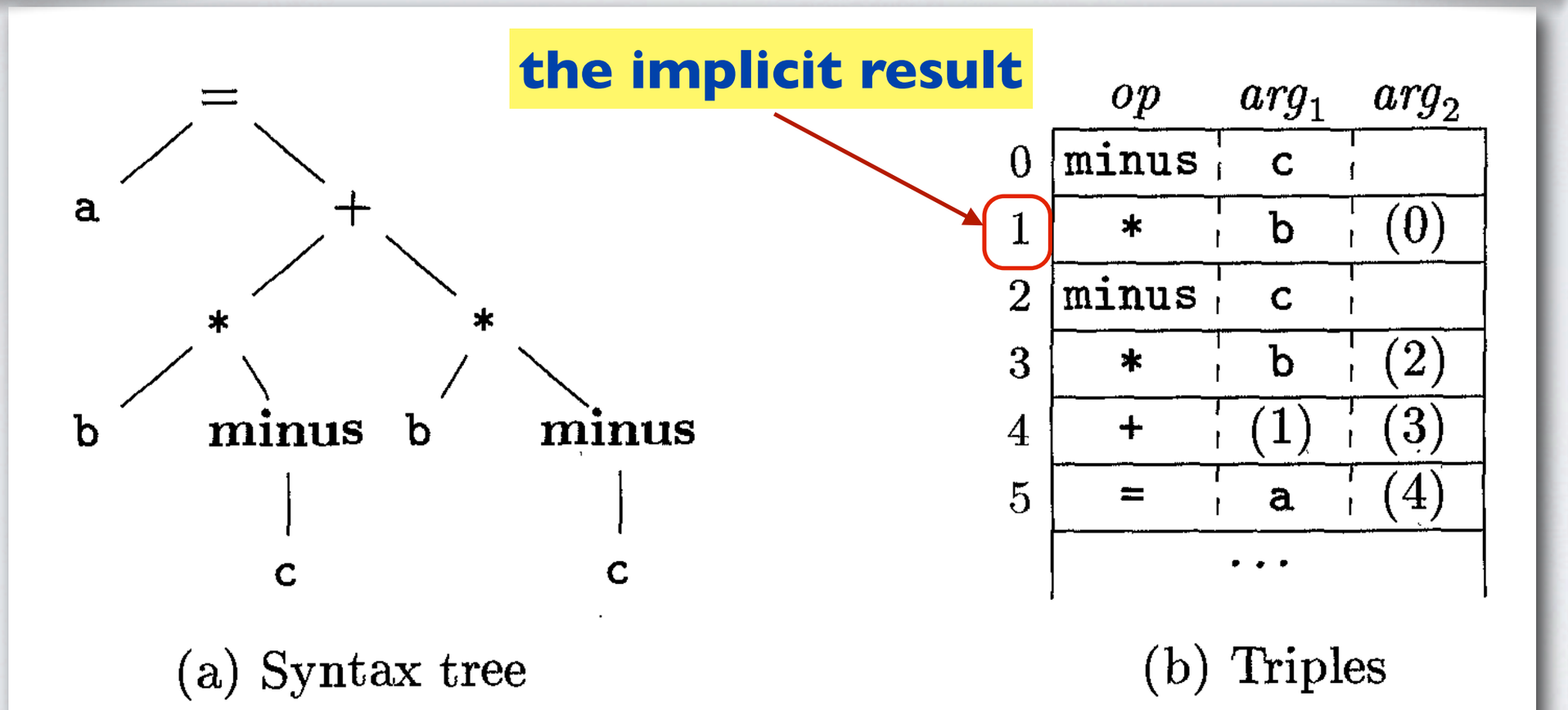
op	arg_1	arg_2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a
…	…	…	…

Triples

triple:

op	*arg₁*	*arg₂*

Using triples, we refer to the **result** of an operation x op y by its position, rather than by an explicit temporary name.



(a) Syntax tree (b) Triples

Representations of **`a + a * (b - c) + (b - c) * d`**

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around

With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result

SOLUTION: indirect triples

a listing of pointers to triples, rather than a listing of triples themselves

instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Indirect triples representation of three-address code

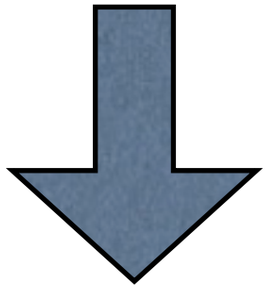
Static Single-Assignment Form

an intermediate representation that facilitates certain code optimizations

all assignments in SSA are to variables with distinct names

`p = a + b`	`p1 = a + b`
`q = p - c`	`q1 = p1 - c`
`p = q * d`	`p2 = q1 * d`
`p = e - p`	`p3 = e - p2`
`q = p + q`	`q2 = p3 + q1`
(a) Three-address code.	(b) Static single-assignment form.
intermediate program in three-address code and SSA	

```
if ( flag) x = -1; else x = 1;
y = x * a;
```



```
if (flag) x1 = -1; else x2 = 1;
x3 = ϕ(x1, x2) ;
y = x3 * a;
```

$$\phi(x_1, x_2) = \begin{cases} x_1 \text{ passes through the true part of the conditional} \\ x_2 \text{ othewise} \end{cases}$$

the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the ϕ-function.

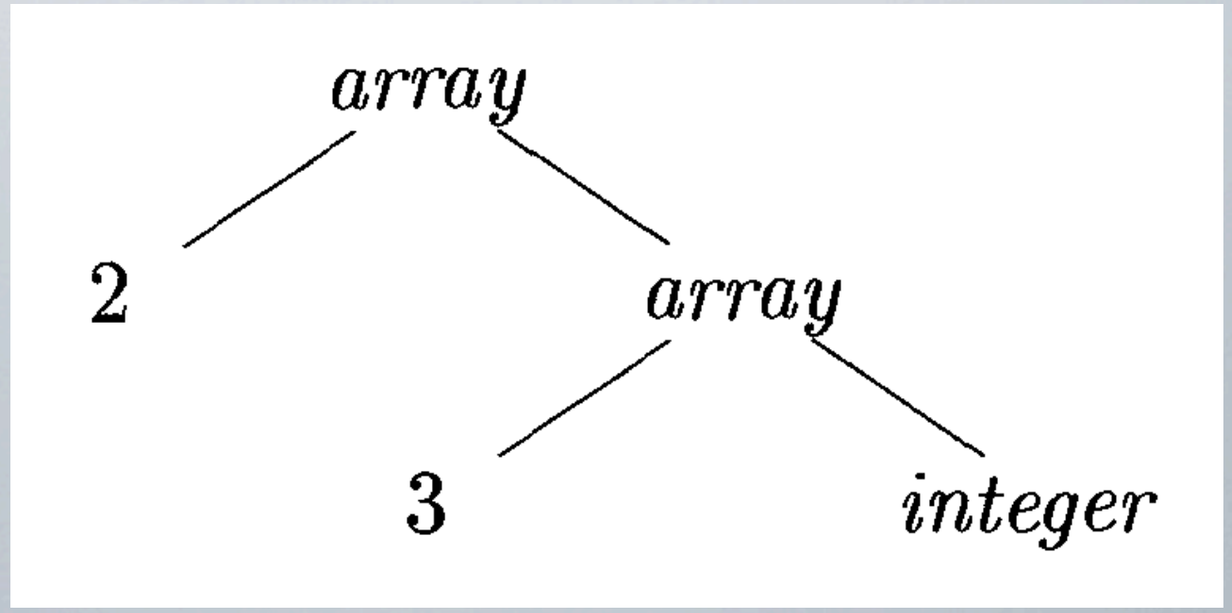
TYPES

***Type checking**:* For example, the && operator in Java expects its two operands to be booleans; the result is also of type boolean.

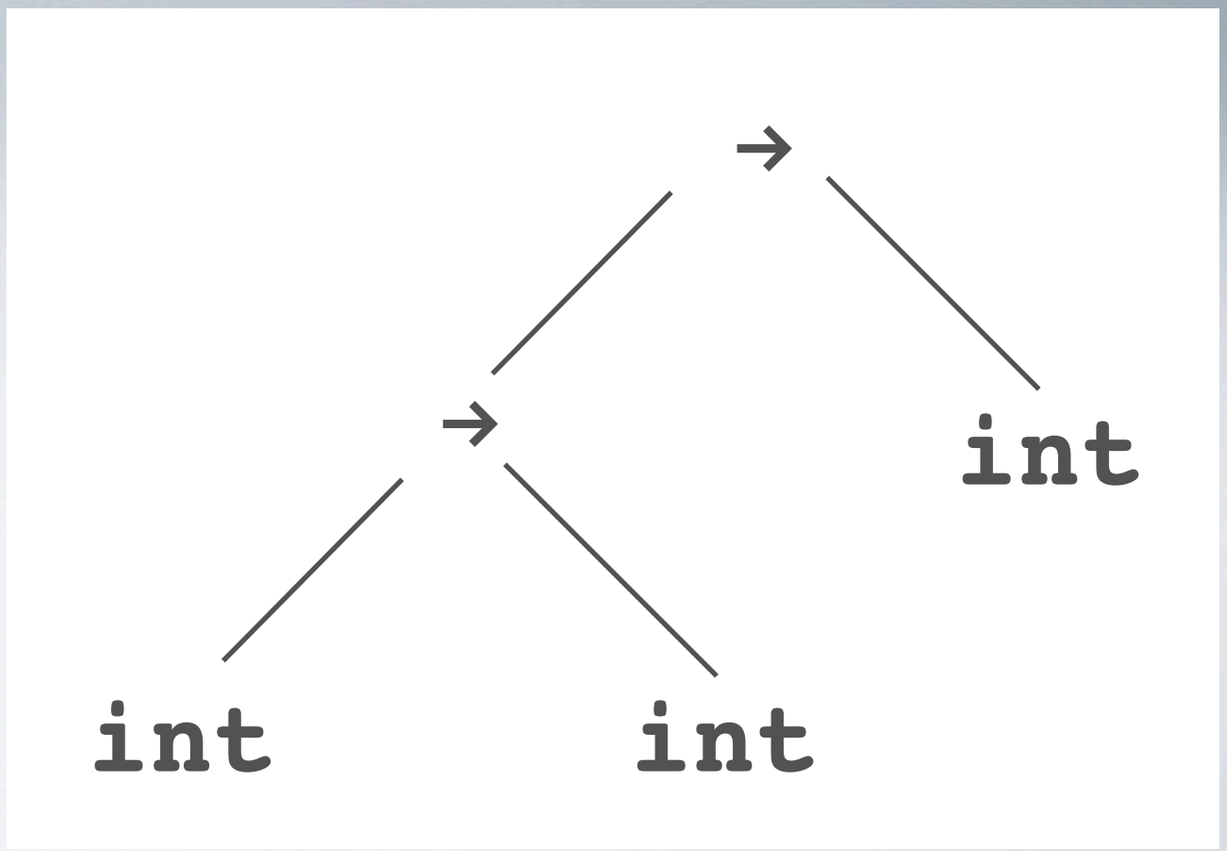
***Translation Applications**.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

Tree representation

```
int[2][3]
```




```
(int → int) → int
```



Type Expressions:

- **basic type:** *char, integer, float,* and *void;*
- **type name**;
- **array:** array(num,type_expr);
- **record:** record{field_names_with_types};
- **function/arrow types:** s→ t;
- **product type:** s×t
- We assume that × associates to the left and that it has higher precedence than→.
- Type expressions may contain variables whose values are type expressions.

type expression representation: graph

type equivalence

When type expressions are represented by graphs, two types are ***structurally equivalent*** if and only if one of the following conditions hold:

1. **They are the same basic type.**
2. **They are formed by applying the same constructor to structurally equivalent types.**
3. **One is a type name that denotes the other.**

name equivalence

type names are treated as standing for themselves + (1) + (2)

recursive types

```
public class Cell {
 int info;
 Cell next;
 ...}
```

Cell is a recursive type

Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

We used this grammar for types and declarations to illustrate inherited attributes. Here we consider the problem of determining the amount of *storage* that will be needed for a name at run time.

This can be determined at *compile time* by the type associated to the name by its declaration.

The information is then saved in the symbol table entry for the name.

Storage Layout

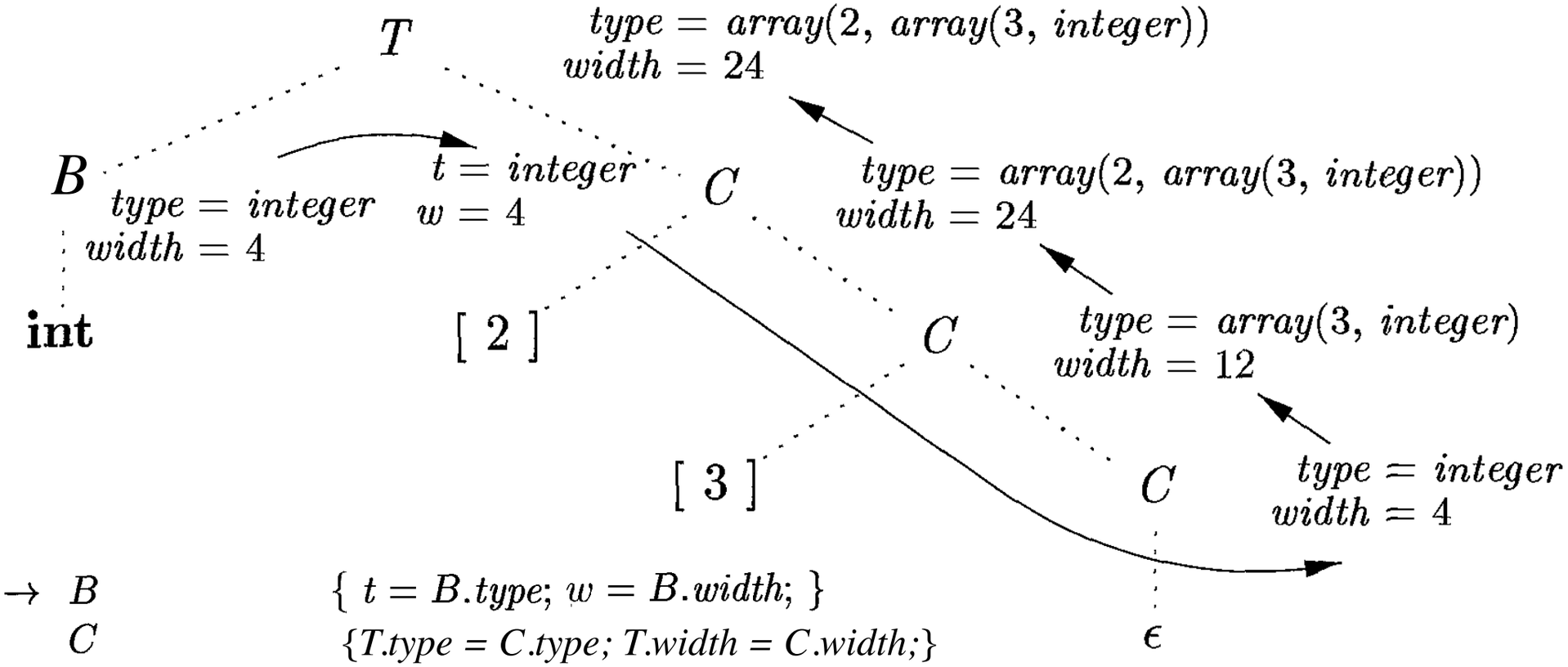
$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \{T.type = C.type;\ T.width = C.width;\} \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

In a SDD, t and w would be inherited attributes for C

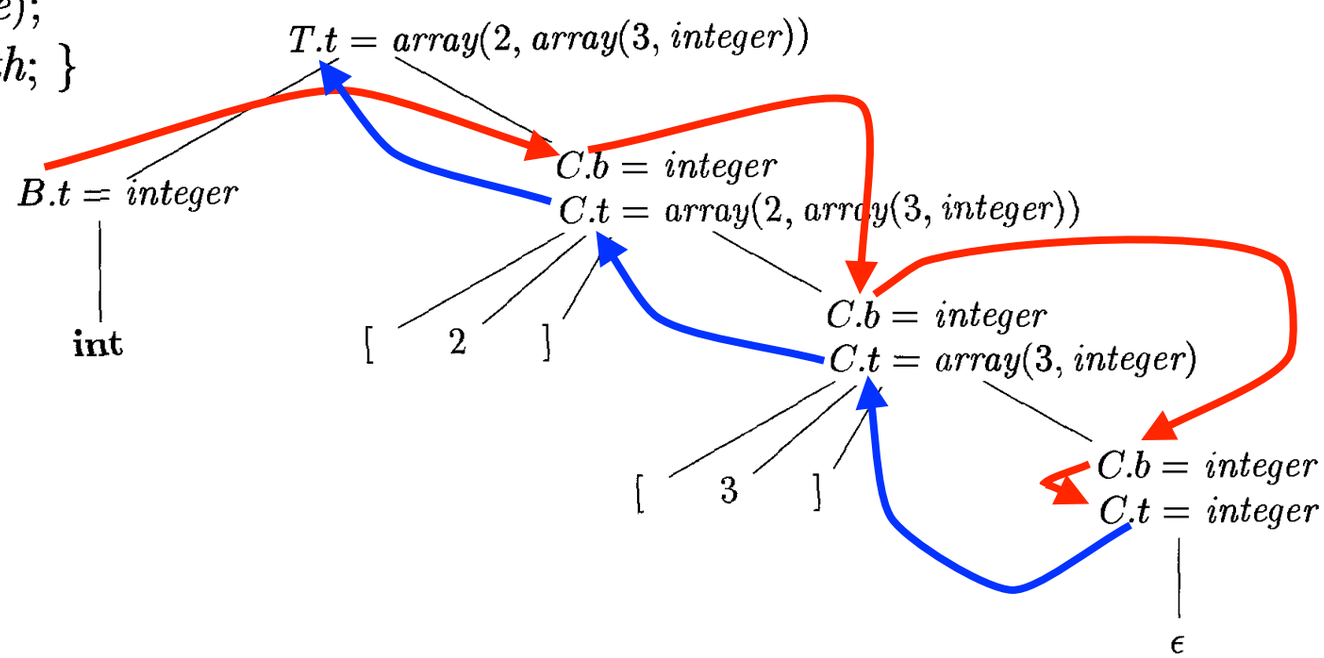
The width of an array is obtained by multiplying the width of an element by the number of elements in the array.

SDD:

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



$T \rightarrow B$		$\{\ t = B.type;\ w = B.width;\ \}$
	C	$\{T.type = C.type;\ T.width = C.width;\}$
$B \rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow$	[**num**] C_1	$\{C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$



Sequences of Declarations

$P \rightarrow \{\ offset = 0;\ \}\ D$
$D \rightarrow T\ \mathbf{id}\ ;\ \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);\ offset = offset + T.width;\ \}\ D_1$
$D \rightarrow \epsilon$

Computing the relative addresses of declared names

Nonterminals generating E, called marker nonterminals, can be used to rewrite productions so that all actions appear at the ends of right sides

$$P \rightarrow M\ D$$
$$M \rightarrow \epsilon \qquad \{\ offset = 0;\ \}$$

Fields in Records and Classes

$T \rightarrow$ **record** '{' D '}'

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D .
- The offset or relative address for a field name is relative to the data area for that record.

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env();\\
 & \quad Stack.push(offset);\ offset = 0;\ \} \\
D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

Translation of Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

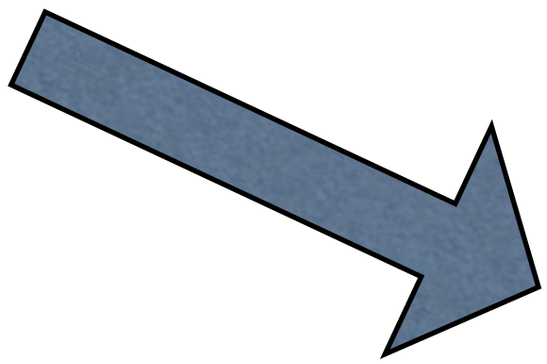
Three-address code for expressions

gen(x '=' y '+' z) ➠ three-address instruction **`x = y + z`**

top denote the current symbol table

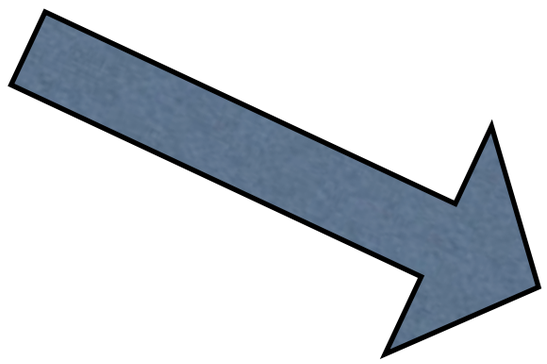
gen(top.get(**id**.lexeme)...) is < top.get(**id**.lexeme),step-of-access-link> ...

