

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

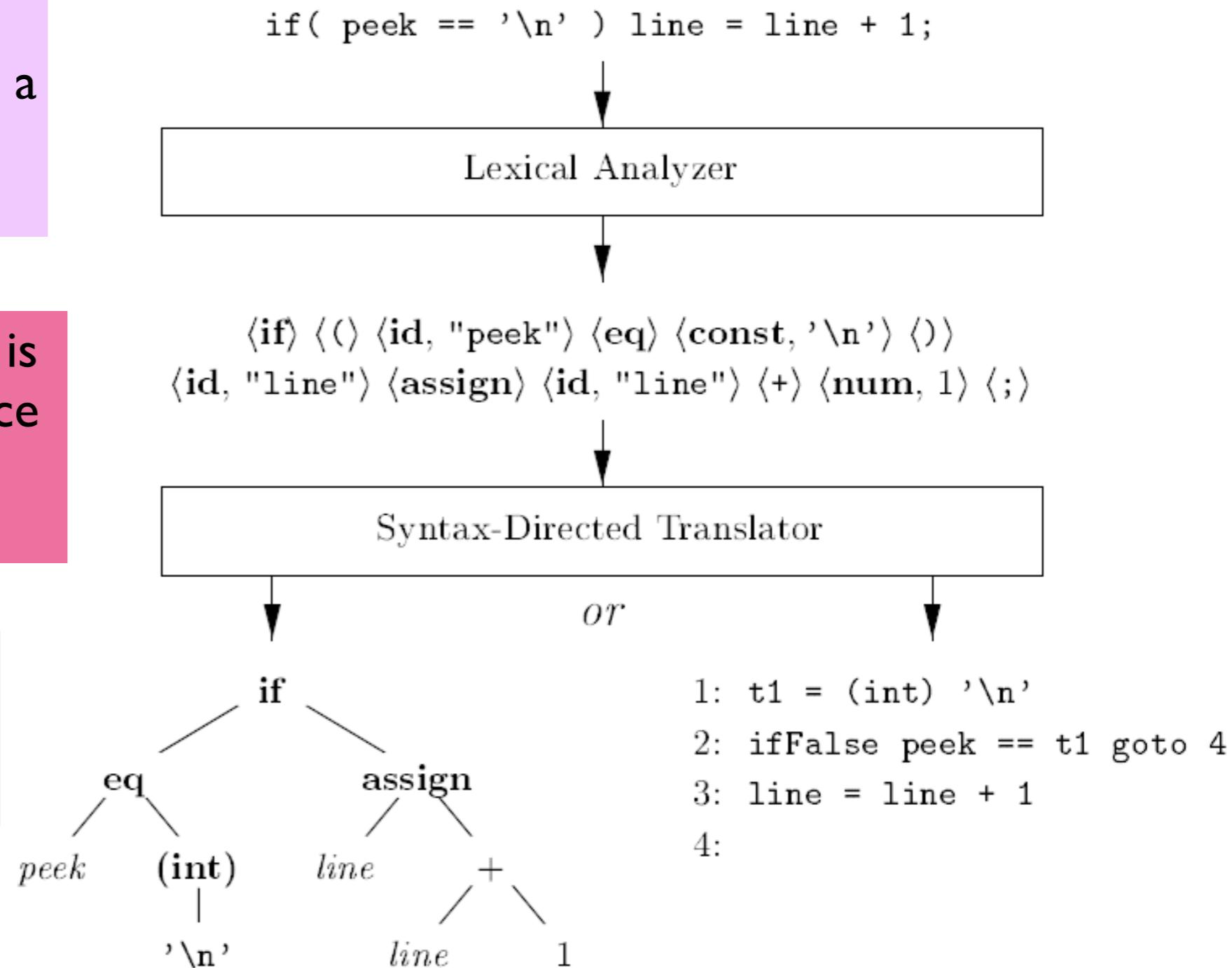


Figure 2.46: Two possible translations of a statement

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

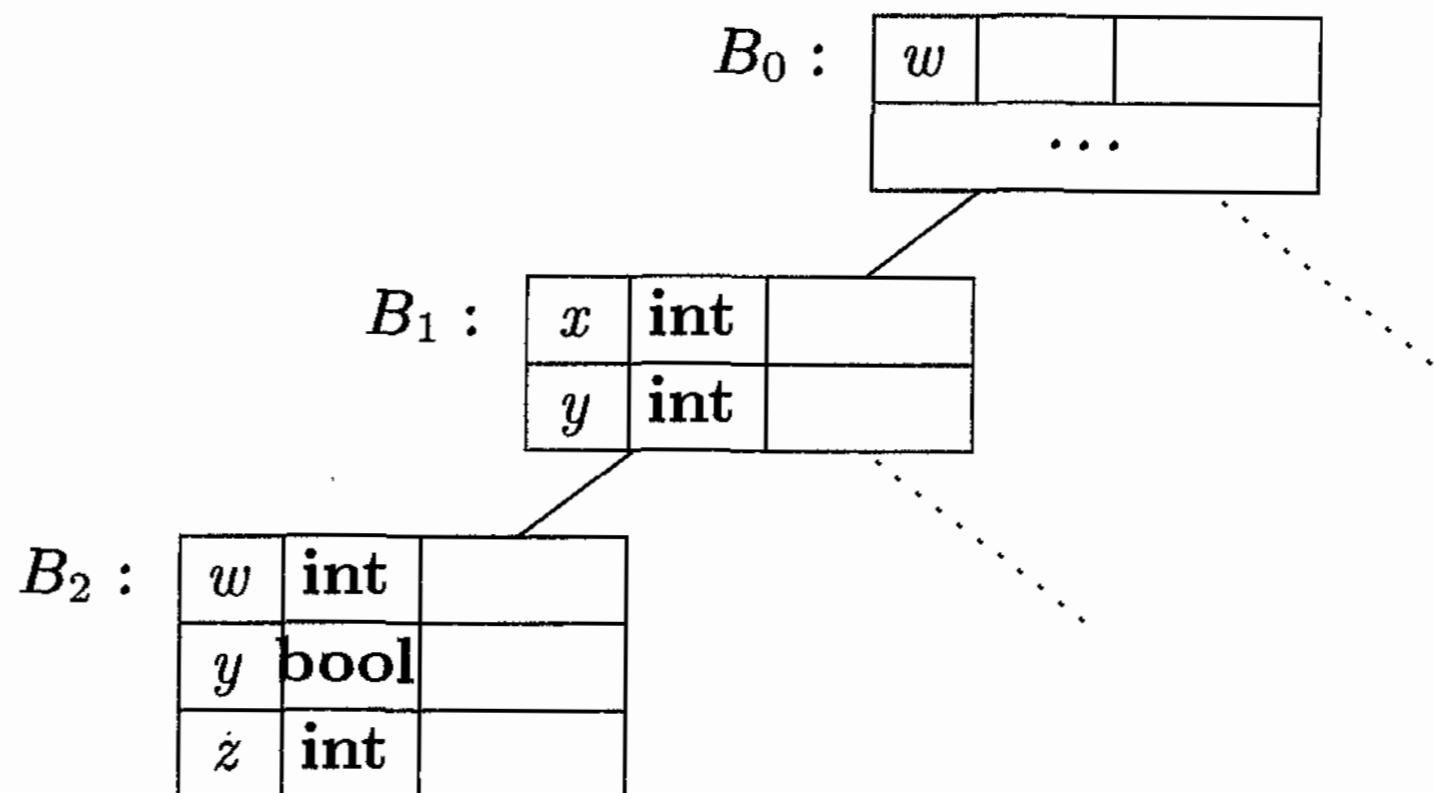
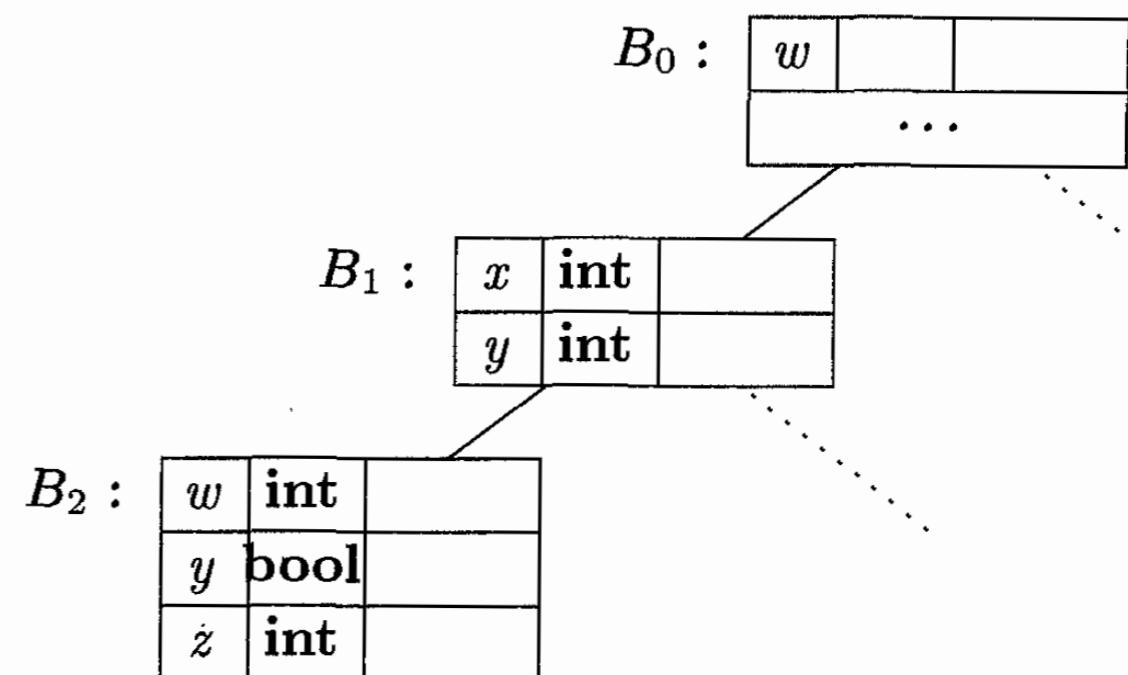


