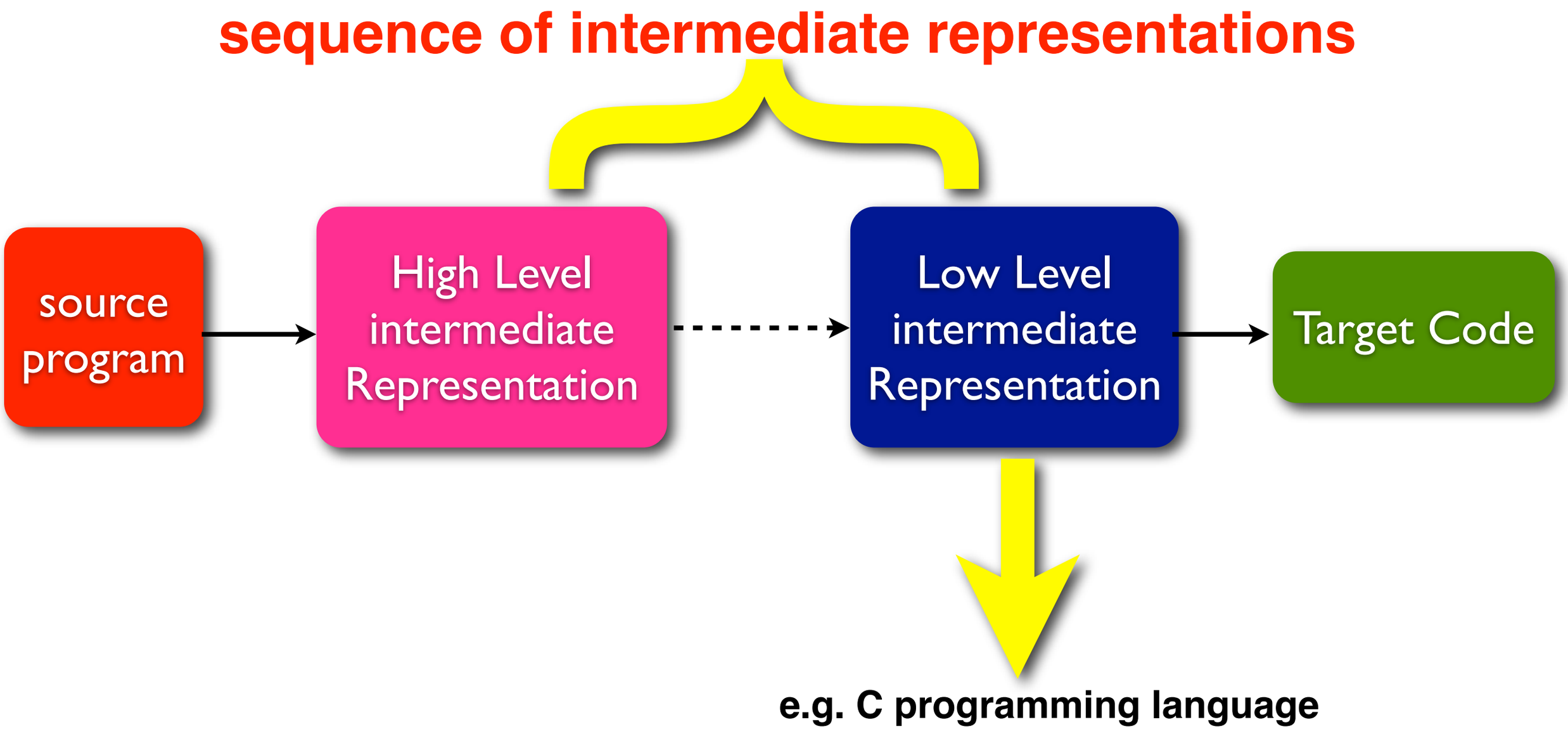
intermediate-Code Generation


sequence of intermediate representations
source program
High Level intermediate Representation
Low Level intermediate Representation
Target Code
e.g. C programming language

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)  }
```

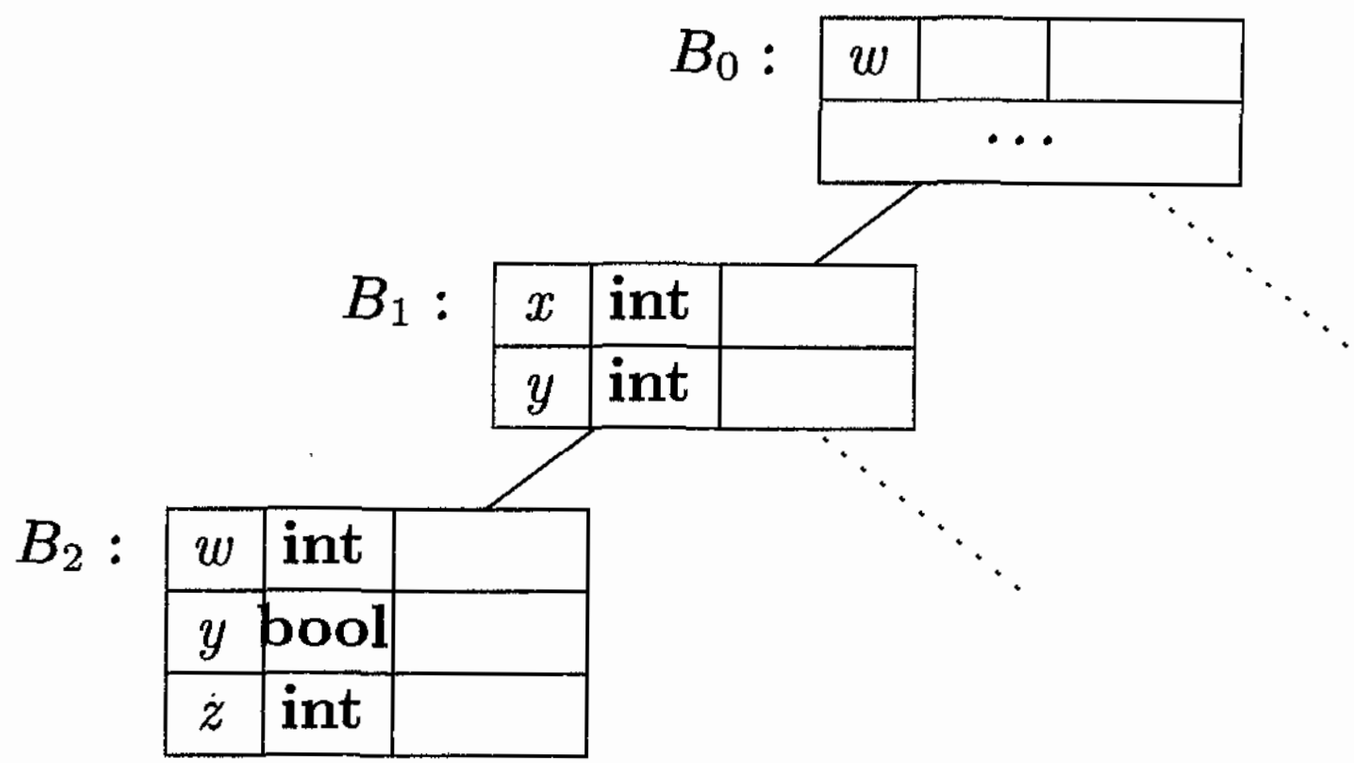


Figure 2.36: Chained symbol tables for Example 2.15

The Java implementation of chained symbol tables in Fig. 2.37 defines a class `Env`, short for *environment*.[9] Class `Env` supports three operations:

- *Create a new symbol table.* The constructor `Env(p)` on lines 6 through 8 of Fig. 2.37 creates an `Env` object with a hash table named `table`. The object is chained to the environment-valued parameter `p` by setting field `next` to `p`. Although it is the `Env` objects that form a chain, it is convenient to talk of the tables being chained.

- *Put* a new entry in the current table. The hash table holds key-value pairs, where:

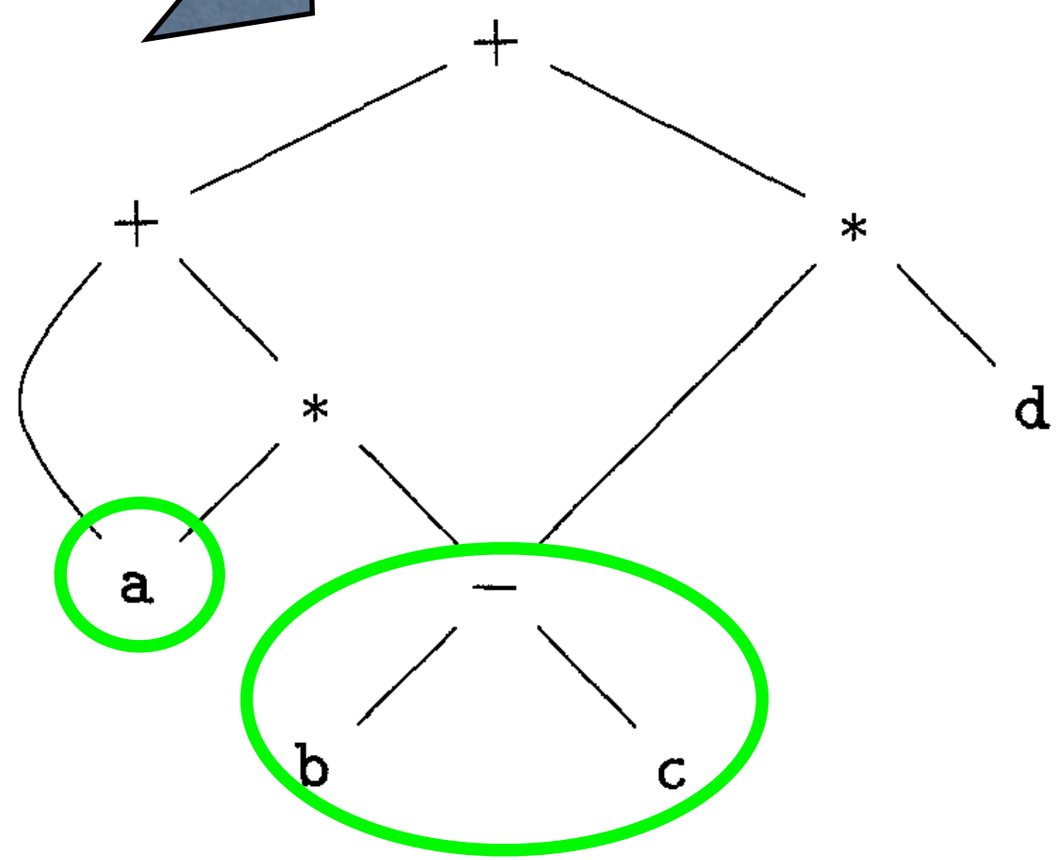
  - The *key* is a string, or rather a reference to a string. We could alternatively use references to token objects for identifiers as keys.
  - The *value* is an entry of class `Symbol`. The code on lines 9 through 11 does not need to know the structure of an entry; that is, the code is independent of the fields and methods in class `Symbol`.

- *Get* an entry for an identifier by searching the chain of tables, starting with the table for the current block. The code for this operation on lines 12 through 18 returns either a symbol-table entry or `null`.

```java
package symbols;
import java.util.*; import lexer.*; import inter.*;

public class Env {

  private Hashtable table;
  protected Env prev;

  public Env(Env n) { table = new Hashtable(); prev = n; }

  public void put(Token w, id i) { table.put(w, i); }

  public id get(Token w) {
     for( Env e = this; e != null; e = e.prev ) {
        id found = (id)(e.table.get(w));
        if( found != null ) return found;
     }
     return null;
  }
}
```

DAG

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

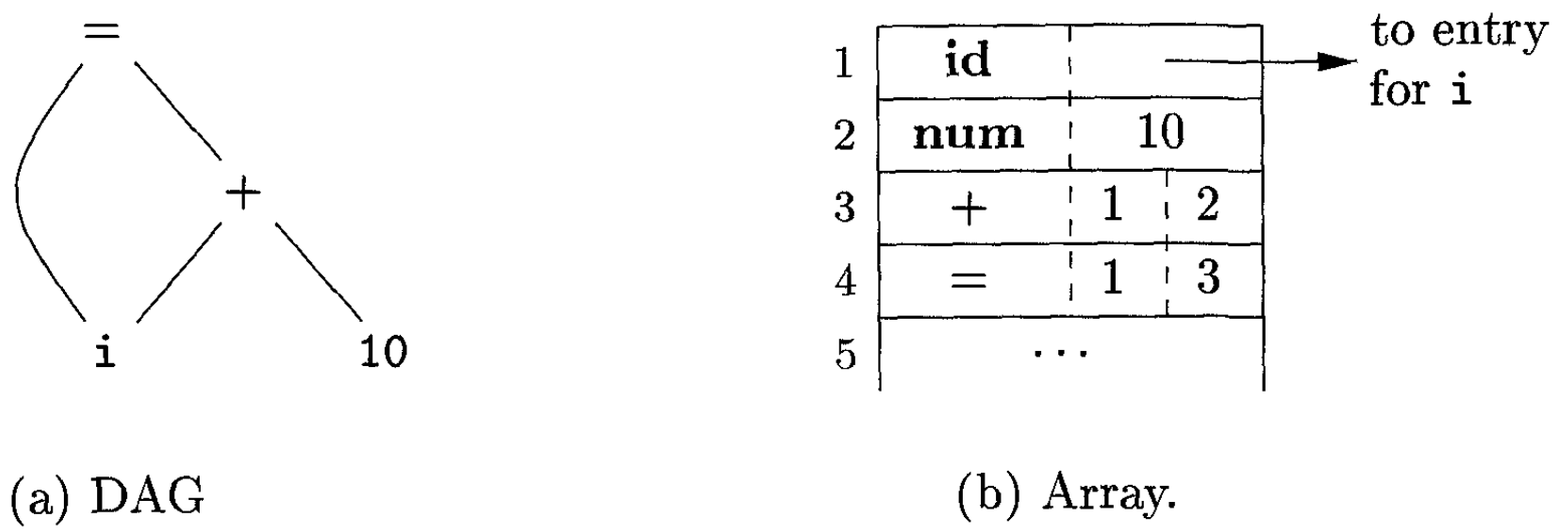


PRODUCTiON	SEMANTiC RULES
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow$ **id**	$T.node$ = **new**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

1) $P_1 = Leaf$ (**id**, $entry\text{-}a)$
2) $P_2 = Leaf$ (**id**, $entry\text{-}a) = P_1$
3) $P_3 = Leaf$ (**id**, $entry\text{-}b)$
4) $P_4 = Leaf$ (**id**, $entry\text{-}c)$
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) $P_S = Leaf$ (**id**, $entry\text{-}b) = P_3$
9) $P_9 = Leaf$ (**id**, $entry\text{-}c) = P_4$
1) $P_{10} = Node("-", P_3, P_4) = P_5$
2) $P_{11} = Leaf$ (**id**, $entry\text{-}d)$
3) $P_{12} = Node("*",P_5,P_{11})$
4) $P_{13} = Node("+",P_7,P_{12})$

Steps for constructing the DAG

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



(a) DAG (b) Array.

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

Algorithm 6.3: *The value-number method for constructing the nodes of a DAG*.
iNPUT: Label ***op***, node ***l***, and node ***r***,

OUTPUT: The value number of a node in the array with signature ⟨***op, l, r***⟩.

METHOD: Search the array for a node *M* with label ***op***, left child (with value-number) ***l***, and right child (with value-number) ***r***, if there is such a node, return the value number of *M*. if not, create in the array a new node *N* with label ***op***, left child ***l***, and right child ***r***, and return its value number

While Algorithm 6.3 yields the desired output, **searching the entire array every time we are asked to locate one node is expensive**, especially if the array holds expressions from an entire program.

A more efficient approach is to use a **hash table**, in which the nodes are put into "buckets," each of which typically will have only a few nodes

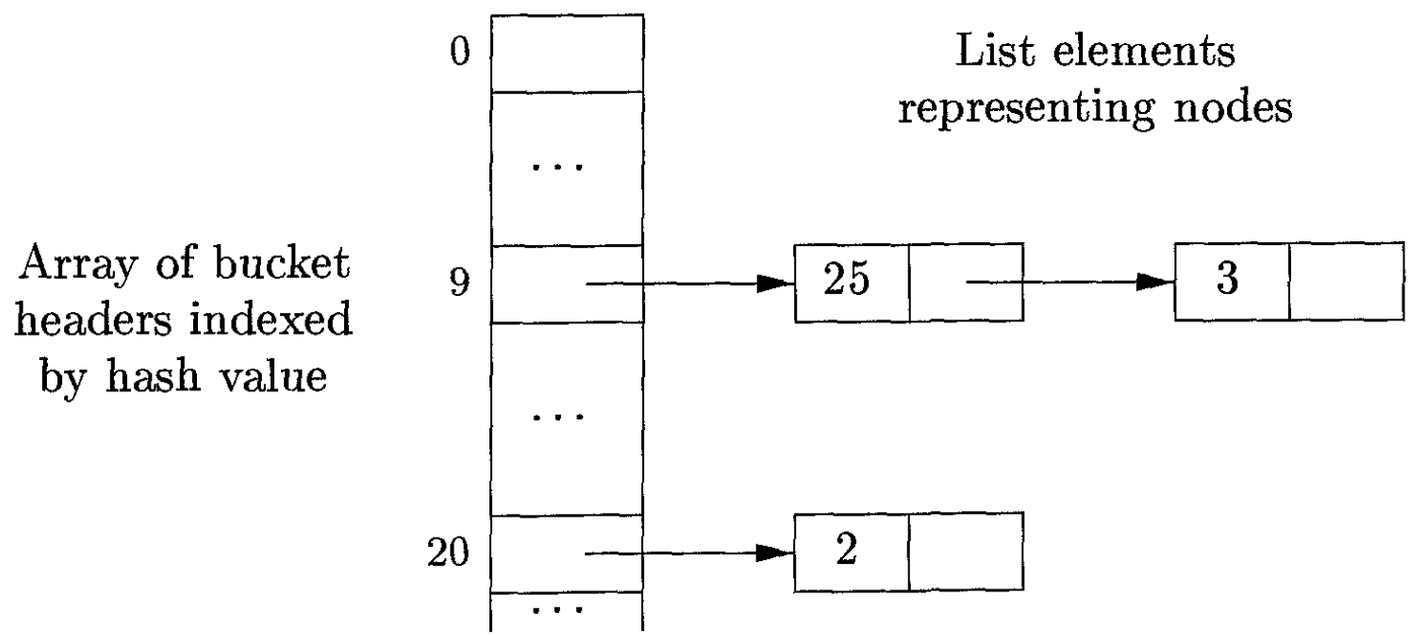
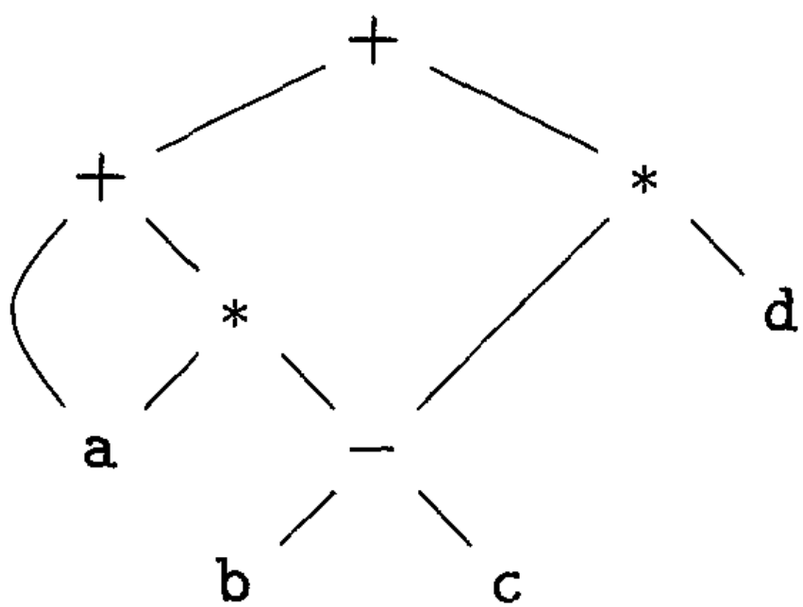


Figure 6.7: Data structure for searching buckets

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 and instructions

An address can be one of the following:

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.

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`, which apply a relational operator (`<`, `==`, `>=`, etc.) to *x* and *y,* and execute the instruction with label *L* next if *x* stands in relation *relop* to *y.* if not, 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+l; while (a[i] < v);
```

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

Quadruples

quadruple (**quad**):

op	*arg*$_1$	*arg*$_2$	*result*

`x = y + z` →

+	*y*	*z*	*x*

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.***

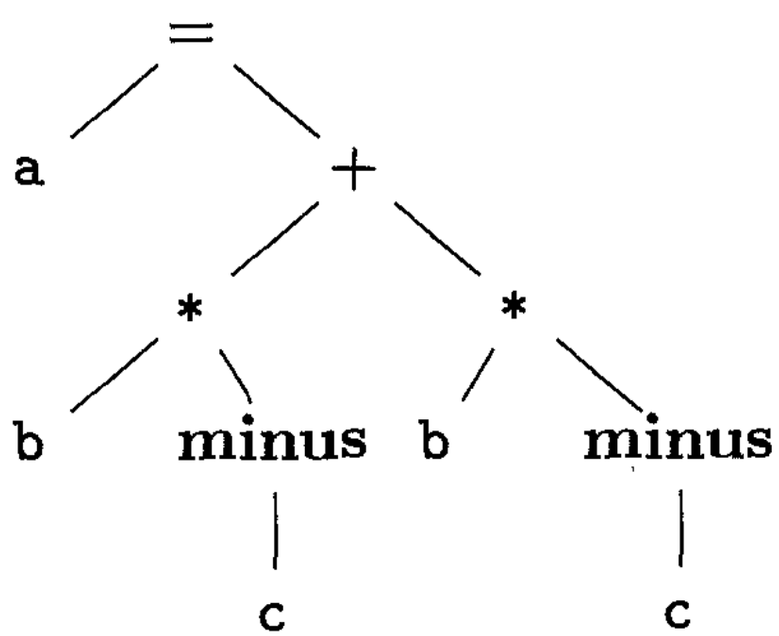
```
t1 =  minus c
t2 =  b*t1
t3 =  minus c
t4 =  b*t3
t5 =  t2+t4
a  =  t5
```

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***



(a) Syntax tree

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

(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)
	...		