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



PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

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



```
t1 = minus c
t2 = b + t1
a = t2
```

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

```
t1 = minus c
t2 = b + t1
a = t2
```

Addressing Array Elements

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type = C.type;\ T.width = C.width;\}$
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**num**] C_1	$\{C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$

T [n] A
base= A[0]
w = width of elements
A[i] has address
base + **i** x **w**

T [n_1][n_2] A
base= A[0][0]
w = width of elements
A[i_1][i_2]
has address
base + (**i_1** x **w_1**) + (**i_2** x **w_2**)

T [n_1]...[n_k] A
base= A[0]...[0]
w = width of elements
A[i_1]...[i_k]
has address
base + (**i_1** x **w_1**) +...+ (**i_k** x **w_k**)

$$L \rightarrow L\ [\ E\ ] \mid \mathbf{id}\ [\ E\ ]$$

T $[n_1]...[n_k]$ A
base= A[0]...[0]
w = width of elements

$(...(...((A[i_1])...)[i_j])...)[i_k]$
has address
base + (i_1 x w_1) +...+ (i_k x w_k)

L.addr denotes a temporary that is used while computing the offset for the array reference

L.array is a pointer to the symbol-table entry for the array name.

L.type is the type of the subarray generated by L.

For any type T, we assume that its width is given by **T.width**.

For any array type T, suppose that **T.elem** gives the element type.

$$S \rightarrow \mathbf{id} = E\ ; \quad \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$$

$$\mid\ L = E\ ; \quad \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$$

$$E \rightarrow E_1 + E_2 \quad \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$$

$$\mid\ \mathbf{id} \quad \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}$$

$$\mid\ L \quad \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}$$

$$L \rightarrow \mathbf{id}\ [\ E\ ] \quad \{\ L.array = top.get(\mathbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \mathbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \}$$

$$\mid\ L_1\ [\ E\ ] \quad \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \mathbf{new}\ Temp\,();\ L.addr = \mathbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \}\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$$

Esercizio

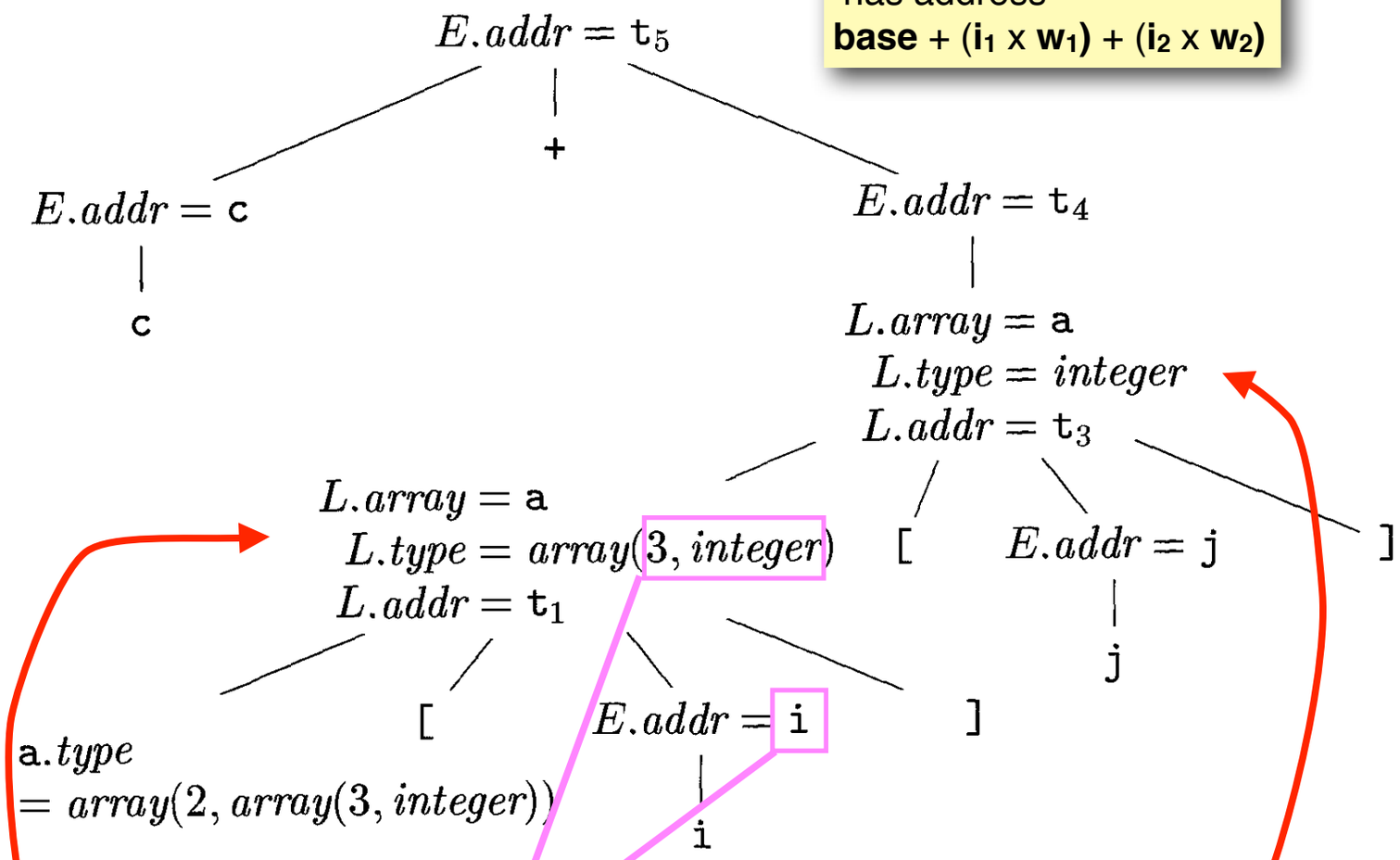
Construct the three-address code for expression

```
c + a[i][j]
```

Assume that we have the declaration

```
int[2][3]a;
```

T [n_1][n_2] A
base= A[0][0]
w = width of elements
A[i_1][i_2]
has address
base + (i_1 x w_1) + (i_2 x w_2)



Annotated parse tree for c + a [i] [j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Three-address code for expression c + a [i][j]

S	$\rightarrow$ **id** = E ;	{ $gen($ $top.get($**id**$.lexeme)$ $'='$ $E.addr)$; }
	| L = E ;	{ $gen(L.addr.base$ $'['$ $L.addr$ $']'$ $'='$ $E.addr)$; }
E	$\rightarrow$ E_1 + E_2	{ $E.addr =$ **new** $Temp\,()$; $gen(E.addr$ $'='$ $E_1.addr$ $'+'$ $E_2.addr)$; }
	| **id**	{ $E.addr = top.get($**id**$.lexeme)$; }
	| L	{ $E.addr =$ **new** $Temp\,()$; $gen(E.addr$ $'='$ $L.array.base$ $'['$ $L.addr$ $']')$; }
L	$\rightarrow$ **id** [E]	{ $L.array = top.get($**id**$.lexeme)$; $L.type = L.array.type.elem$; $L.addr =$ **new** $Temp\,()$; $gen(L.addr$ $'='$ $E.addr$ $'*'$ $L.type.width)$; }
	| L_1 [E]	{ $L.array = L_1.array$; $L.type = L_1.type.elem$; $t =$ **new** $Temp\,()$; $L.addr =$ **new** $Temp\,()$; $gen(t$ $'='$ $E.addr$ $'*'$ $L.type.width)$; } $gen(L.addr$ $'='$ $L_1.addr$ $'+'$ $t)$; }

L.addr **denotes a temporary that is used while computing the offset for the array reference**

L.array **is a pointer to the symbol-table entry for the array name.**

L.type **is the type of the subarray generated by *L*.**

For any type *T*, we assume that its width is given by *T.width*.

For any array type *T*, suppose that *T.elem* gives the element type.

TYPE CHECKING

Type synthesis

if *f* has type s → *t* **and** *x* has type s
then expression *f (x)* has type *t*

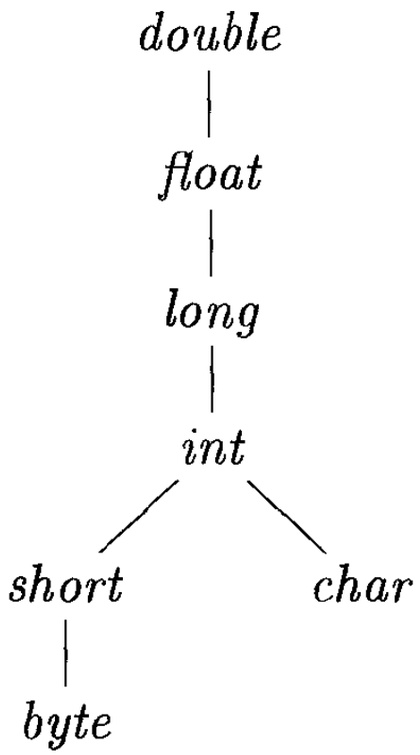
Type inference

if *f (x)* is an expression,
then for some α and β, *f* has type α → β
and *x* has type α

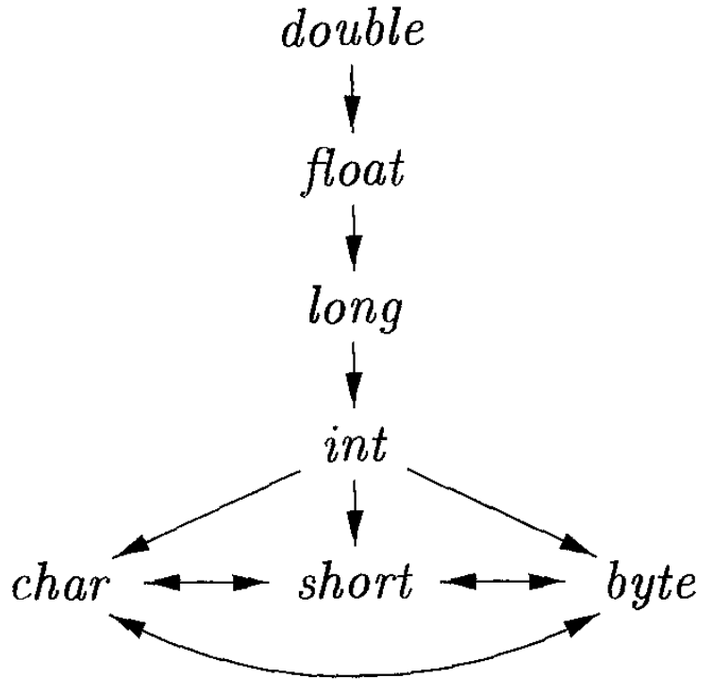
Type Conversions

widening conversions (**Promotions**), which are intended to preserve information,

narrowing conversions, which can lose information



(a) Widening conversions



(b) Narrowing conversions

max(t_1, t_2) takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy

widen(a,t,w) generates type conversions if needed to widen an address *a* of type *t* into a value of type *w.*

```
Addr widen(Addr a, Type t, Type w){
    if (t == w) return a;
    else if (t == integer && w == float)
           {
    temp = new Temp();
    gen(temp '=' '(float)' a);
    return temp;
    }
    else error;
}
```

$$E \rightarrow E_1 + E_2 \ \{E.type = max(E_1.type, E_2.type);$$
$$a_1 = widen(E_1.addr, E_1.type, E.type);$$
$$a_2 = widen(E_2.addr, E_2.type, E.type);$$
$$E.addr = new\ Temp();$$
$$gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}$$

Overloading of Functions and Operators

if f can have type $s_i \to t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
and x has type s_k, for some $1 \leq k \leq n$
then expression $f(x)$ has type t_k

Type inference and Polymorphic Functions

Control Flow

Boolean Expressions

B → B || B | B && B | !B | (B) | E rel E | true | false

Short-Circuit Code

in *short-circuit* (or *jumping)* code, the boolean operators **&&**, **||**, and **!** translate into jumps.

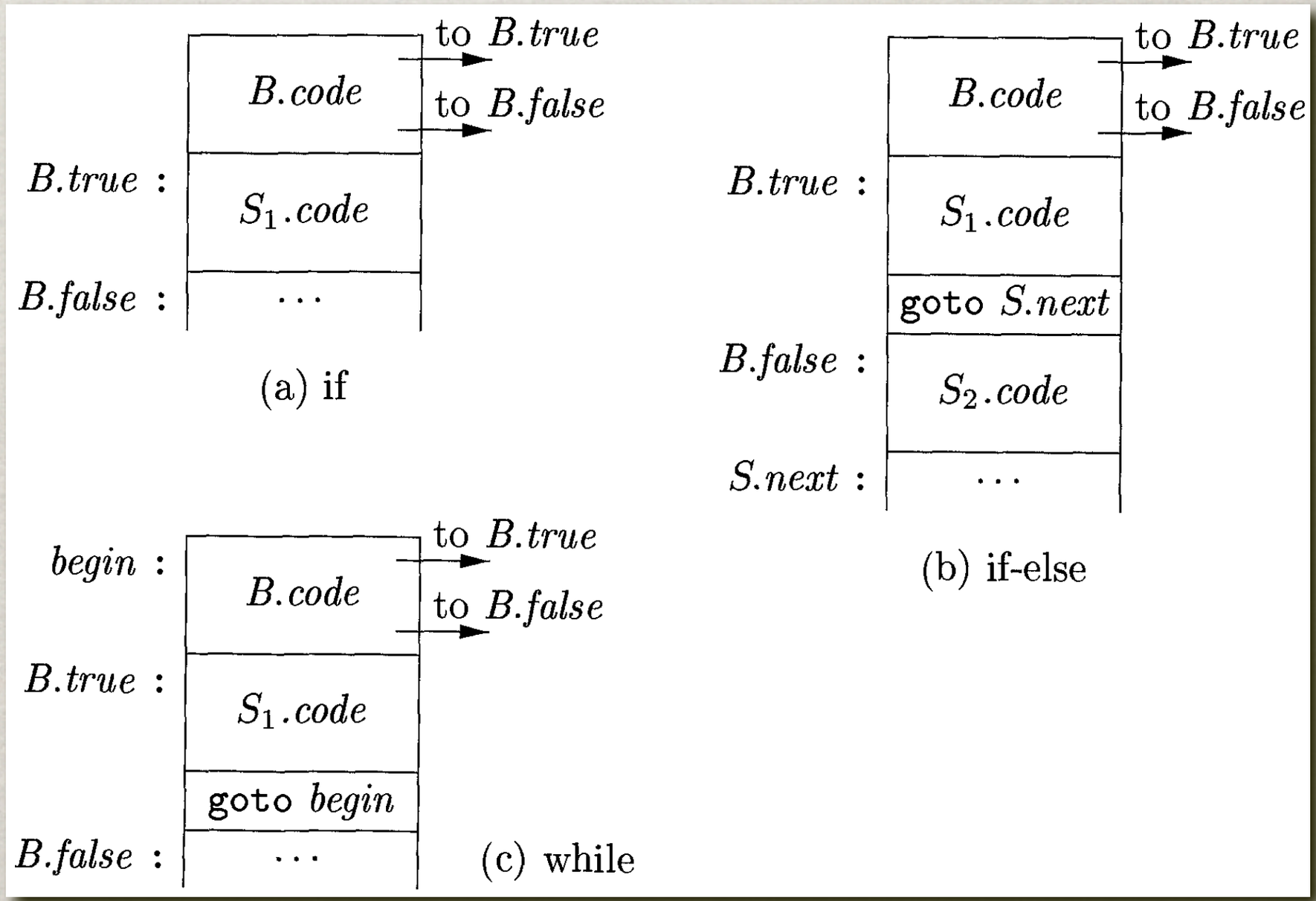
- E1 || E2 need not evaluate E2 if E1 is known to be true.
- E1 && E2 need not evaluate E2 if E1 is known to be false.

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

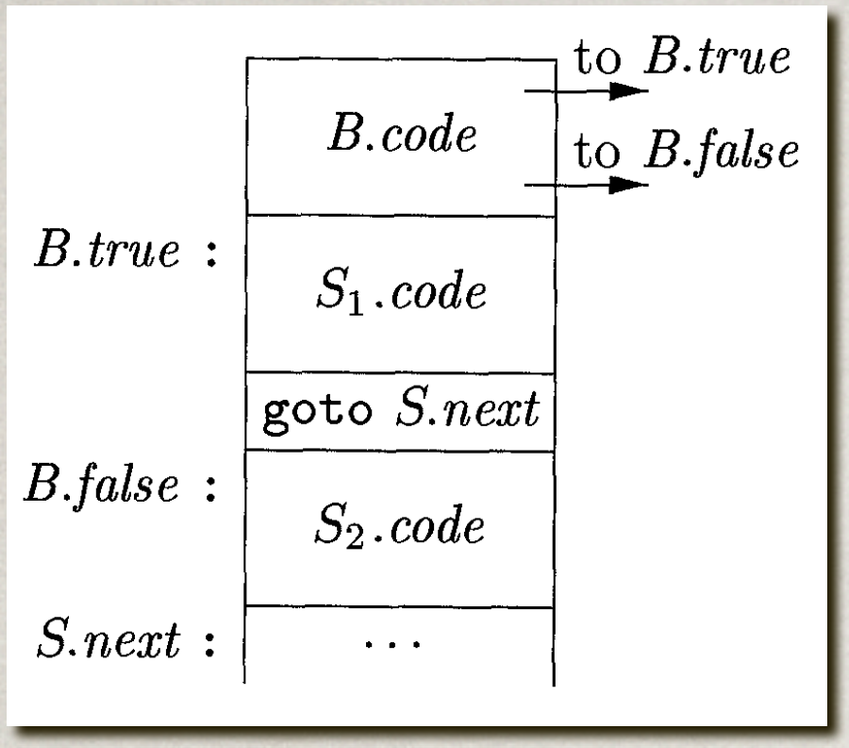
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0;
L1:     ...
```

Flow-of-Control Statements

S → **if** (*B*) S_1
S → **if** (*B*) S_1 **else** S_2
S → **while** (*B*) S_1



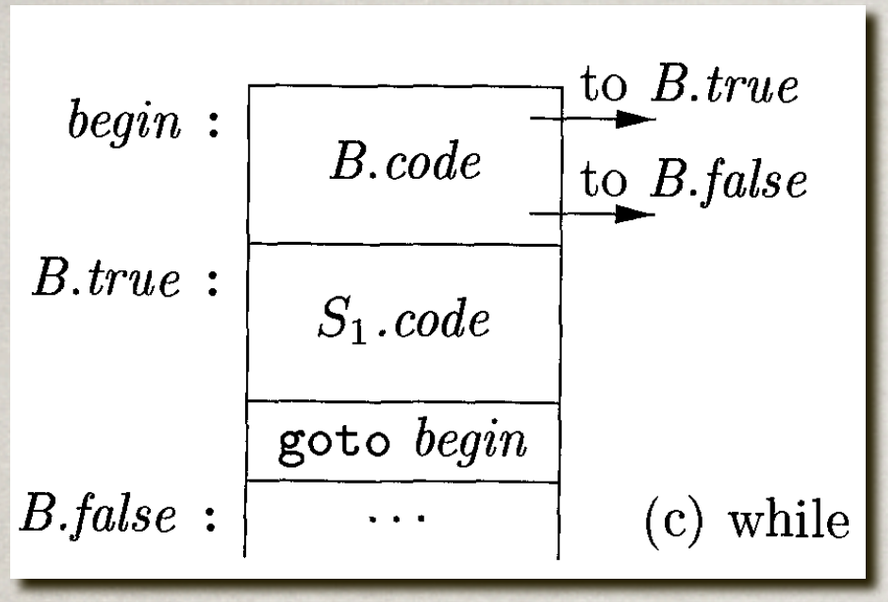
PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$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



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

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

We assume that *newlabel()* creates a new label each time it is called, and that *label(L)* attaches label L to the next three-address instruction to be generated



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

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

$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Concrete Syntax	Abstract Syntax
=	**assign**
||	**cond**
&&	**cond**
== !=	**rel**
< <= >= >	**rel**
+ -	**op**
* / %	**op**
!	**not**
$^{-}_{unary}$	**minus**
[]	**access**

: Concrete and abstract syntax for several Jav