Figure 2.36: Chained symbol tables for Example 2.15

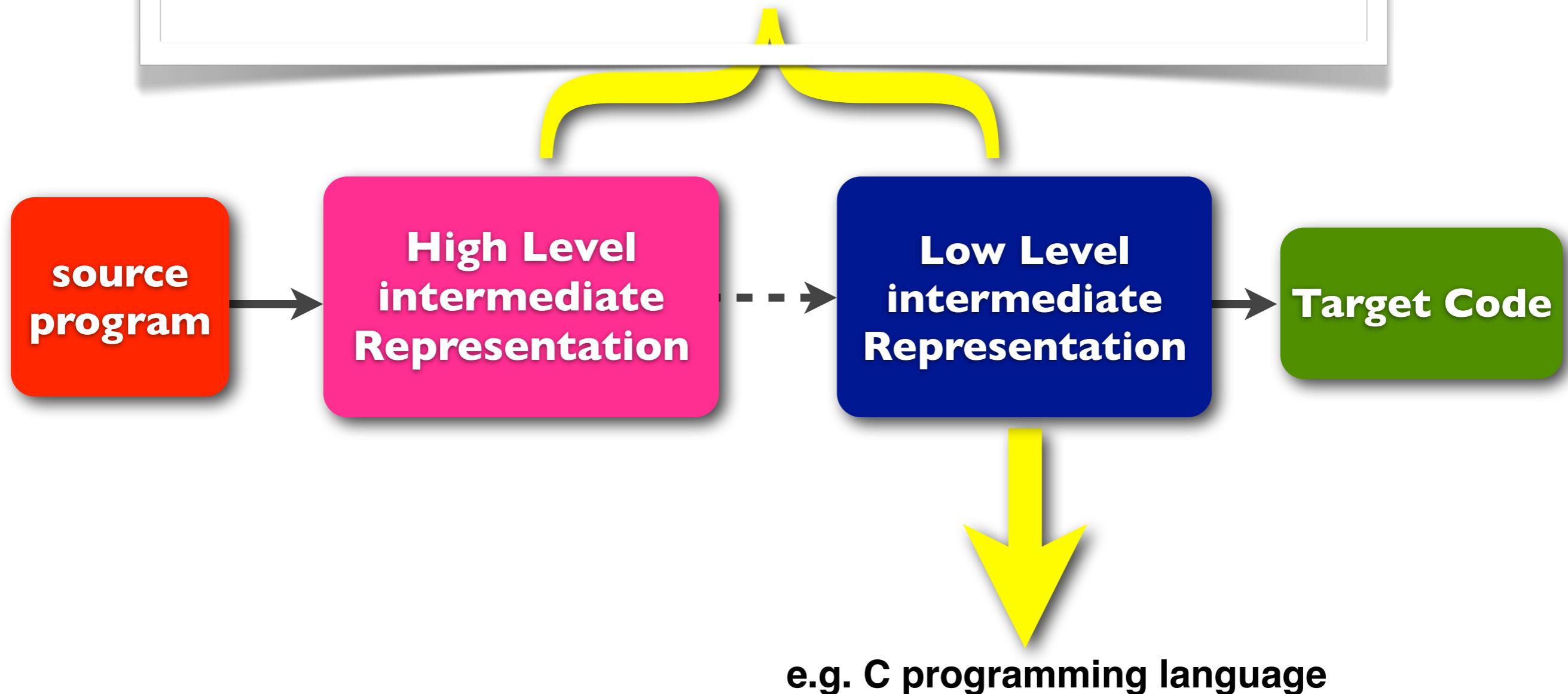
```

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) }

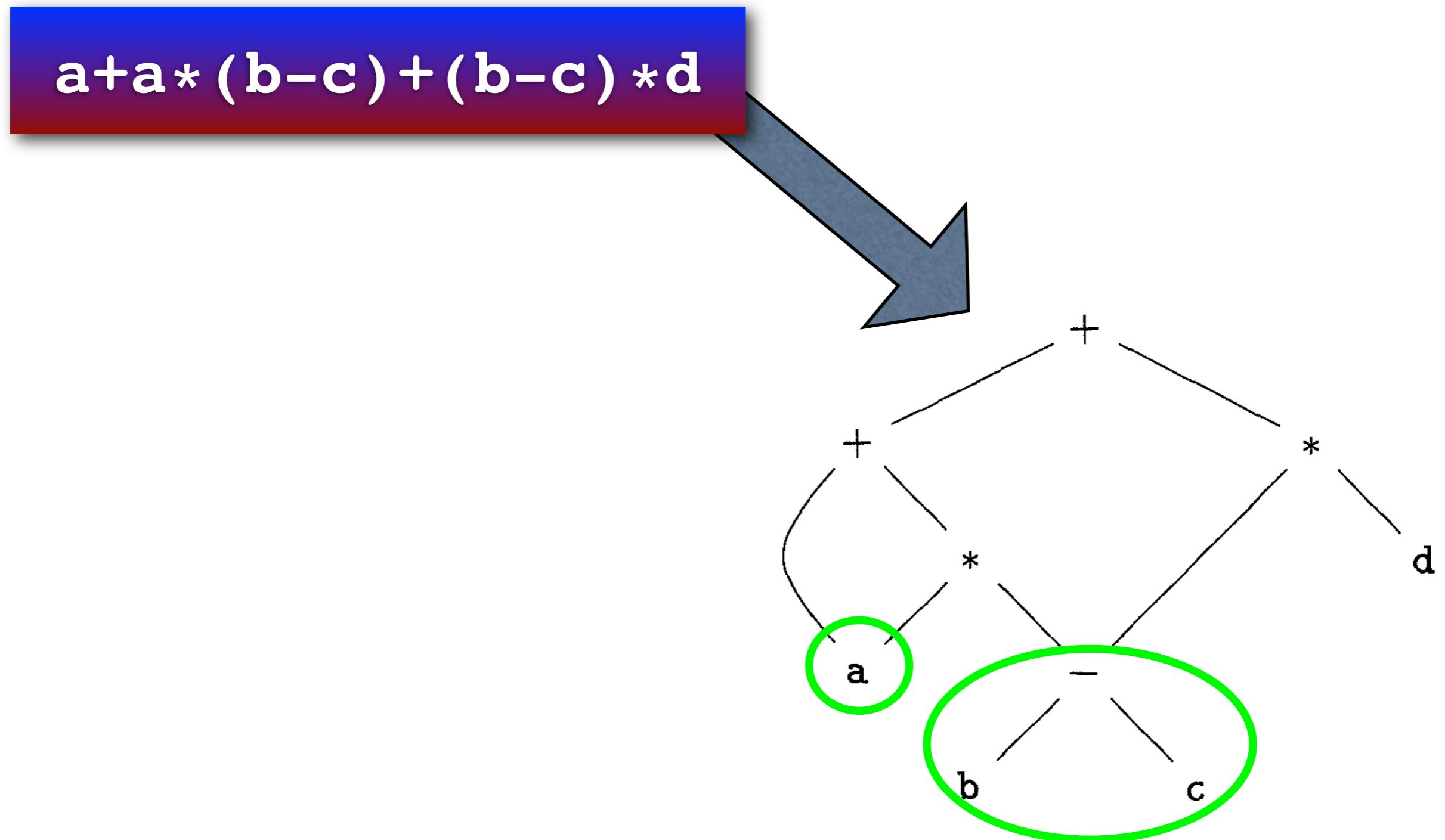
```



Sequence of intermediate representations



DAG (variants of syntax trees)



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

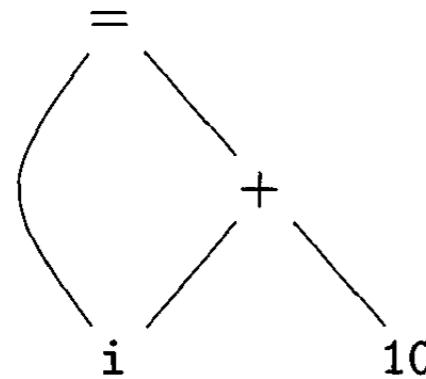
1. $P_1 = \text{Leaf} (\text{id}, \text{entry-}a)$
2. $P_2 = \text{Leaf} (\text{id}, \text{entry-}a) = P_1$
3. $P_3 = \text{Leaf} (\text{id}, \text{entry-}b)$
4. $P_4 = \text{Leaf} (\text{id}, \text{entry-}c)$
5. $P_5 = \text{Node} ("-", P_3, P_4)$
6. $P_6 = \text{Node} ("*", P_1, P_5)$
7. $P_7 = \text{Node}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

Steps for constructing the DAG

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

When the call to $\text{Leaf} (\text{id}, \text{entry-}a)$ 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 as that returned at step 5; i.e., $p_{10} = p_5$. \square

The Value-Number Method for constructing DAGs



(a) DAG

	id		
1		—	
2	num	10	
3	+	1	2
4	=	1	3
5	...		

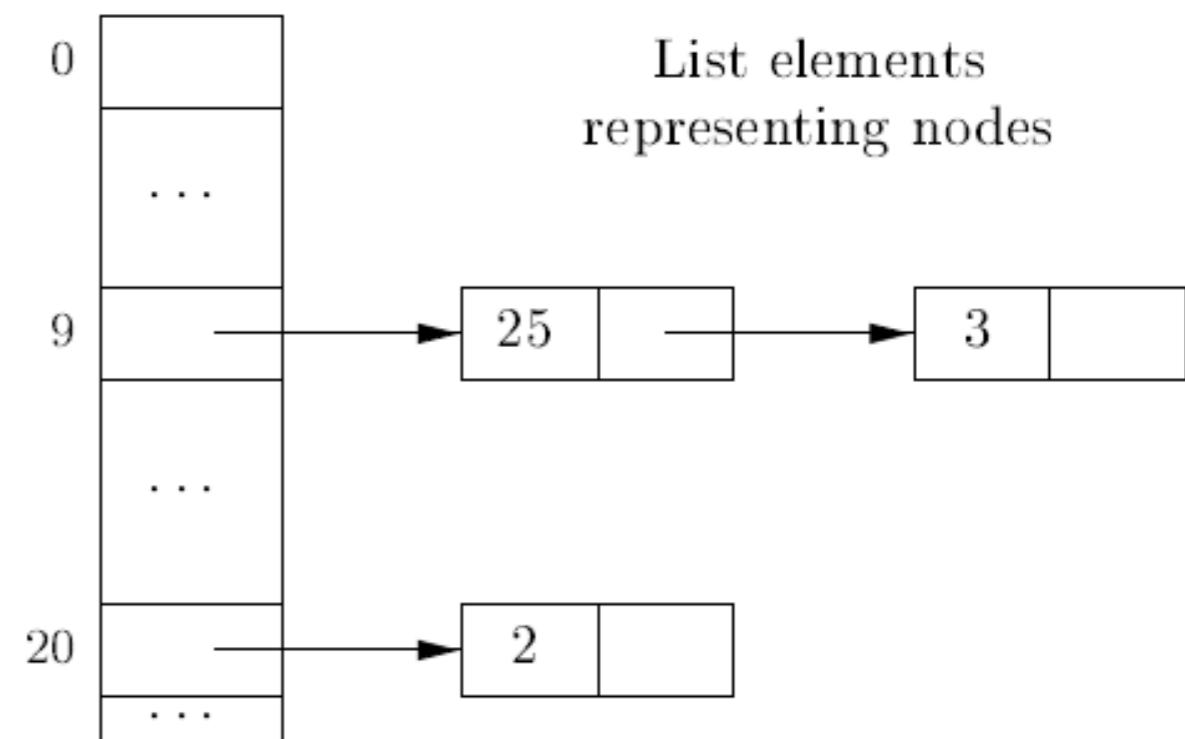
An arrow points from the entry for `i` in the array to the `i` node in the DAG.

(b) Array.

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

List elements
representing nodes

Array of bucket
headers indexed
by hash value

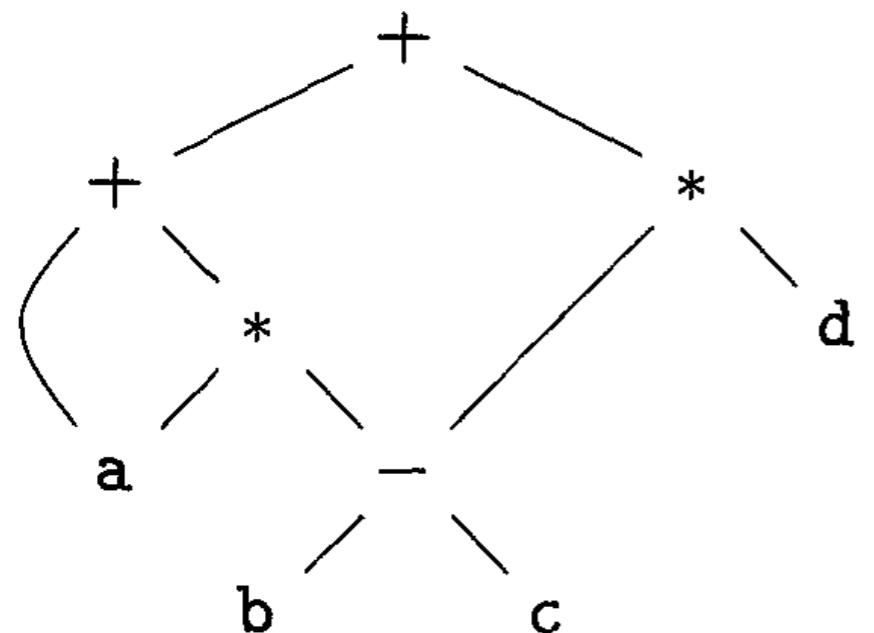


An efficient implementation is obtained by using **dictionaries** such as hash table

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 ($<$, $=$, \geq , 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
 $p(x_1, x_2, \dots, x_n)$.***

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);
```

<pre>L: t₁=i+1 i=t₁ t₂=i*8 t₃=a[t₂] if t₃ < v goto L</pre>	<pre>100: t₁=i+1 101: i=t₁ 102: t₂=i*8 103: t₃=a[t₂] 104: if t₃ < v goto 100</pre>
(a) Symbolic labels.	(b) Position numbers.

Quadruples

quadruple (quad):

<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
-----------	------------------------	------------------------	---------------

x = y + z

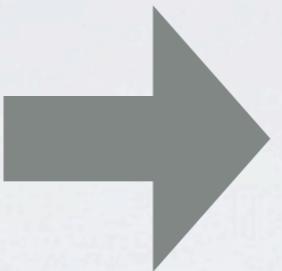


+	y	z	x
---	---	---	---

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
```