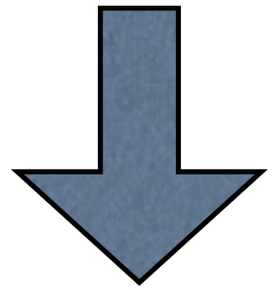
indirect triples representation of three-address code

Static Single-Assignment Form

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

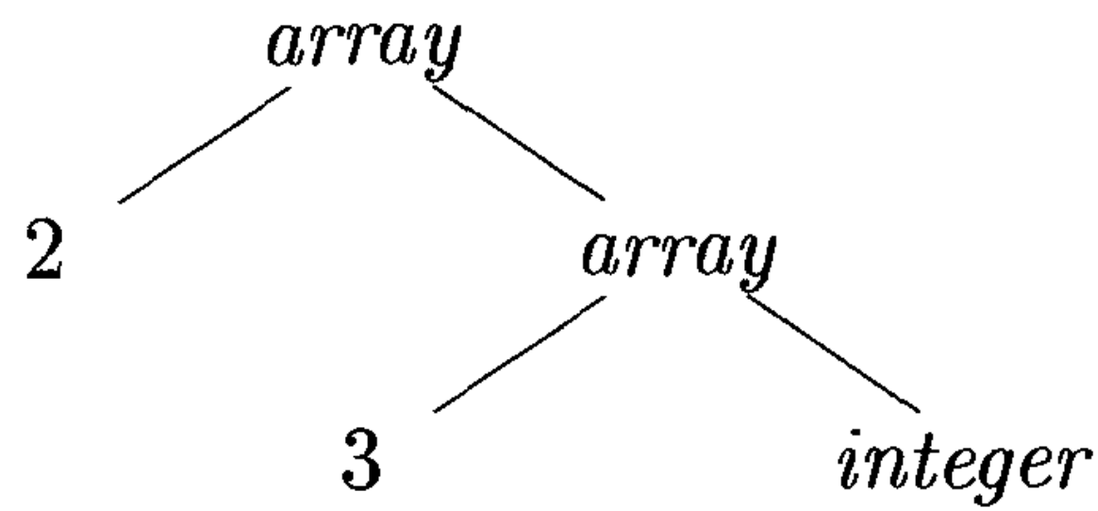
(lezioni I semestre)

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

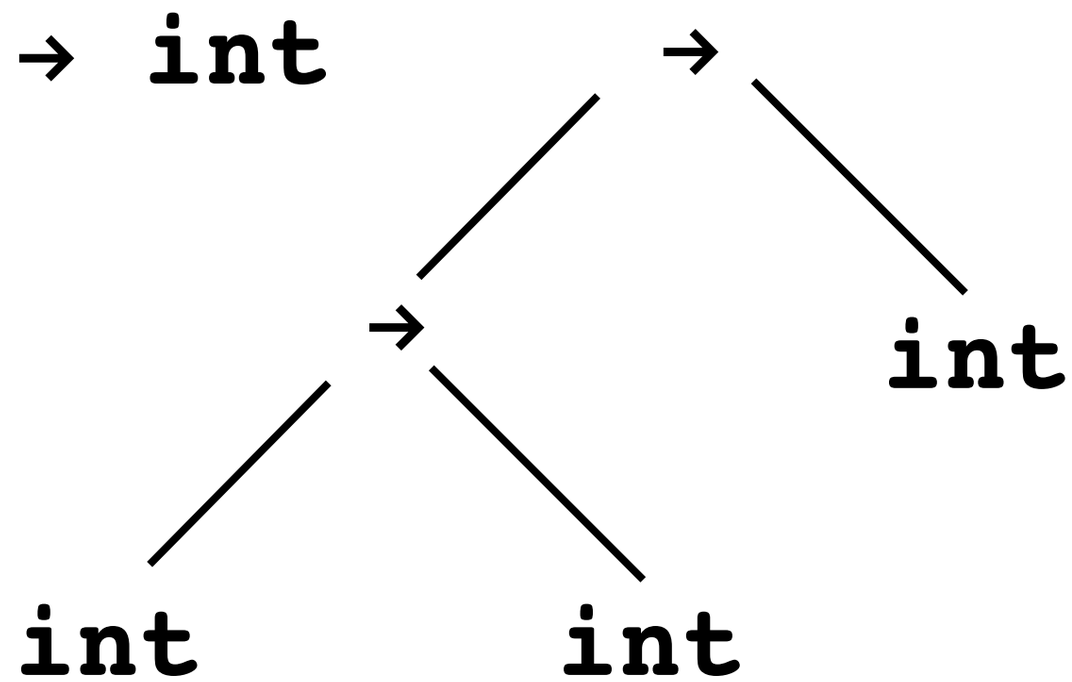
***Translation Applications**.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of 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:** $s \rightarrow t$;
- **product type:** $s \times t$
- We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$.
- 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 is true:

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

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

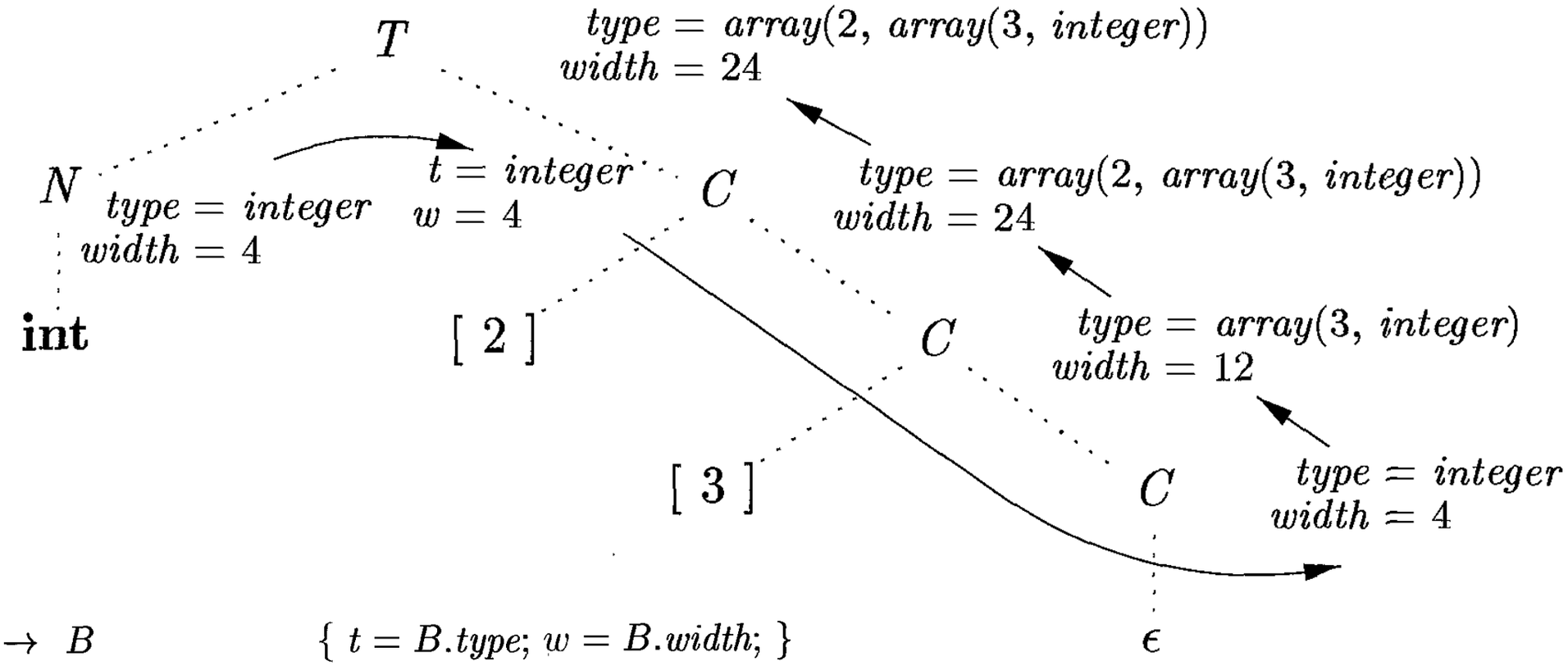
Storage Layout

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

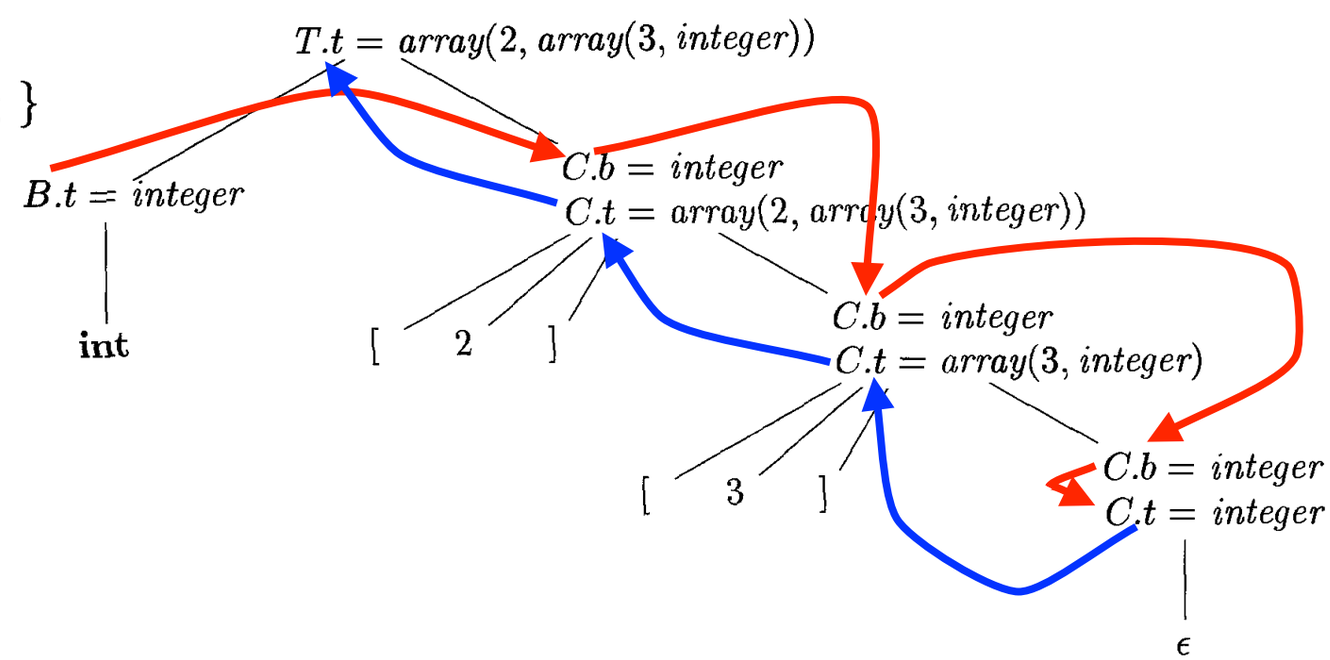
The width of an array is obtained by multiplying the width of an element by the number of elements in the array.

SDD

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



$T \rightarrow B$		$\{\ t = B.type;\ w = B.width;\ \}$
$\quad\ \ C$		
$B \rightarrow$	**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	$\{\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$



Sequences of Declarations

$$
\begin{array}{lll}
P & \rightarrow & \{\ \mathit{offset} = 0;\ \} \\
 & & D \\
D & \rightarrow & T\ \mathbf{id}\ ; \quad \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
 & & \quad \mathit{offset} = \mathit{offset} + T.\mathit{width};\ \} \\
 & & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

Computing the relative addresses of declared names

$$
\begin{array}{lll}
P & \rightarrow & M\ D \\
M & \rightarrow & \epsilon \qquad \{\ \mathit{offset} = 0;\ \}
\end{array}
$$

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;
```

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

Translation of Expressions

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

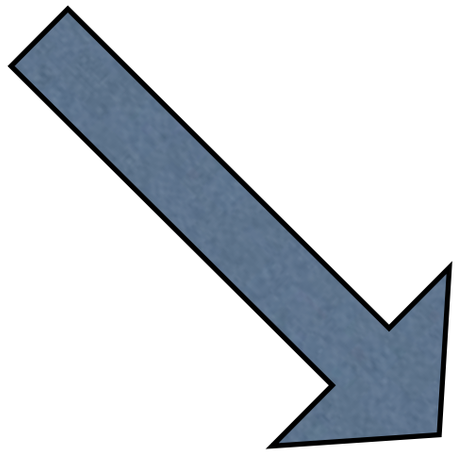
Three-address code for expressions

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
```

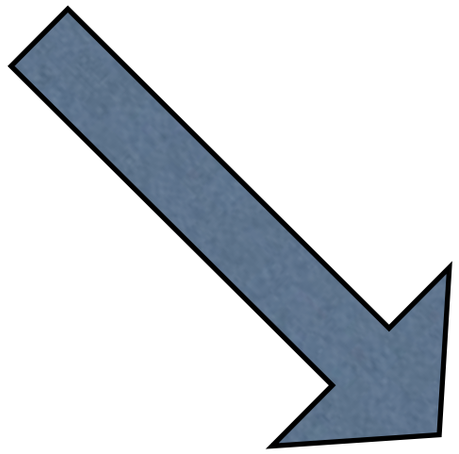


?

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

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

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



t_1 = minus c

t_2 = b + t_1

a = t_2

Incremental Translation

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

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

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

$$\mid\ -E_1 \quad \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \}$$

$$\mid\ (\ E_1\ ) \quad \{\ E.addr = E_1.addr;\ \}$$

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

The approach can also be used to build a syntax tree.

E.g. the semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$E \rightarrow E_1 + E_2$ {E.addr = new Node('+', E_1.addr,E_2.addr); }

Here, attribute ***addr*** represents the address of a node rather than a variable or constant.

Addressing Array Elements

1-D array: A[i].

Assumptions:

- lower bound in address = **low**
- element data width = **w**
- starting address = **start_addr**

Address for A[i]

- **=start addr+(i–low)∗w**
- **=i∗w+(start addr – low∗w)**
- The value, called base, **(start addr – low ∗ w)** can be computed at compile time, and then stored at the symbol table.

Addressing 2-D array elements

2-D array A[i1,i2].

Row major: the preferred mapping method.

- A[1,1],A[1,2],A[1,3],A[2,1],A[2,2],...
- A[i] means the ith row.
- Advantage: A[i,j] = A[i][j].

Column major:

- A[1,1],A[2,1],A[1,2],A[2,2],A[1,3],...

Row major:

Address for A[i1,i2]

=start addr+((i1 –low1)$*$n2 +(i2 –low2))$*$w

=(i1 $*$n2 +i2)$*$w + (start addr–low1 $*$n2 $*$w–low2 $*$w)

- **n2** is the number of elements in a row.
- **low1** is the lower bound of the first coordinate.
- **low2** is the lower bound of the second coordinate.

The value, called base, **(start addr–low1 $*$n2 $*$w–low2 $*$w)** can be computed at compiler time, and then stored at the symbol table.

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

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

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.

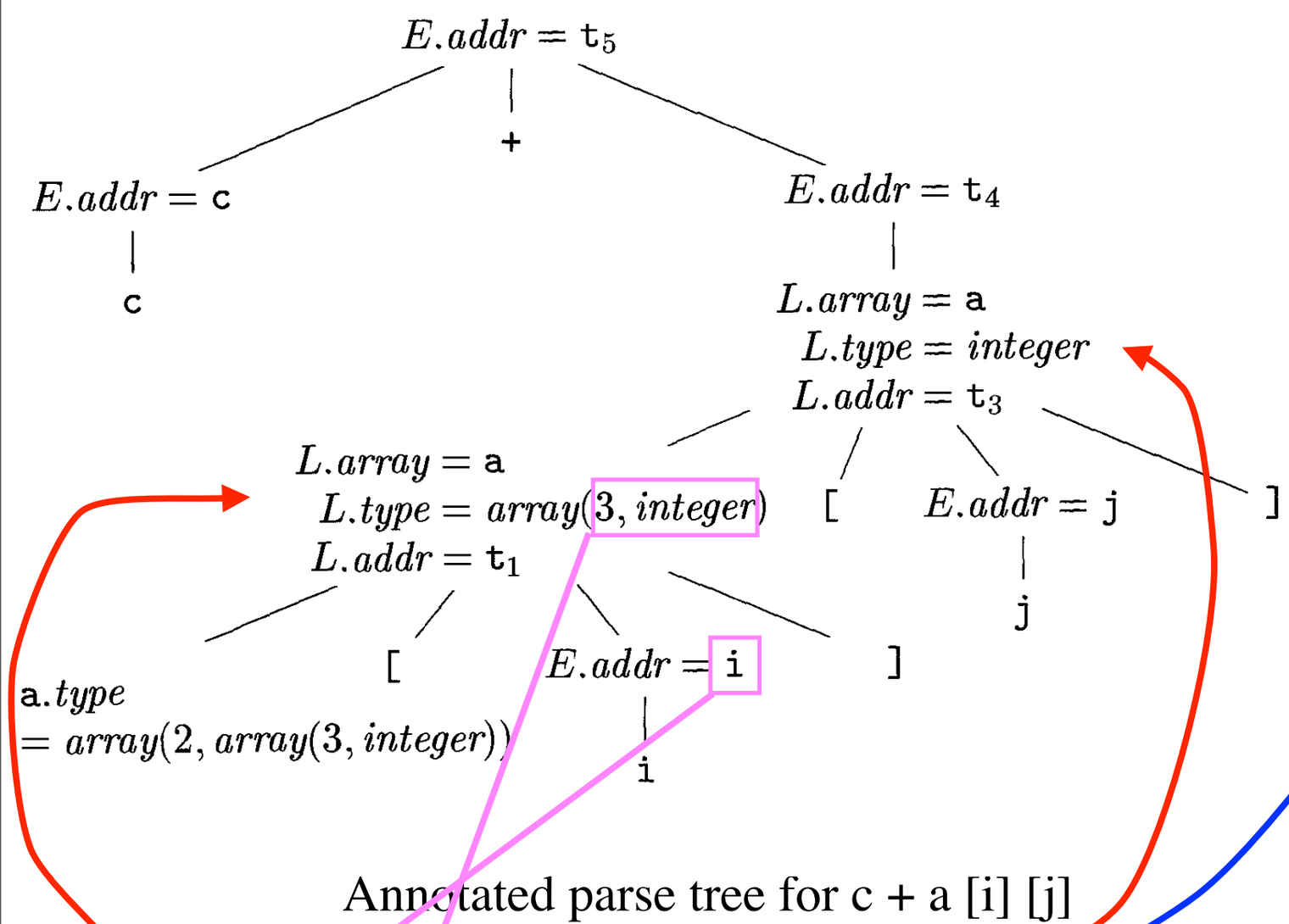
Three-address code for expression

`c + a[i][j]`

?

supponendo di aver dichiarato

`int[2][3]a;`



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

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

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

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

TYPE CHECKiNG

Type synthesis

Type inference

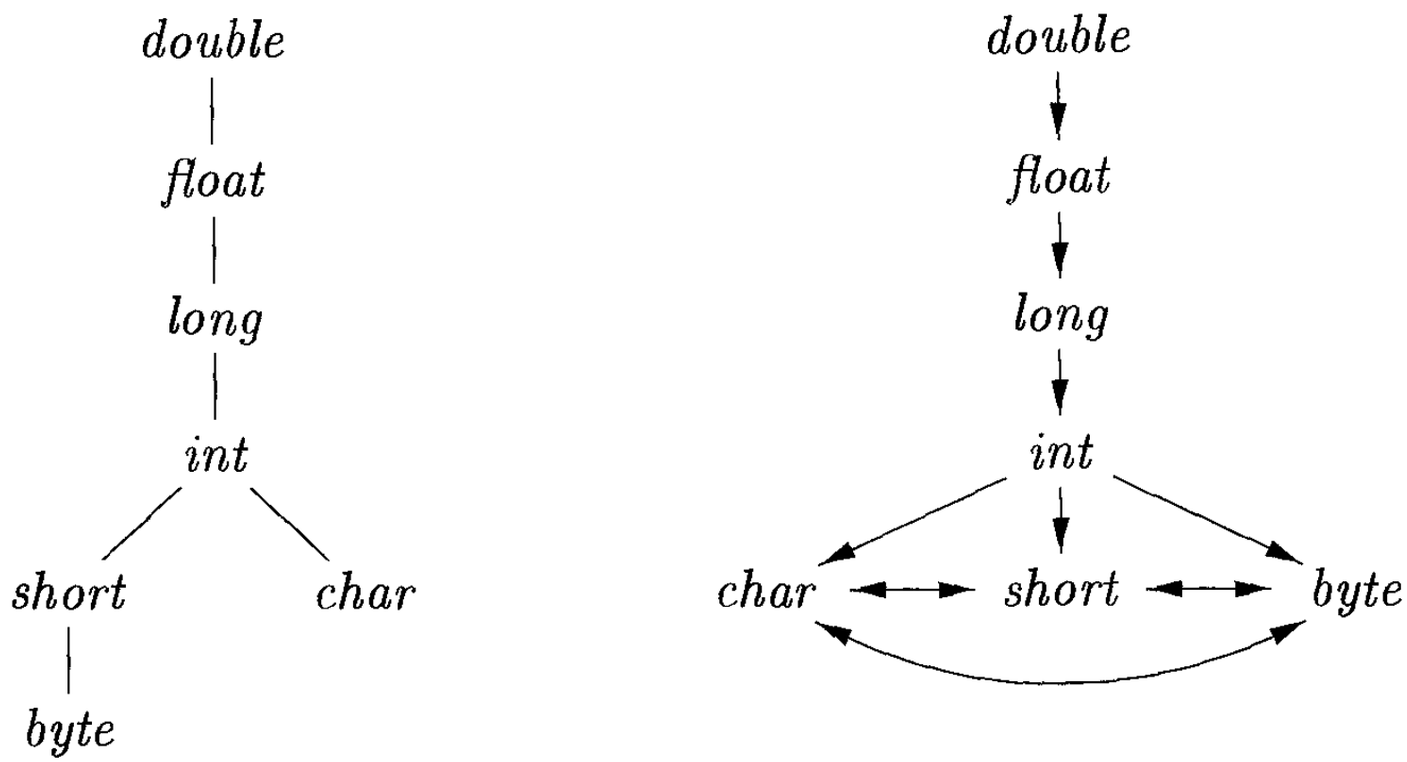
if *f* has type s → *t* **and** *x* has type s
then expression *f (x)* has type *t*

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 $s_i \rightarrow t_i$, 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

(corso I semestre)

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.