```
Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op(op, y, z) or a Rel(op, y, z) node ) {
        t = new temporary;
        emit string for t = rvalue(y) op rvalue(z);
        return a new node for t;
    }
    else if ( x is an Access(y, z) node ) {
        t = new temporary;
        call lvalue(x), which returns Access(y, z');
        emit string for t = Access(y, z');
        return a new node for t;
    }
    else if ( x is an Assign(y, z) node ) {
        z' = rvalue(z);
        emit string for lvalue(y) = z';
        return z';
    }
}
```

Figure 2.45: Pseudocode for function *rvalue*

Control-Flow Translation of Boolean Expressions

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

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

B of the form $a < b$ translates into:

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

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

if B_1 is true, **then** ***B* itself is true** ➽ $B_1.true = B.true.$

if B_1 is false, **then** B_2 must be evaluated, so we make *B1.false* be the label of the first instruction in the code for B_2.

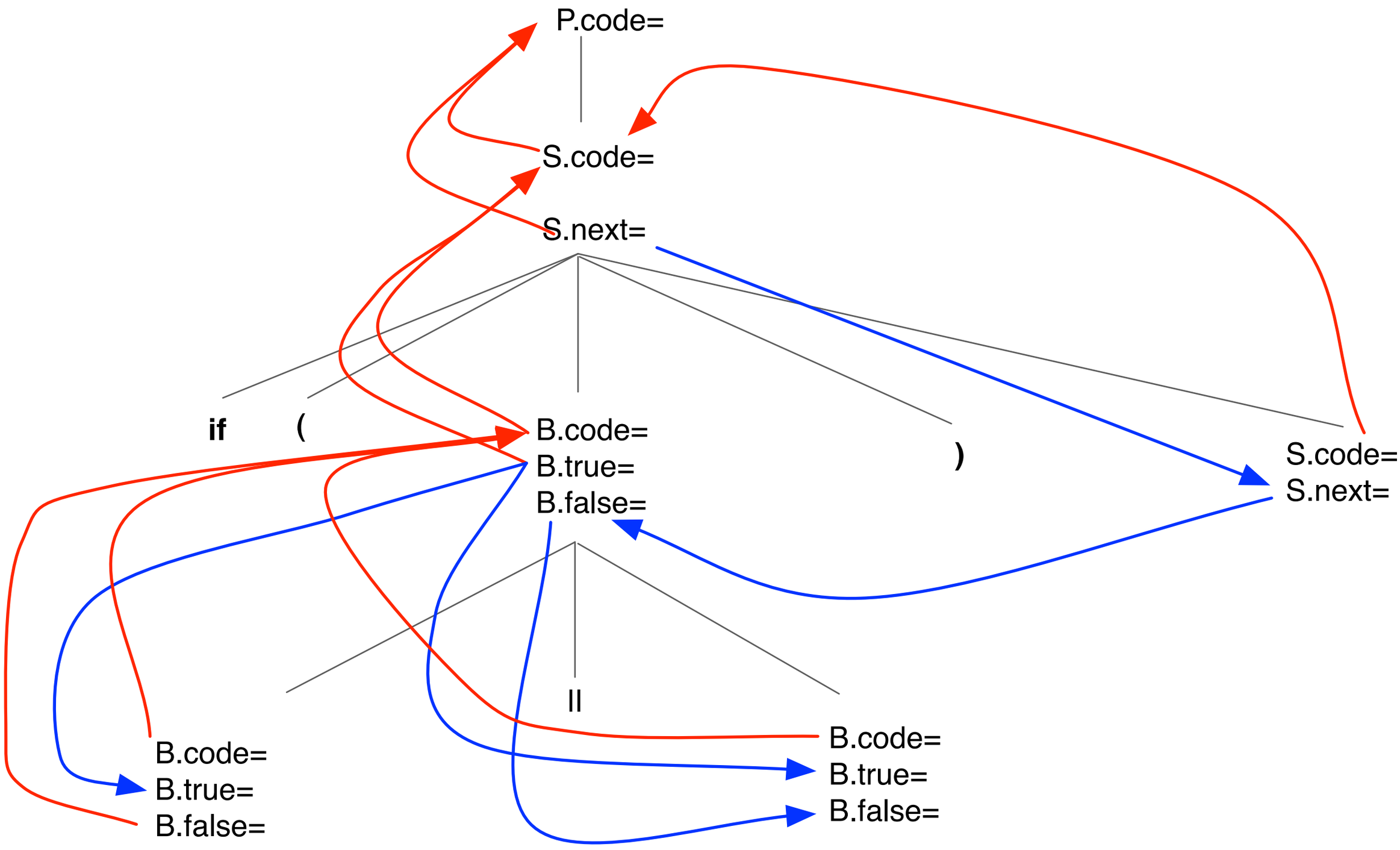
The true and false exits of B_2 are the same as the true and false exits of B_1 respectively.

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

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

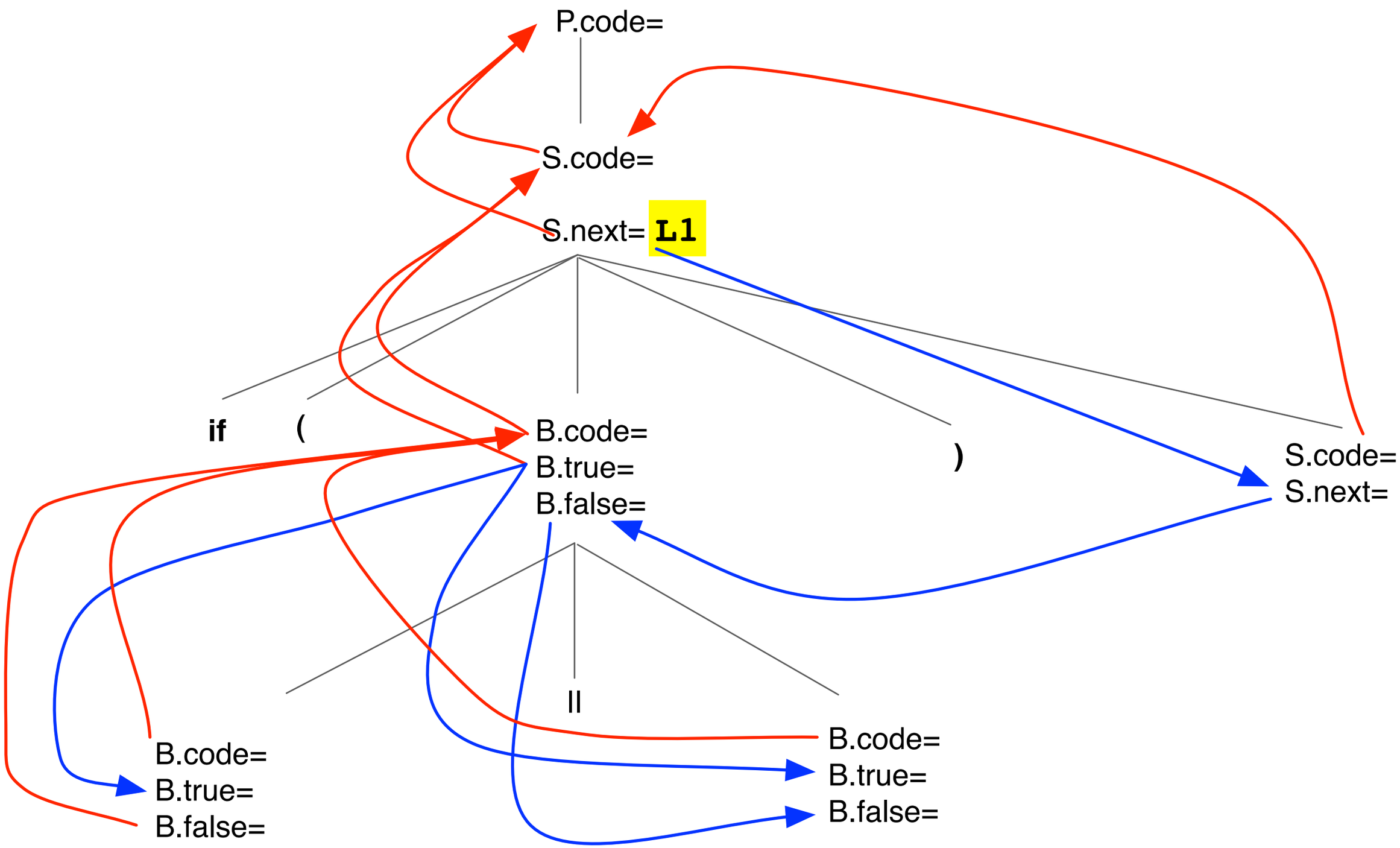
P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code

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



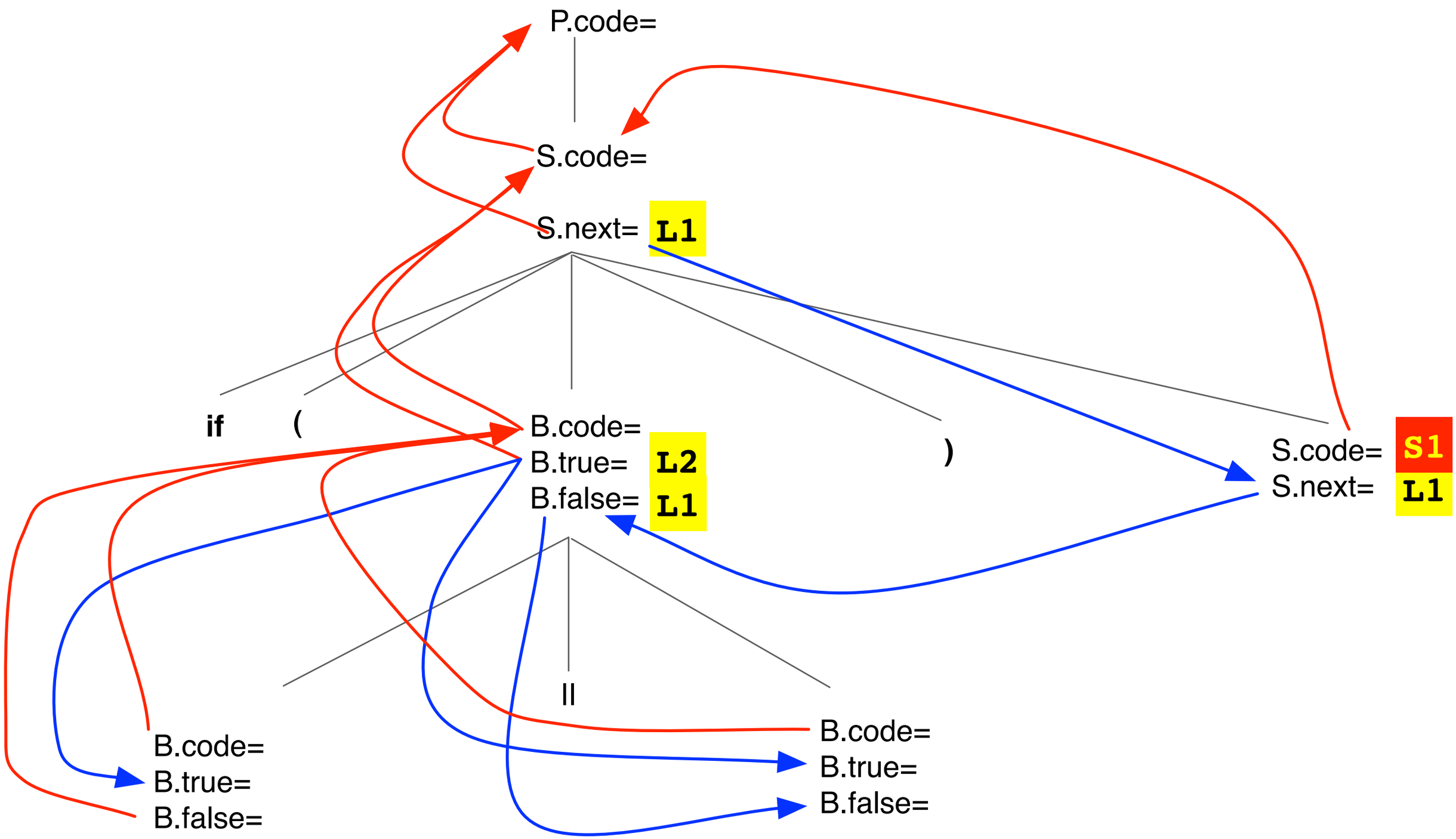
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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



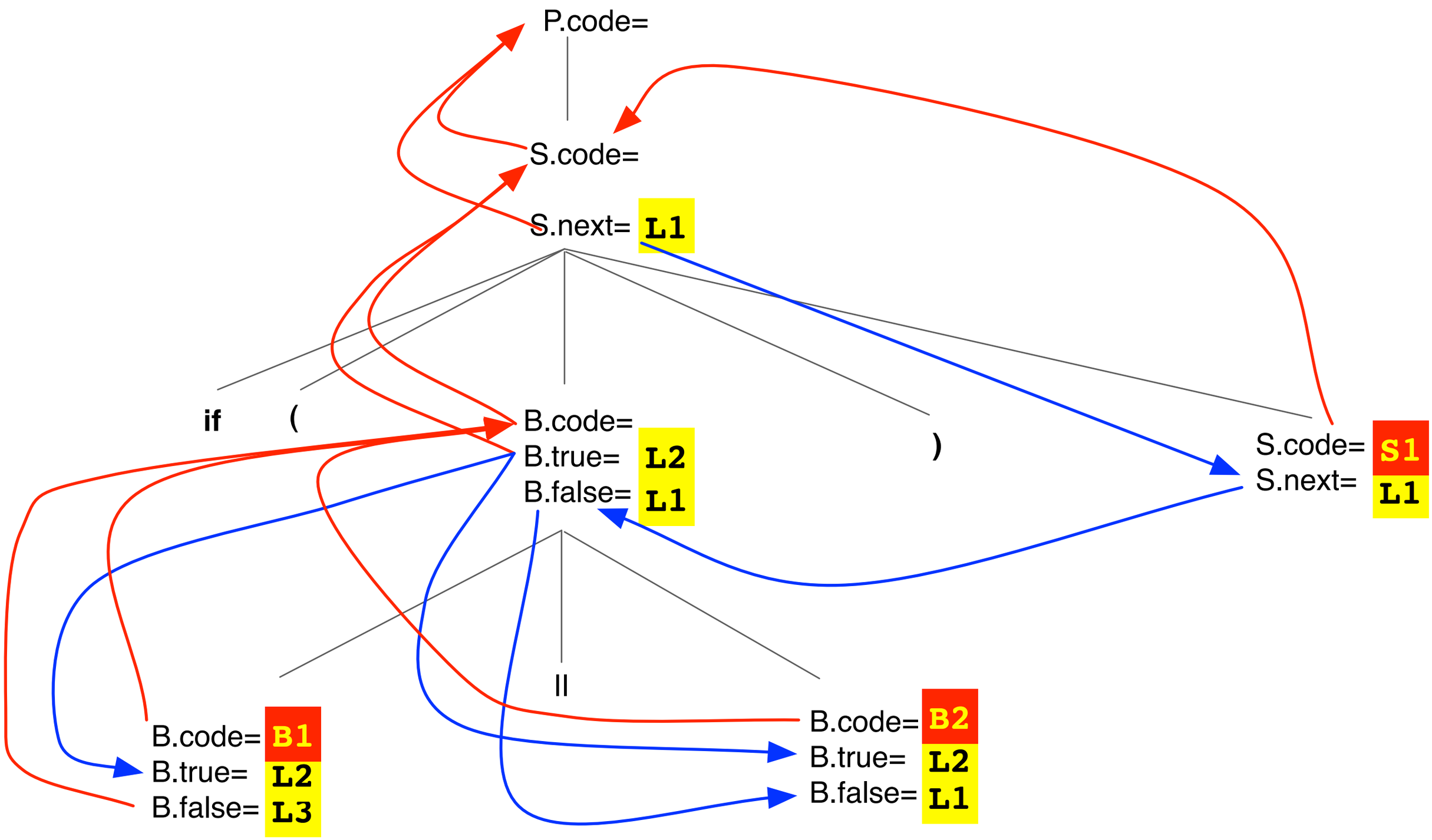
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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



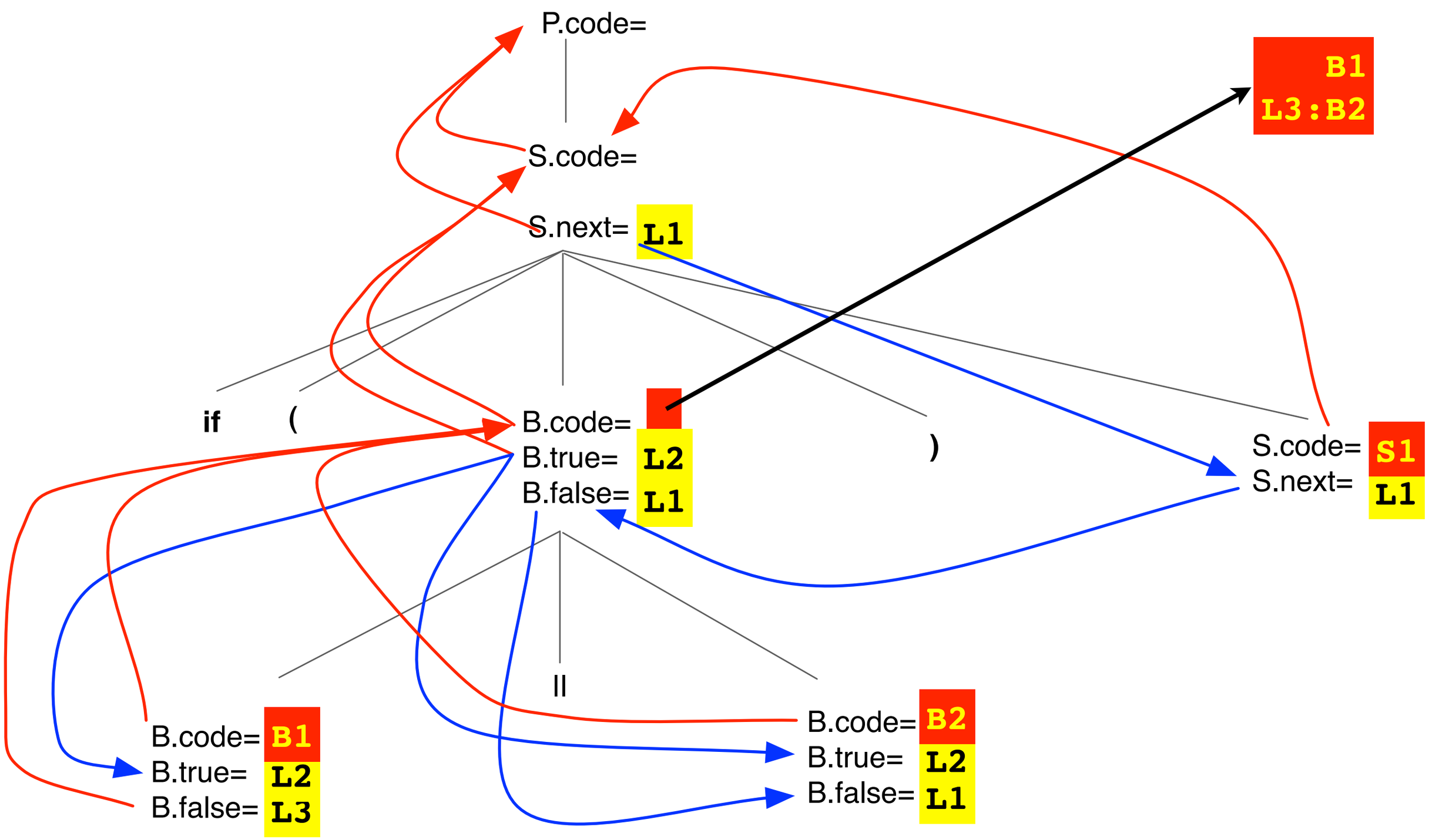
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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



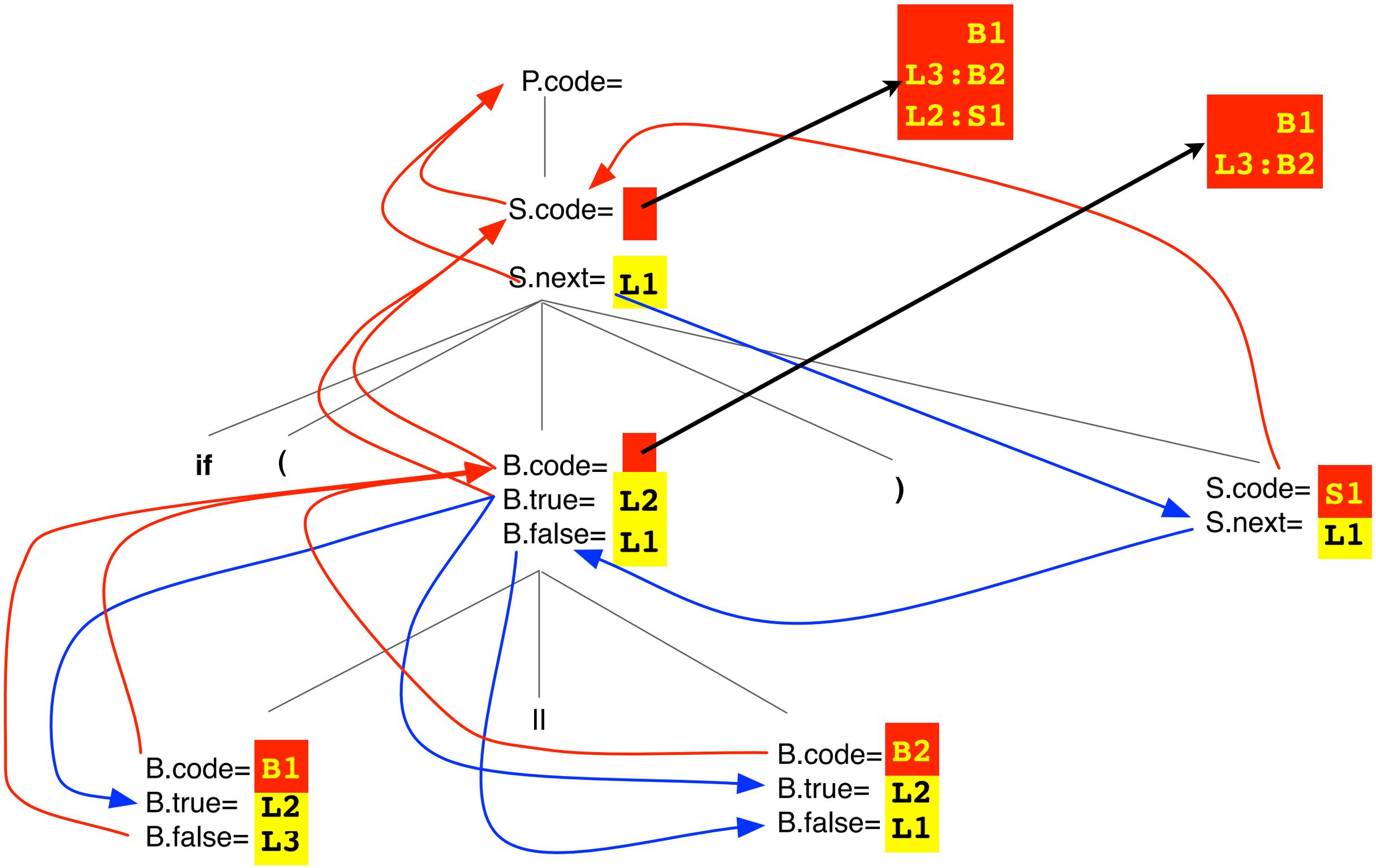