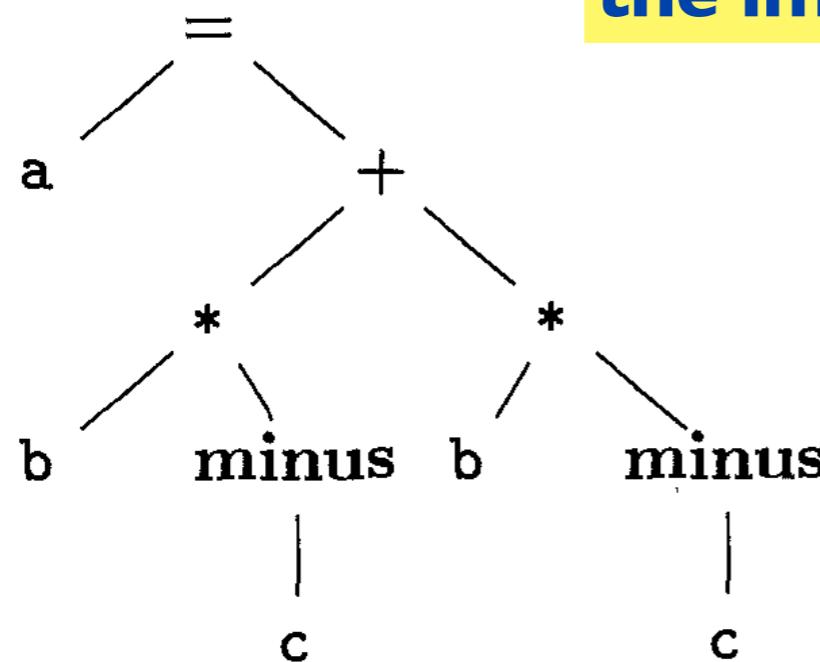
op	arg ₁	arg ₂	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a
...

Triples

triple:

<i>op</i>	<i>arg</i> ₁	<i>arg</i> ₂
-----------	-------------------------	-------------------------

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



the implicit result

	<i>op</i>	<i>arg</i> ₁	<i>arg</i> ₂
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
			...

(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 arg₁ arg₂

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

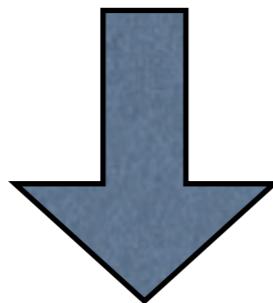
Static Single-Assignment Form

an intermediate representation that facilitates certain code optimizations

all assignments in SSA are to variables with distinct names

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$
(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;
```

$$\Phi(x_1, x_2) = \begin{cases} x_1 & \text{x}_1 \text{ passes through the true part of the conditional} \\ x_2 & \text{ootherwise} \end{cases}$$

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.

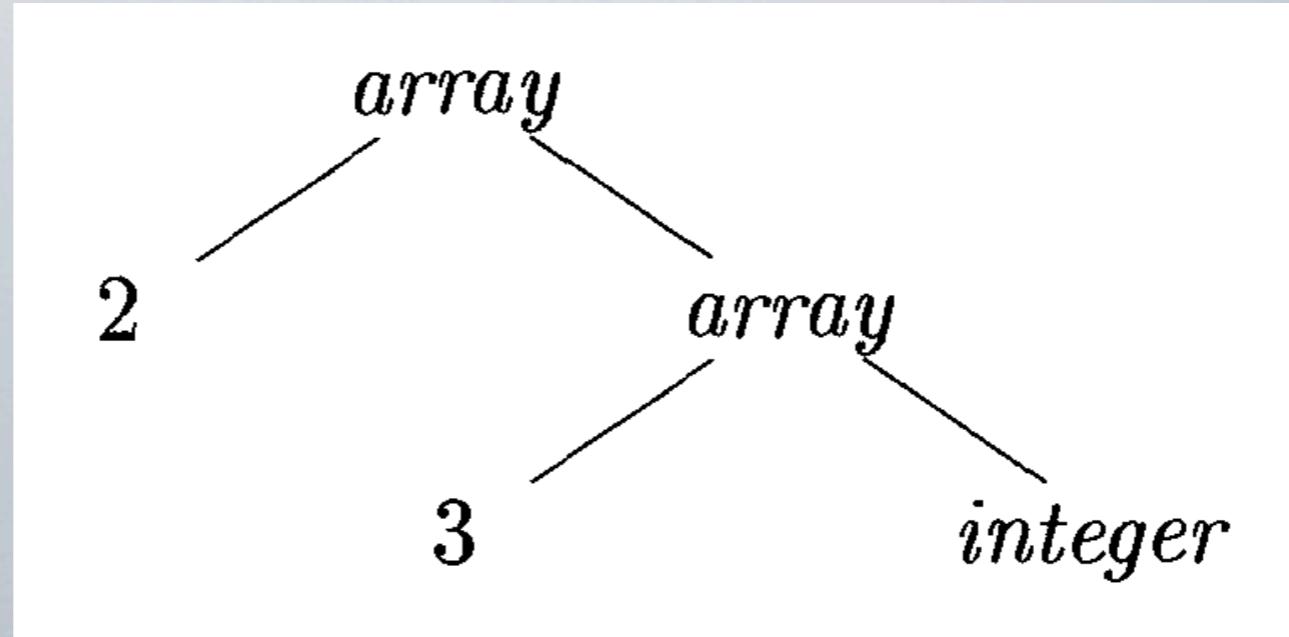
TYPES

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

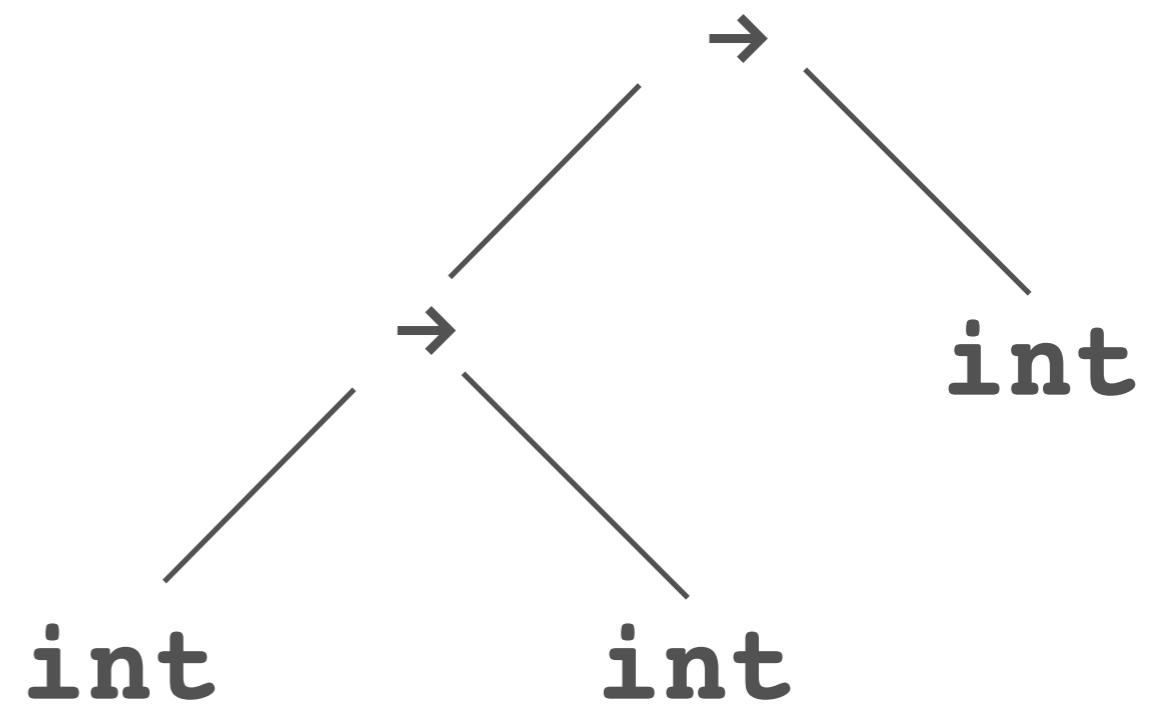
Translation Applications. From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic 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 \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 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 \text{ id } ; \ D \mid \epsilon \\ T & \rightarrow & B \ C \mid \text{record } \{ \ D \ \} \\ B & \rightarrow & \text{int} \mid \text{float} \\ C & \rightarrow & \epsilon \mid [\ \text{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.

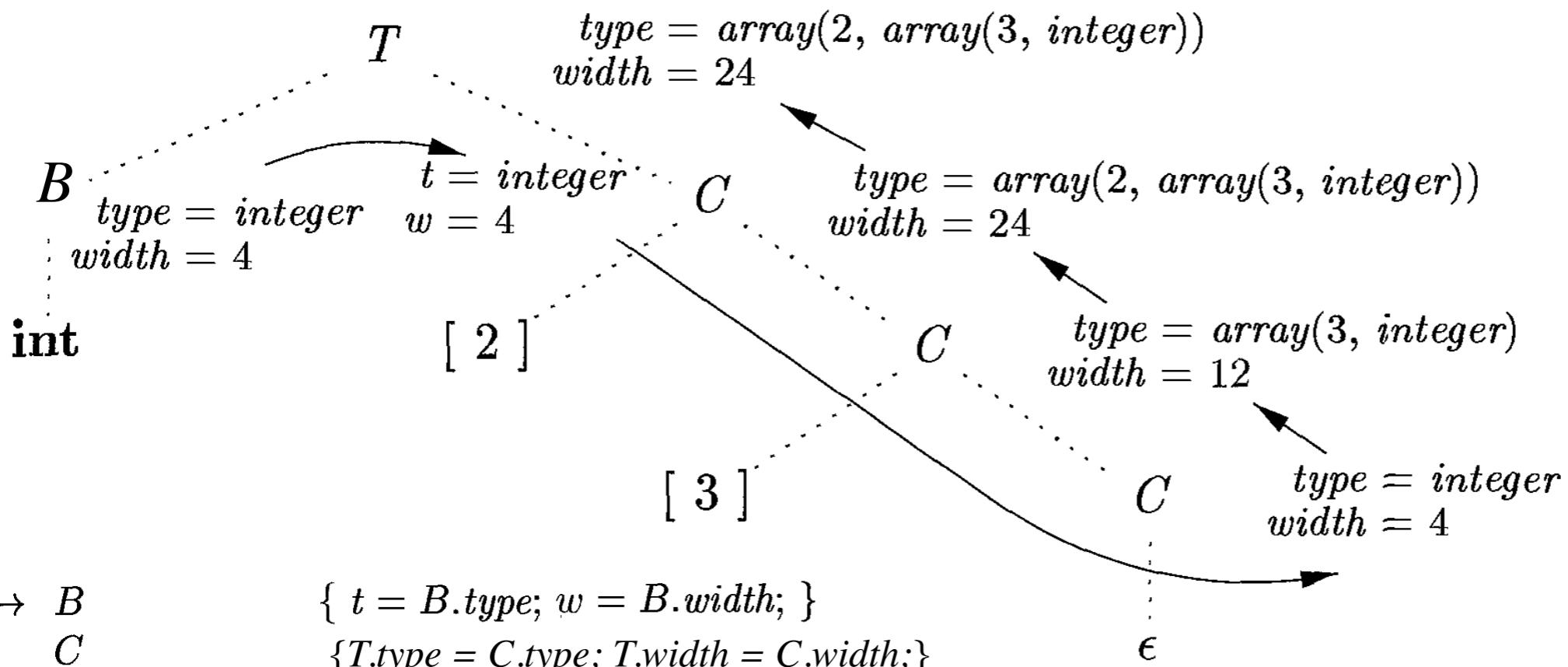
Storage Layout

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$	In a SDD, t and w would be inherited attributes for C
$T \rightarrow C$	$\{ T.type = C.type; T.width = C.width; \}$	
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$	
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$	
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$	
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$	

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

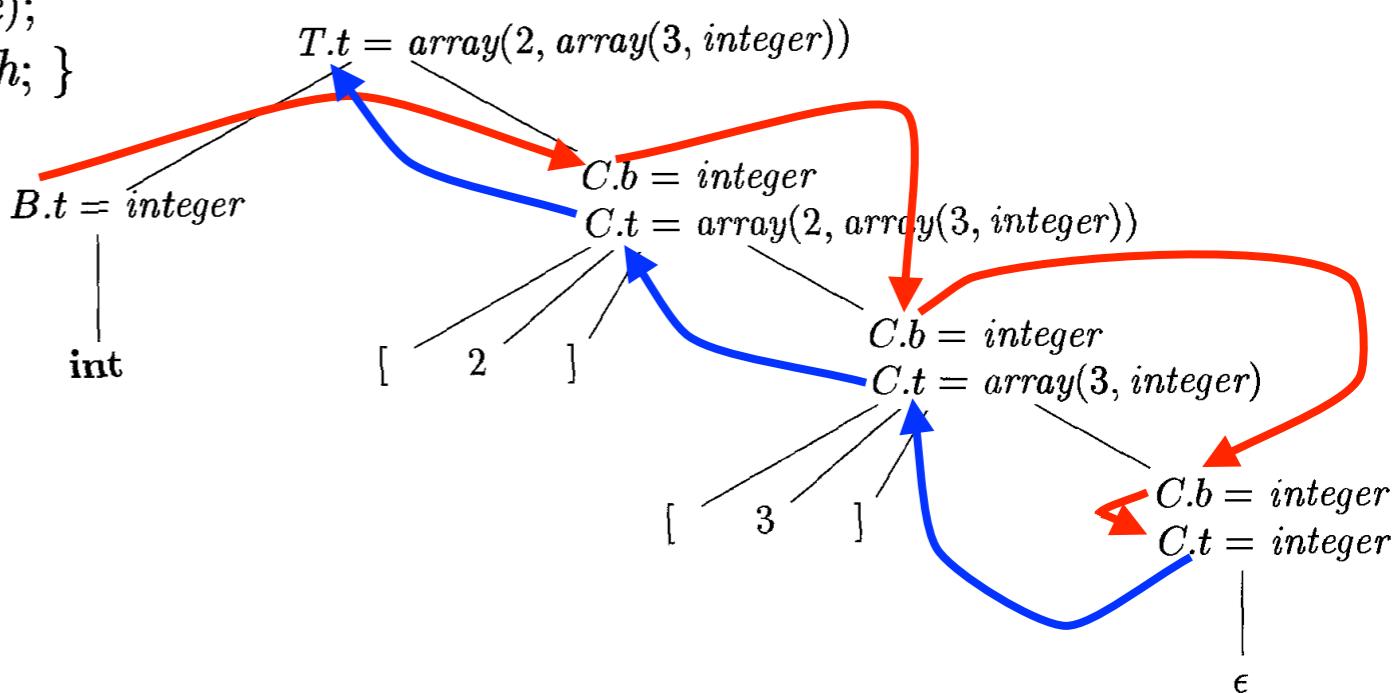


$B \rightarrow \text{int} \quad \{ B.type = \text{integer}; B.width = 4; \}$

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

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

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



Sequences of Declarations

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

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

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

Fields in Records and Classes

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

- 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;
```