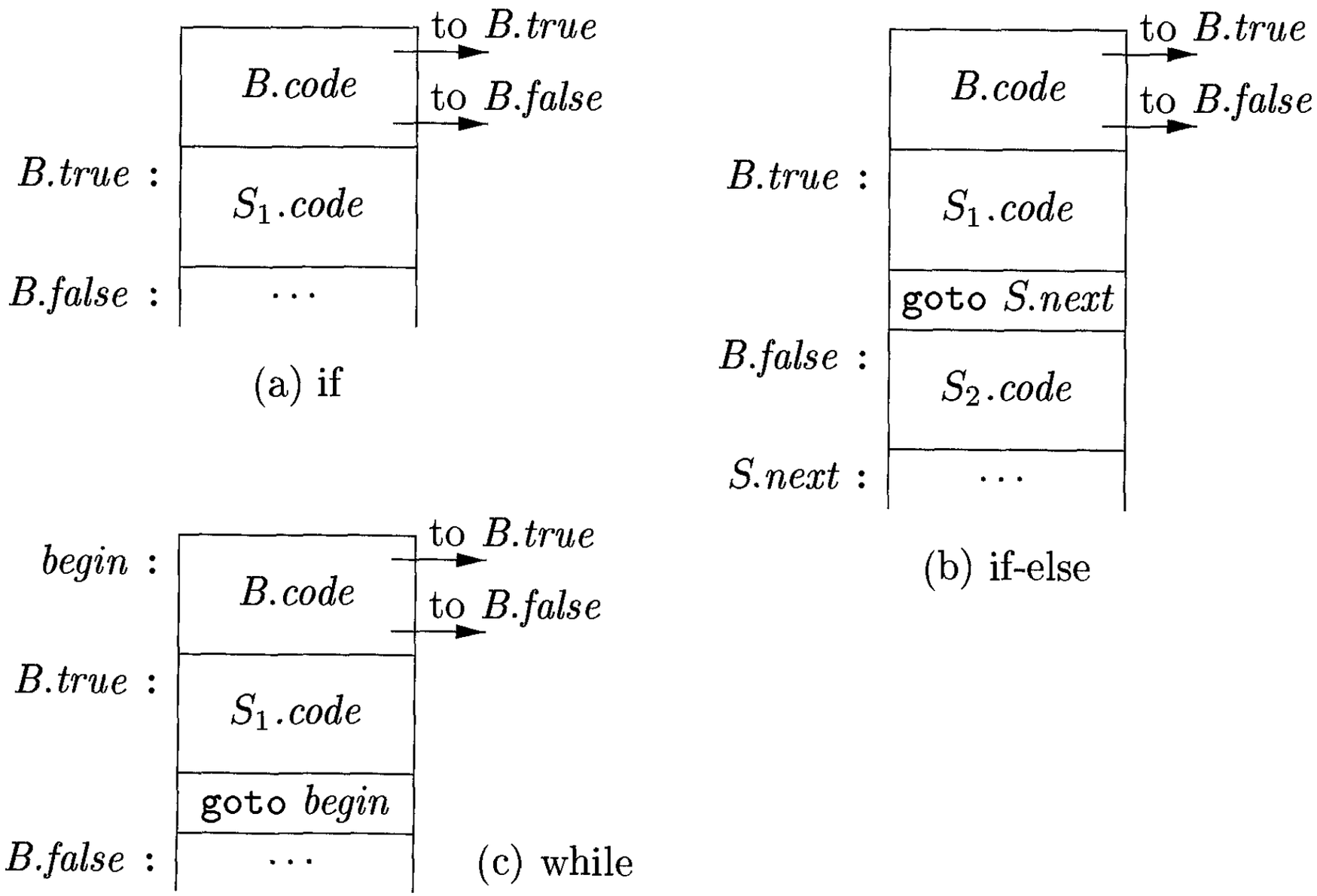
- E1 || E2 need not evaluate E2 if E1 is known to be true.
- E1 && E2 need not evaluate E2 if E1 is known to be false.