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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

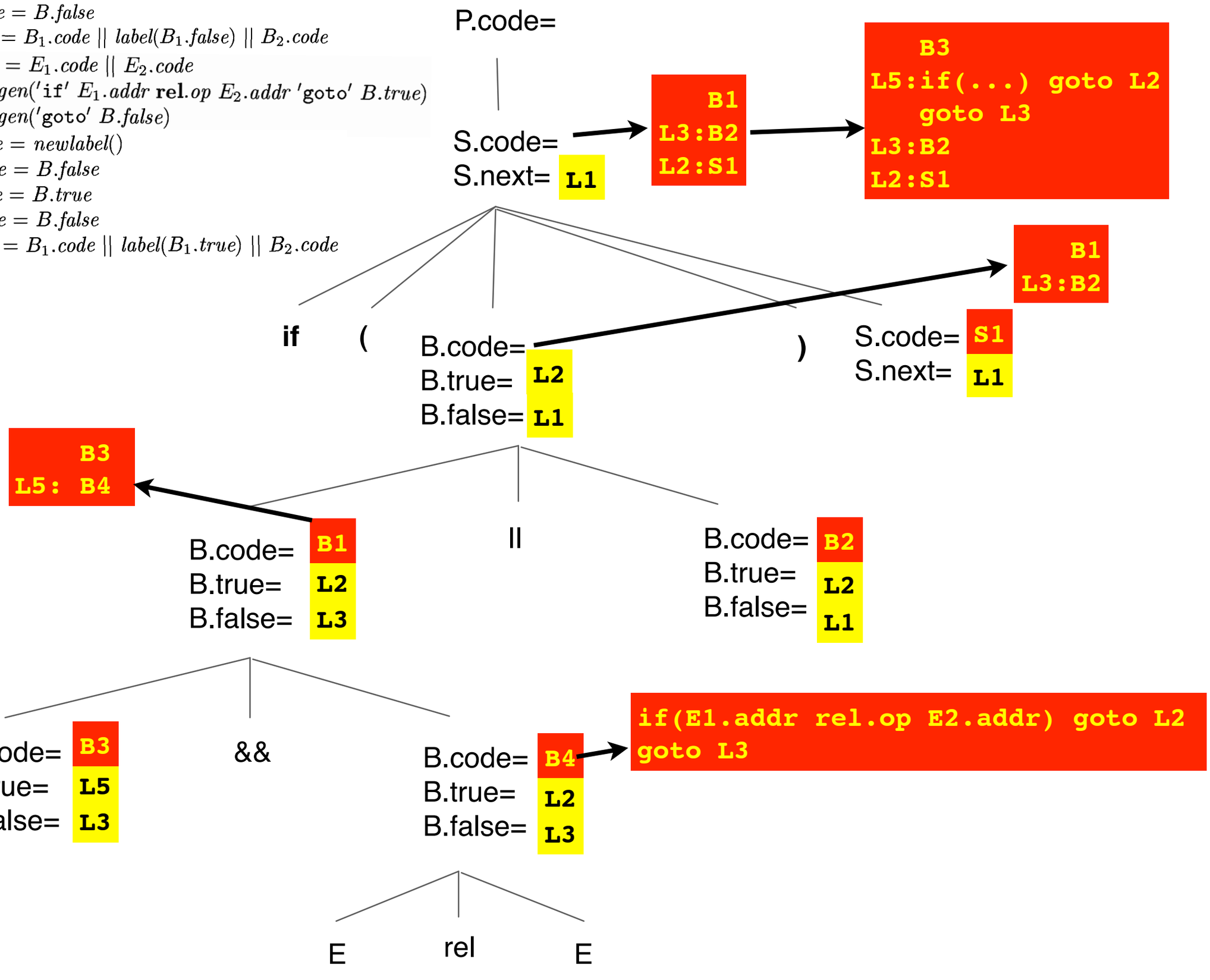


P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

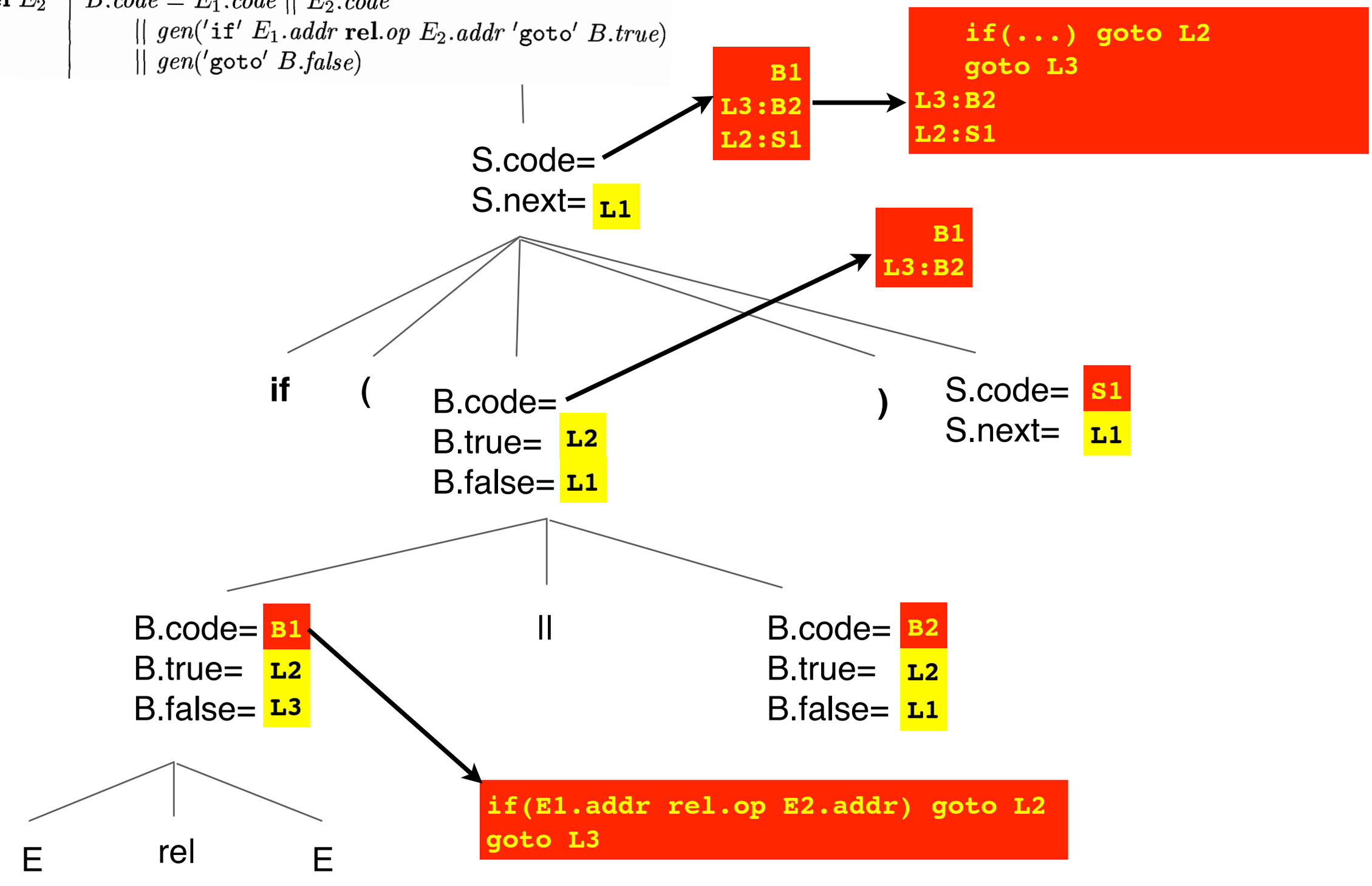
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$



$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$



$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$



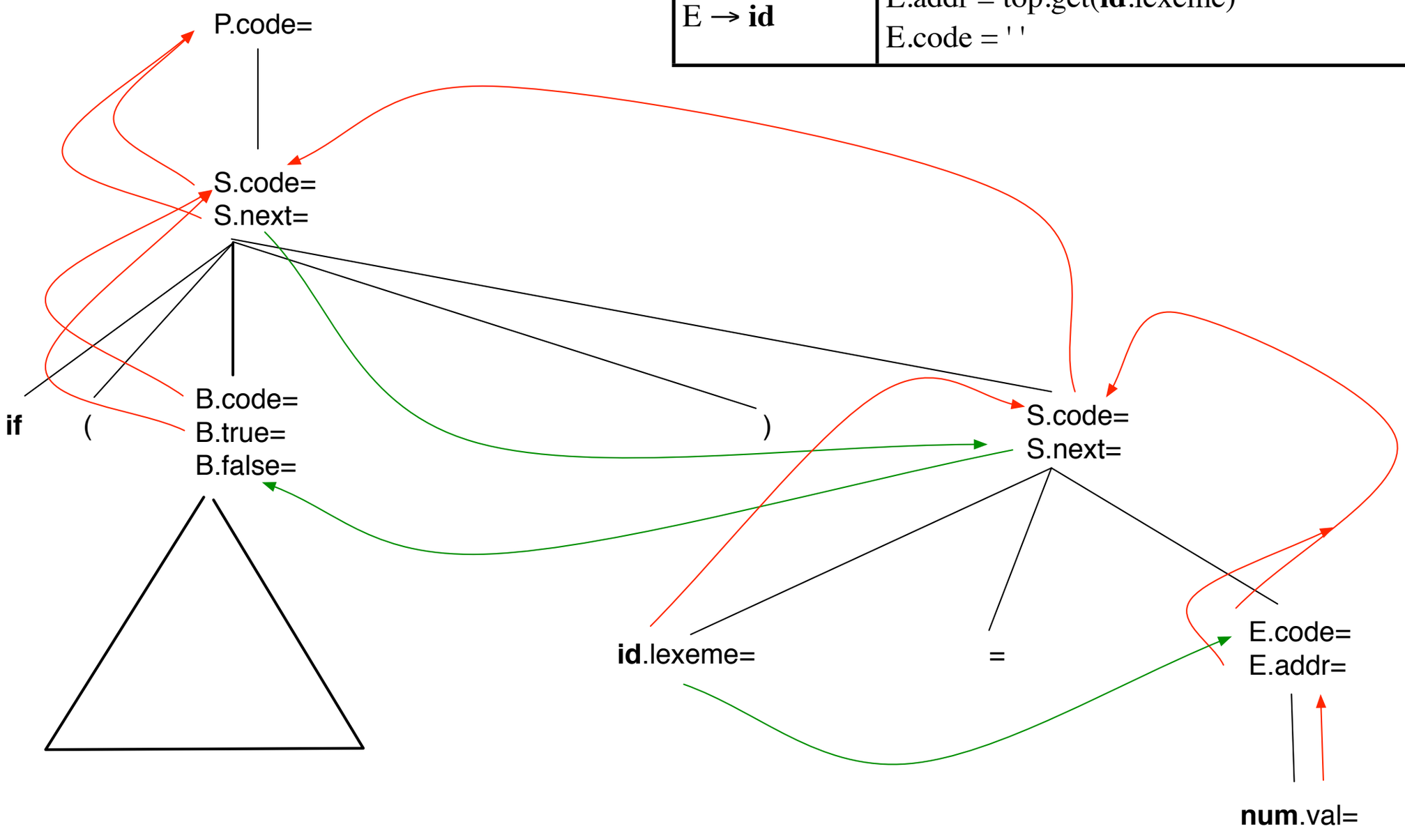
Esercizio

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

P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **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 → **id** = E;	S.code = E.code || gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

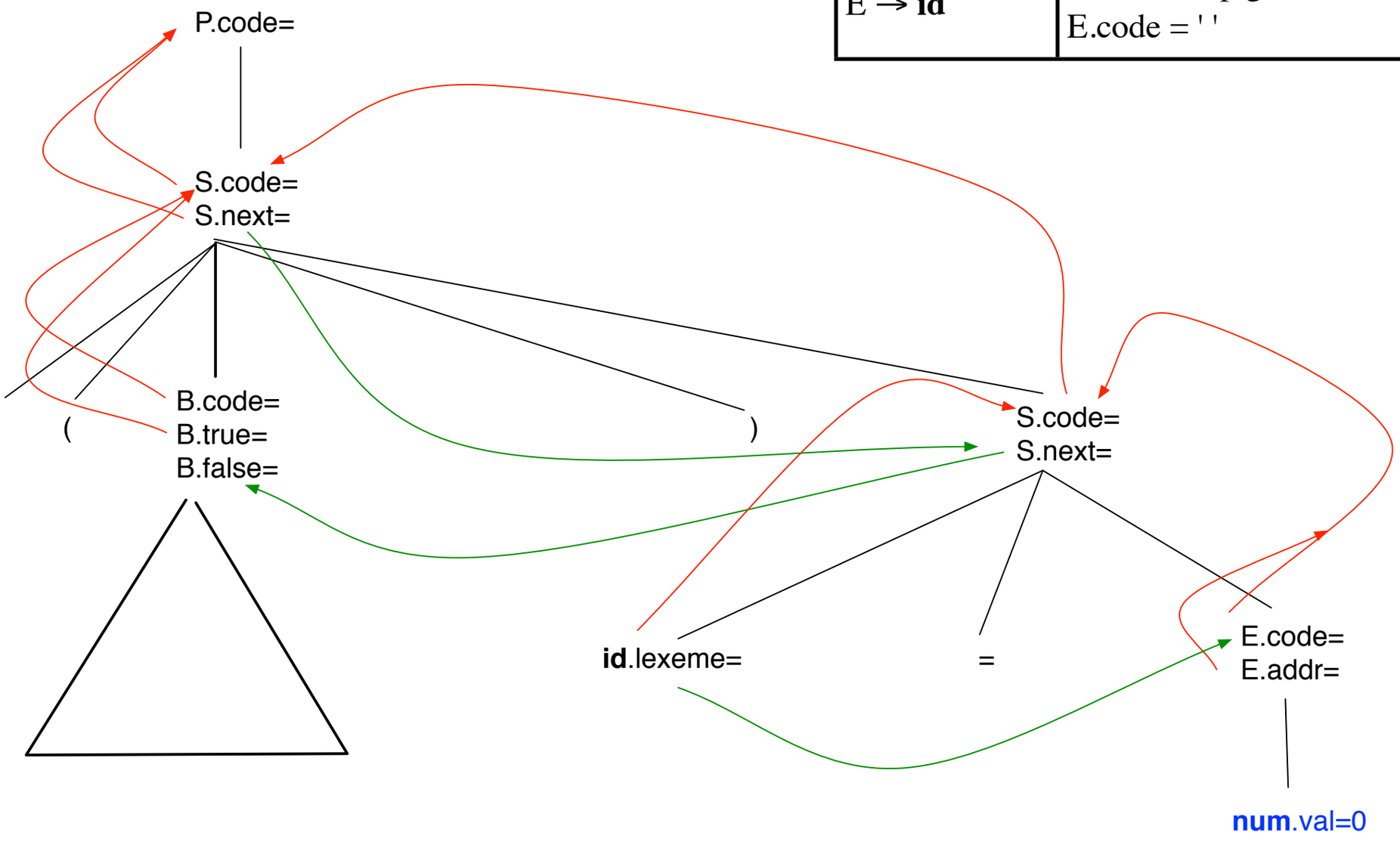
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **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 → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



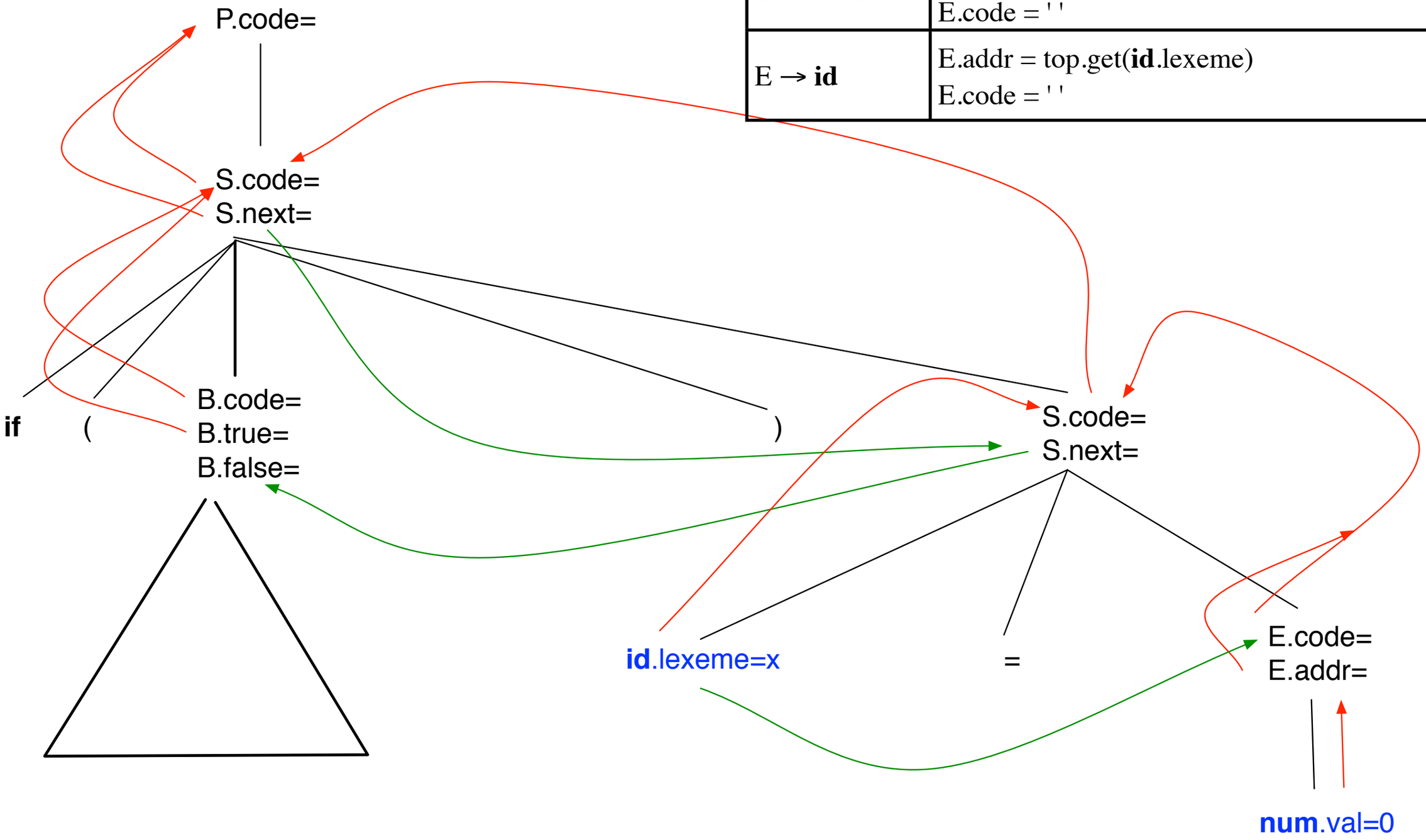
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **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 → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



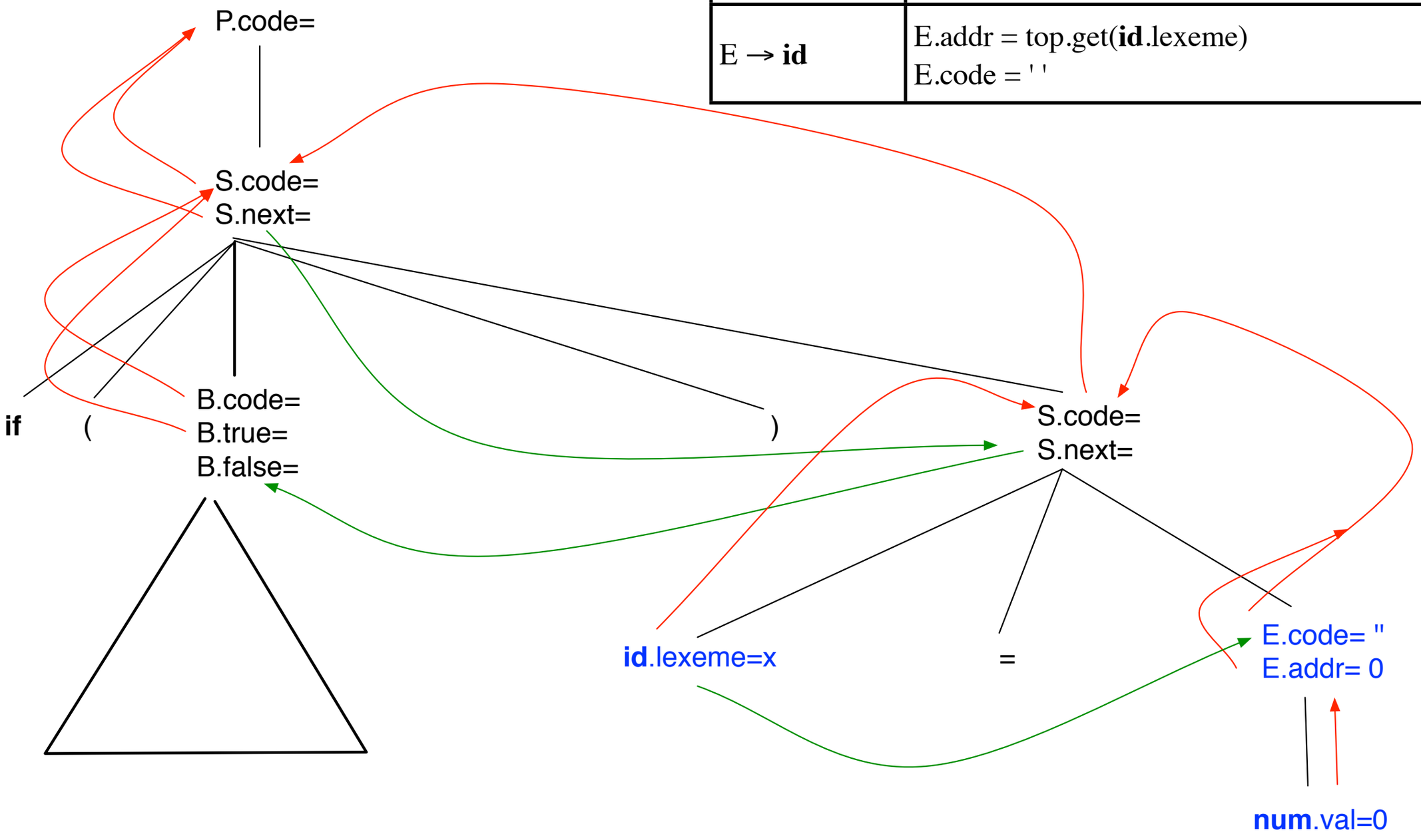
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **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 → **id** = E;	S.code = E.code || gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



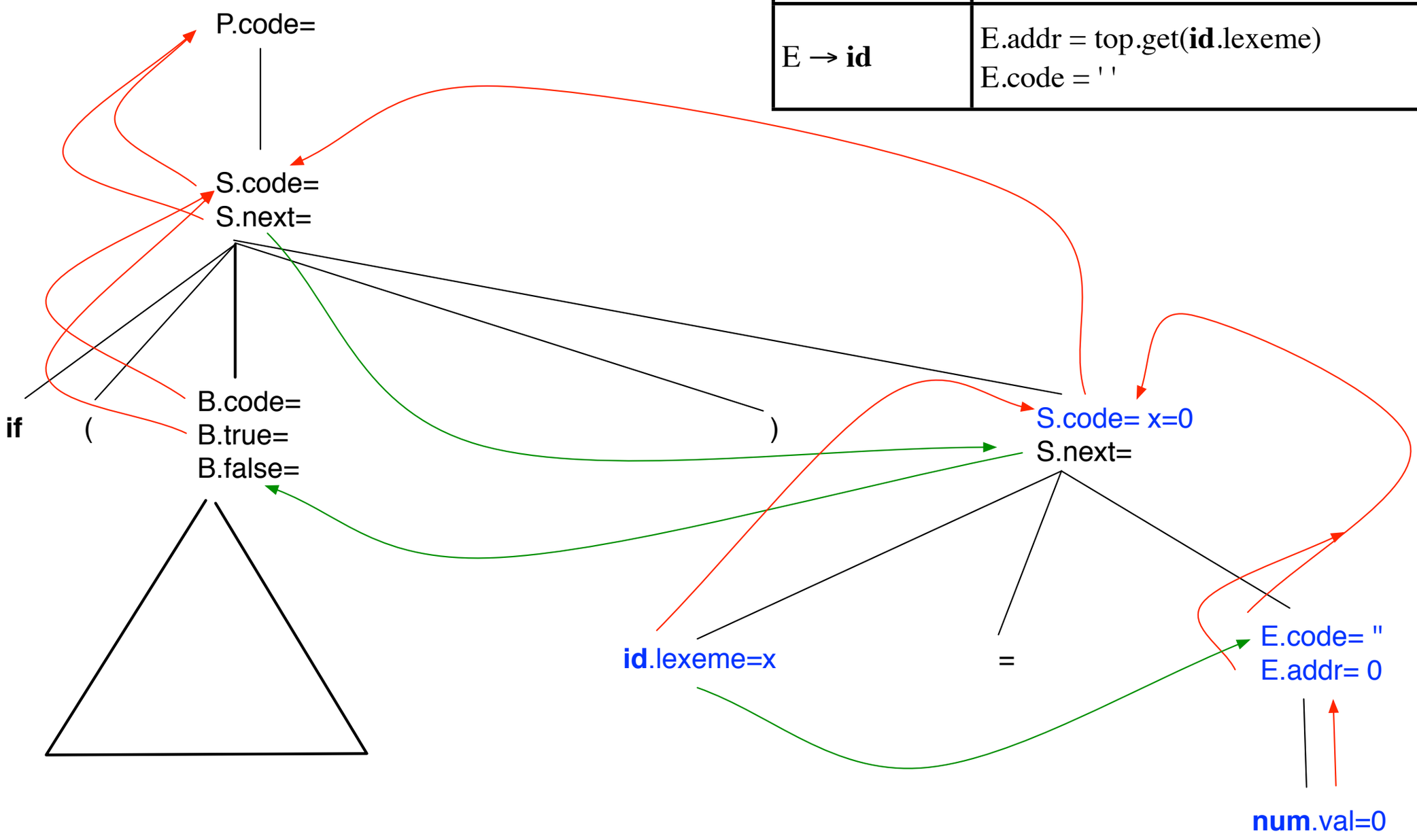
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **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 → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



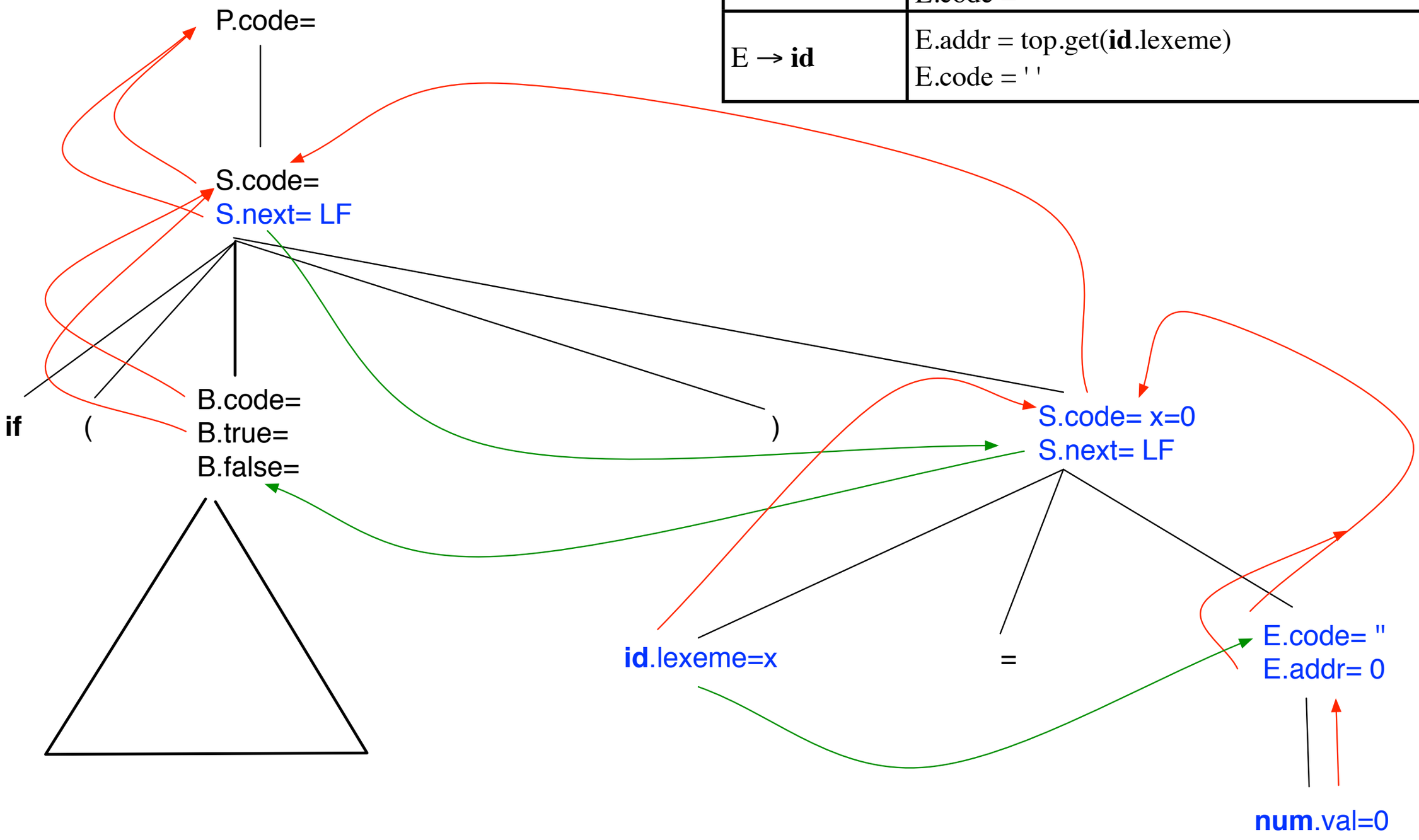
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **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 → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



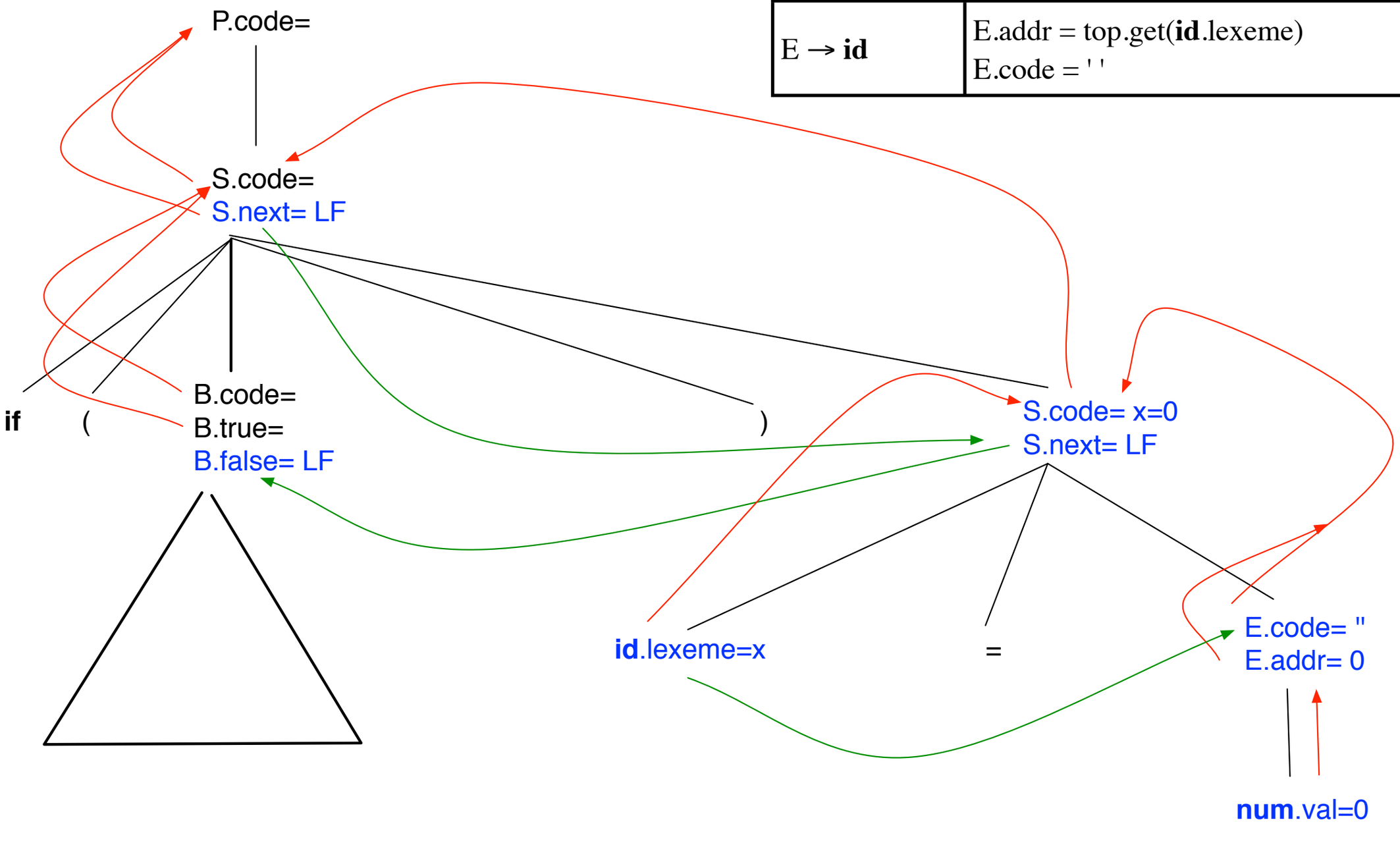
```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **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 → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