$T \rightarrow \mathbf{record} \{ \begin{array}{l} \{ Env.push(top); top = \mathbf{new} Env(); \\ Stack.push(offset); offset = 0; \} \end{array}$

$D \}' \begin{array}{l} \{ T.type = record(top); T.width = offset; \\ top = Env.pop(); offset = Stack.pop(); \} \end{array}$

Translation of Expressions

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

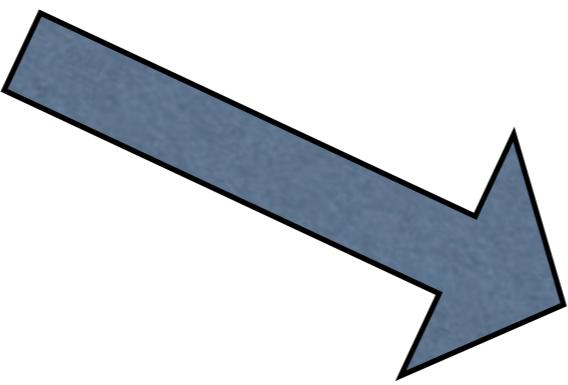
Three-address code for expressions

$\text{gen}(x ' =' y ' +' z)$ ➔ three-address instruction $x = y + z$

top denote the current symbol table

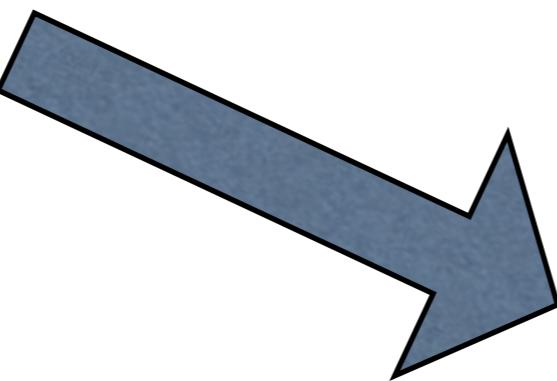
$\text{gen}(\text{top.get(id.lexeme)}...)$ is $\langle \text{top.get(id.lexeme)}, \text{step-of-access-link} \rangle \dots$

a = b + -c



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

a = b + -c



t₁ = minus c
t₂ = b + t₁
a = t₂

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

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

$t_1 = \text{minus } c$
 $t_2 = b + t_1$
 $a = t_2$

Addressing Array Elements

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

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

T [n₁][n₂] A
base= A[0][0]
w = width of elements
A[i₁][i₂]
has address
base + (i₁ x w₁) + (i₂ x w₂)

T [n₁...n_k] A
base= A[0]...[0]
w = width of elements
A[i₁]...[i_k]
has address
base + (i₁ x w₁) + ... + (i_k x w_k)

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

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

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

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

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

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

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

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

$T [n_1]...[n_k] A$
 $\text{base} = A[0]...[0]$
 $w = \text{width of elements}$

$(...((A[i_1])...)[i_j])...)[i_k]$
 has address
 $\text{base} + (i_1 \times w_1) + \dots + (i_k \times w_k)$

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.

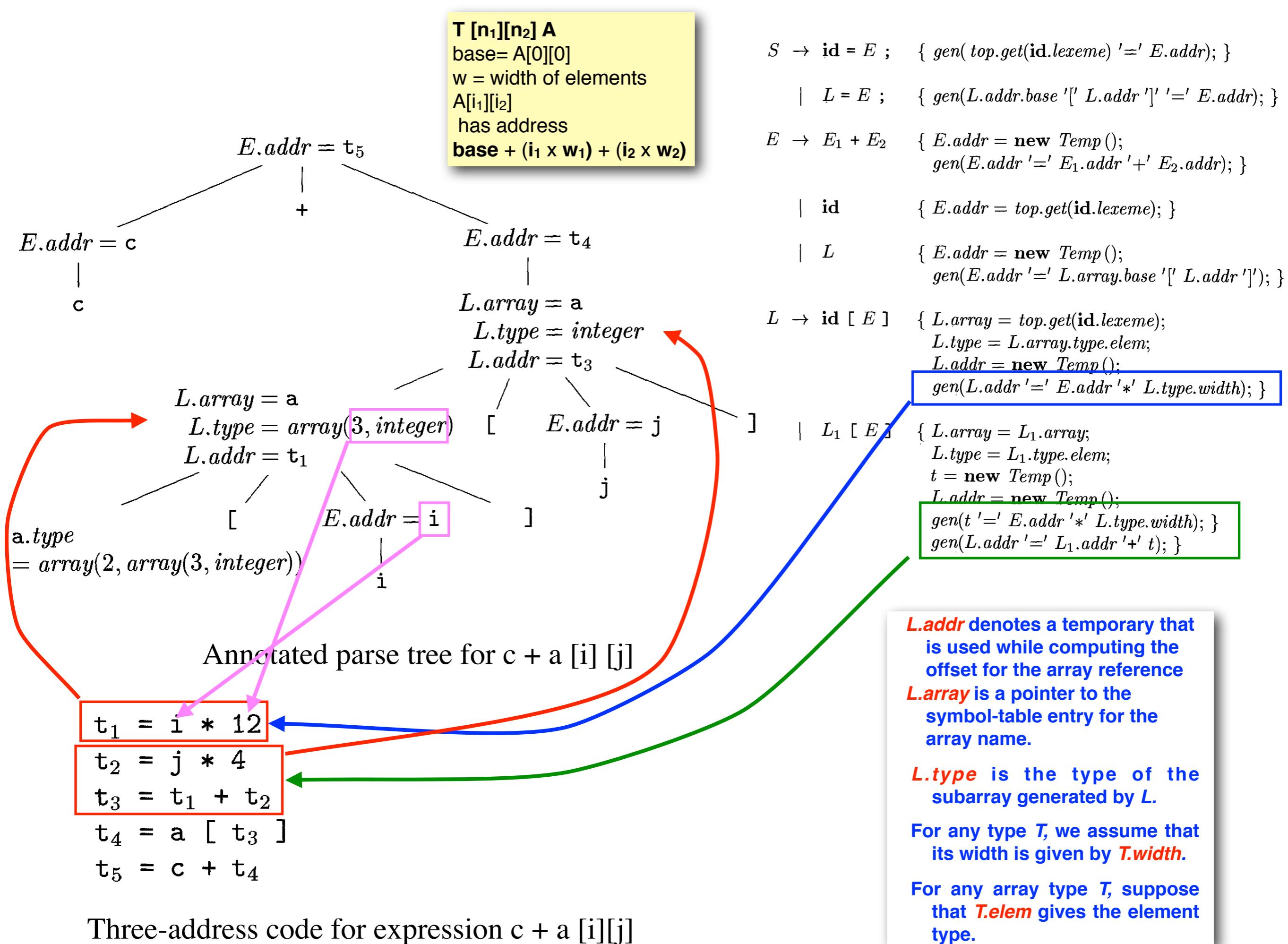
Esercizio

Construct the three-address code for expression

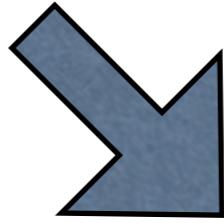
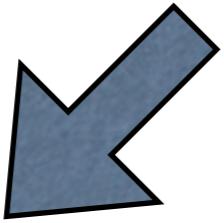
c + a[i][j]

Assume that we have the declaration

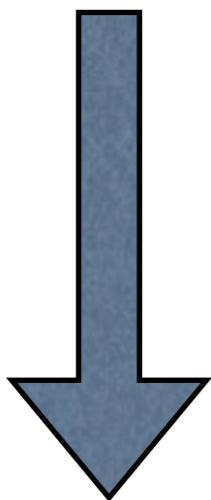
int [2][3] a;



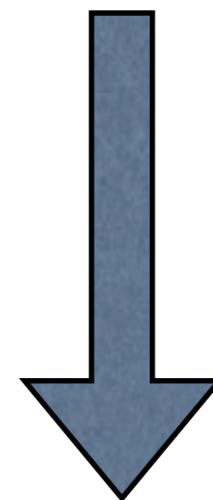
TYPE CHECKING



Type synthesis



Type inference



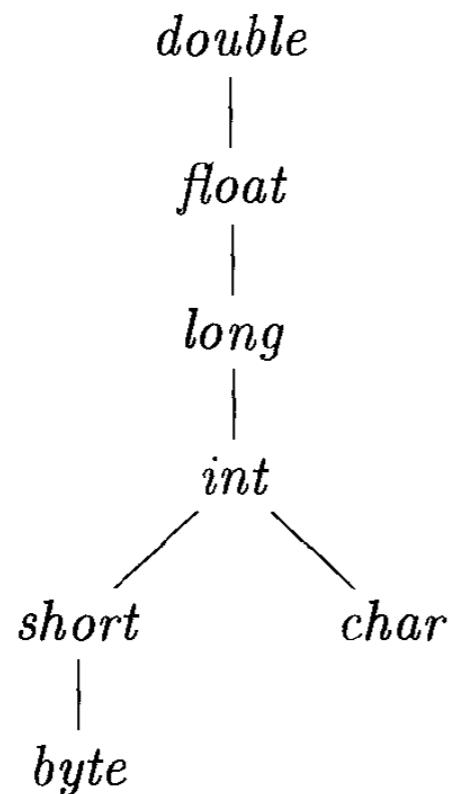
if f has type $s \rightarrow t$ **and** x has type s
then expression $f(x)$ has type t

if $f(x)$ is an expression,
then for some α and β , f has type $\alpha \rightarrow \beta$
and x has type α