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

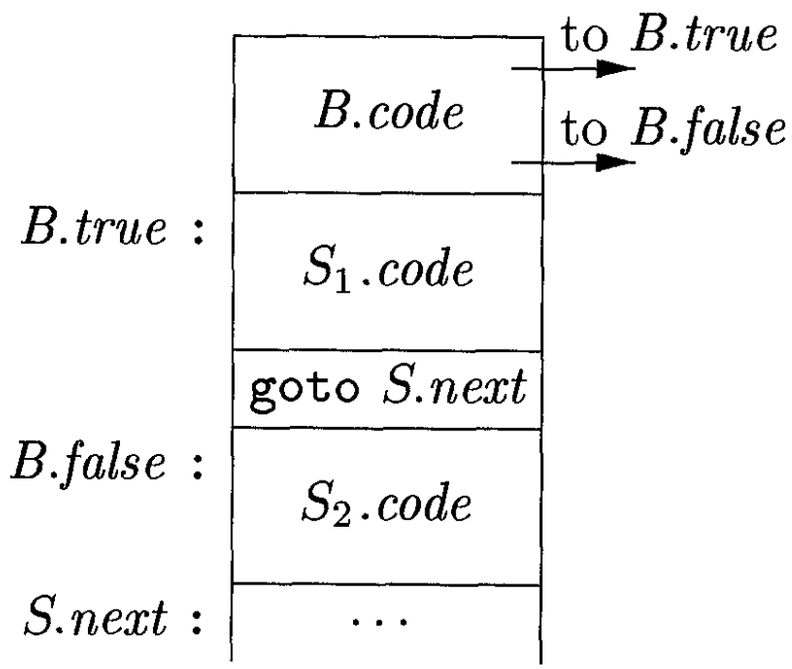
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0;
L1:     ...
```

Flow-of-Control Statements

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

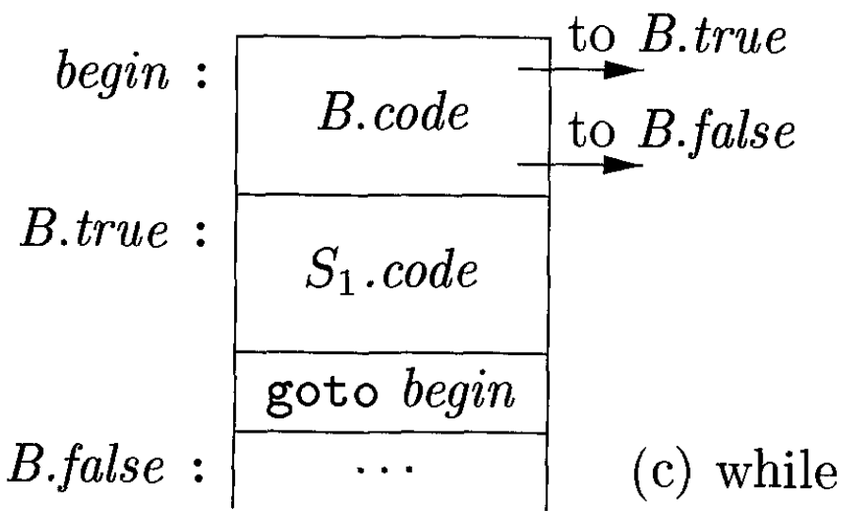


PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$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



$$
\begin{array}{l|l}
S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2 & B.true = newlabel() \\
& B.false = newlabel() \\
& S_1.next = S_2.next = S.next \\
& S.code = B.code \\
& \qquad || \; label(B.true) \; || \; S_1.code \\
& \qquad || \; gen('\texttt{goto}' \; S.next) \\
& \qquad || \; label(B.false) \; || \; S_2.code
\end{array}
$$

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



(c) while

$$S \rightarrow \textbf{while} \; (\; B \;) \; S_1$$

$$\begin{aligned}
&begin = newlabel() \\
&B.true = newlabel() \\
&B.false = S.next \\
&S_1.next = begin \\
&S.code = label(begin) \; || \; B.code \\
&\qquad || \; label(B.true) \; || \; S_1.code \\
&\qquad || \; gen('\texttt{goto}' \; begin)
\end{aligned}$$

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

If implemented literally, the semantic rules will generate lots of labels and may attach more than one label to a three-address instruction. The backpatching approach creates labels only when they are needed. Alternatively, unnecessary labels can be eliminated during a subsequent optimization phase.

Control-Flow Translation of Boolean Expressions

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

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

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

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

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

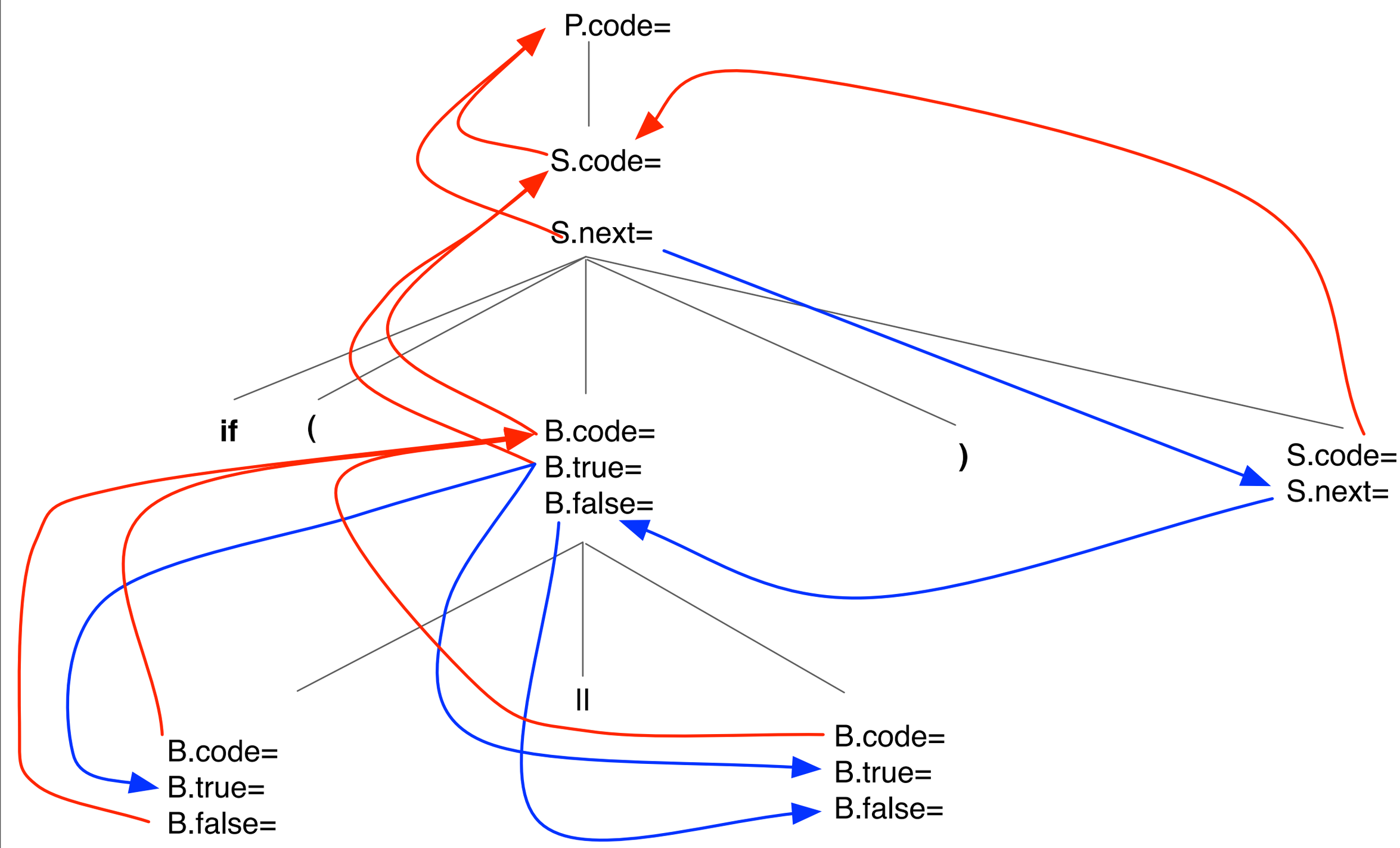
if B_1 is true, **then** ***B* itself is true** ➽ $B_1.true = B.true.$

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.

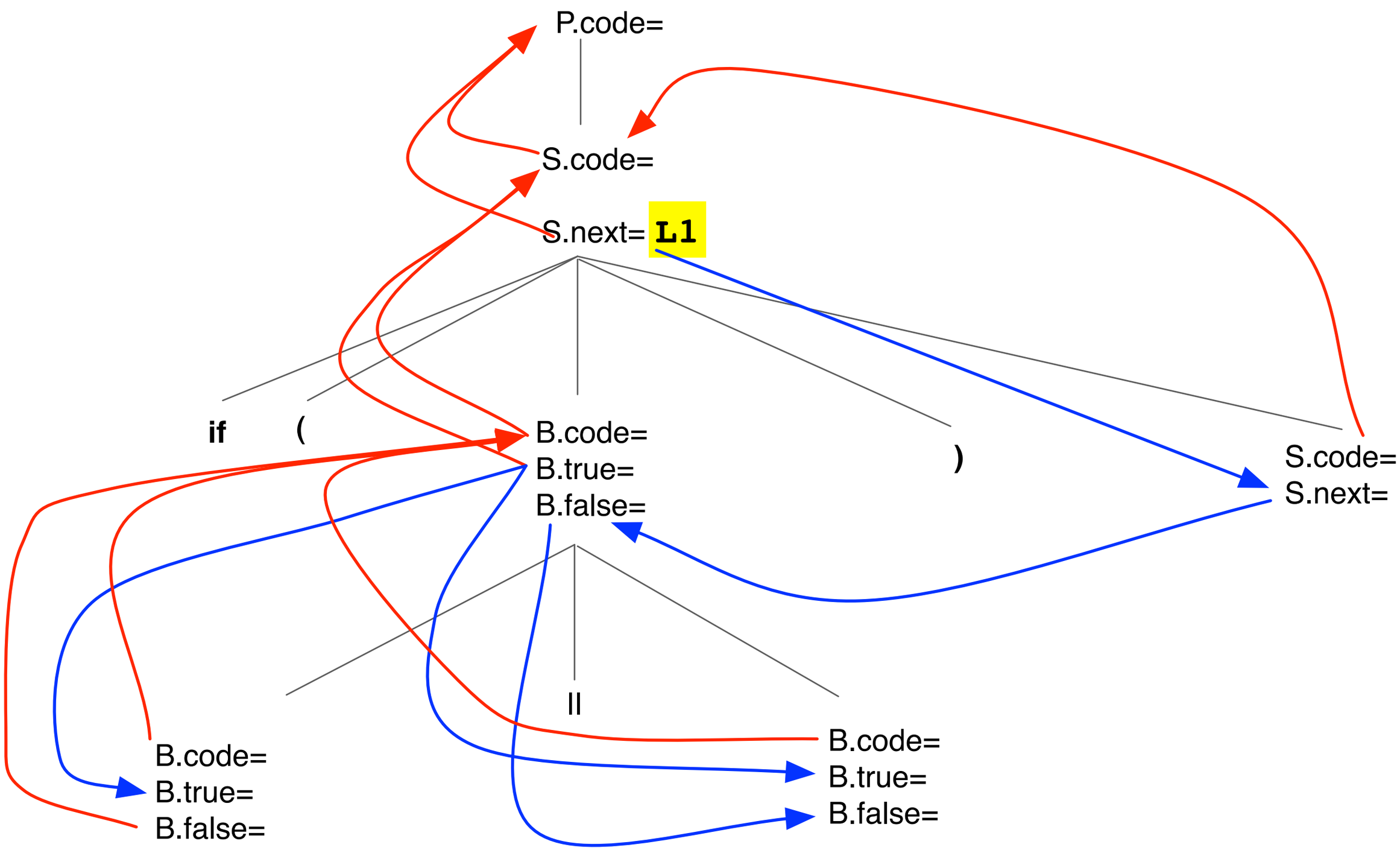
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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



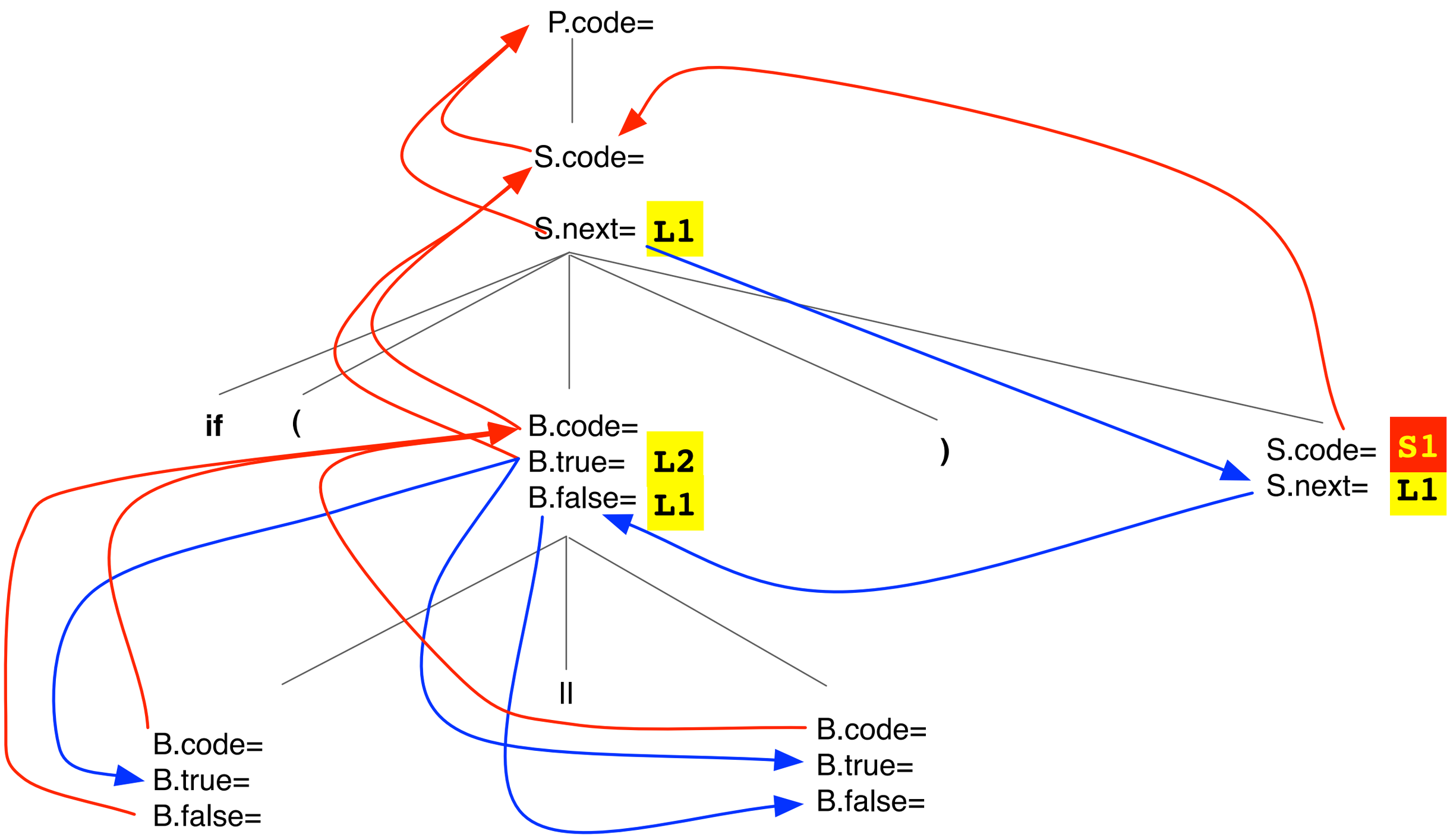
P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code

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



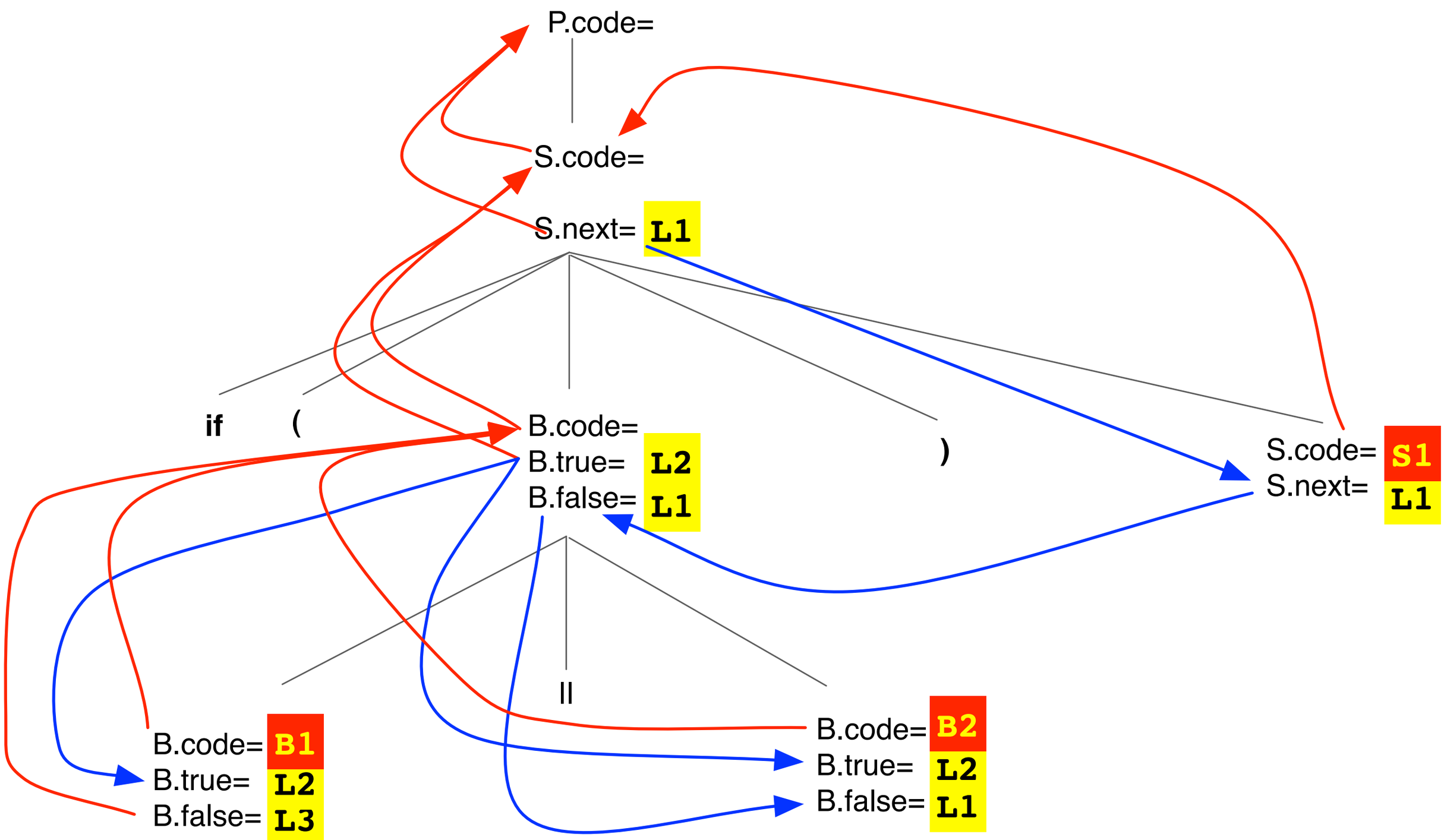
P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code

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



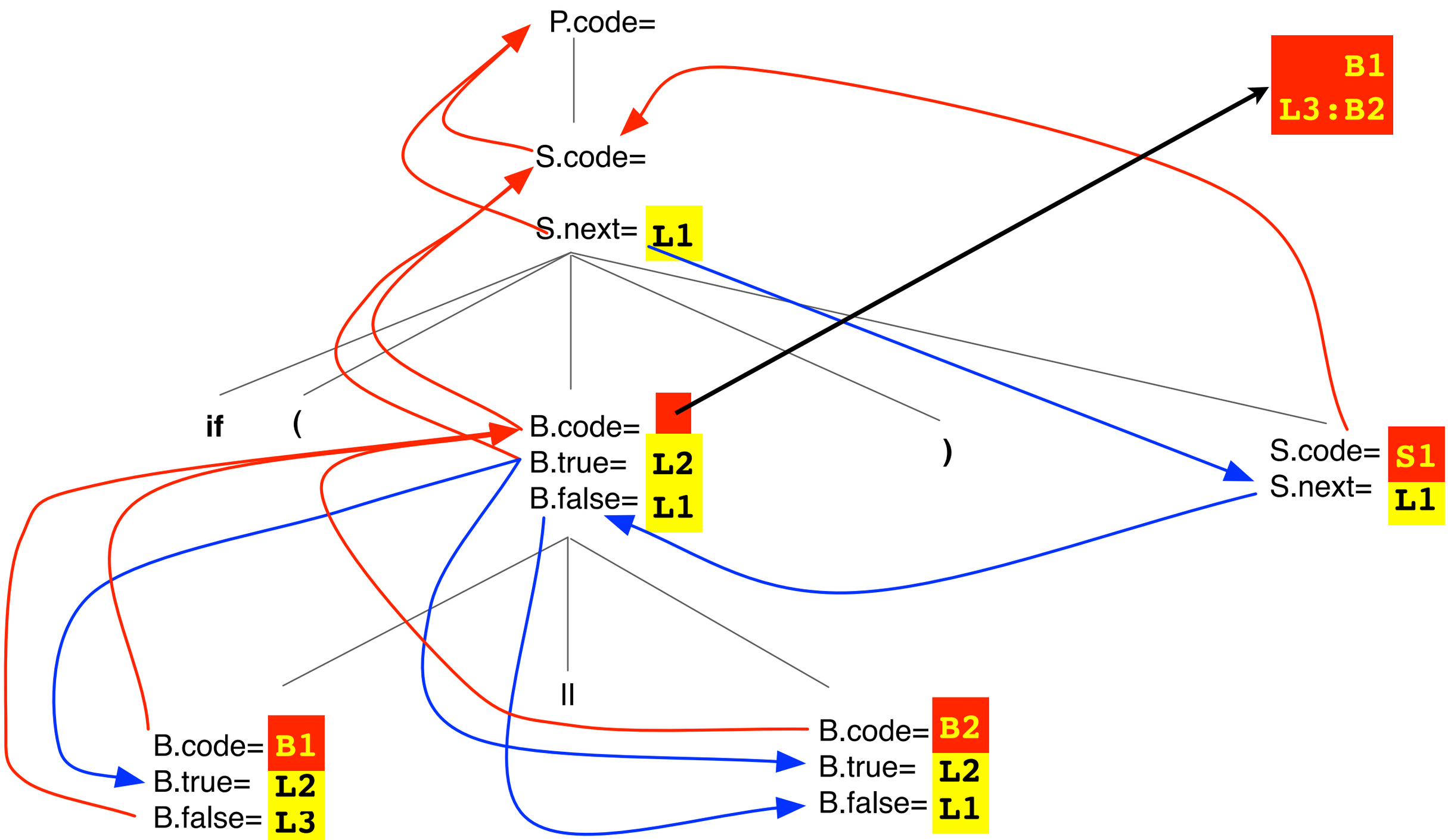
P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code

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



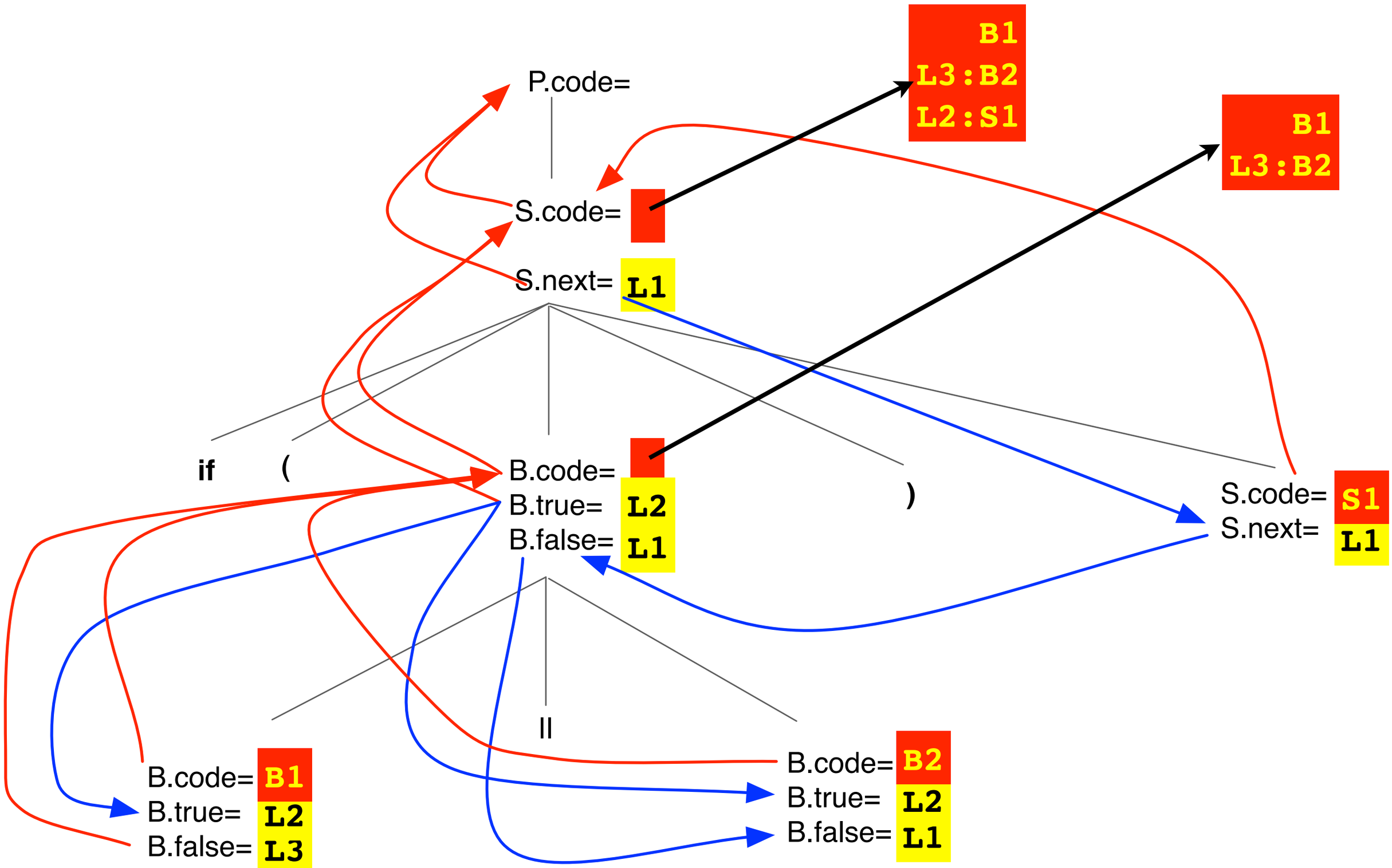
P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code

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

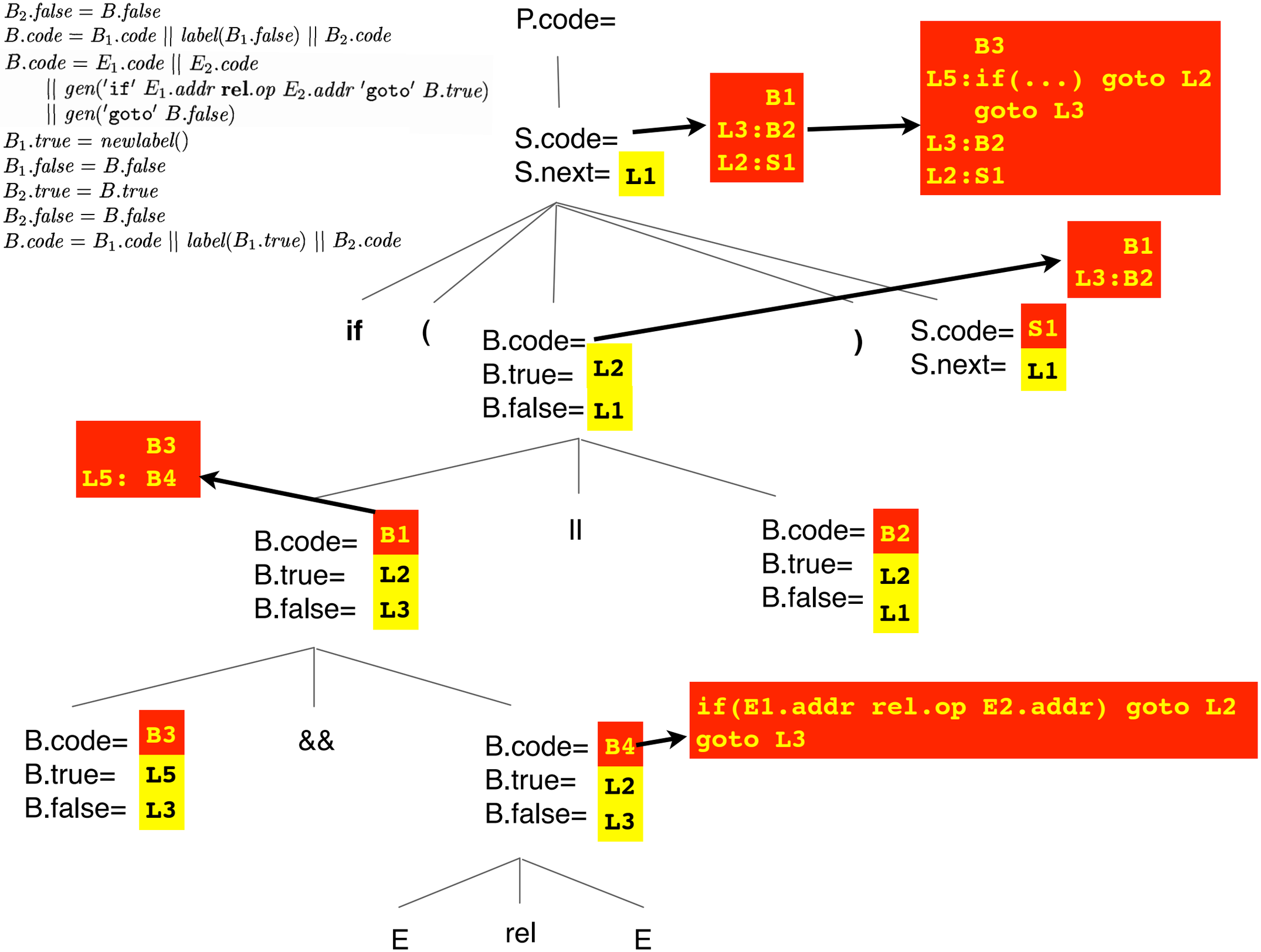


P → S	S.next = newlabel() P.code = S.code || label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code

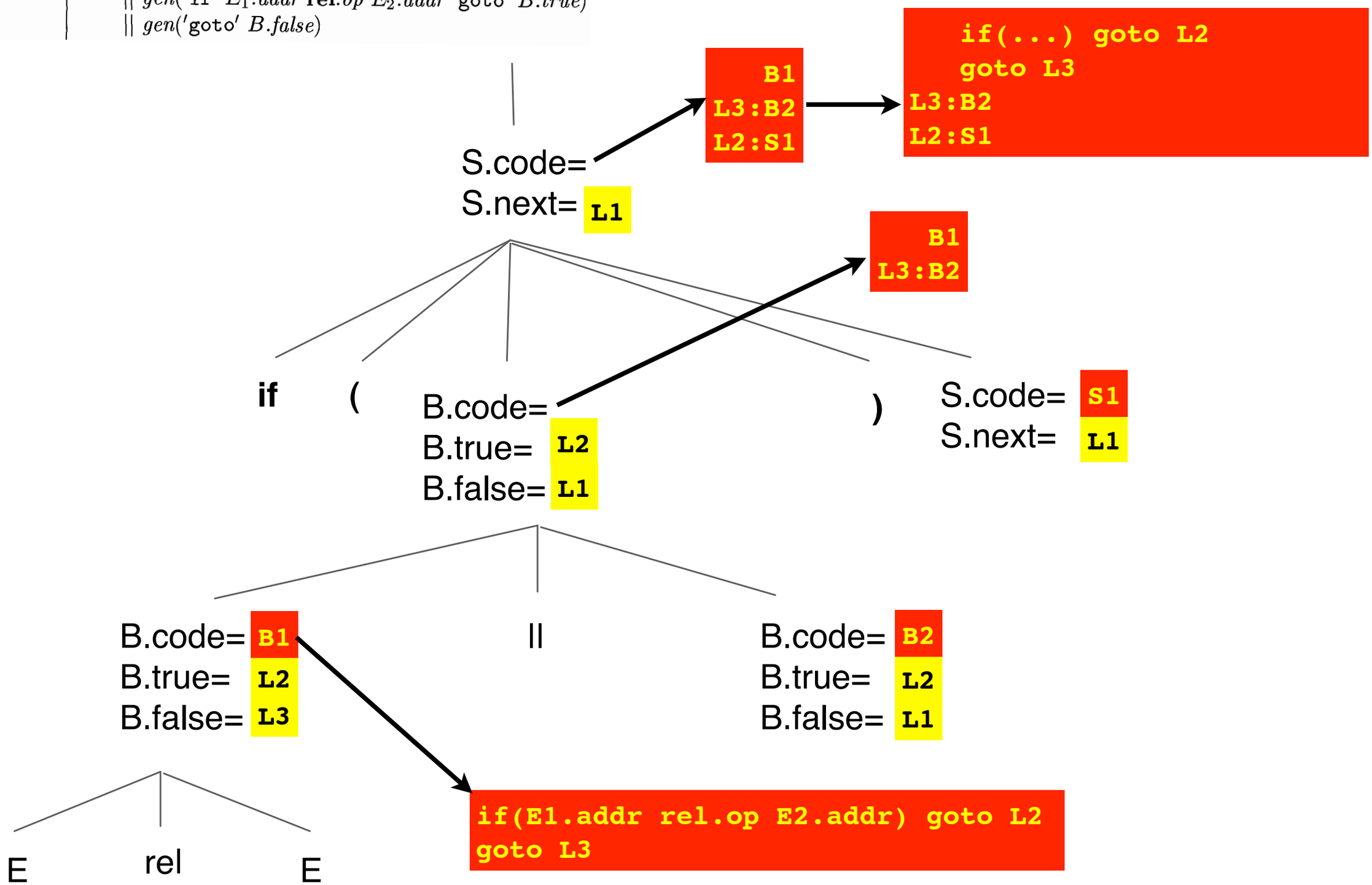
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$



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



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

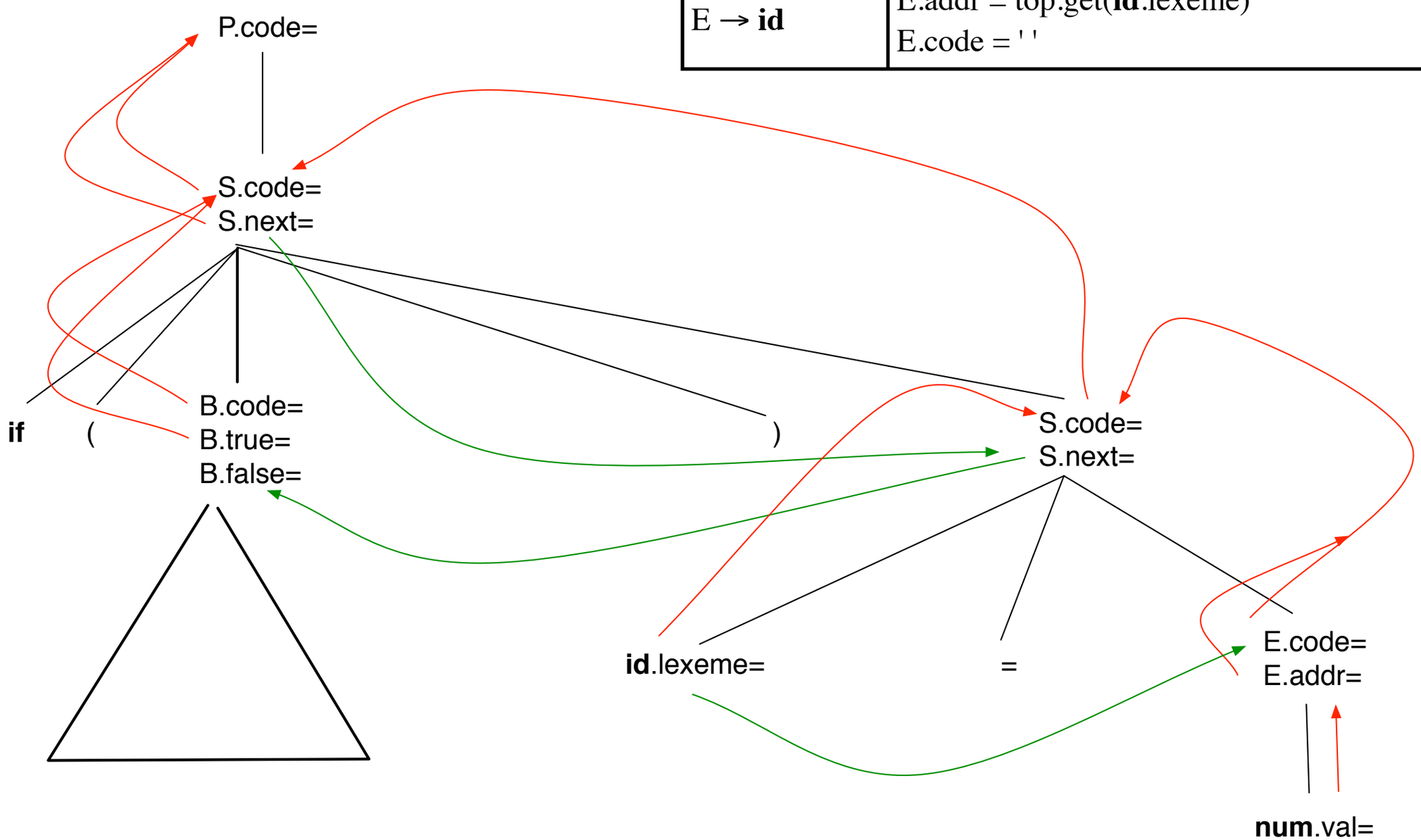