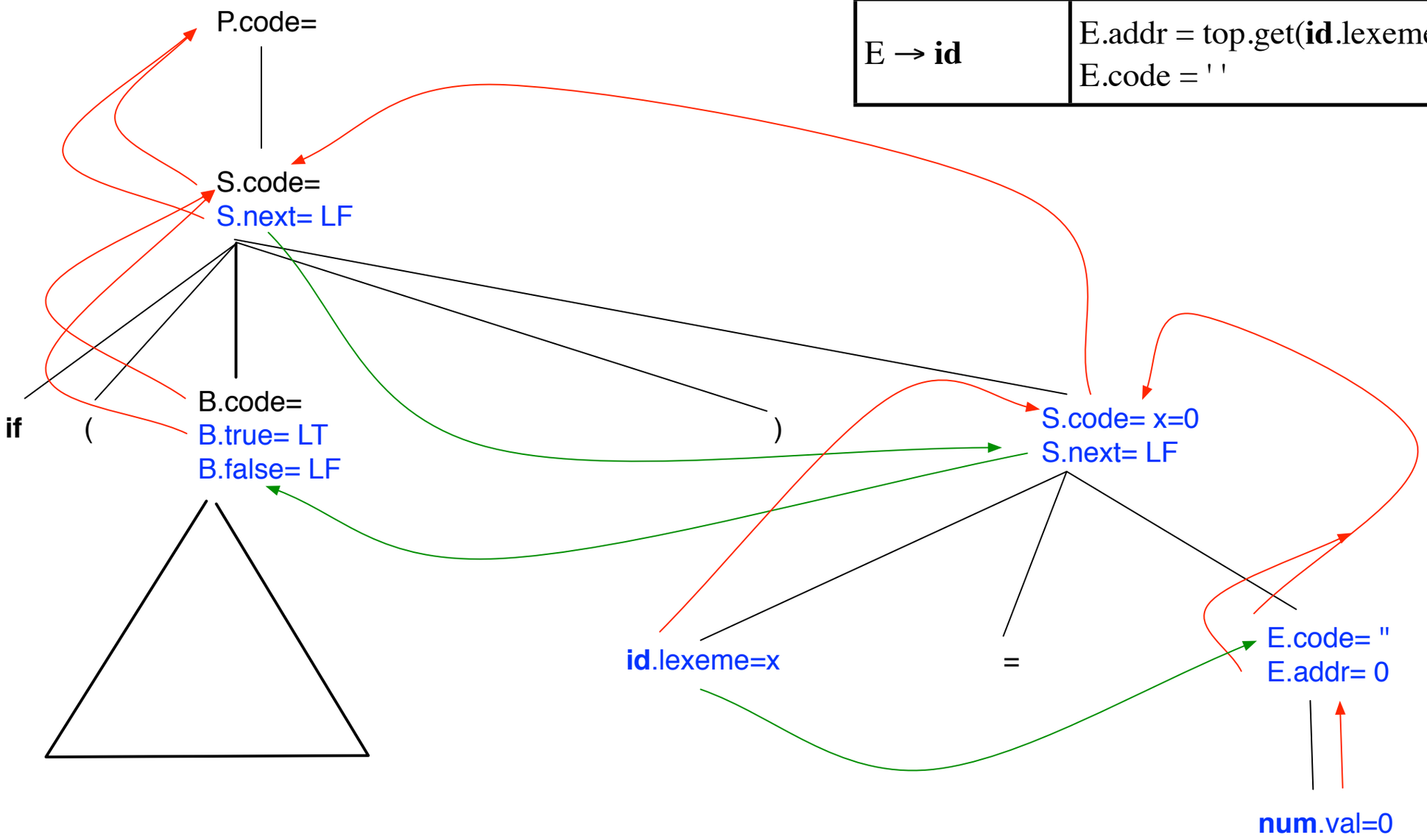


```
if(x<100 || x > 200 && x!=y) x=0;
```