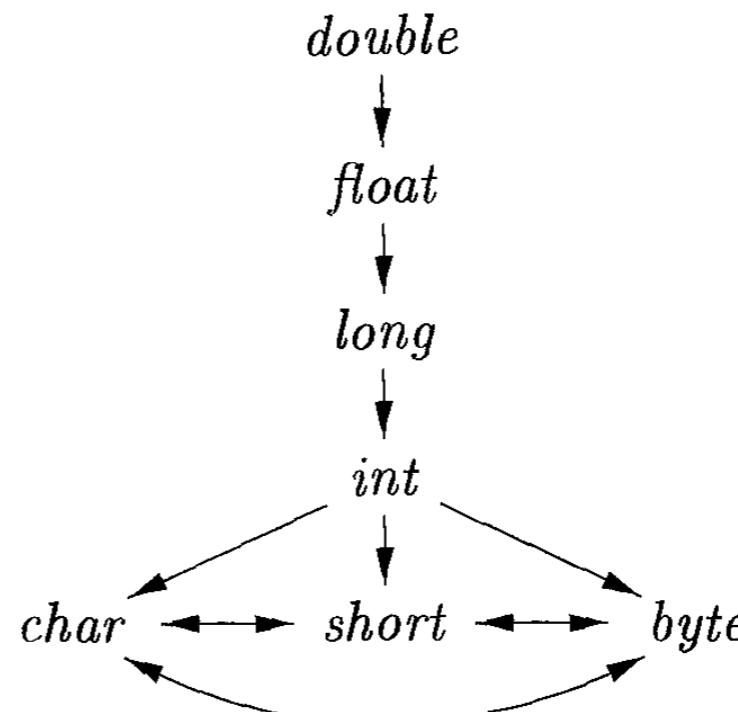
Type Conversions

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

$\text{widen}(a, t, w)$ 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;  
}
```

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

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

Control Flow

Boolean Expressions

$B \rightarrow B \text{ II } B \mid B \text{ && } B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$

Short-Circuit Code

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

- $E_1 \text{ II } E_2$ need not evaluate E_2 if E_1 is known to be true.
- $E_1 \text{ && } E_2$ need not evaluate E_2 if E_1 is known to be false.

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



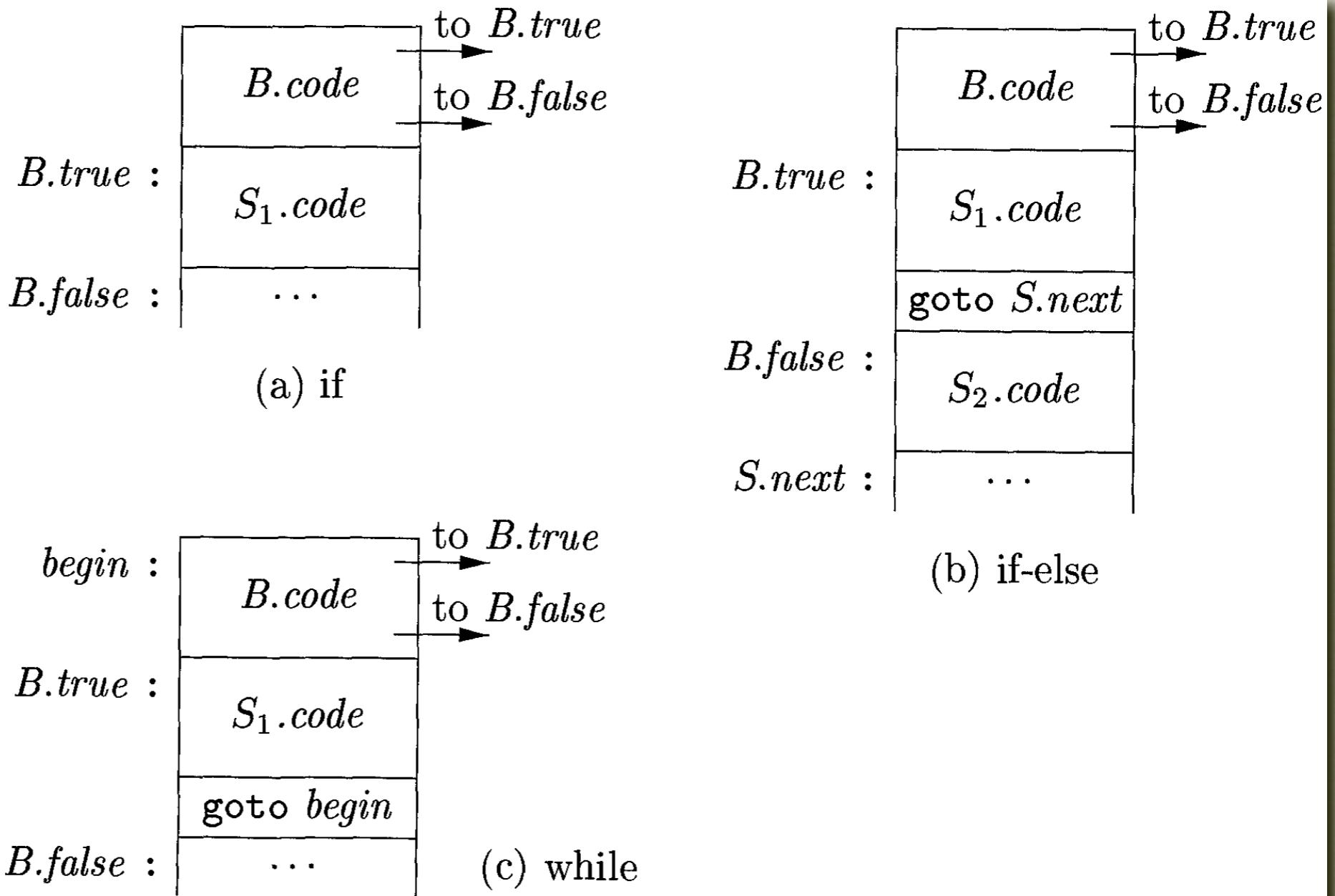
```
if x < 100 goto L2
ifFalse x > 200 goto L1
ifFalse x != y goto L1
```

```
L2: x = 0;
```

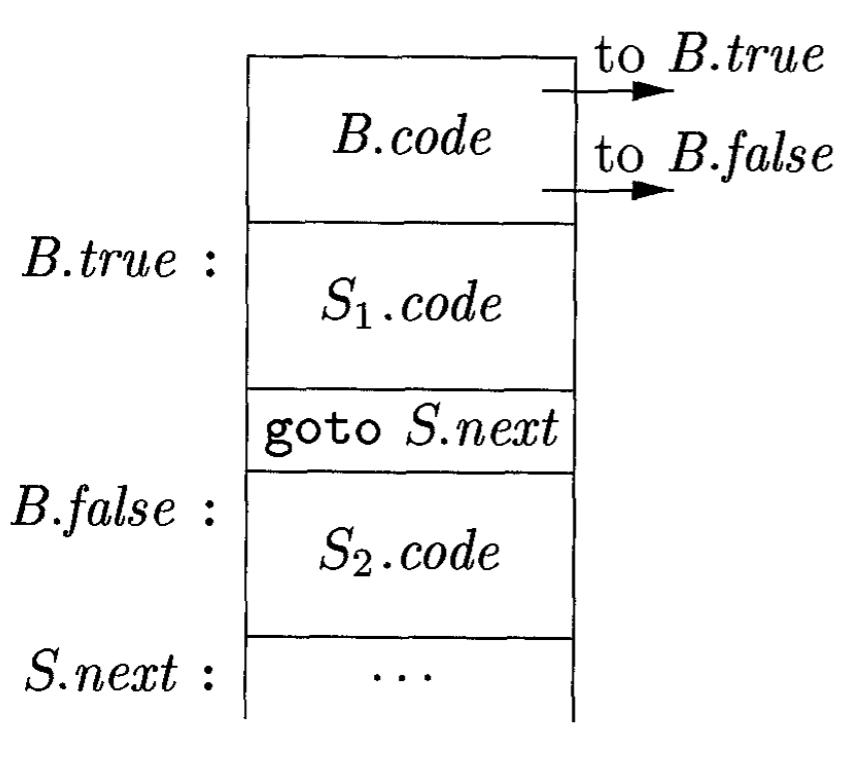
```
L1: ...
```

Flow-of-Control Statements

$S \rightarrow \text{if } (B) S_1$
 $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$
 $S \rightarrow \text{while } (B) S_1$



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

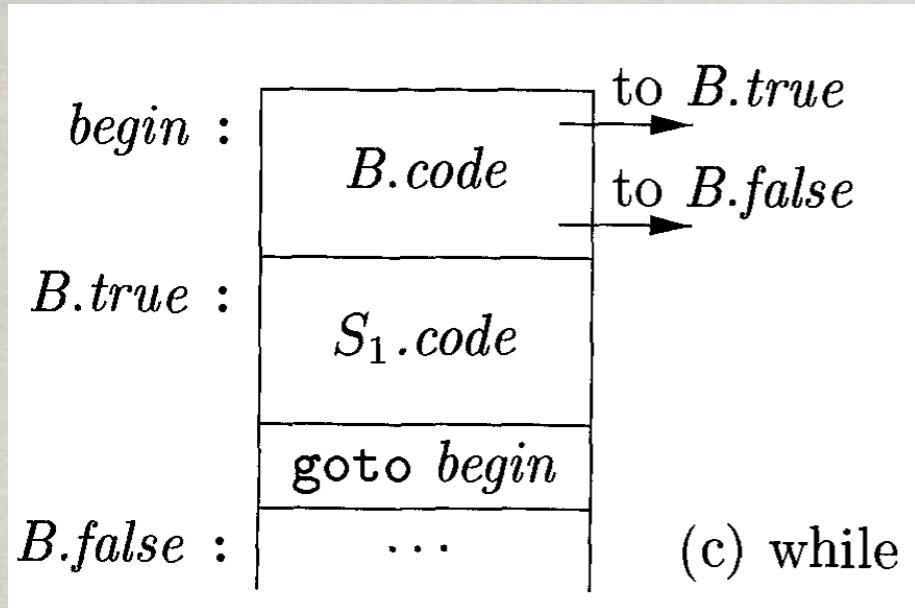


$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
-------------------	--------------------------------------------------------------------

$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$

$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel \text{gen('goto' } S.next)$ $\parallel label(B.false) \parallel S_2.code$

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



$$P \rightarrow S$$

$$\begin{cases} S.next = newlabel() \\ P.code = S.code \parallel label(S.next) \end{cases}$$

S \rightarrow **while** (*B*) *S₁*

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

$$S \rightarrow S_1 \ S_2$$
$$\begin{aligned} S_1.next &= newlabel() \\ S_2.next &= S.next \\ S.code &= S_1.code \parallel label(S_1.next) \parallel S_2.code \end{aligned}$$

CONCRETE SYNTAX	ABSTRACT SYNTAX
=	assign
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
-unary	minus
[]	access

: Concrete and abstract syntax for several Java operators.

```

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

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

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

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

$$B \rightarrow B_1 \parallel B_2$$

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

if B_1 is true, then B itself is true $\Rightarrow B_1.true = B.true$.

if B_1 is false, then B_2 must be evaluated, so we make $B_1.false$ 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.