```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '

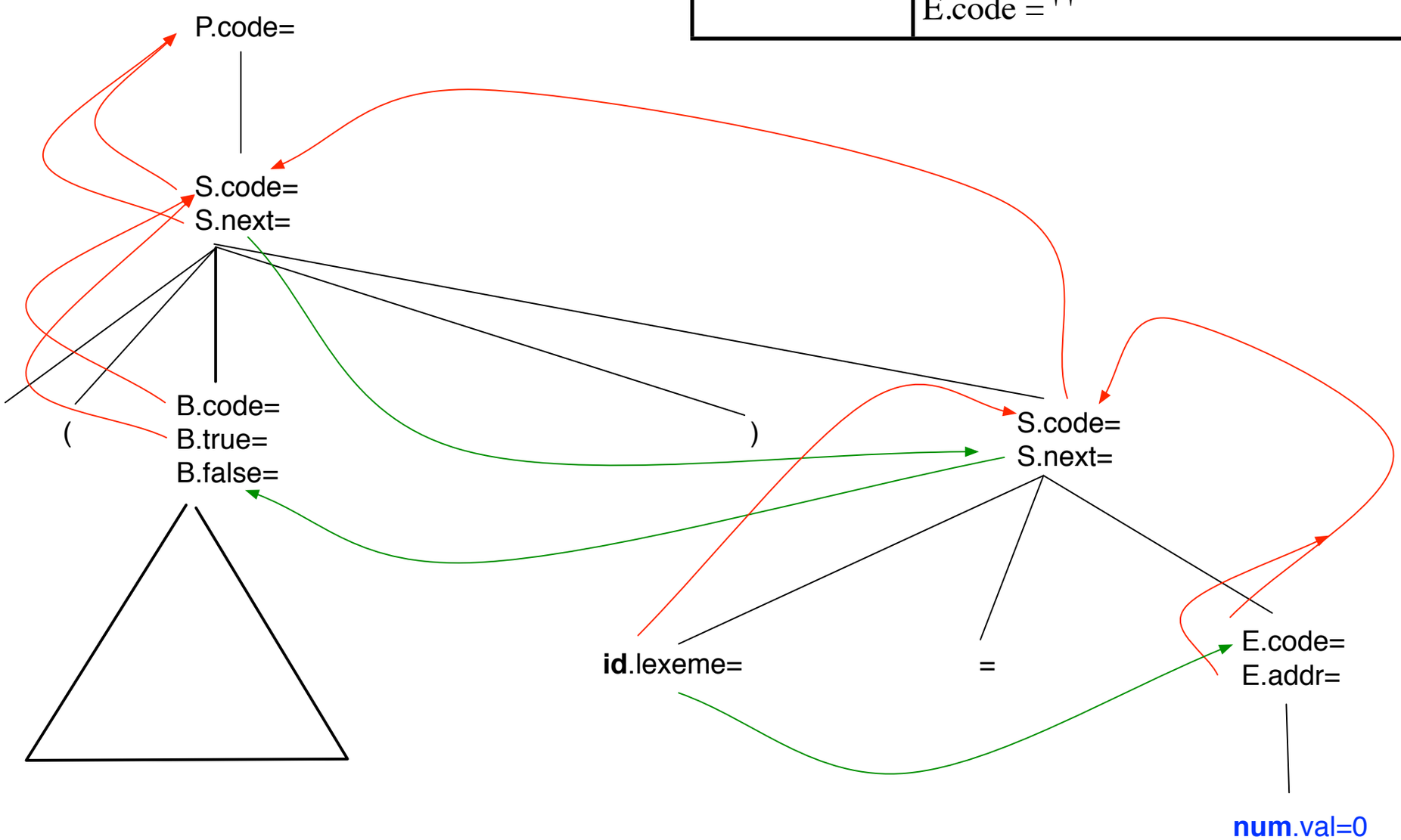
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



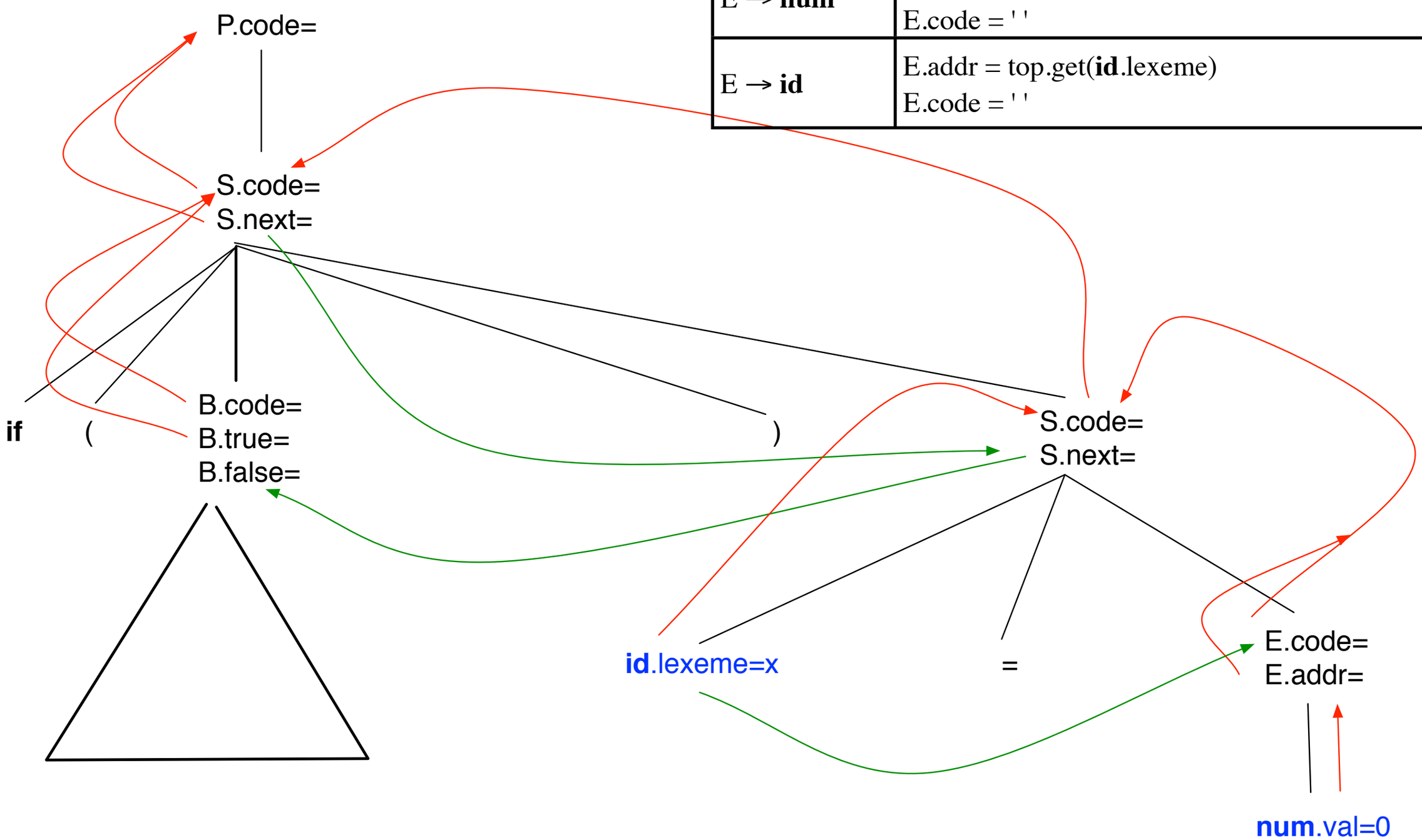
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



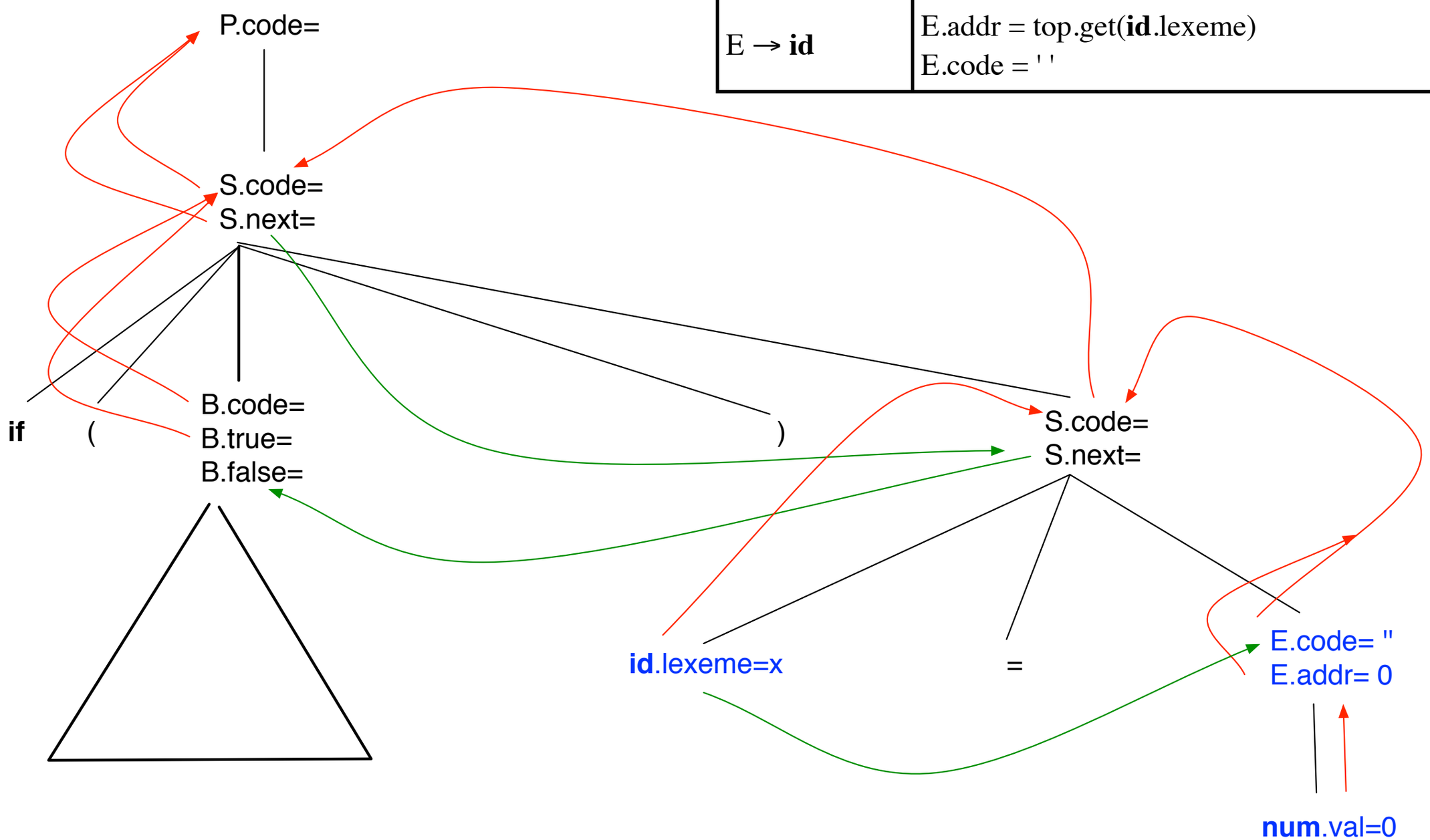
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



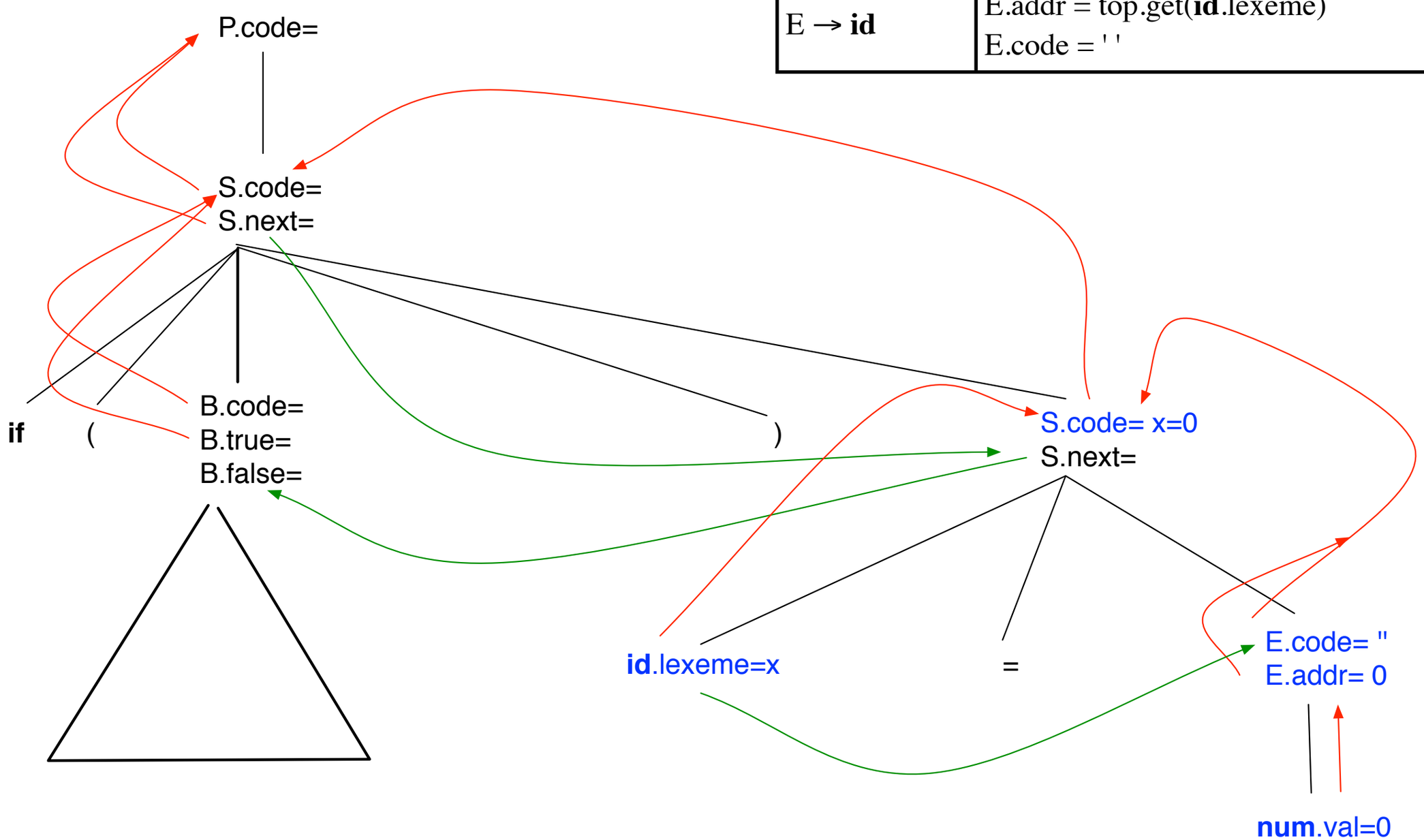
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



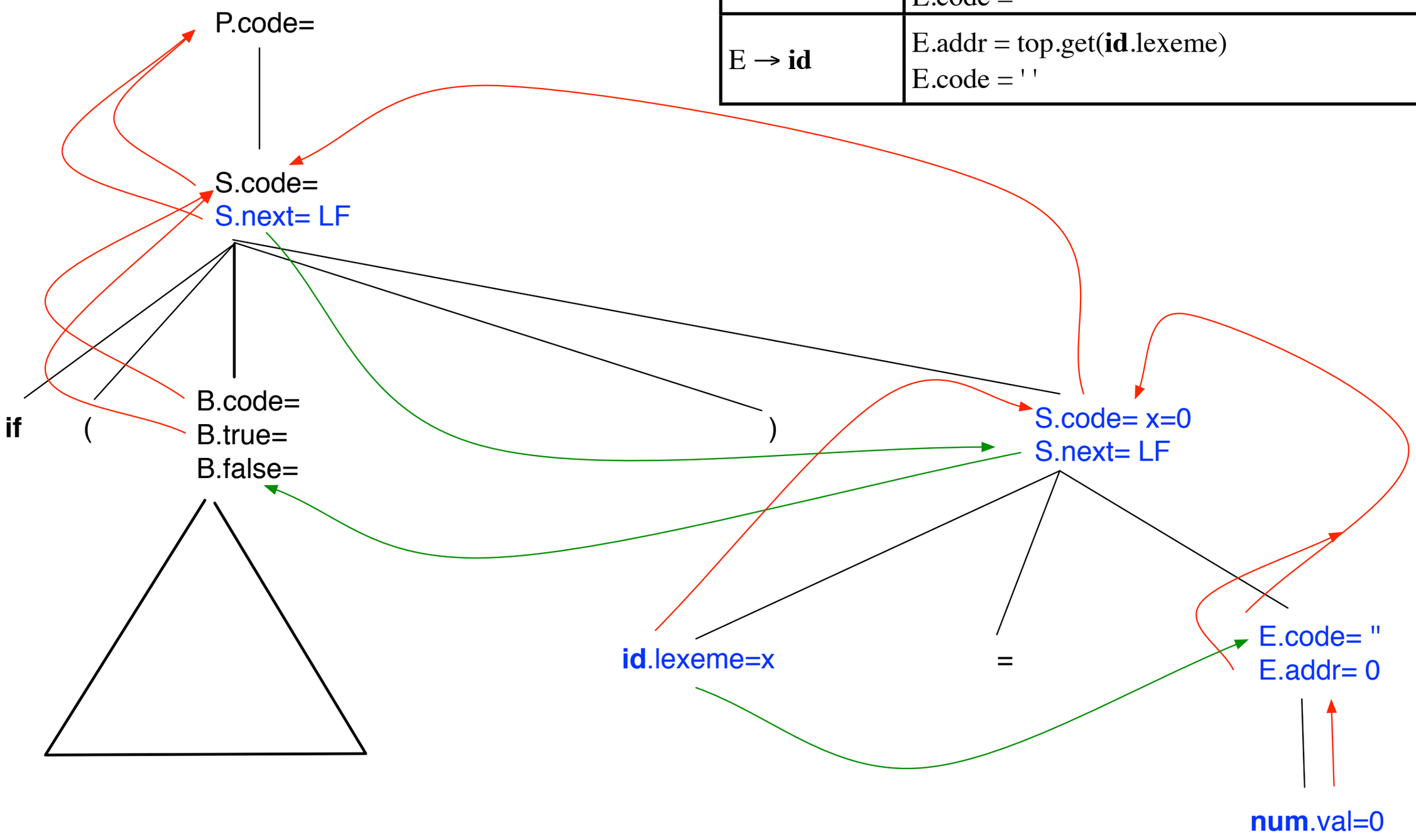
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