P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **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 → **id** = E;	S.code = E.code || gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

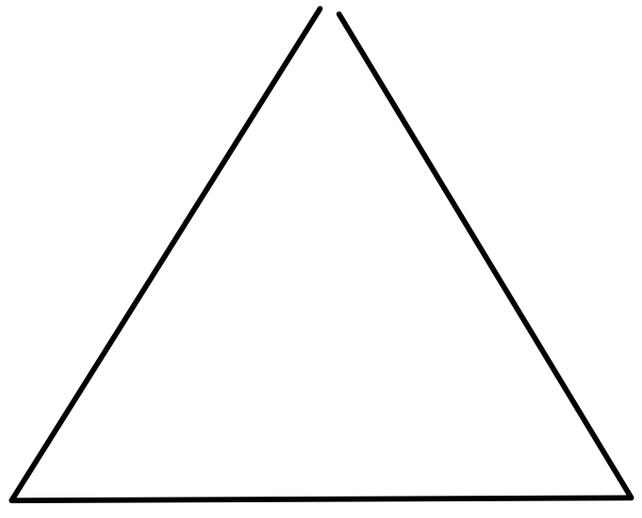


P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code
S → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



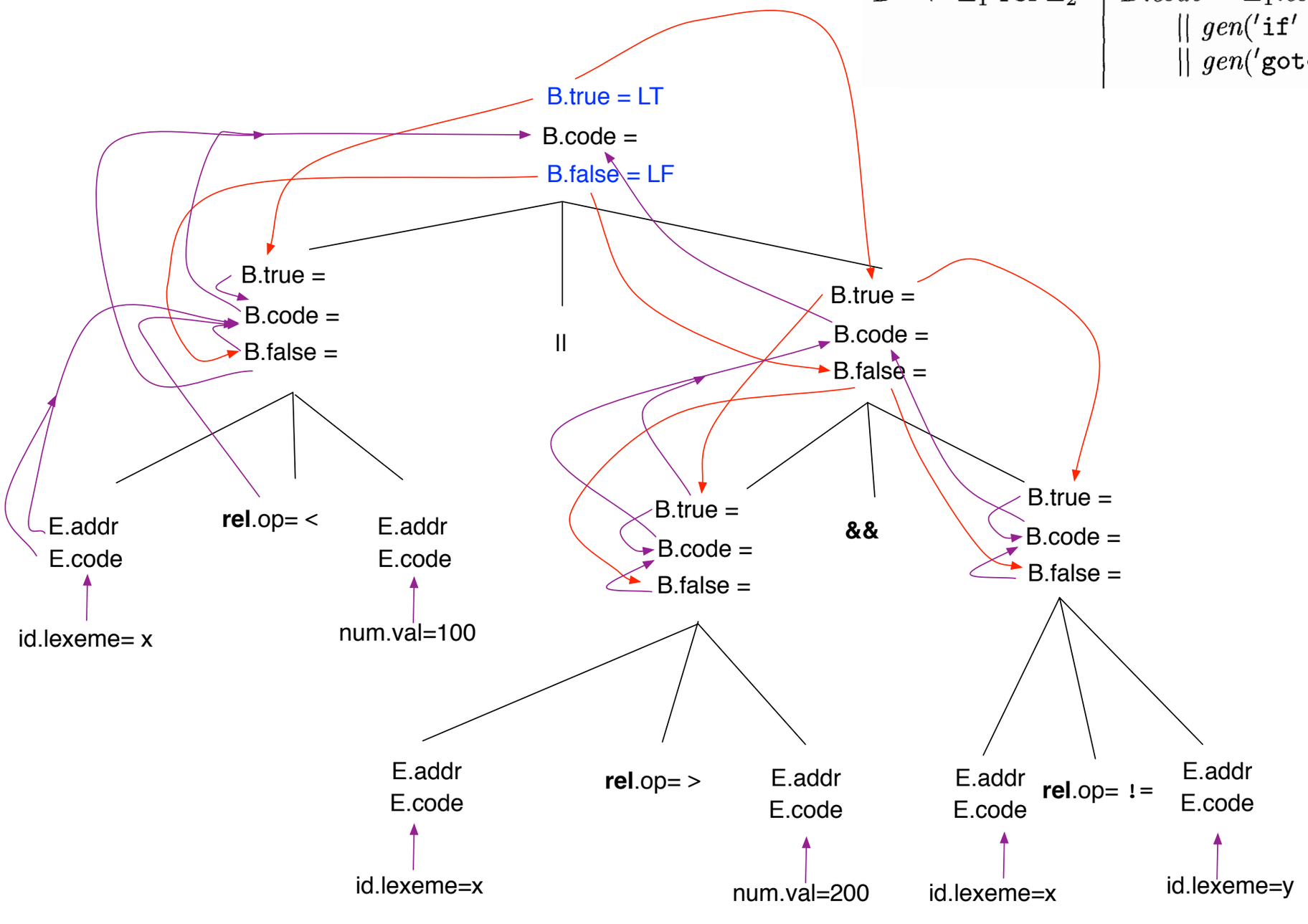
```
x<100 || x > 200 && x!=y
```

B.code=
B.true= LT
B.false= LF



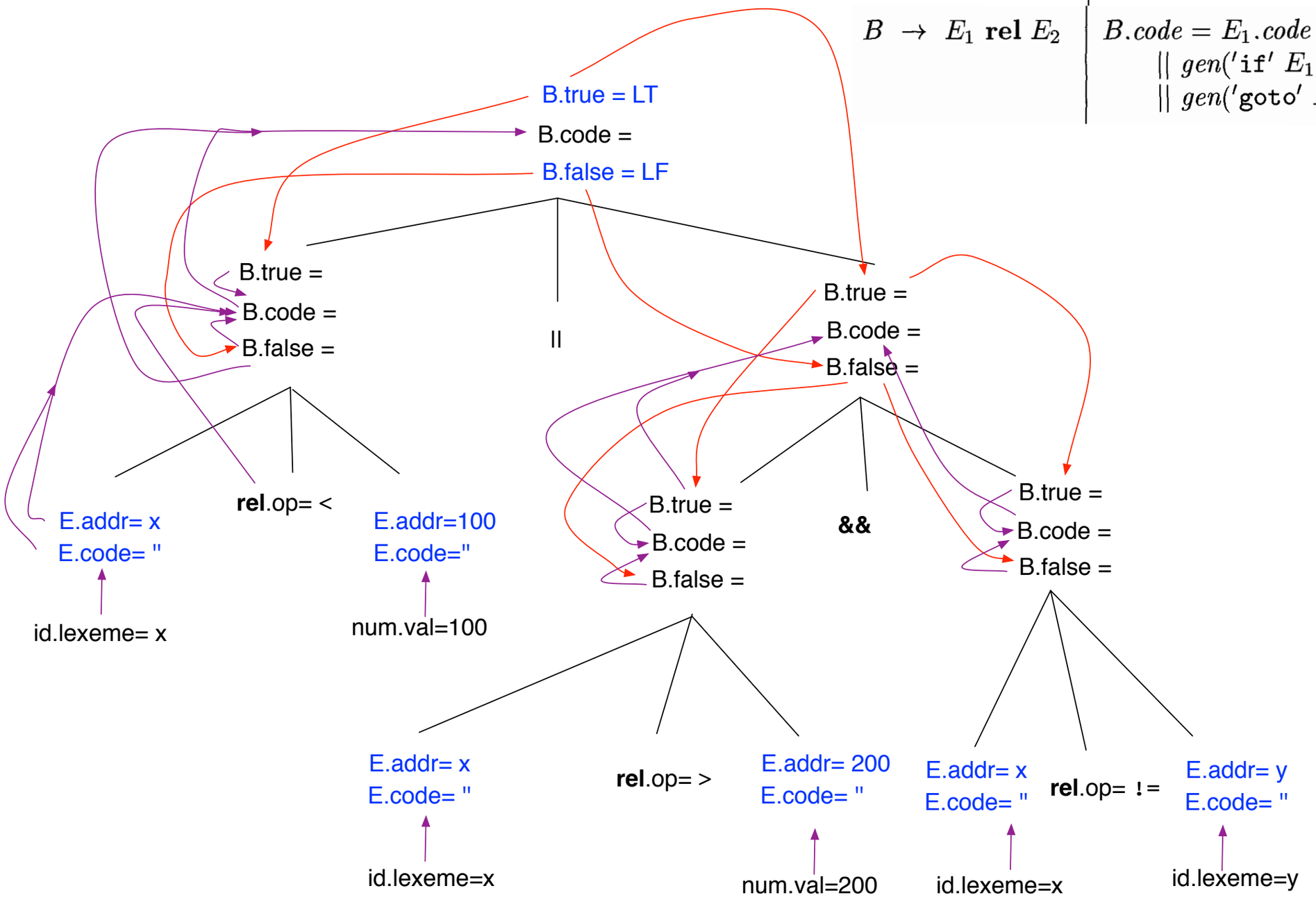
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



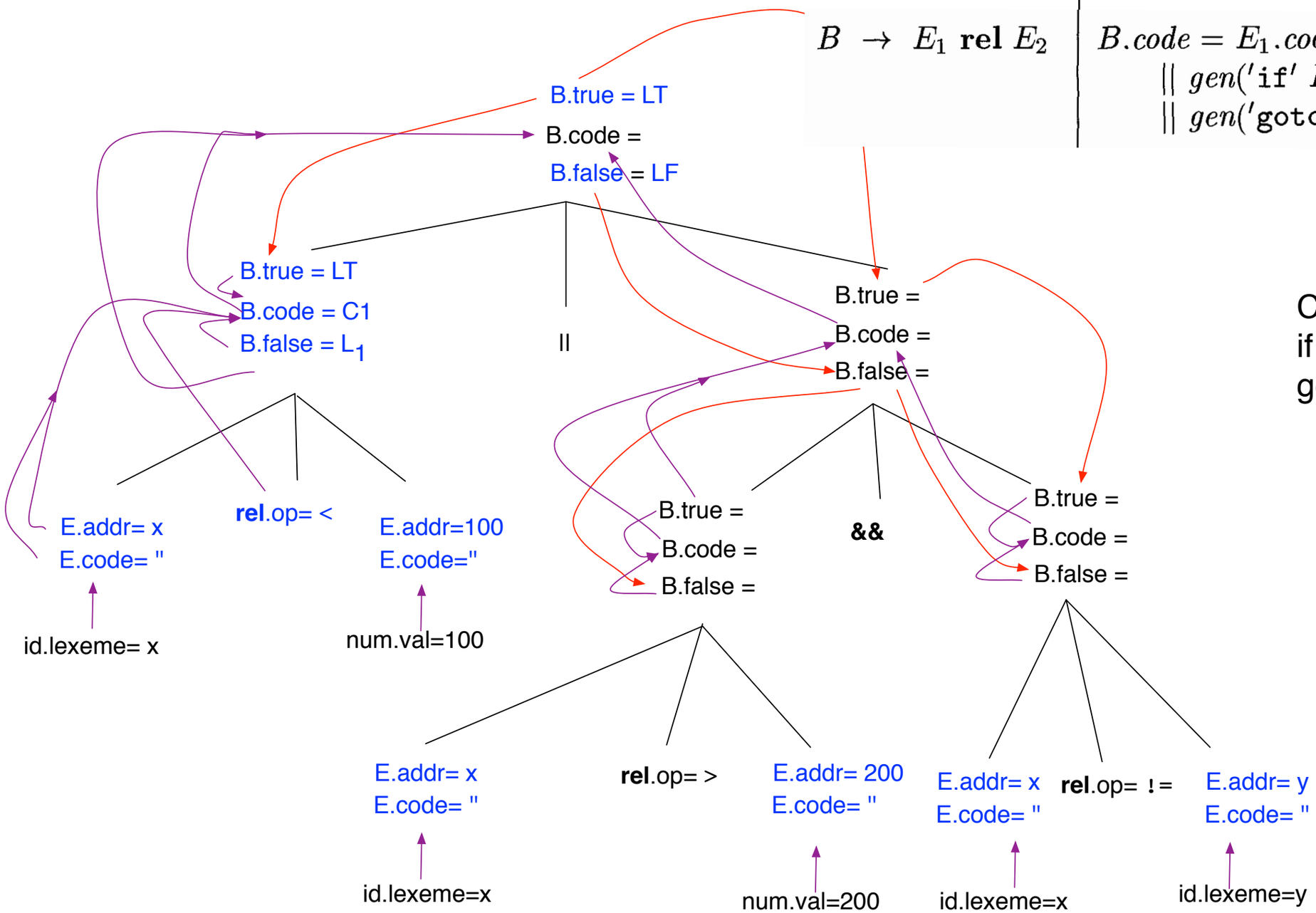
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

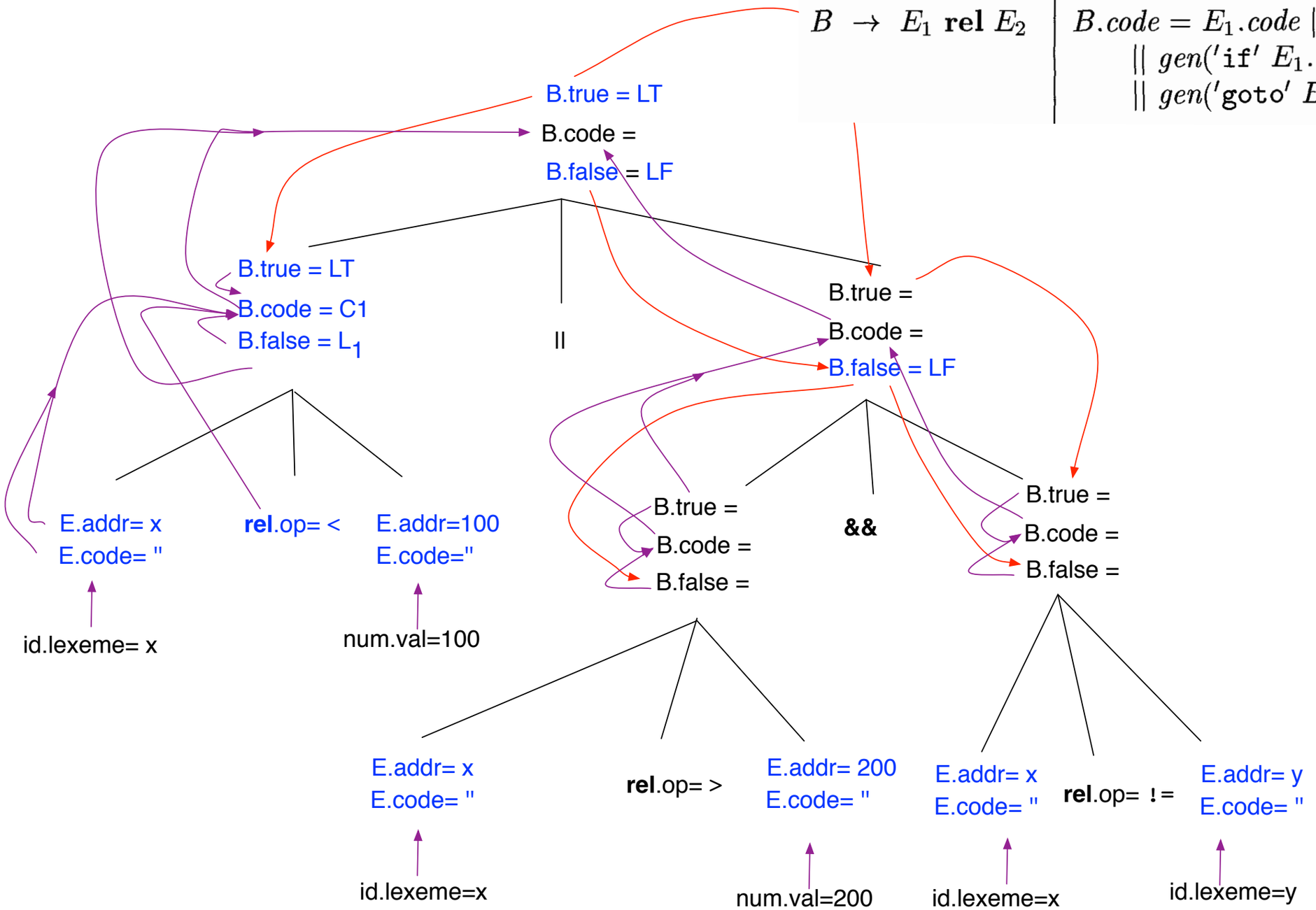
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



C1 =
if x < 100 goto LT
goto L_1

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

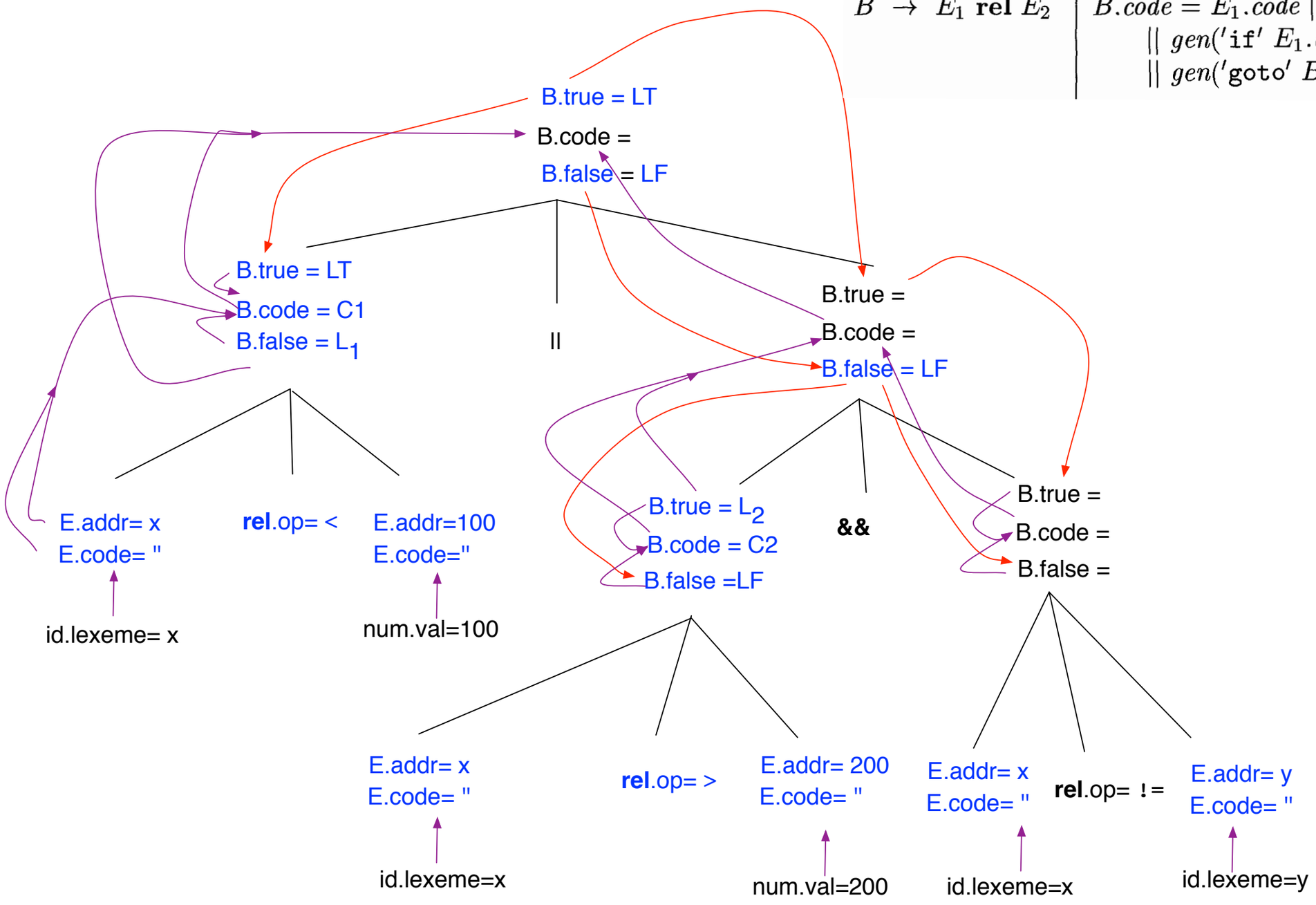
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