$B \rightarrow B_1 \text{ || } B_2$

$B_1.\text{true} = B.\text{true}$
$B_1.\text{false} = \text{newlabel}()$
$B_2.\text{true} = B.\text{true}$
$B_2.\text{false} = B.\text{false}$
$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$

 $B \rightarrow B_1 \text{ && } B_2$

$B_1.\text{true} = \text{newlabel}()$
$B_1.\text{false} = B.\text{false}$
$B_2.\text{true} = B.\text{true}$
$B_2.\text{false} = B.\text{false}$
$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

$P \rightarrow S$

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

$S \rightarrow \text{if } (B) S_1$

$B.true = newlabel()$
 $B.false = S_1.next = S.next$
 $S.code = B.code \parallel label(B.true) \parallel S_1.code$

$B \rightarrow B_1 \parallel B_2$

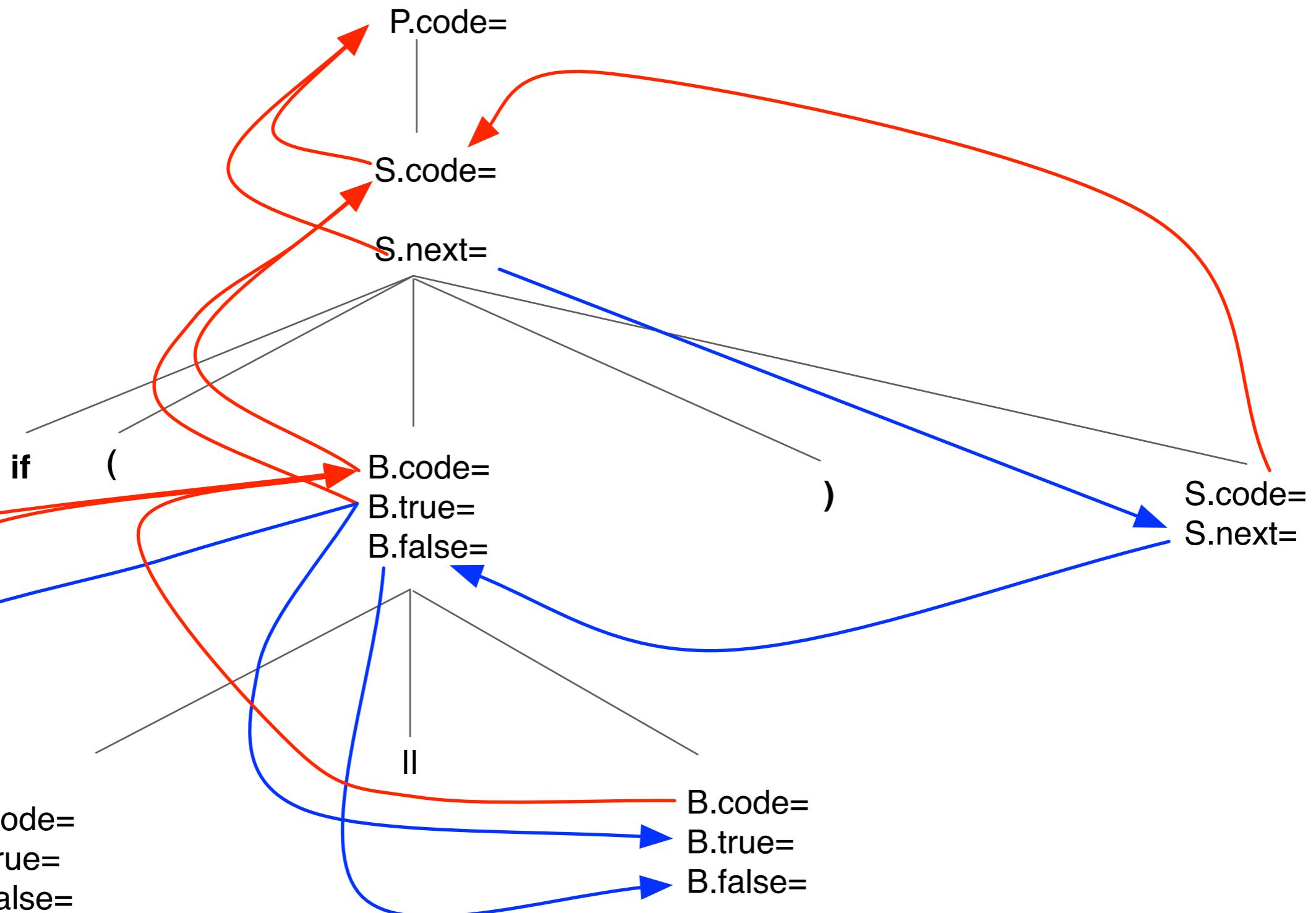
$B_1.true = B.true$

$B_1.false = newlabel()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \parallel label(B_1.false) \parallel B_2.code$



$P \rightarrow S$

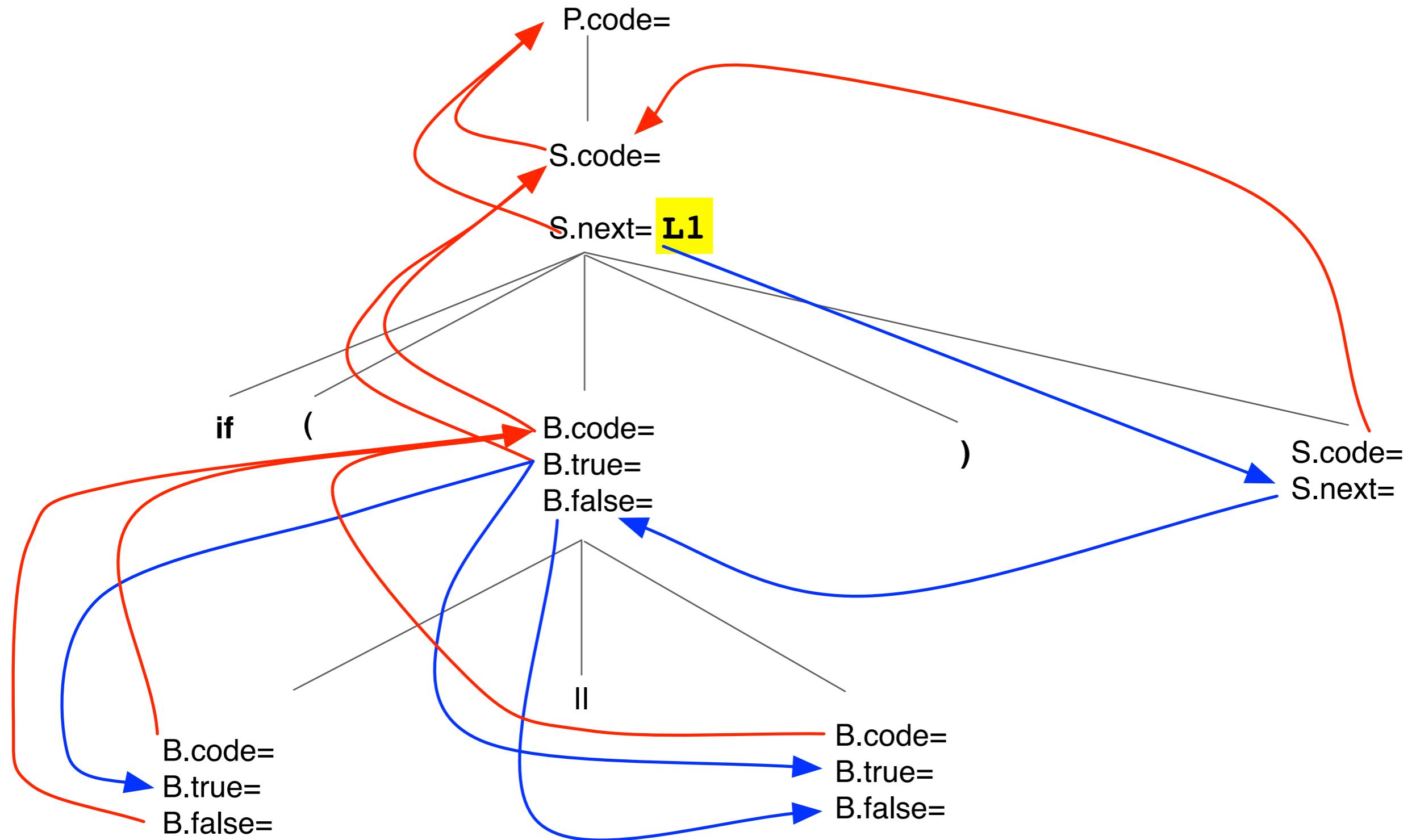
S.next = newlabel()	$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$
P.code = S.code label(S.next)		$B_1.false = newlabel()$

$S \rightarrow \text{if } (B) S_1$

B.true = newlabel()	$B_2.true = B.true$
B.false = $S_1.\text{next} = S.\text{next}$	$B_2.false = B.false$
S.code = B.code label(B.true) $S_1.\text{code}$	$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$

$B \rightarrow B_1 \parallel B_2$

$B_1.true = B.true$	$B_1.false = newlabel()$
$B_2.true = B.true$	$B_2.false = B.false$
$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$	



$P \rightarrow S$

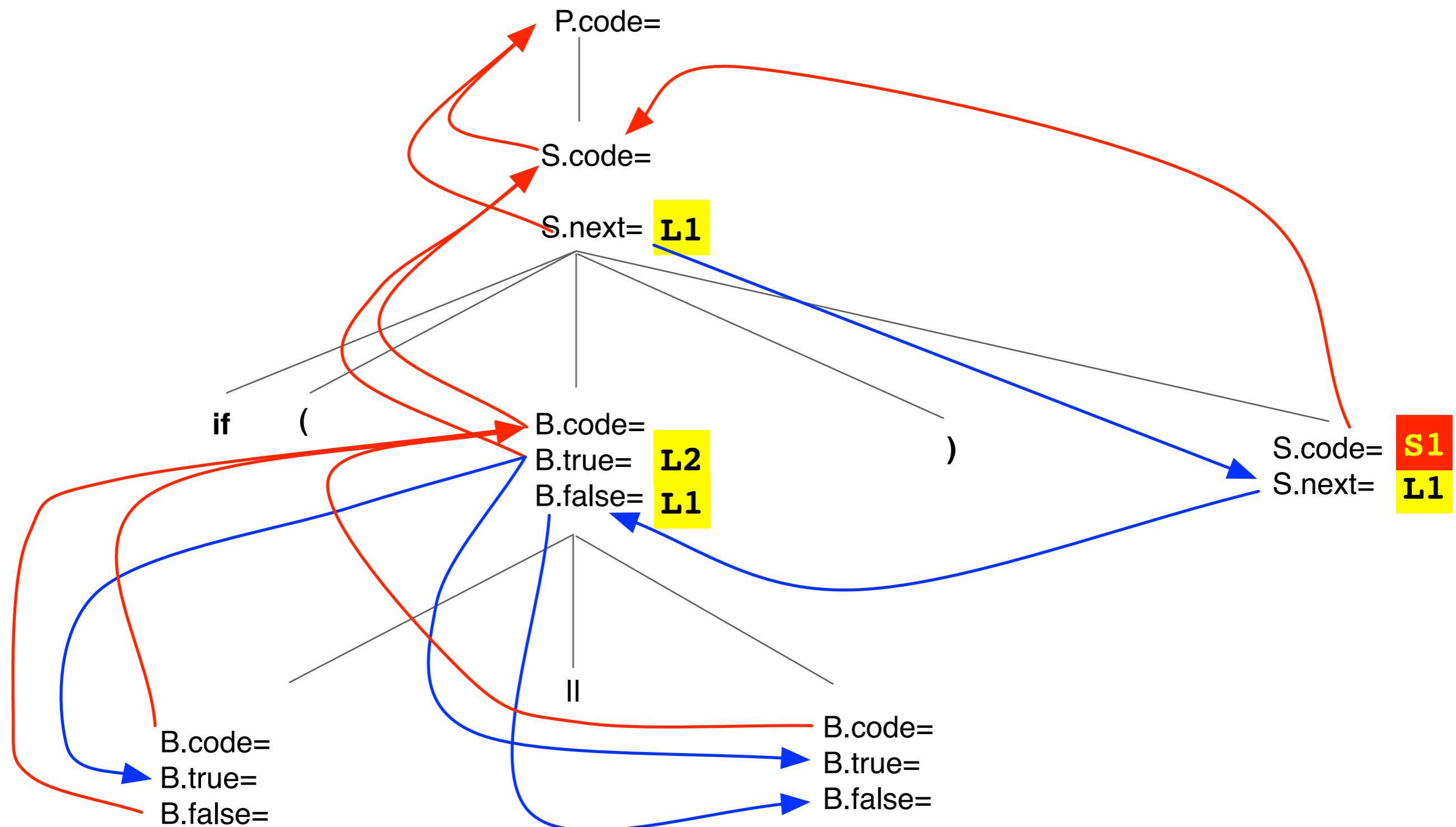
S.next = newlabel()	$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$
P.code = S.code label(S.next)		$B_1.false = newlabel()$