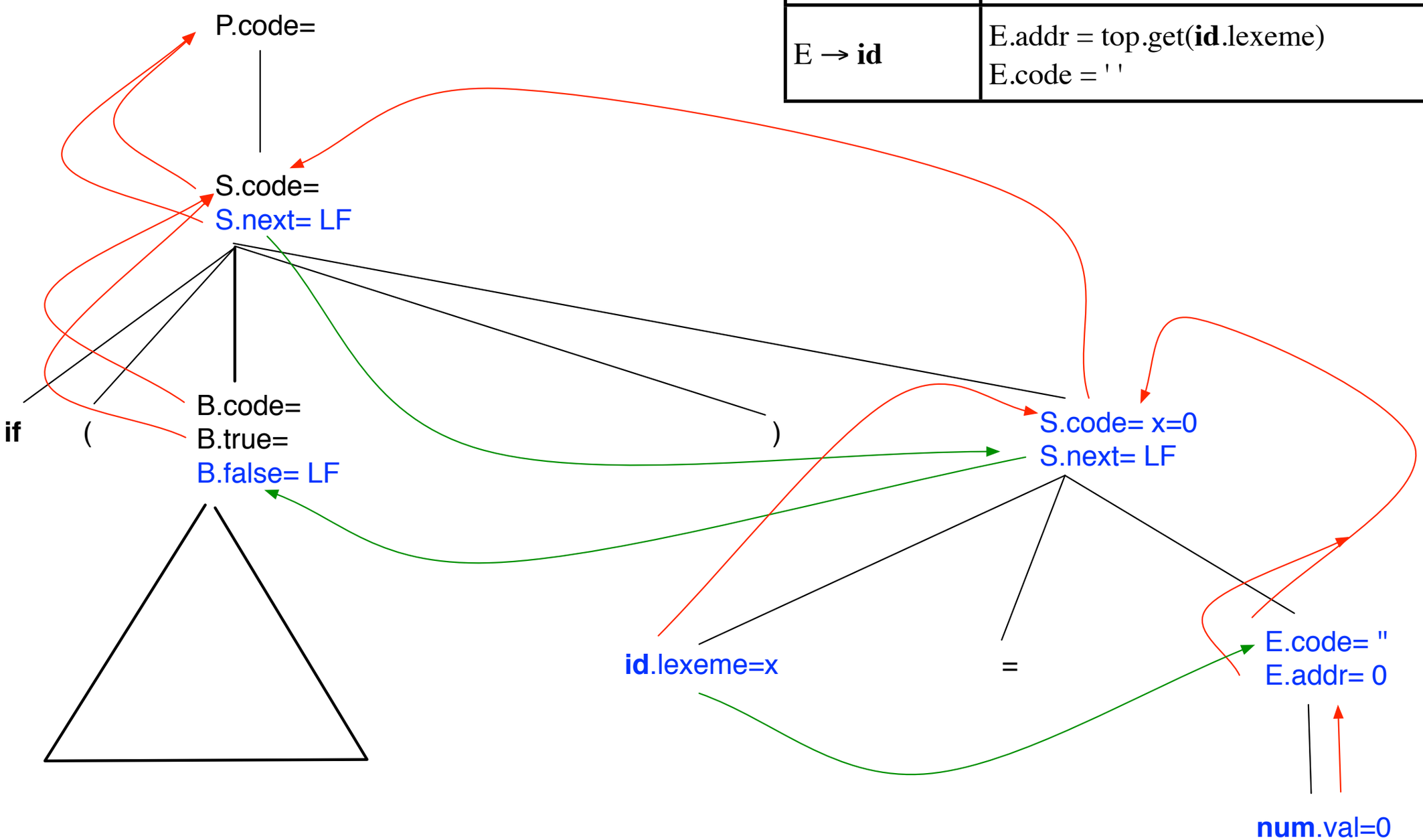
```
if(x<100 || x > 200 && x!=y) x=0;
```