C1 =
if x < 100 goto LT
goto L_1

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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

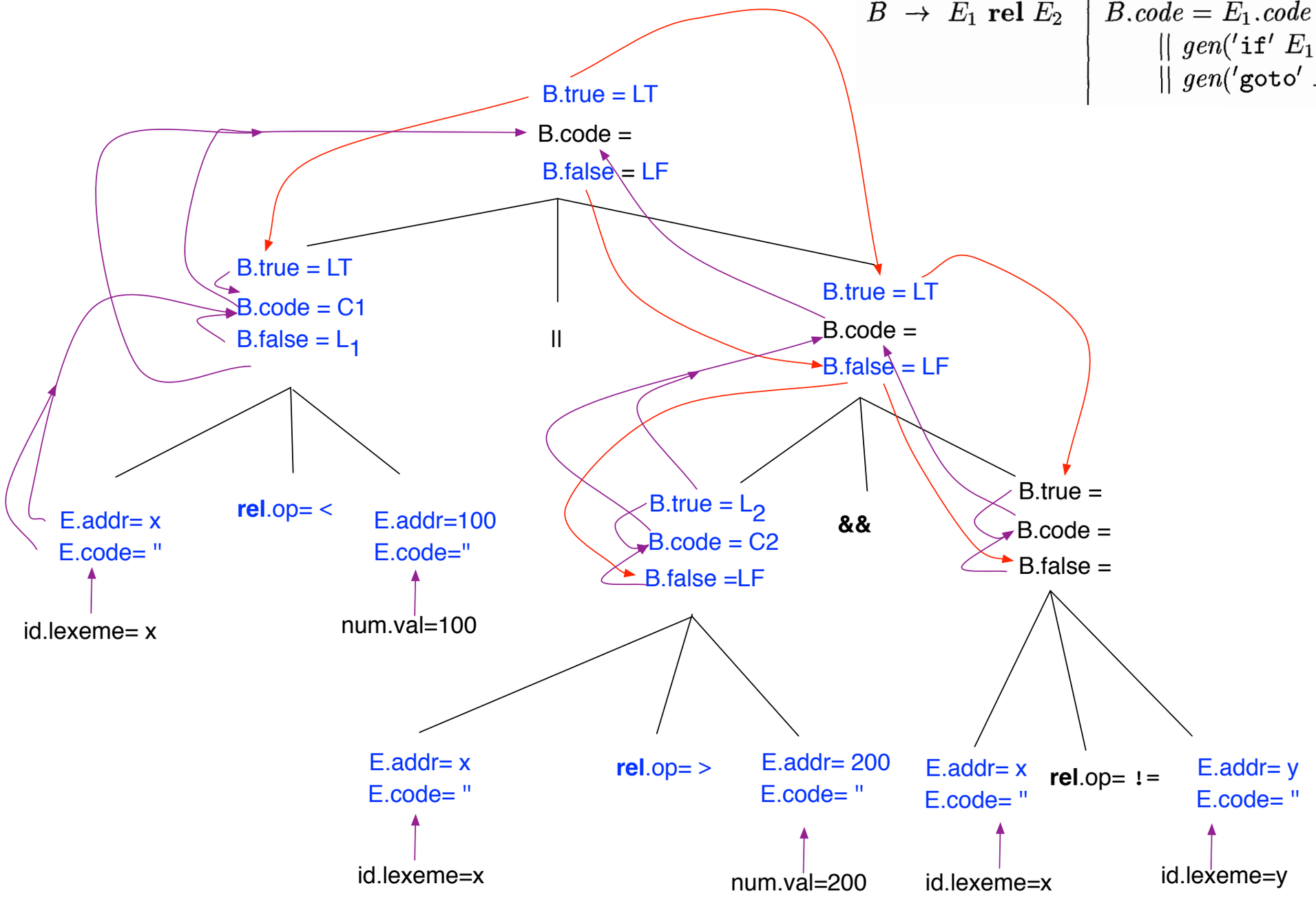


C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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

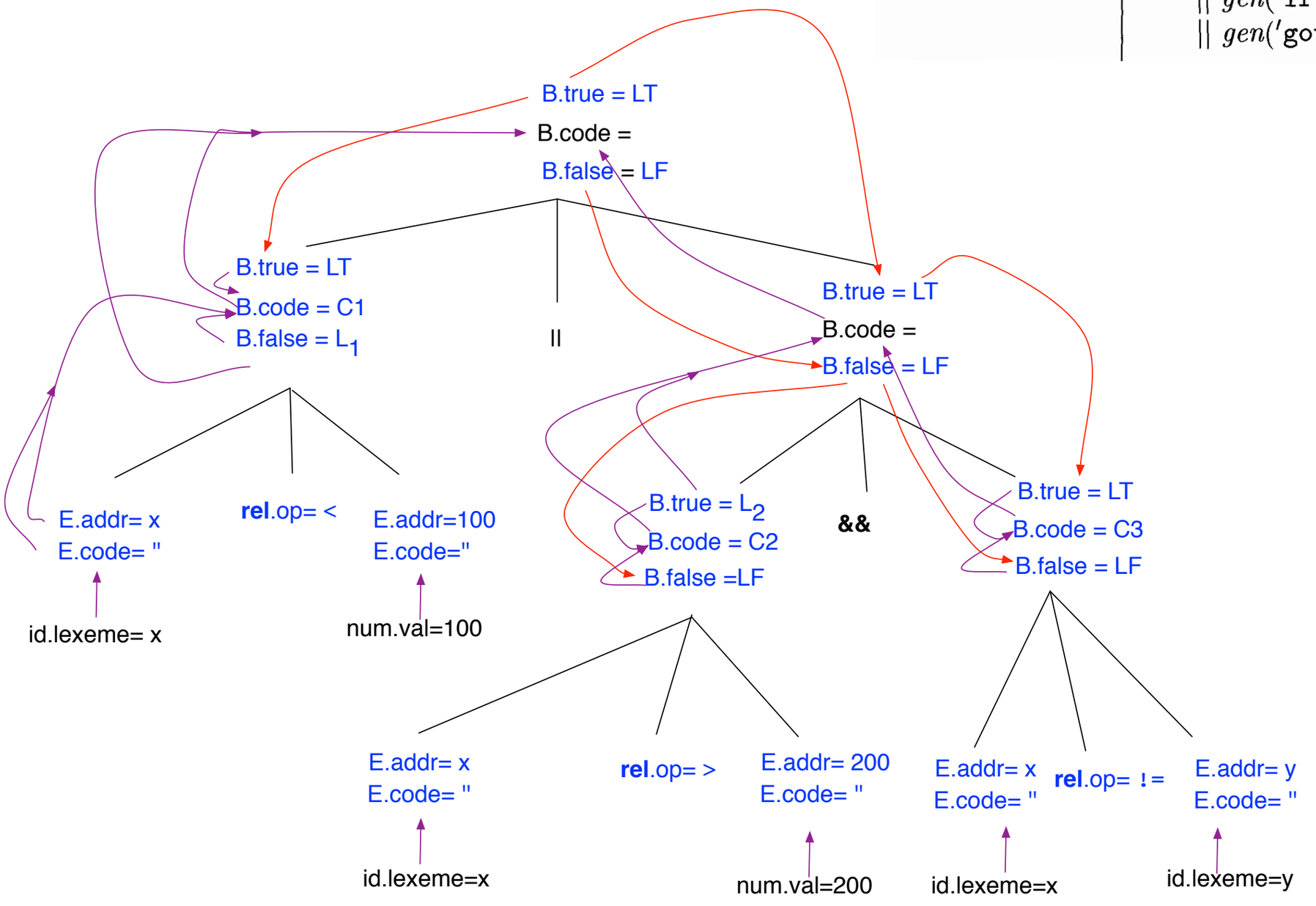


C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



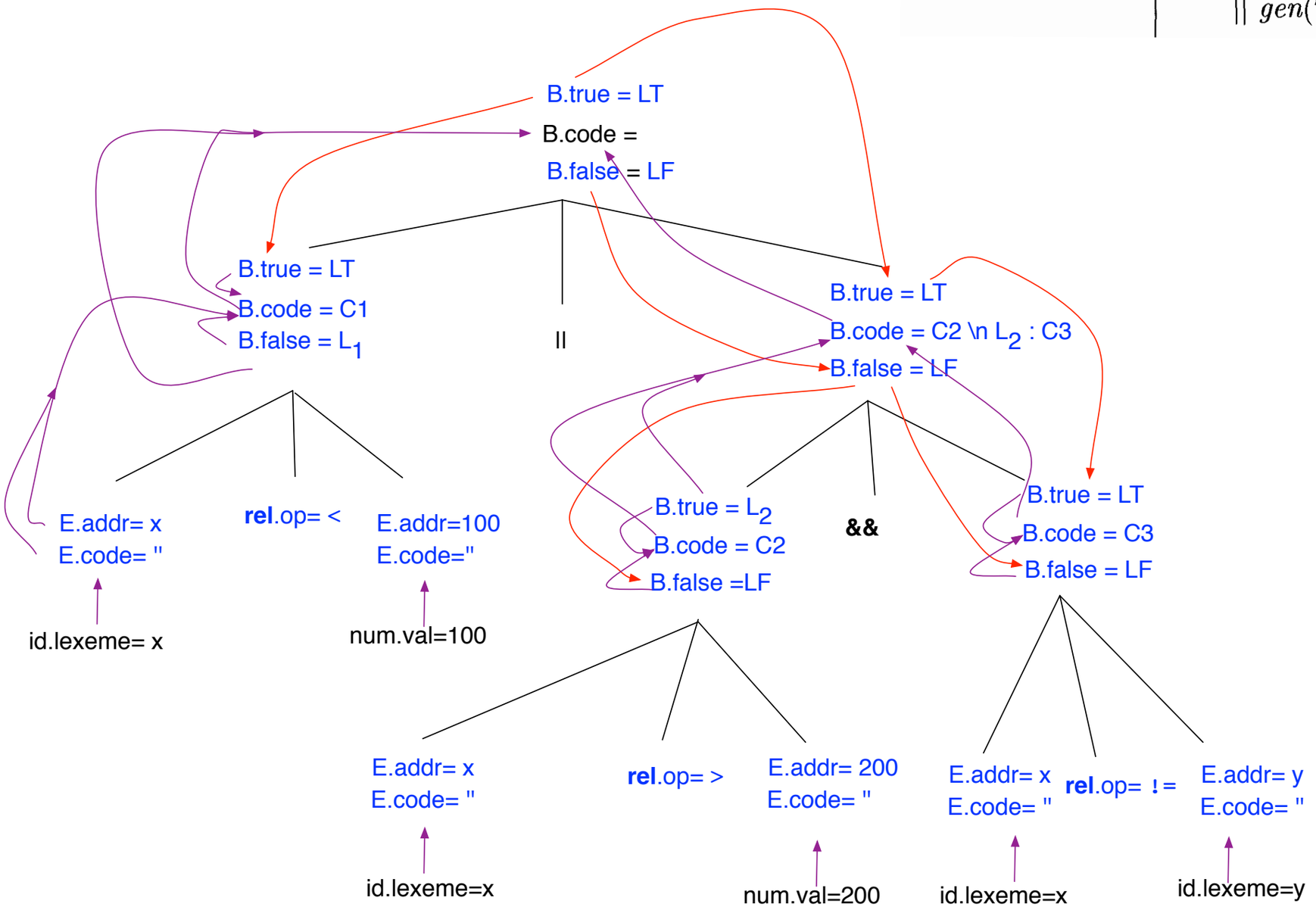
C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

C3 =
if x != y goto LT
goto LF

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



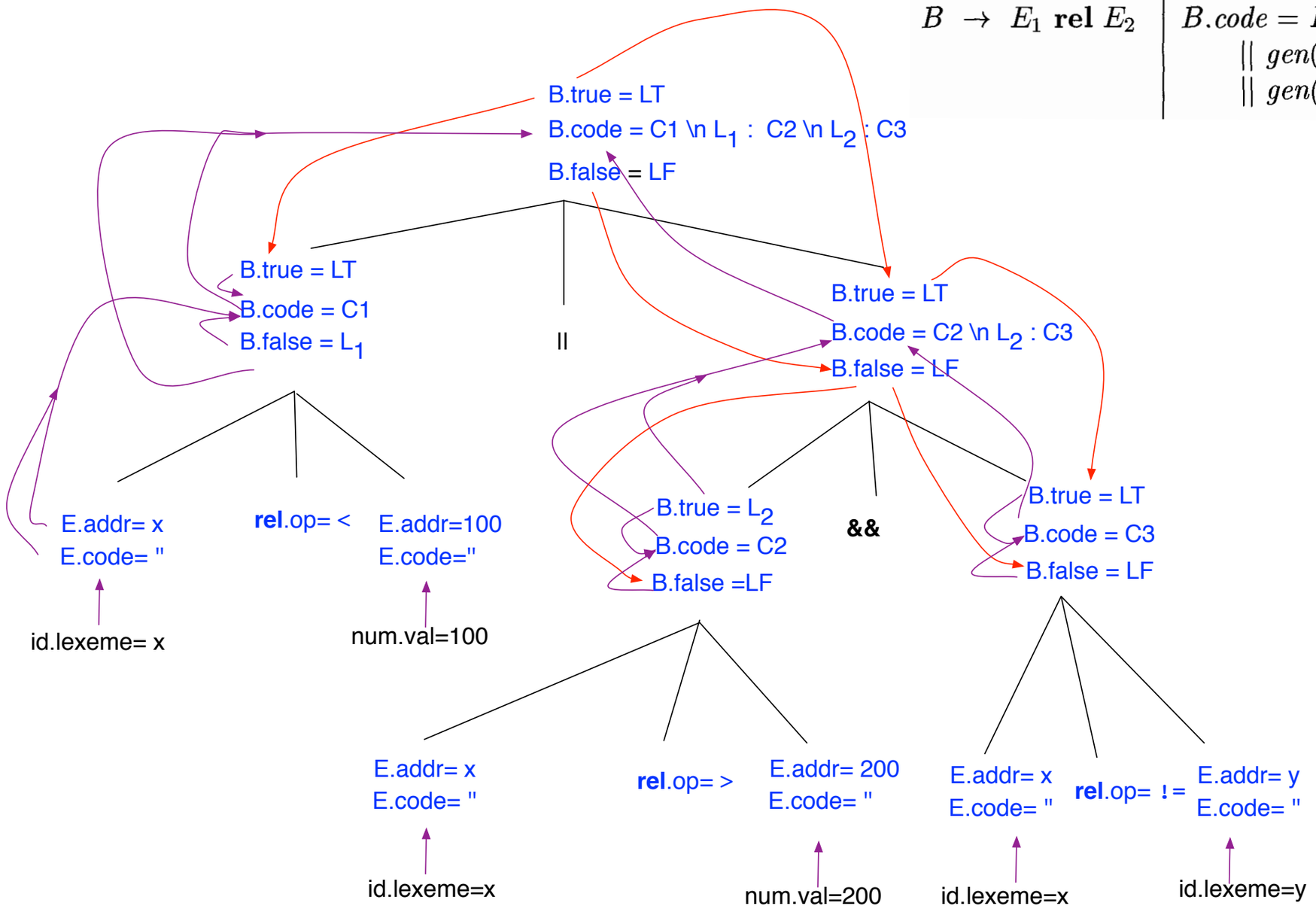
C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

C3 =
if x != y goto LT
goto LF

E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF

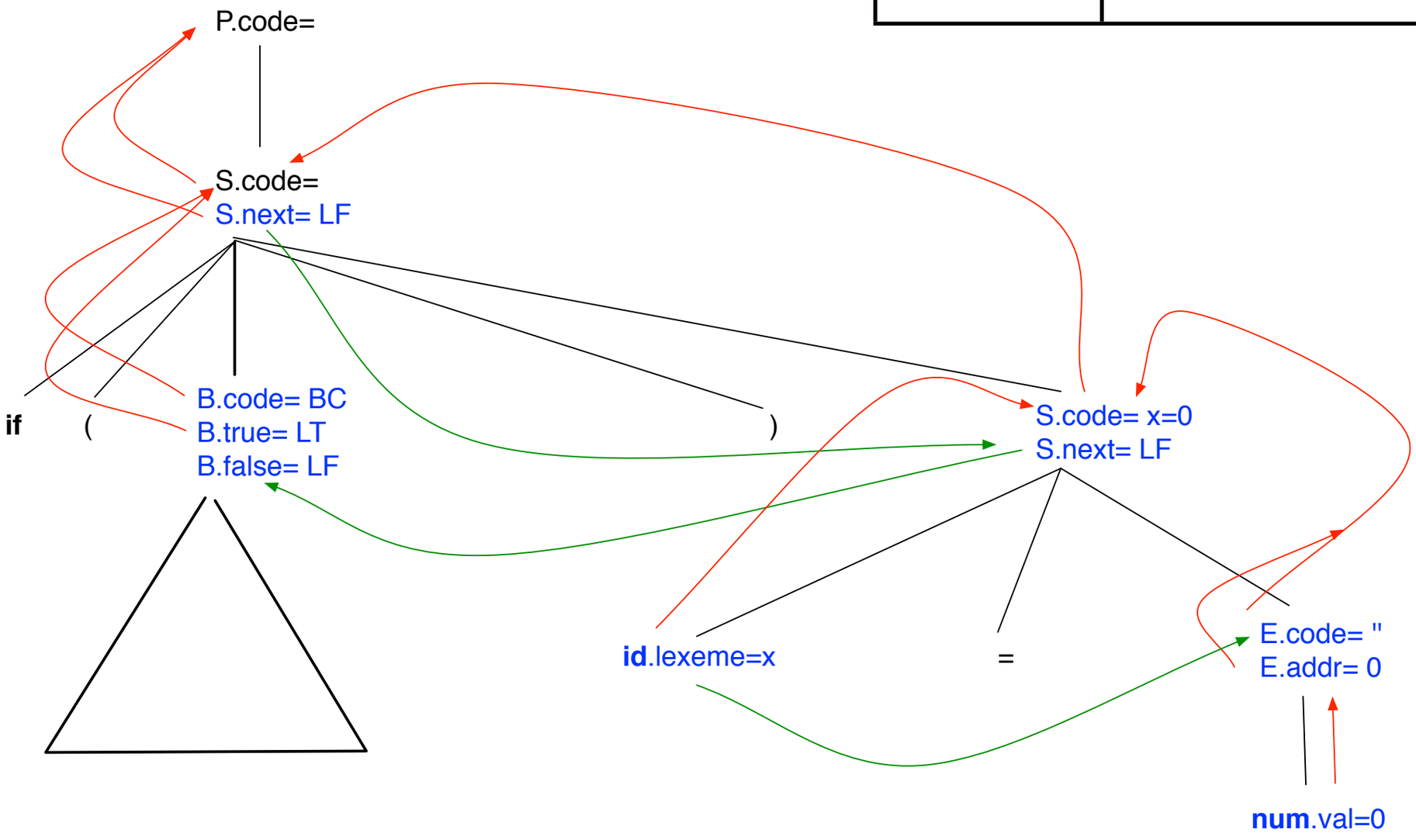
C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