$S \rightarrow \text{if } (B) S_1$

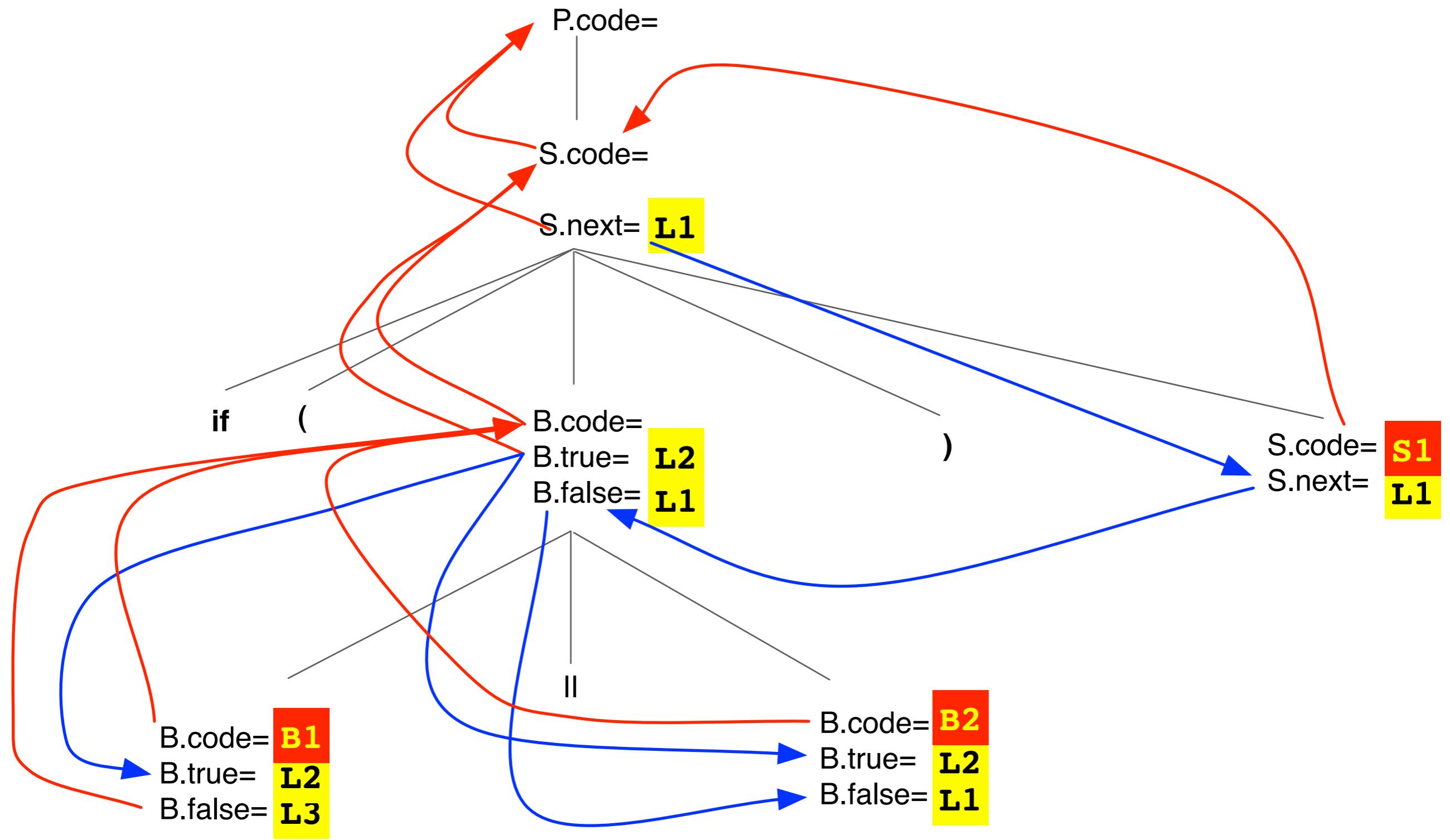
B.true = newlabel()	$B_2.true = B.true$
B.false = $S_1.\text{next} = S.\text{next}$	$B_2.false = B.false$
S.code = B.code label(B.true) $S_1.\text{code}$	$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$

$B \rightarrow B_1 \parallel B_2$

$B_1.true = B.true$	$B_1.false = newlabel()$
$B_2.true = B.true$	$B_2.false = B.false$
$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$	



$P \rightarrow S$	$S.\text{next} = \text{newlabel}()$ $P.\text{code} = S.\text{code} \parallel \text{label}(S.\text{next})$	$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$S \rightarrow \text{if (} B \text{) } S_1$	$B.\text{true} = \text{newlabel}()$ $B.\text{false} = S_1.\text{next} = S.\text{next}$ $S.\text{code} = B.\text{code} \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code}$		



$P \rightarrow S$

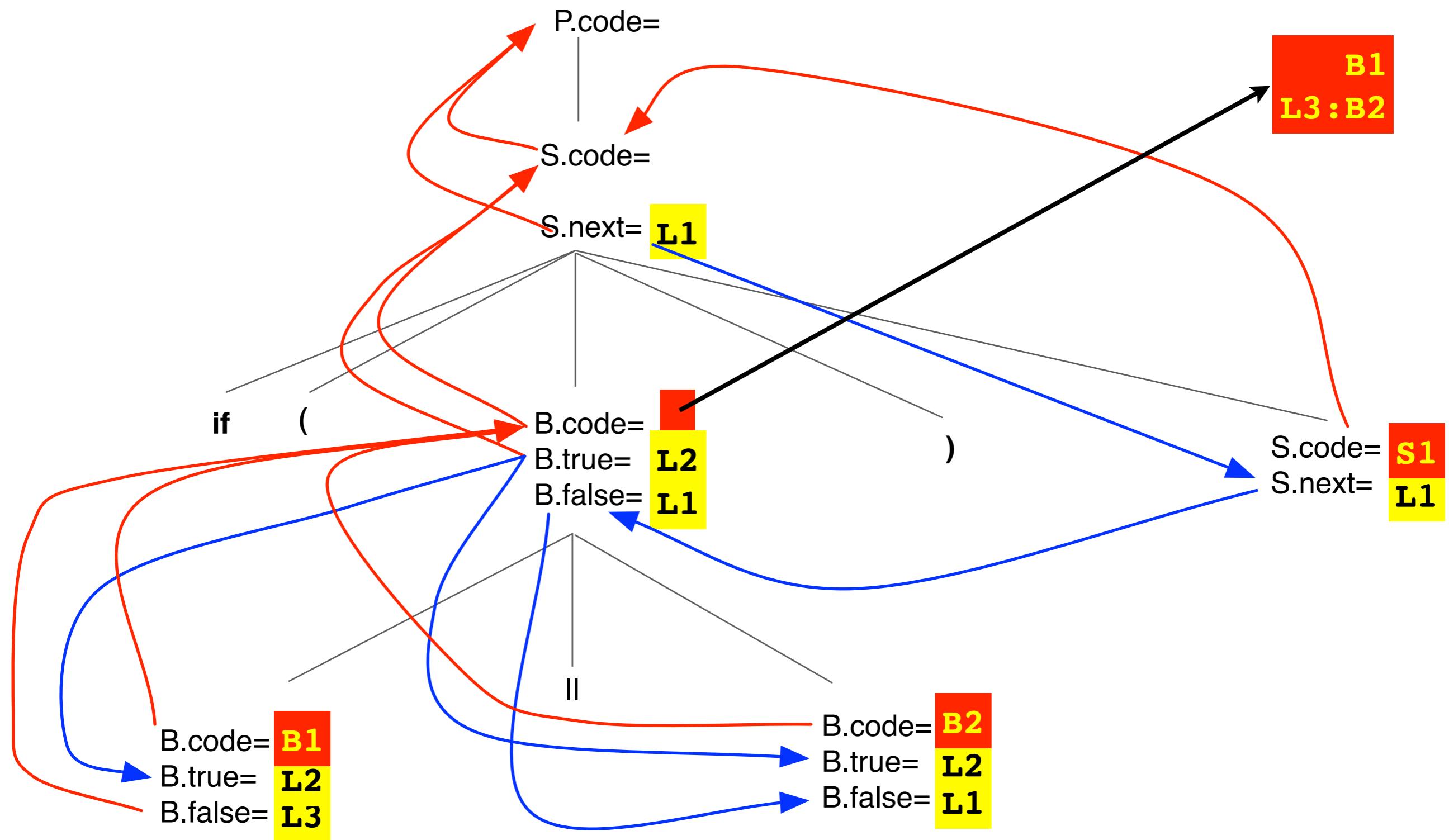
S.next = newlabel()	$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$
P.code = S.code label(S.next)		$B_1.false = newlabel()$

$S \rightarrow \text{if } (B) S_1$

B.true = newlabel()	$B_2.true = B.true$
B.false = $S_1.\text{next} = S.\text{next}$	$B_2.false = B.false$
S.code = B.code label(B.true) $S_1.\text{code}$	$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$

$B \rightarrow B_1 \parallel B_2$

$B_1.true = B.true$	$B_1.false = newlabel()$
$B_2.true = B.true$	$B_2.false = B.false$
$B.\text{code} = B_1.\text{code} \parallel label(B_1.false) \parallel B_2.\text{code}$	



$P \rightarrow S$

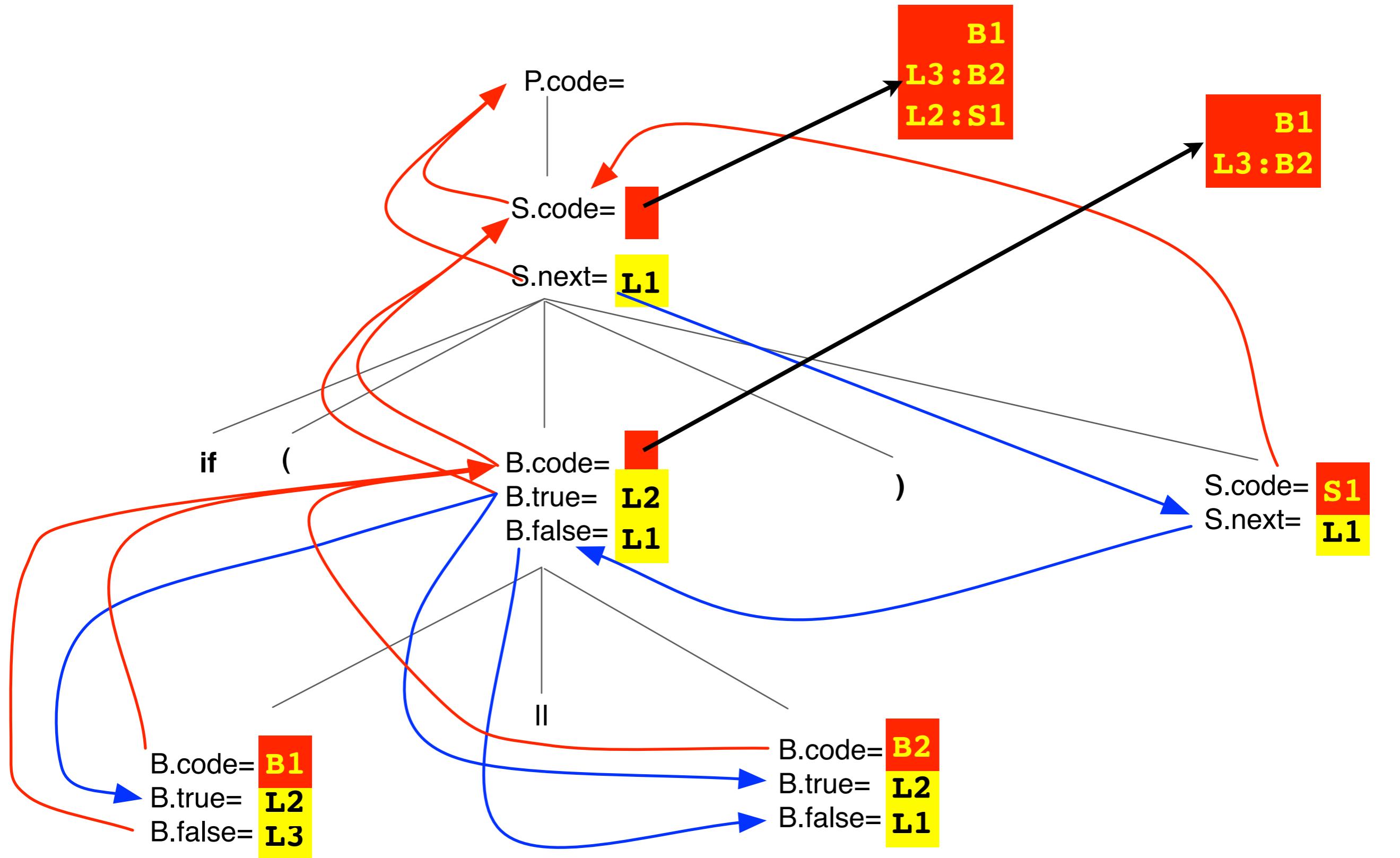
S.next = newlabel()	$B \rightarrow B_1 \parallel B_2$	$B_1.true = B.true$
P.code = S.code label(S.next)		$B_1.false = newlabel()$