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

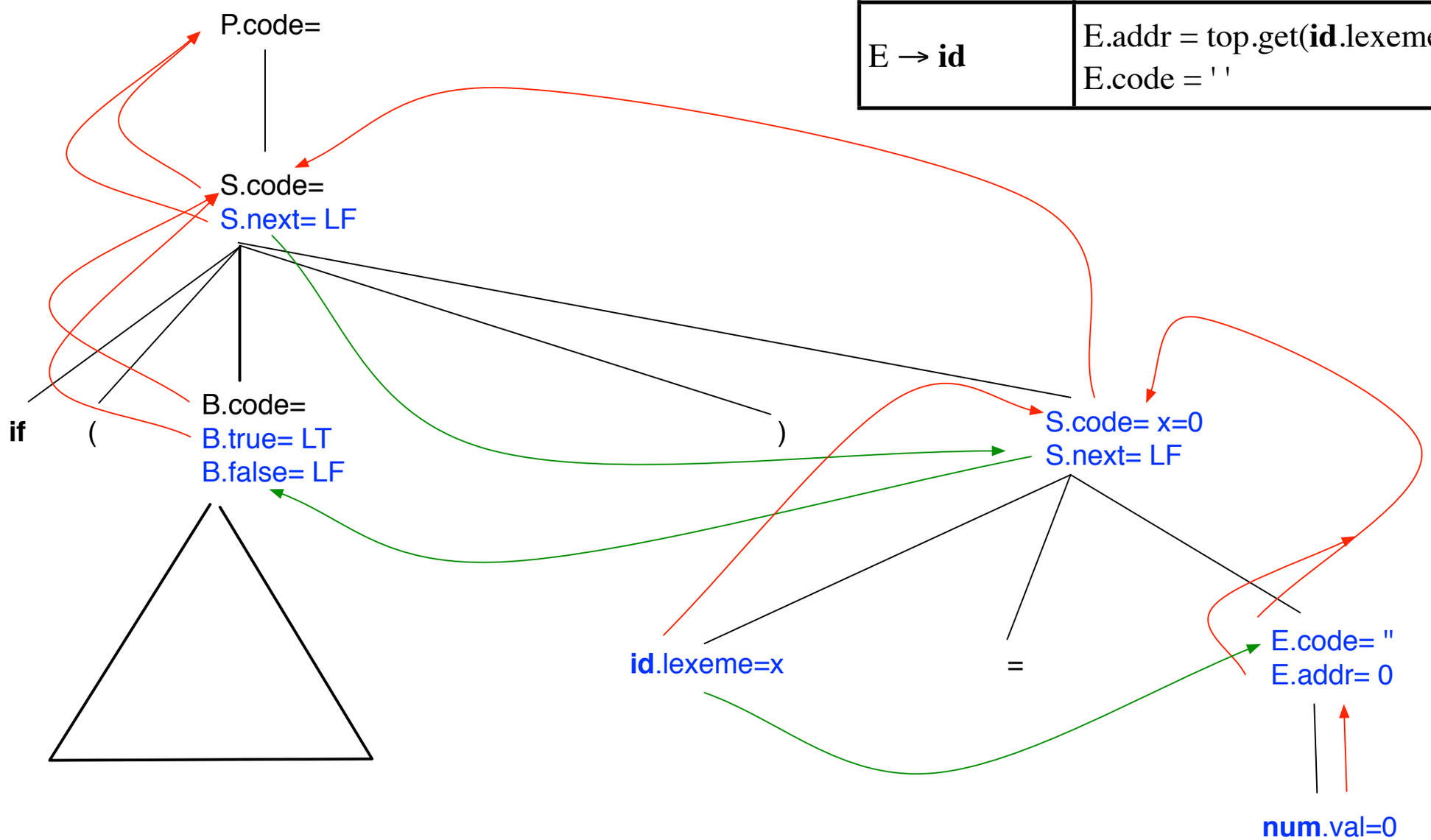


```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '

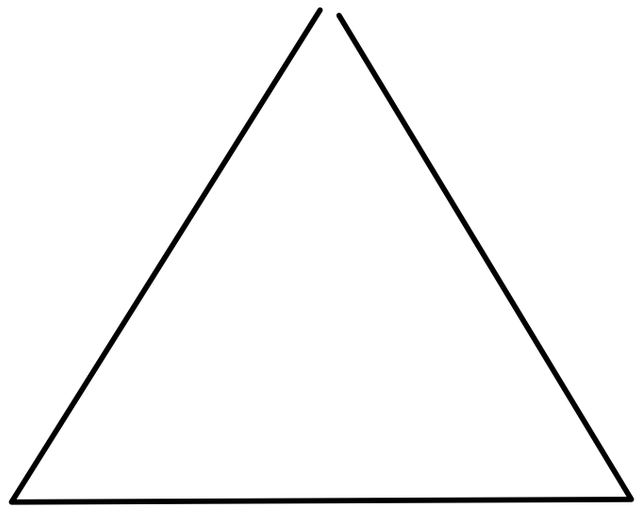


P → S	S.next = newlabel() P.code = S.code ‖ label(S.next)
S → **if** (B) S_1	B.true = newlabel() B.false = S_1.next = S.next S.code = B.code ‖ label(B.true) ‖ S_1.code
S → **id** = E;	S.code = E.code ‖ gen(top.get(**id**.lexeme)'=' E.addr
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr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



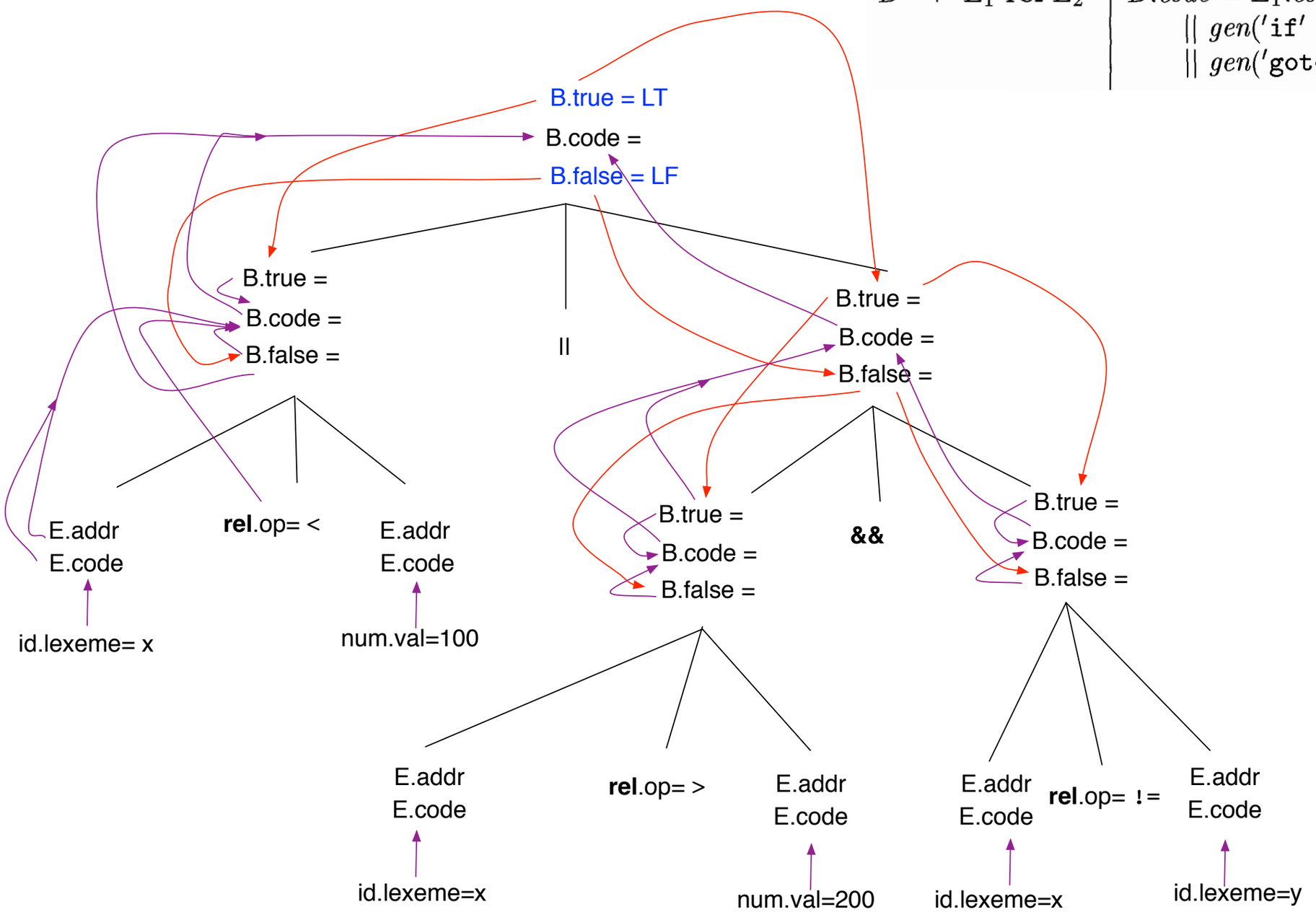
```
x<100 || x > 200 && x!=y
```

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



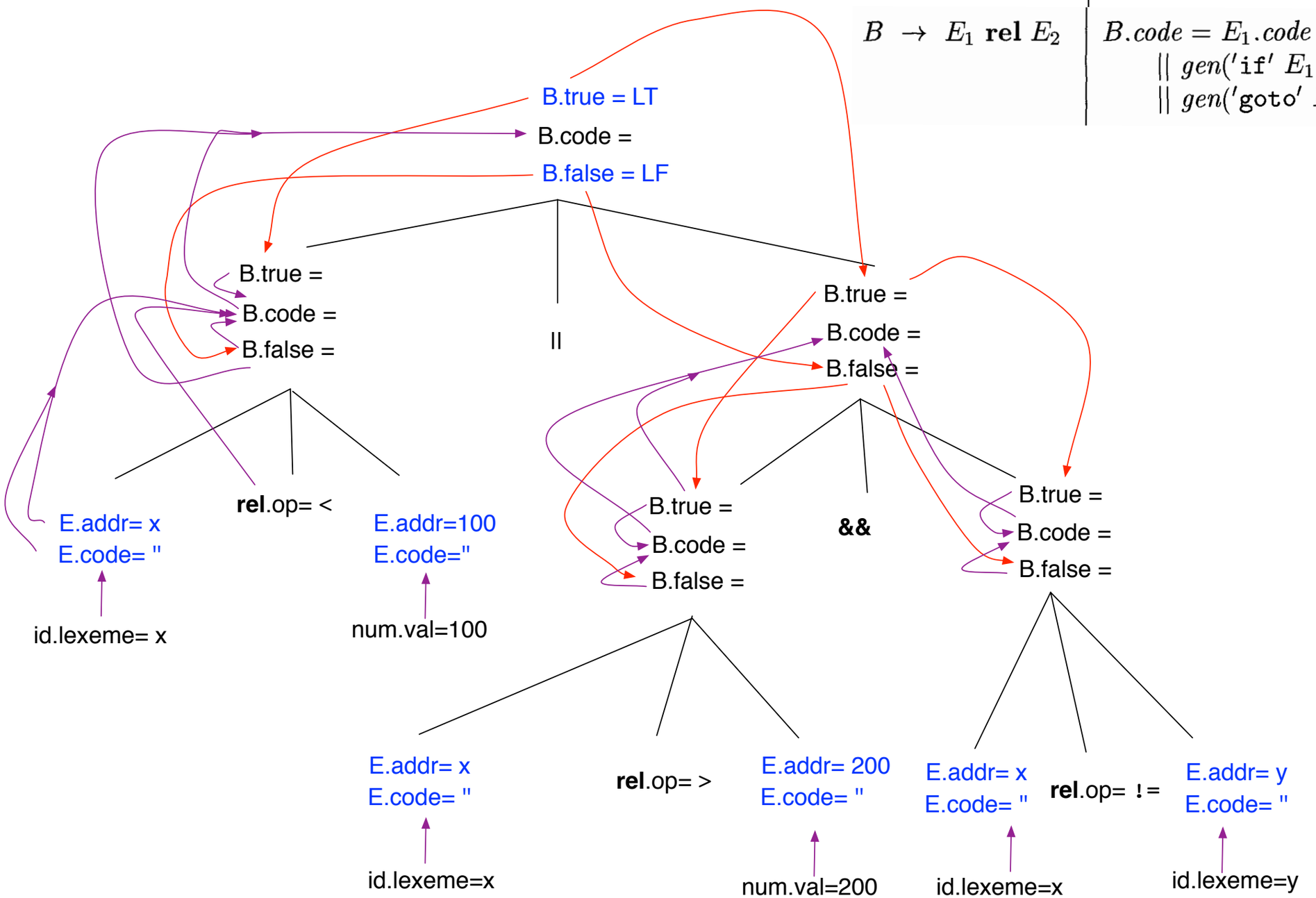
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



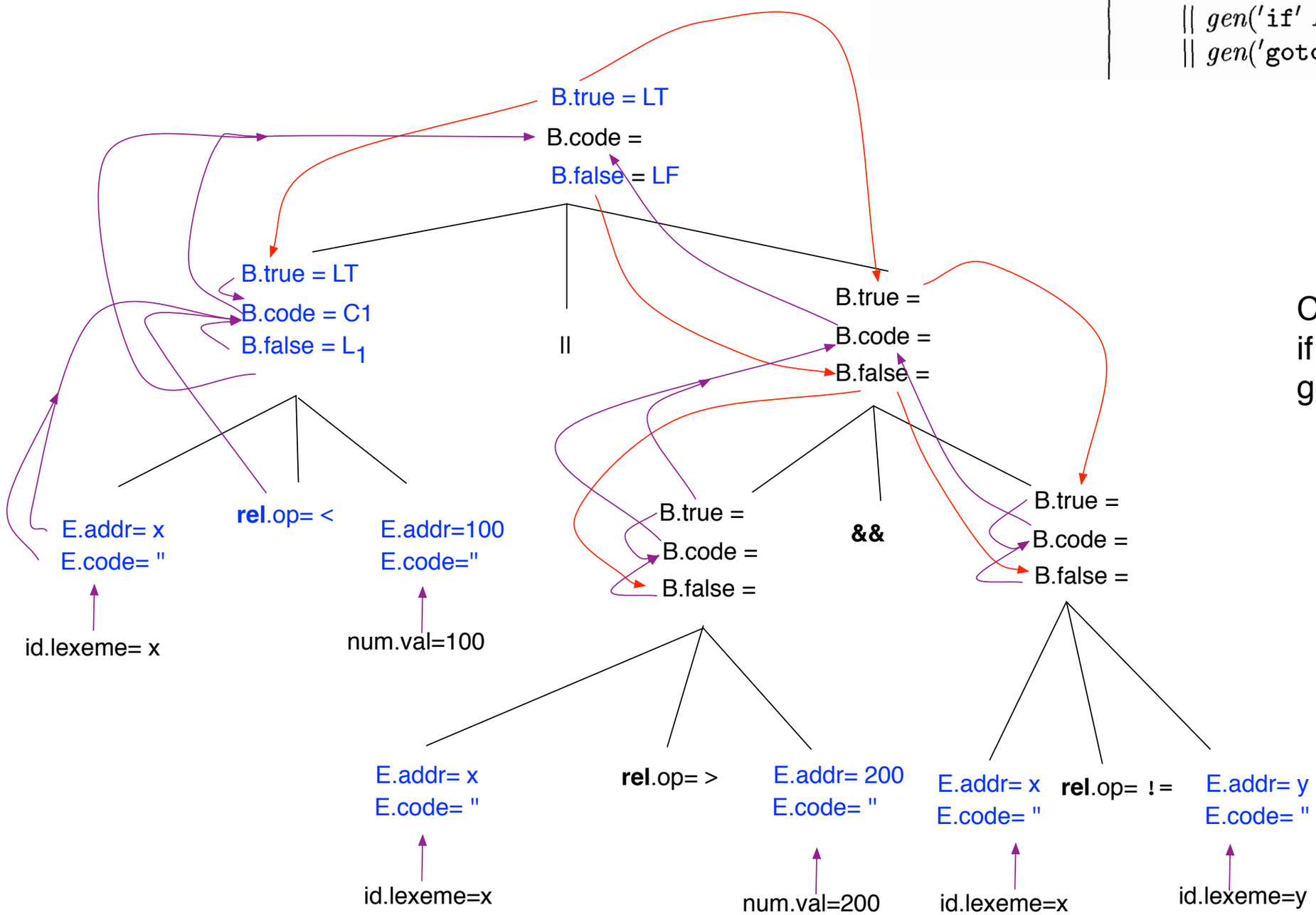
E → **num**	E.addr = **num**.val E.code = ' '
E → **id**	E.addr = top.get(**id**.lexeme) E.code = ' '

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



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

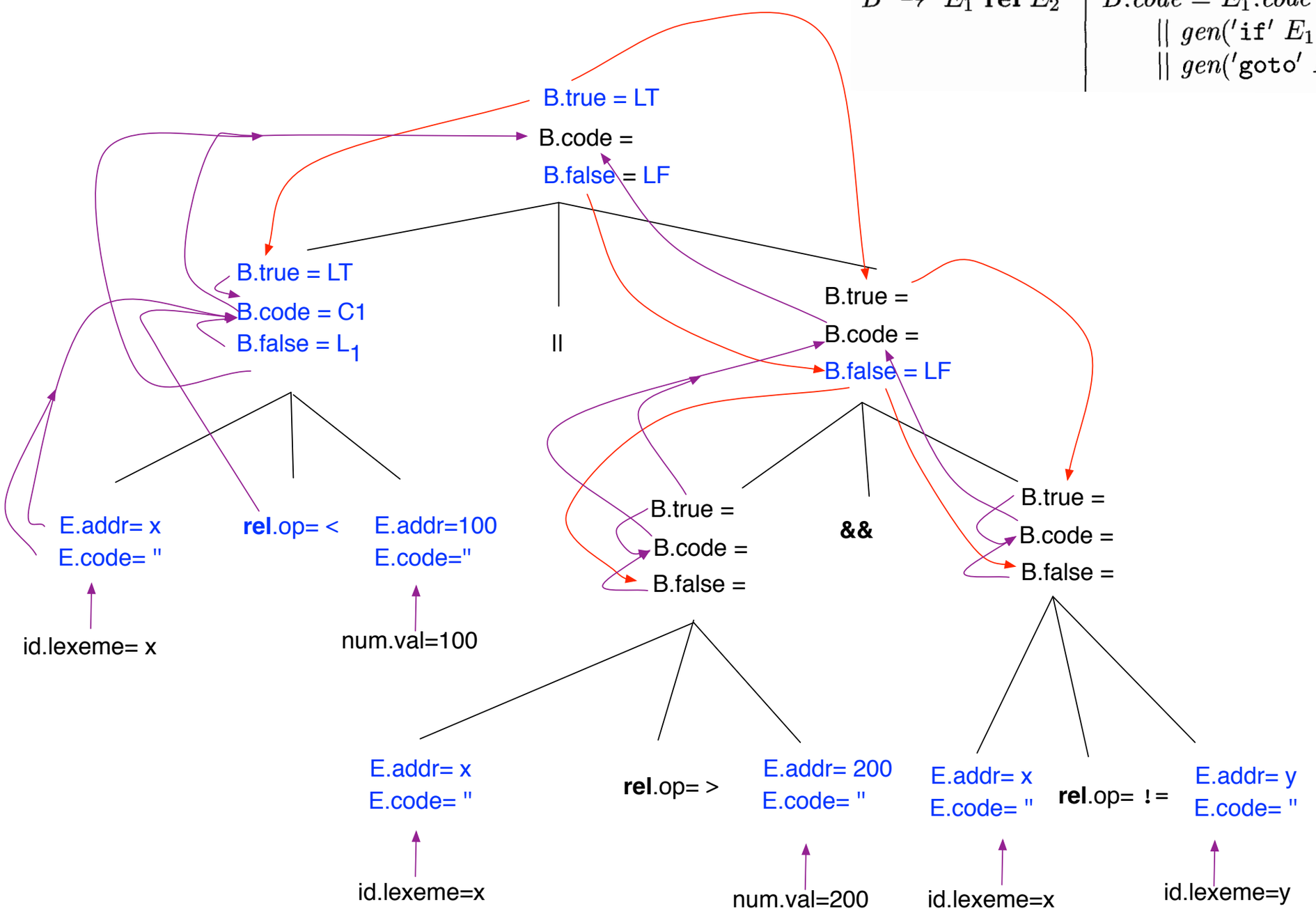
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



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 = ' '

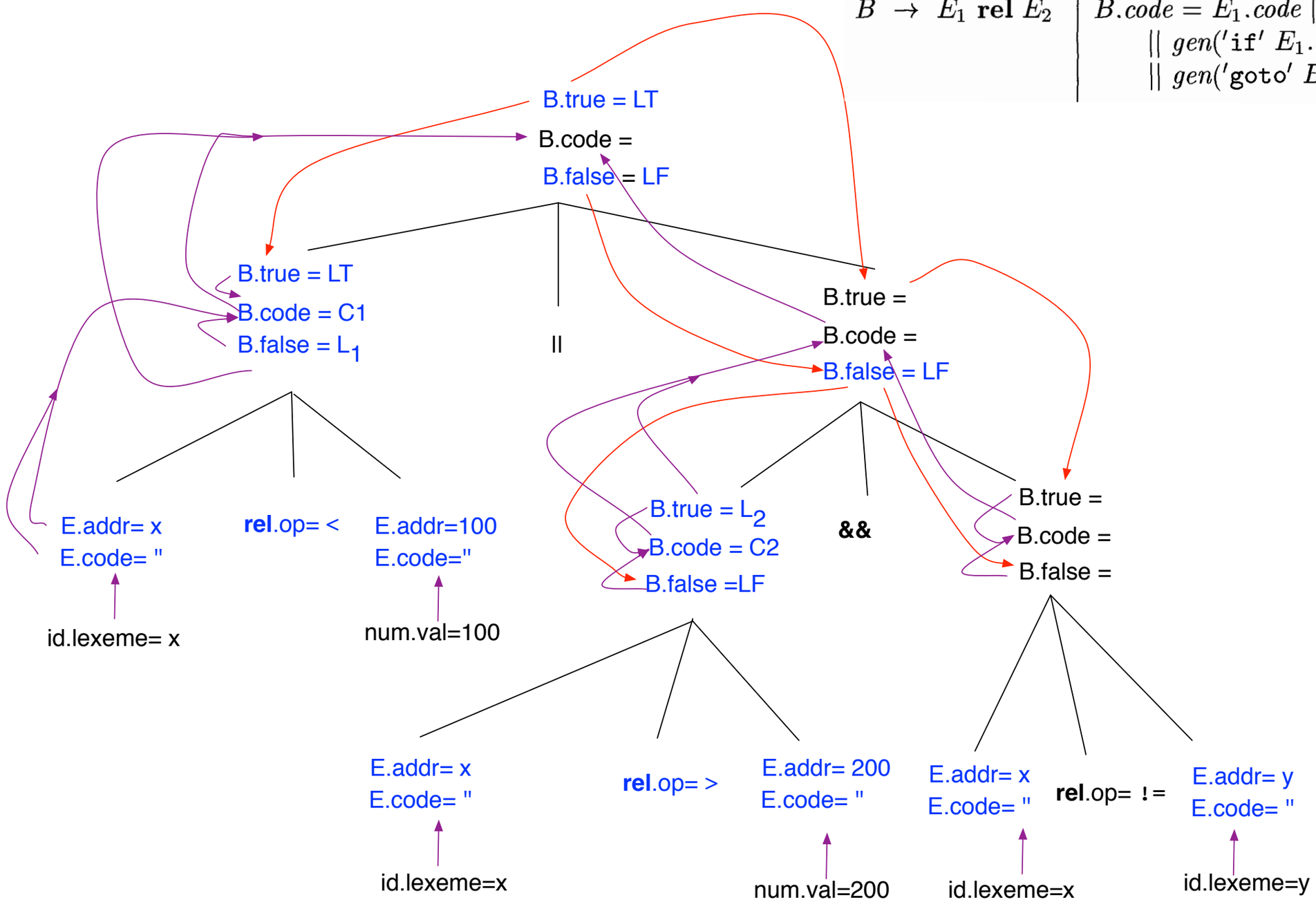
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



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 = ' '

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

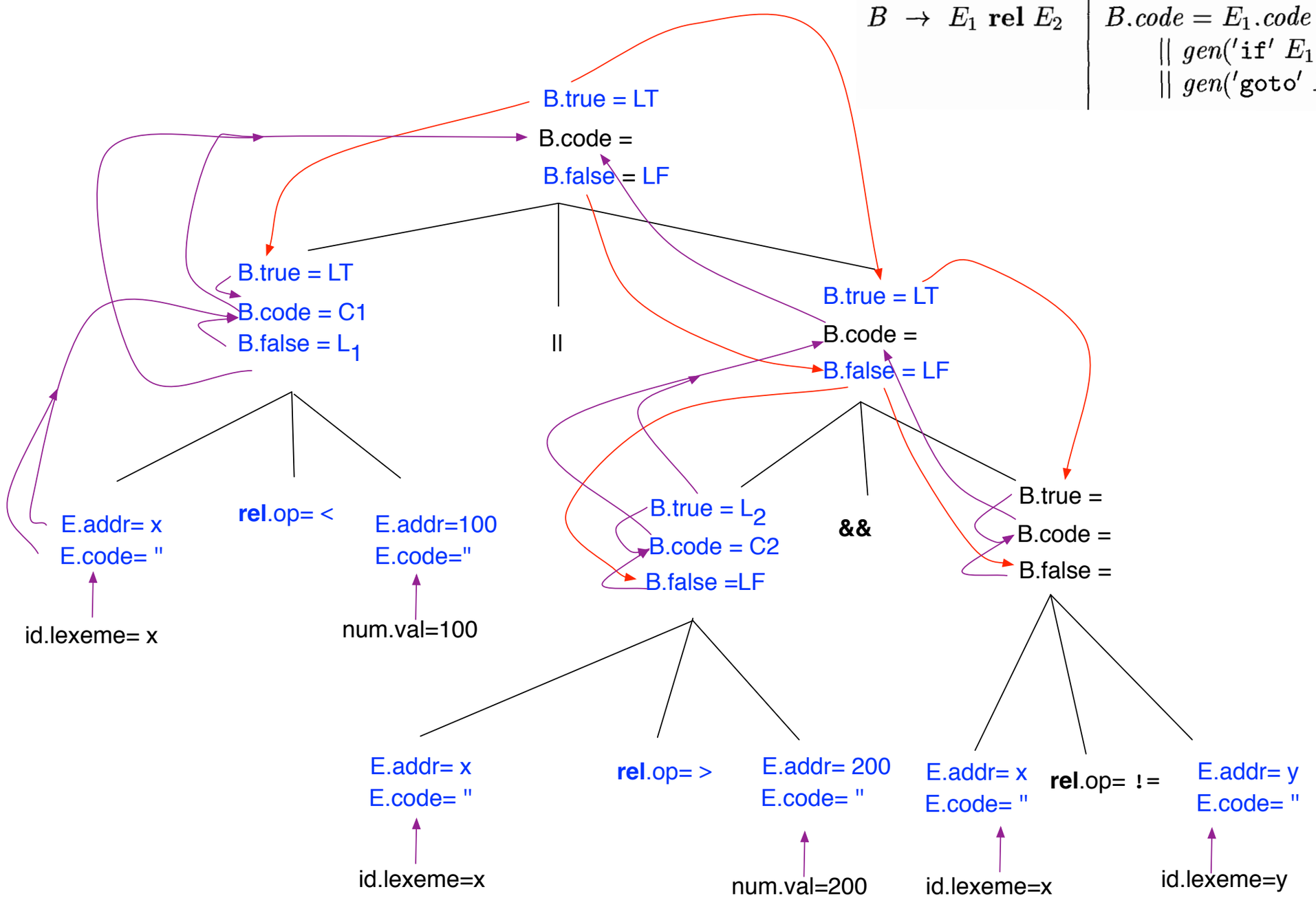


C1 =
if x < 100 goto LT
goto L_1

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 = ' '

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

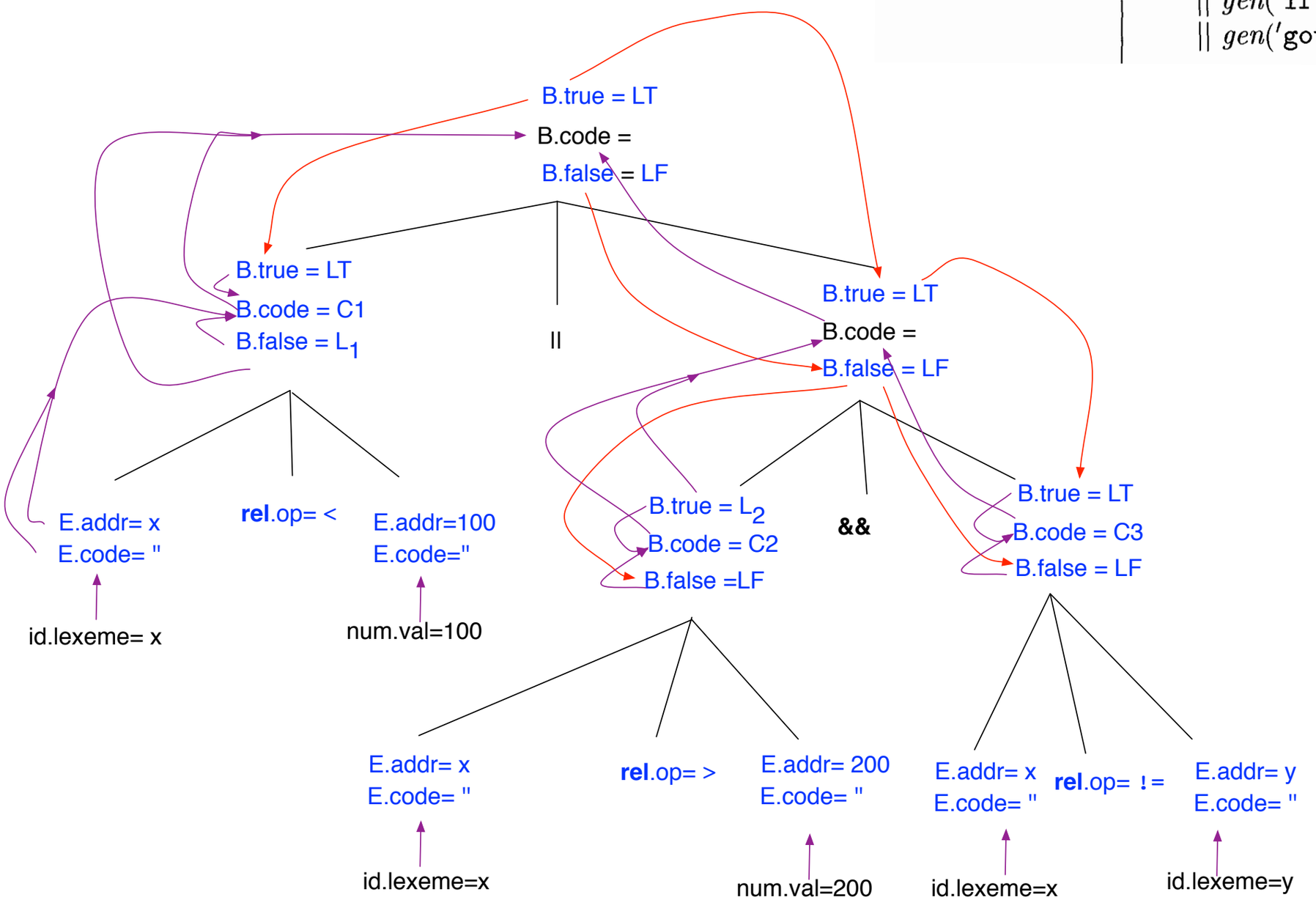


C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

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

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



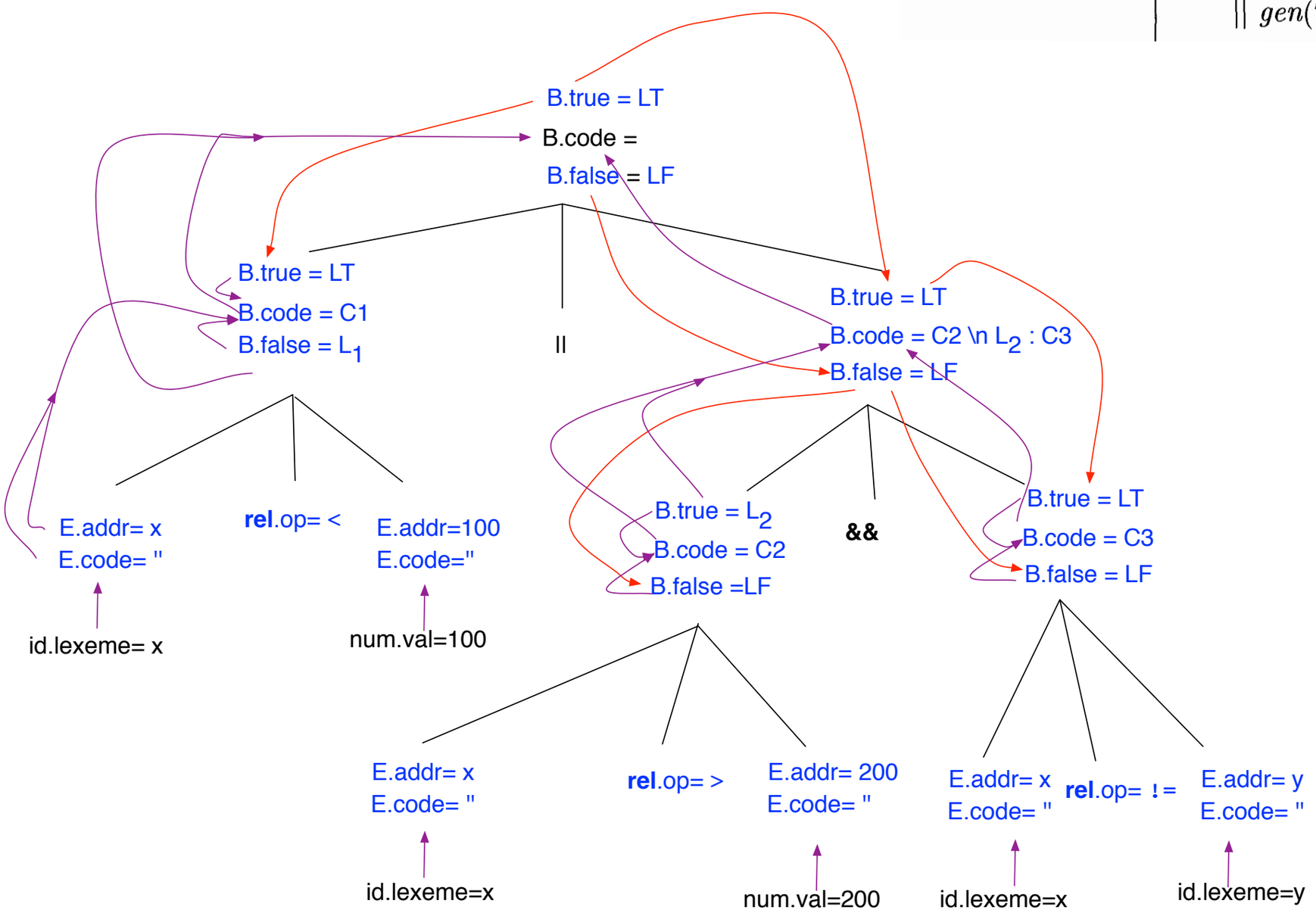
C1 =
if x < 100 goto LT
goto L_1

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 = ' '

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



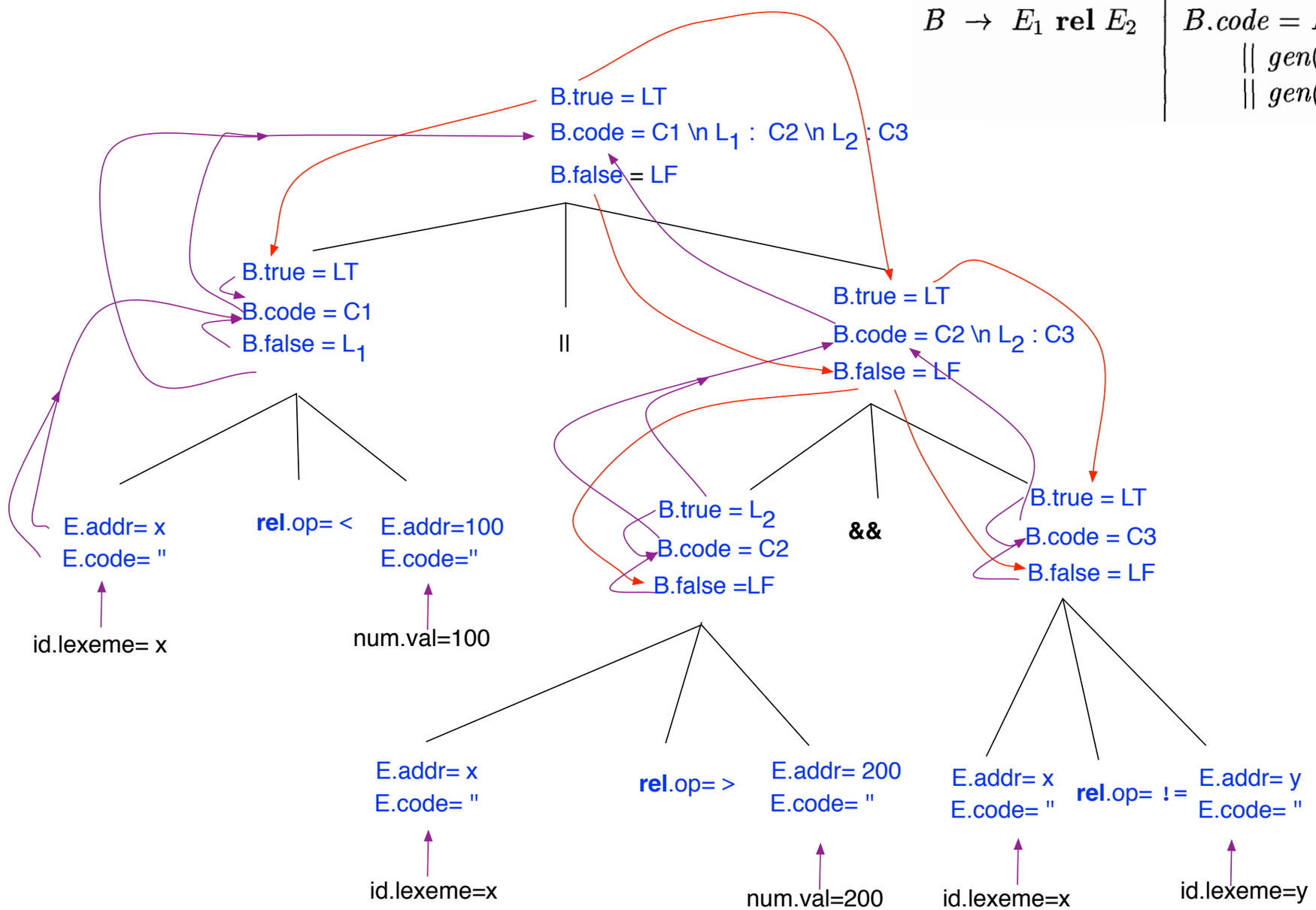
C1 =
if x < 100 goto LT
goto L_1

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

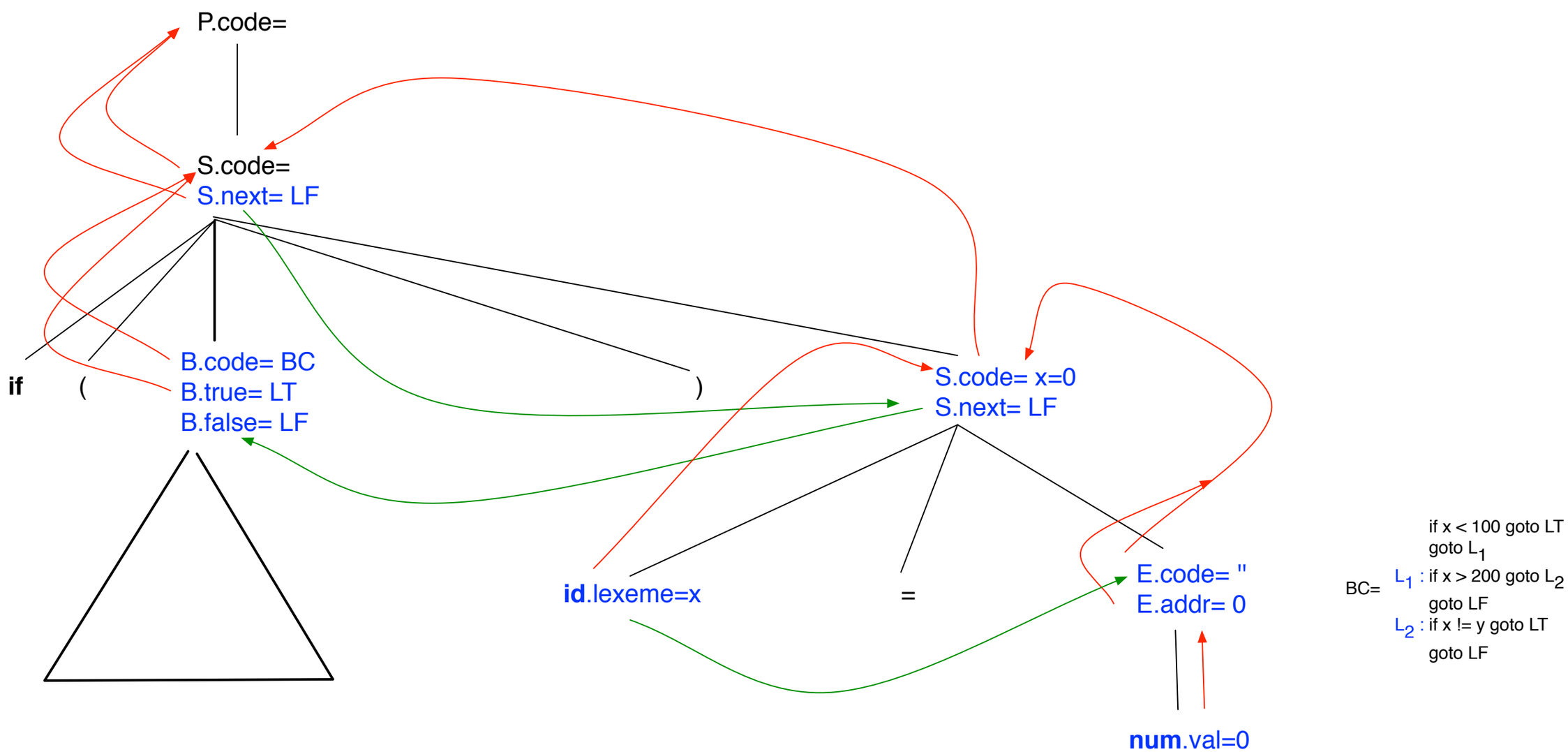
C1 =
if x < 100 goto LT
goto L_1

C2 =
if x > 200 goto L_2
goto LF

C3 =
if x != y goto LT
goto LF

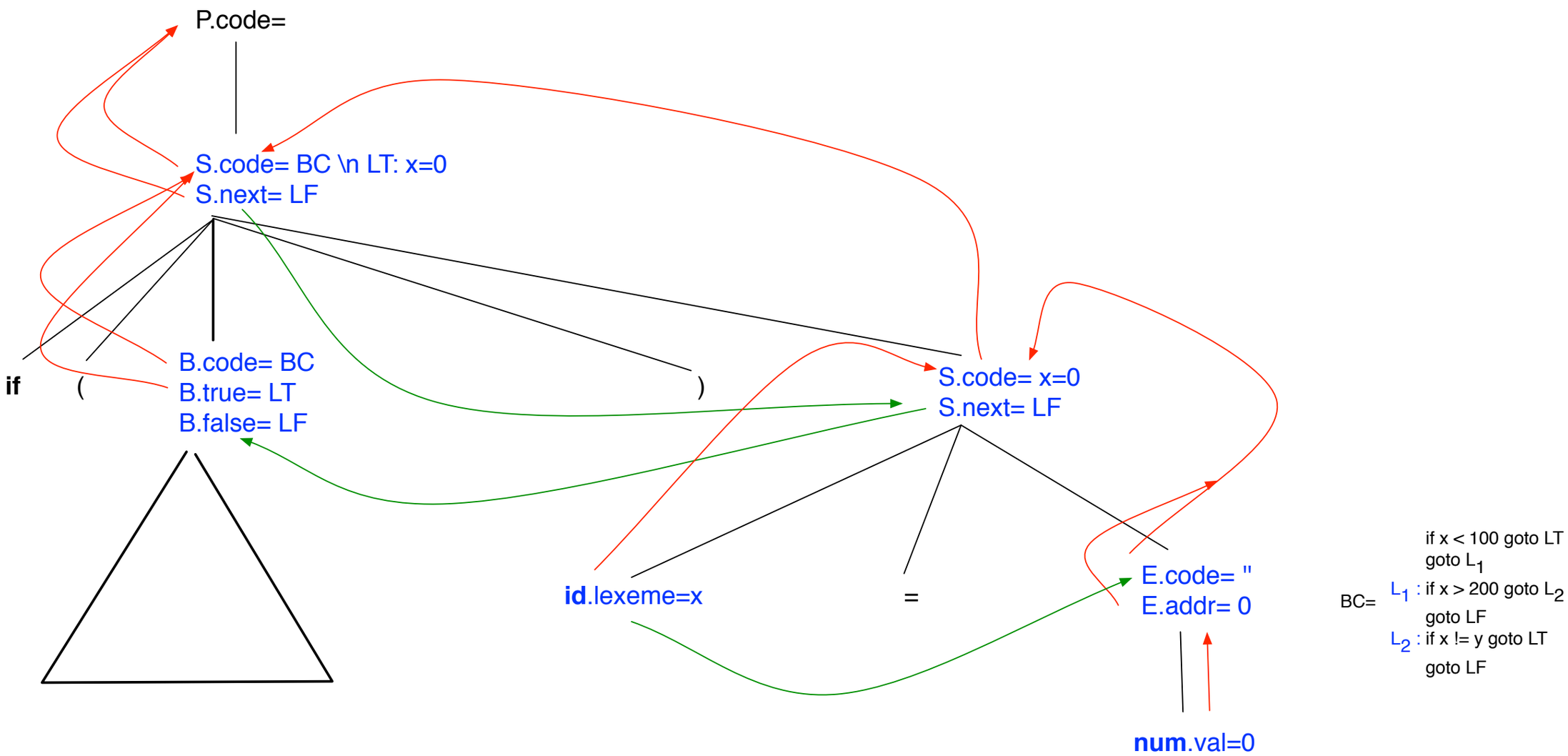
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



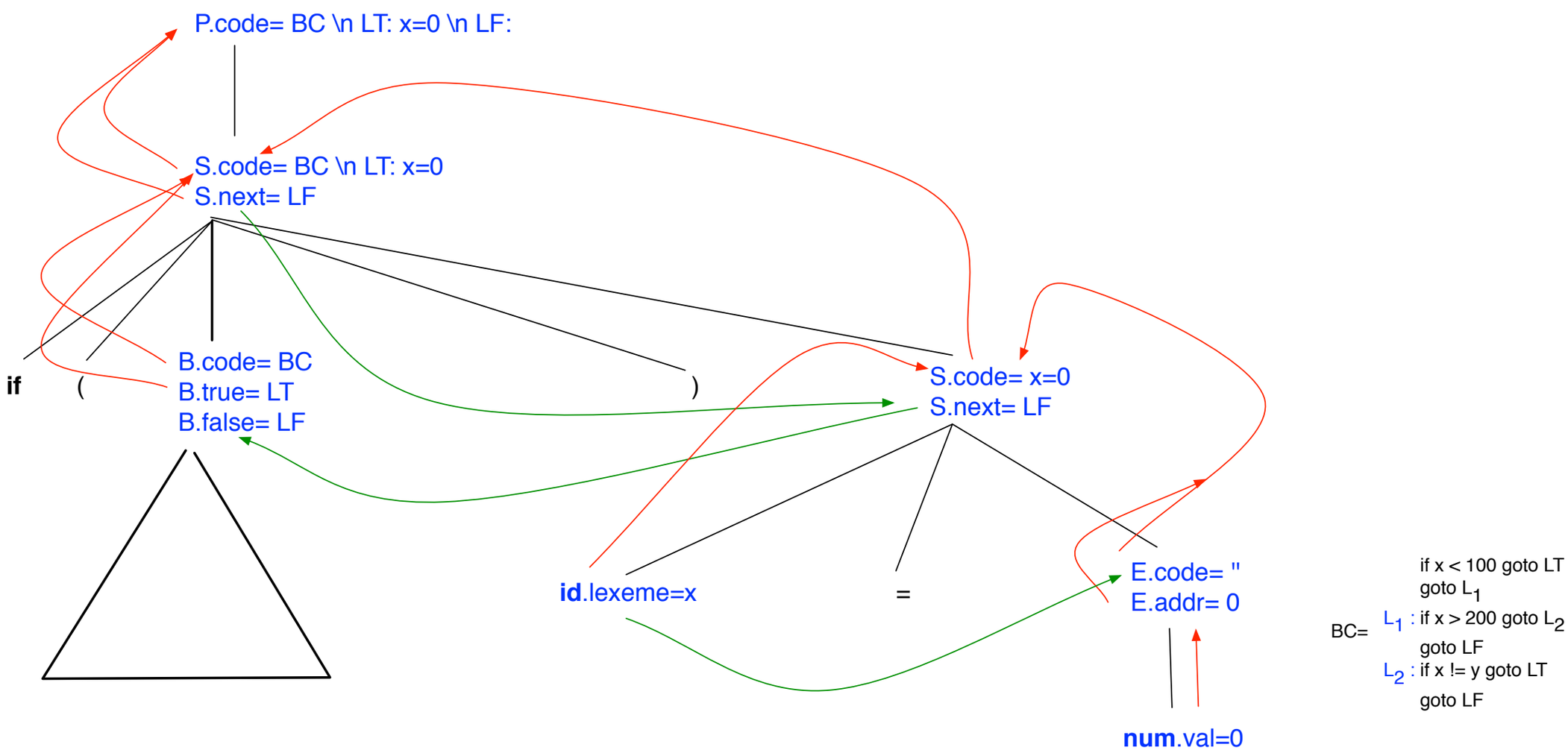
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



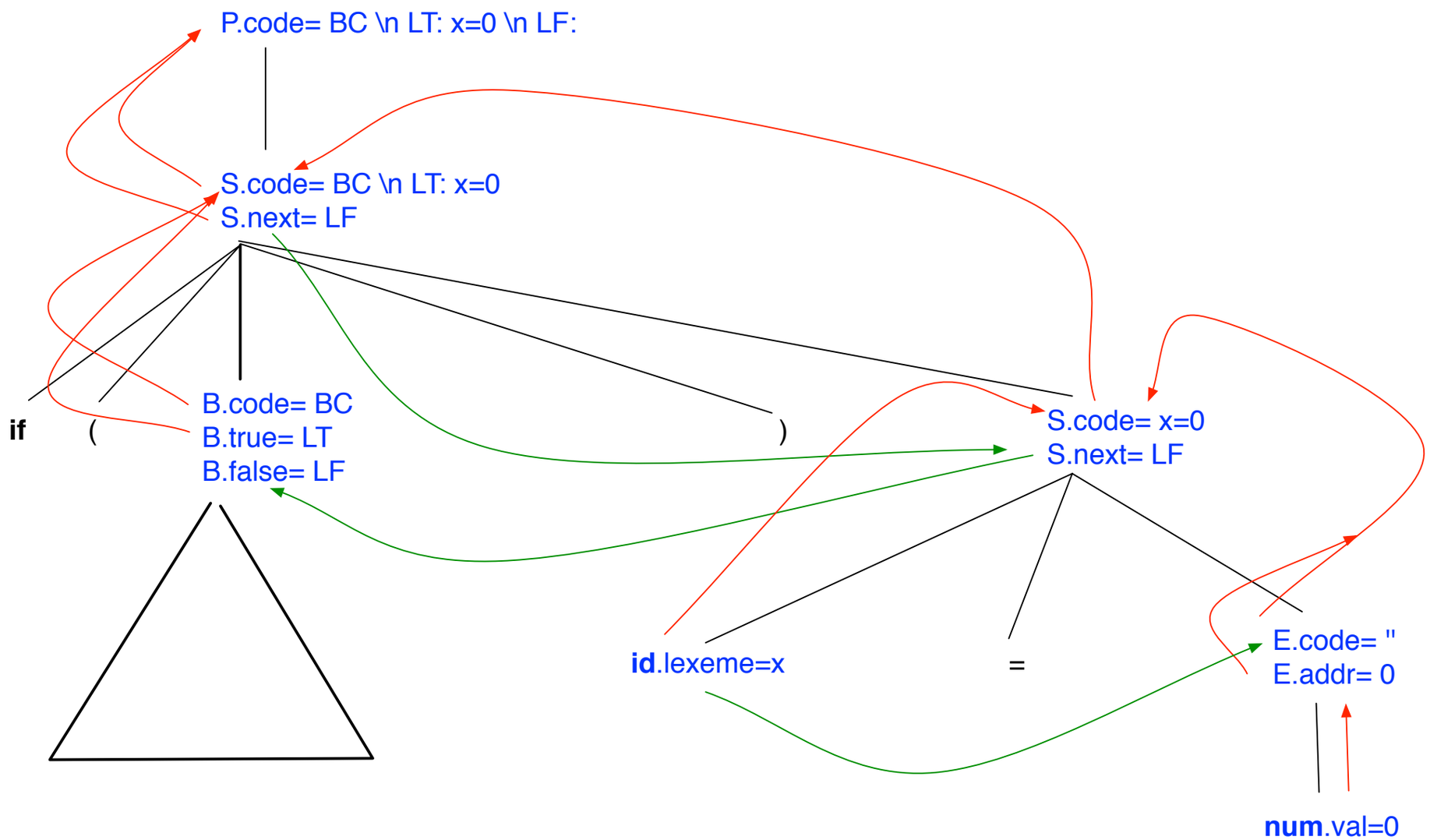
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ' '



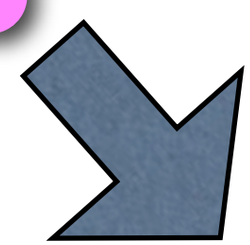
if x < 100 goto LT
goto L_1
L_1 : if x > 200 goto L_2
goto LF
L_2 : if x != y goto LT
goto LF
LT: x=0
LF:

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

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

$$
\begin{array}{l|l}
B \rightarrow B_1 \mid\mid B_2 & B_1.true = B.true \\
& B_1.false = newlabel() \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code
\end{array}
$$

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

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.*

Boolean Values and Jumping Code

Boolean expression → flow of control in statements

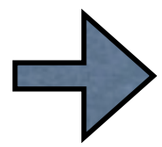
↓

may also be evaluated for its value

- ***Use two passes.*** Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.
- ***Use one pass for statements, but two passes for expressions.*** With this approach, we would translate **E** in **while (E)** S_1 before S_1 is examined. The translation of **E**, however, would be done by building its syntax tree and then walking the tree.

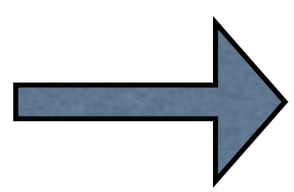
S → **id** = B	