C3 =
if x != y goto LT
goto LF

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

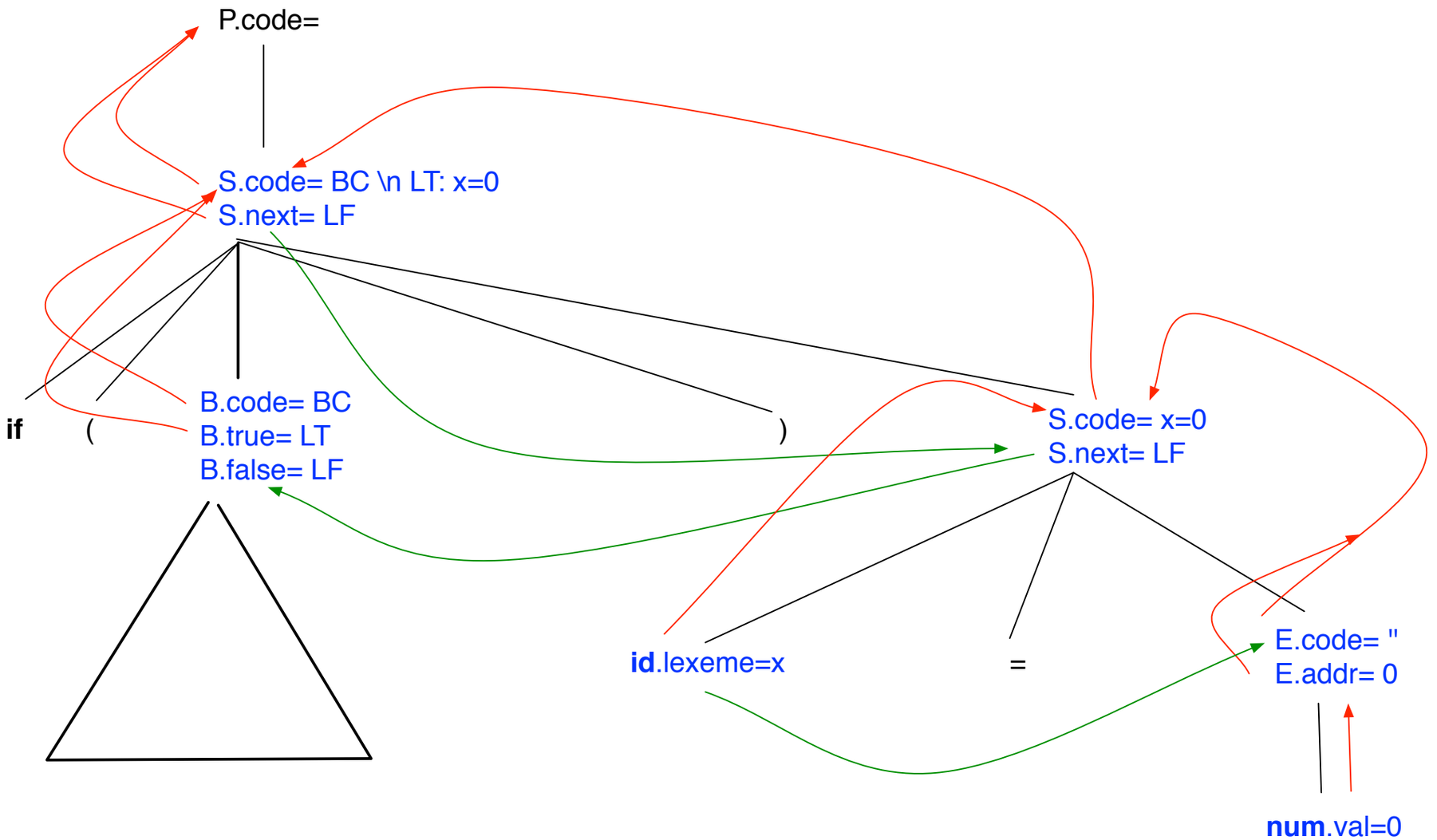
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code
S → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



BC= if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF

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

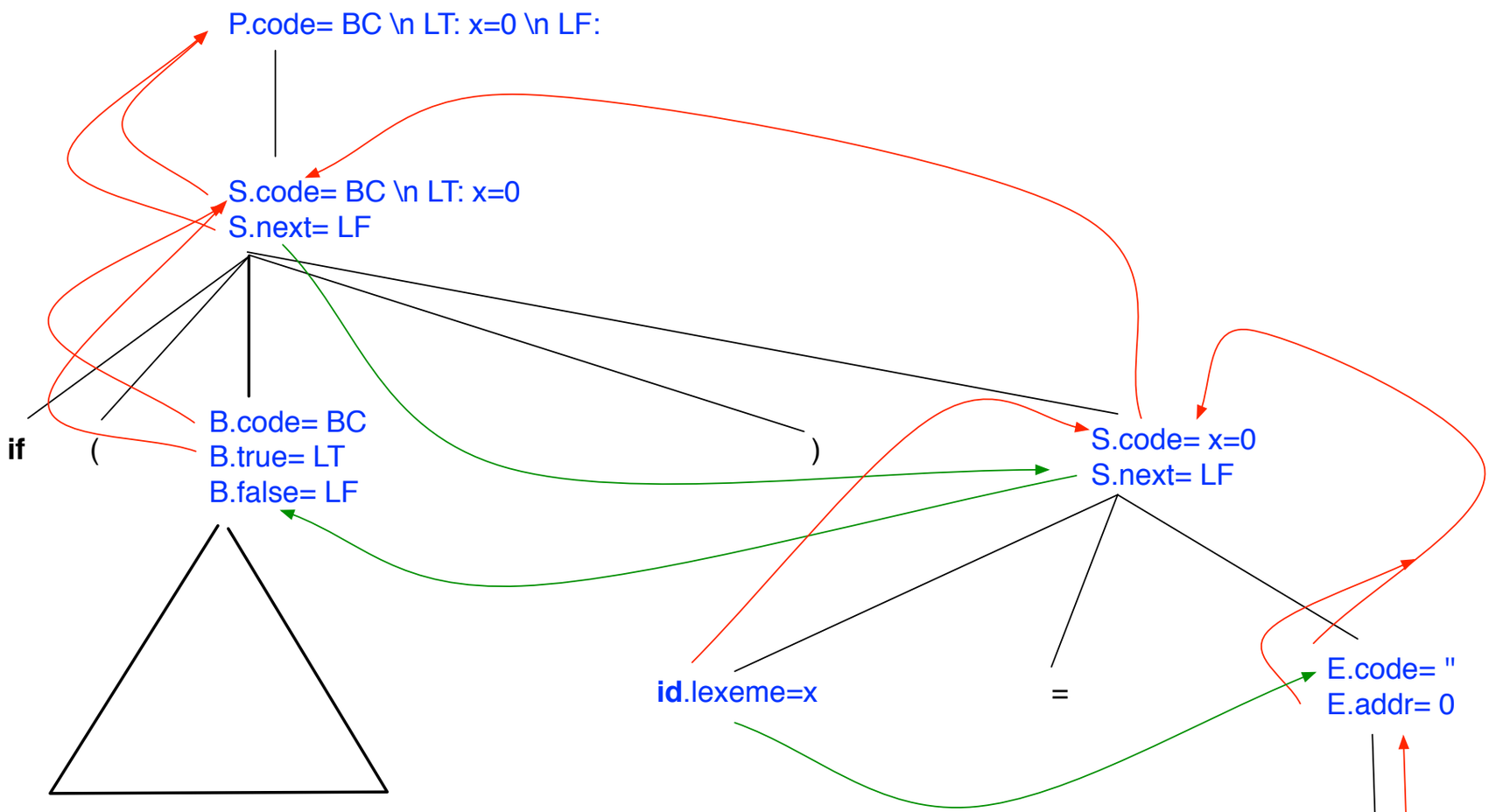
P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
S → **id** = E;	S.code = E.code || gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



BC= if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF

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

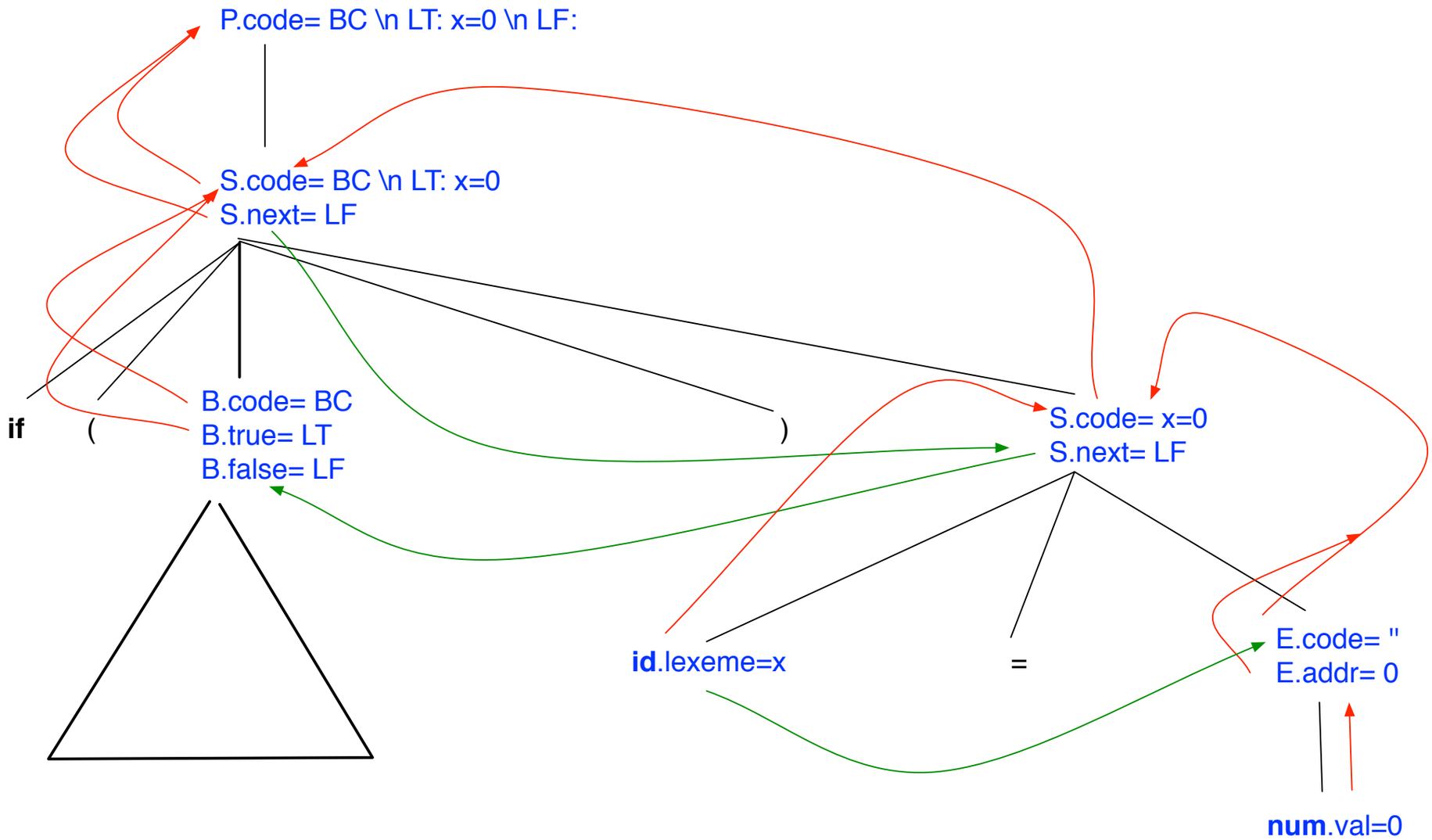
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code
S → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



BC= L_1 : if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF

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

P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **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 → **id** = E;	S.code = E.code || gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '



if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF
LT: x=0
LF:

BC= if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF

Esercizio

Generare il codice a tre indirizzi per i comandi:

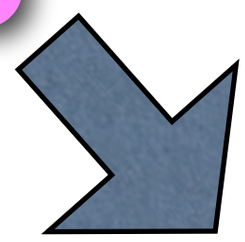
```
x=2;while(x < 3 && 1 < 2) x=x+4;
```

```
z=1;if(x<3 && z>5) x=11; else x=0;
```

AVOIDING REDUNDANT GOTOS

```
x > 200
```

```
     if x > 200 goto L4
     goto L1
L4:  ...
```



```
     ifFalse x > 200 goto L1
L4:  ...
```

fall means: **do not generate jump**

production	semantic rule
$S \rightarrow \textbf{if}(B)\ S_1$	B. true = fall B.false = S_1.next = S.next S.code = B.code ll S_1.code $B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$B \rightarrow E_1\ \textbf{rel}\ E_2$	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) ll 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 ll E_2.code ll s $B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$

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

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel()$ $B_1.false = fall$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = \textbf{if } B.true \neq fall \textbf{ then } B_1.code \mid\mid B_2.code$ $\textbf{else } B_1.code \mid\mid B_2.code \mid\mid label(B_1.true)$

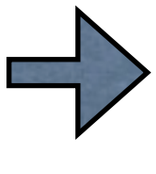
Note that the meaning of label *fall* for *B* is different from its meaning for B_1.

Suppose ***B.true* is *fall***; i.e, control falls through *B,* if *B* evaluates to true. Although *B* evaluates to true if *B1* does, $B_1.true$ must ensure that control jumps over the code for *B2* to get to the next instruction after *B.*

On the other hand, if *B1* evaluates to false, the truth-value of *B* is determined by the value of *B2,* so the rules ensure that $B_1.false$ corresponds to control falling through from *B1* to the code for *B2.*

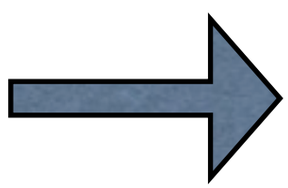
S → **id** = B

`x<100 || x > 200 && x!=y`



```
     if x < 100 goto LT
     goto L1
L1  :if x > 200 goto L2
     goto LF
L2  :if x != y goto LT
     goto LF
```

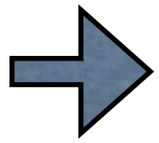
`p = x<100 || x > 200 && x!=y`



```
     if x < 100 goto LT
     goto L1
L1: if x > 200 goto L2
     goto LF
L2: if x != y goto LT
     goto LF
LT: t = true
     goto L
LF: t = false
L:  p = t
```

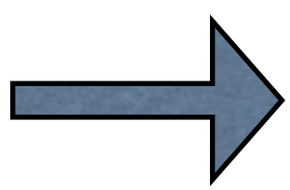
S → **id** = B	S.addr = **new** Temp() S.lab = newlabel() B.true = newlabel() B.false= newlabel() S.code = B.code ‖ label(B.true) ‖ gen(S.addr '= `true`') ‖ gen('`goto`' S.lab) ‖ label(B.false)‖ gen(S.addr '= `false`') ‖ label(S.lab) ‖ gen(top.get(**id**.lexeme) '=' S.addr)

```
x<100 || x > 200 && x!=y
```



```
    if x < 100 goto LT
    goto L1
L1  :if x > 200 goto L2
    goto LF
L2  :if x != y goto LT
    goto LF
```

```
p = x<100 || x > 200 && x!=y
```



```
    if x < 100 goto LT
    goto L1
L1: if x > 200 goto L2
    goto LF
L2: if x != y goto LT
    goto LF
LT: t = true
    goto L
LF: t = false
L:  p = t
```

$S \rightarrow id = E\ ; \mid if\ (\ E\ )\ S \mid while\ (\ E\ )\ S \mid S\ S$
$E \rightarrow E \mid\mid E \mid E\ \&\&\ E \mid E\ rel\ E \mid E + E \mid (E) \mid id \mid true \mid false$

a single nonterminal **E** for expressions

We can handle these two roles of expressions by using separate code-generation functions. Suppose that, attribute **E.n** denotes the syntax-tree node for an expression *E* and that nodes are objects. Let method ***jump*** generate jumping code at an expression node, and let method ***rvalue*** generate code to compute the value of the node into a temporary.

When *E* appears in S → **while** (*E*) S_1, method *jump* is called at node *E.n.* The implementation of *jump* is based on the rules for boolean expression. Specifically, 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.*

When *E* appears in S → **id** = *E;,* method *rvalue* is called at node *E.n.* If *E* has the form $E_1 + E_2$, the method call *E.n.rvalue()* generates code. 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 the true and false exits, respectively, from the jumping code.

$T \rightarrow B$ $\quad \{ t = B.type; w = B.width; \}$

$\quad\quad C$ $\quad \{T.type = C.type; T.width = C.width;\}$

$B \rightarrow \textbf{int}$ $\quad \{ B.type = integer; B.width = 4; \}$

$B \rightarrow \textbf{float}$ $\quad \{ B.type = float; B.width = 8; \}$

$C \rightarrow \epsilon$ $\quad \{ C.type = t; C.width = w; \}$

$C \rightarrow \texttt{[}\ \textbf{num}\ \texttt{]}\ C_1$ $\quad \{C.type = array(\textbf{num}.value, C_1.type);$
$C.width = \textbf{num}.value \times C_1.width; \}$

$P \rightarrow$ $\quad \{ offset = 0; \}$

$\quad\quad D$

$D \rightarrow T\ \textbf{id}\ ;$ $\quad \{ top.put(\textbf{id}.lexeme, T.type, offset);$
$offset = offset + T.width; \}$

$\quad\quad D_1$

$D \rightarrow \epsilon$

$E \rightarrow \texttt{num}$

$E \rightarrow \texttt{id}$

$E \rightarrow E_1\ \texttt{mod}\ E_2$

$E \rightarrow E_1\ \texttt{binop}\ E_2$

$E \rightarrow E_1[E_2]$

$T \rightarrow B$ C { $t = B.type$; $w = B.width$; } {$T.type = C.type$; $T.width = C.width$;}

$B \rightarrow$ **int** { $B.type = integer$; $B.width = 4$; }

$B \rightarrow$ **float** { $B.type = float$; $B.width = 8$; }

$C \rightarrow \epsilon$ { $C.type = t$; $C.width = w$; }

$C \rightarrow$ [**num**] C_1 {$C.type = array($**num**$.value, C_1.type)$; $C.width =$ **num**$.value \times C_1.width$; }

$P \rightarrow D$ { $offset = 0$; }

$D \rightarrow T$ **id** ; D_1 { $top.put($**id**$.lexeme, T.type, offset)$; $offset = offset + T.width$; }

$D \rightarrow \epsilon$

$E \rightarrow$ num { ***E.type*** := integer }

$E \rightarrow$ id { ***E.type*** := ***lookup*** (id.***entry***) }

$E \rightarrow E_1$ mod E_2
{ ***E.type*** := if E_1. ***type*** = integer and ***sequiv*** $(E_1.type, E_2.type)$ then integer else ***type_error*** }

$E \rightarrow E_1$ binop E_2
{ ***E.type*** := if ***sequiv*** $(E_1.type, E_2.type)$ then $E_1.type$ else ***type_error*** }

$E \rightarrow E_1[E_2]$
{ ***E.type*** := if E_2. ***type*** = integer and $E_1.type$ = ***array(s,t)*** then t else ***type_error*** }

$$S \rightarrow \texttt{id} := E$$

$$S \rightarrow \texttt{if}\ E\ \texttt{then}\ S_1$$

$$S \rightarrow \texttt{while}\ E\ \texttt{do}\ S_1$$

$$S \rightarrow S_1; S_2$$

$S \rightarrow$ `id` $:=$ E { ***S.type*** $:=$ `if` ***sequiv*** (`id.`***type***,***E.type***)
`then` ***void*** `else` ***type_error*** }

$S \rightarrow$ `if` E `then` S_1
{ ***S.type*** $:=$ `if` ***E.type*** $=$ **boolean**
`then` S_1***.type*** `else` ***type_error*** }

$S \rightarrow$ `while` E `do` S_1
{ ***S.type*** $:=$ `if` ***E.type*** $=$ **boolean**
`then` S_1***.type*** `else` ***type_error*** }

$S \rightarrow S_1; S_2$
{ ***S.type*** $:=$ `if` S_1***.type*** $=$ ***void*** `and`
S_2***.type*** $=$ ***void***
`then` ***void*** `else` ***type_error*** }