```
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
```

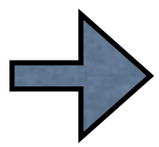
```
p = x<100 || x > 200 && x!=y
```



```
    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
```

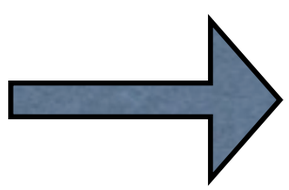
S → **id** = B	S.addr = **new** Temp() S.lab = newlabel() B.true = newlabel() B.false= newlabel() S.code = B.code ‖ label(B.true) ‖ gen(S.addr '= `true`') ‖ gen('`goto`' S.lab) ‖ label(B.false)‖ gen(S.addr '= `false`') ‖ label(S.lab) ‖ gen(top.get(**id**.lexeme) '=' S.addr)

```
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
```

One-Pass Code Generation Using Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump

For example, the translation of the boolean expression *B* in **if (B) S** **contains a jump, for when *B* is false, to the instruction following the code for *S*.** In a one-pass translation, *B* must be translated before *S* is examined. What then is the target of the goto that jumps over the code for *S?*

We have addressed this problem by **passing labels as inherited attributes to where the relevant jump instructions were generated**. But a **separate pass is then needed** to bind labels to addresses.

Backpatching can be used to **generate code** for boolean expressions and flow-of-control statements in **one pass**

Intermediate Code for Procedures

```
n = f(a[i]);
```

1)	t1 = i * 4
2)	t2 = a [t1]
3)	param t2
4)	t3 = call f, 1
5)	n = t3

D	→	**define** T **id** (F) { S }
F	→	ε | T **id** , F
S	→	**return** E ;
E	→	**id** (A)
A	→	ε | E, A

Function definitions and function calls can be translated using concepts that have already been introduced

Function types. The type of a function must encode the return type and the types of the formal parameters. Let *void* be a special type that represents no parameter or no return type. Function types can be represented by using a constructor *fun* applied to the return type and an ordered list of types for the parameters.

Symbol tables*.* Let *s* be the top symbol table when the function definition is reached. The function name is entered into *s* for use in the rest of the program. The formal parameters of a function can be handled in analogy with field names in a record

In the production for *D,* after seeing define and the function name, we push *s* and set up a new symbol table

Env.push(top); top = **new** *Env(top);*

Call the new symbol table, *t.* Note that *top* is passed as a parameter in new *Env (top),* so the new symbol table *t* can be linked to the previous one, *s.* The new table *t* is used to translate the function body. We revert to the previous symbol table s after the function body is translated.

Type checking. Within expressions, a function is treated like any other operator.

Function calls. When generating three-address instructions for a function call id(E, *E, ... ,E),* it is sufficient to generate the three-address instructions for evaluating or reducing the parameters *E* to addresses, followed by a param instruction for each parameter.