$S \rightarrow \text{if } (B) S_1$

B.true = newlabel()	$B_2.true = B.true$
B.false = $S_1.\text{next} = S.\text{next}$	$B_2.false = B.false$
S.code = B.code label(B.true) $S_1.\text{code}$	$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.false) \parallel B_2.\text{code}$

$B \rightarrow B_1 \parallel B_2$

$B_1.true = B.true$	$B_1.false = newlabel()$
$B_2.true = B.true$	$B_2.false = B.false$
$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.false) \parallel B_2.\text{code}$	



$S \rightarrow \text{if} (B) S_1$

```
B.true = newlabel()
B.false = S1.next = S.next
S.code = B.code || label(B.true) || S1.code
```

$B \rightarrow B_1 \sqcup \bar{B}_2$

```
B1.true = B.true
B1.false = newlabel()
B2.true = B.true
B2.false = B.false
B.code = B1.code || label(B1.false) || B2.code
```

$B \rightarrow E_1 \text{ rel } E_2$

```
B.code = E1.code || E2.code
|| gen('if' E1.addr rel.op E2.addr 'goto' B.true)
|| gen('goto' B.false)
```

$B \rightarrow B_1 \&& B_2$

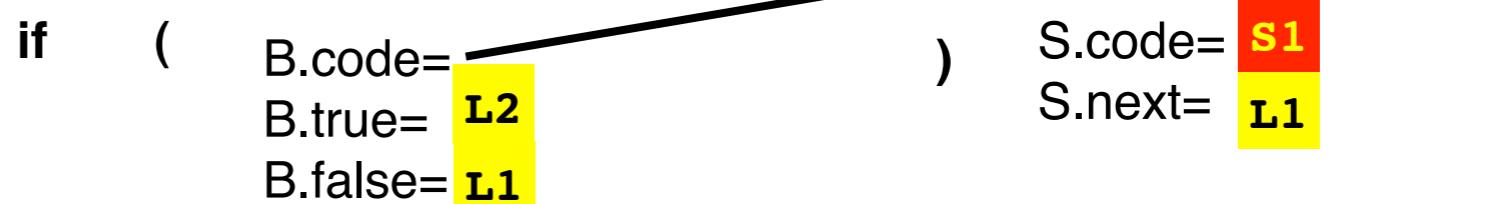
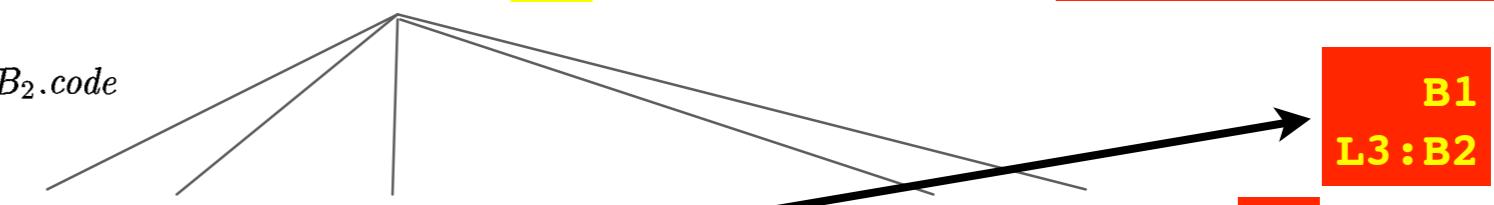
```
B1.true = newlabel()
B1.false = B.false
B2.true = B.true
B2.false = B.false
B.code = B1.code || label(B1.true) || B2.code
```

P.code =

S.code =
S.next = L₁

B₁
L₃:B₂
L₂:S₁

B₃
L₅:if(....) goto L₂
goto L₃
L₃:B₂
L₂:S₁



B₃
L₅: B₄

B.code = B₁
B.true = L₂
B.false = L₃

B.code = B₂
B.true = L₂
B.false = L₁

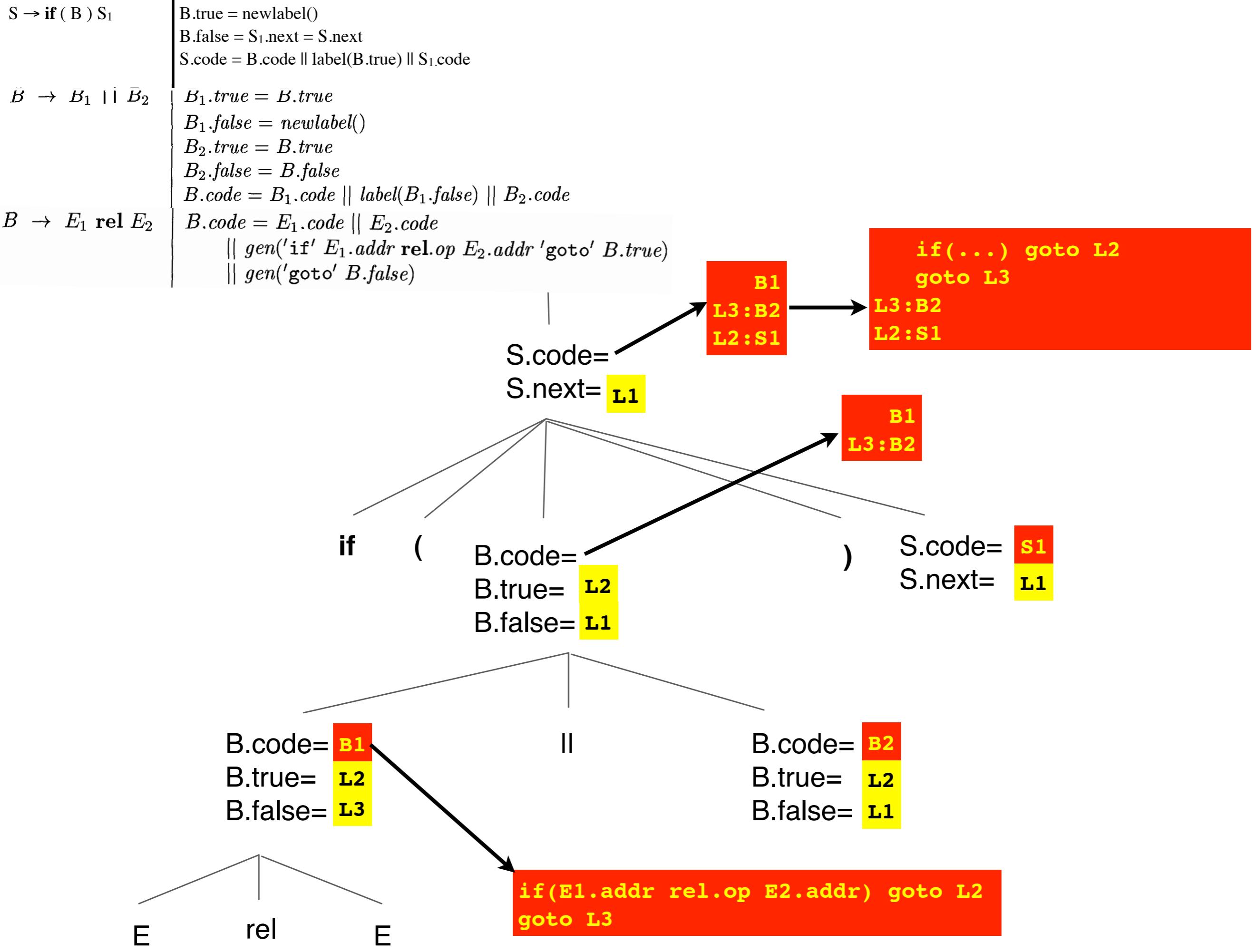
B.code = B₃
B.true = L₅
B.false = L₃

&&

B.code = B₄
B.true = L₂
B.false = L₃

if(E₁.addr rel.op E₂.addr) goto L₂
goto L₃

E rel E



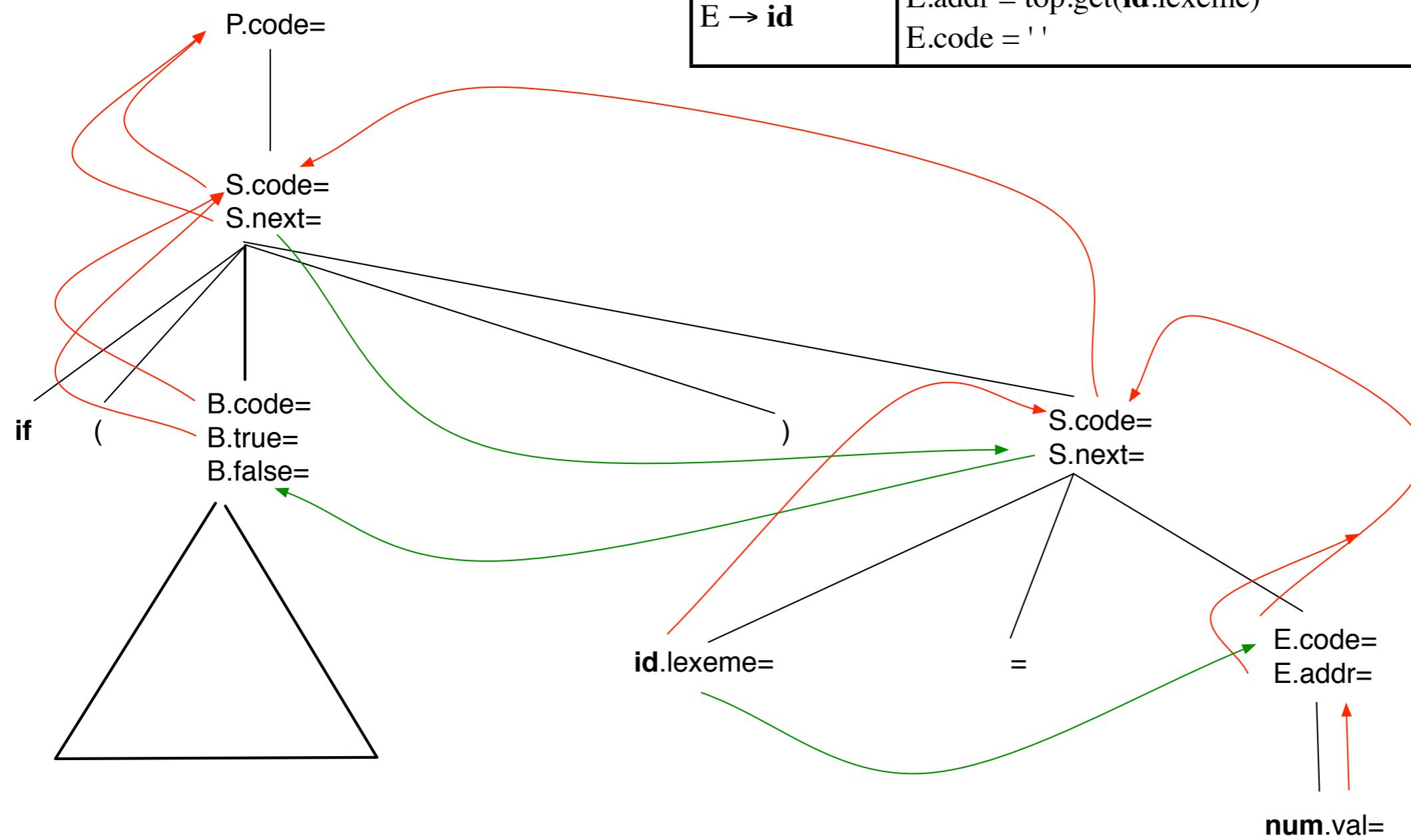
Esercizio

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

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

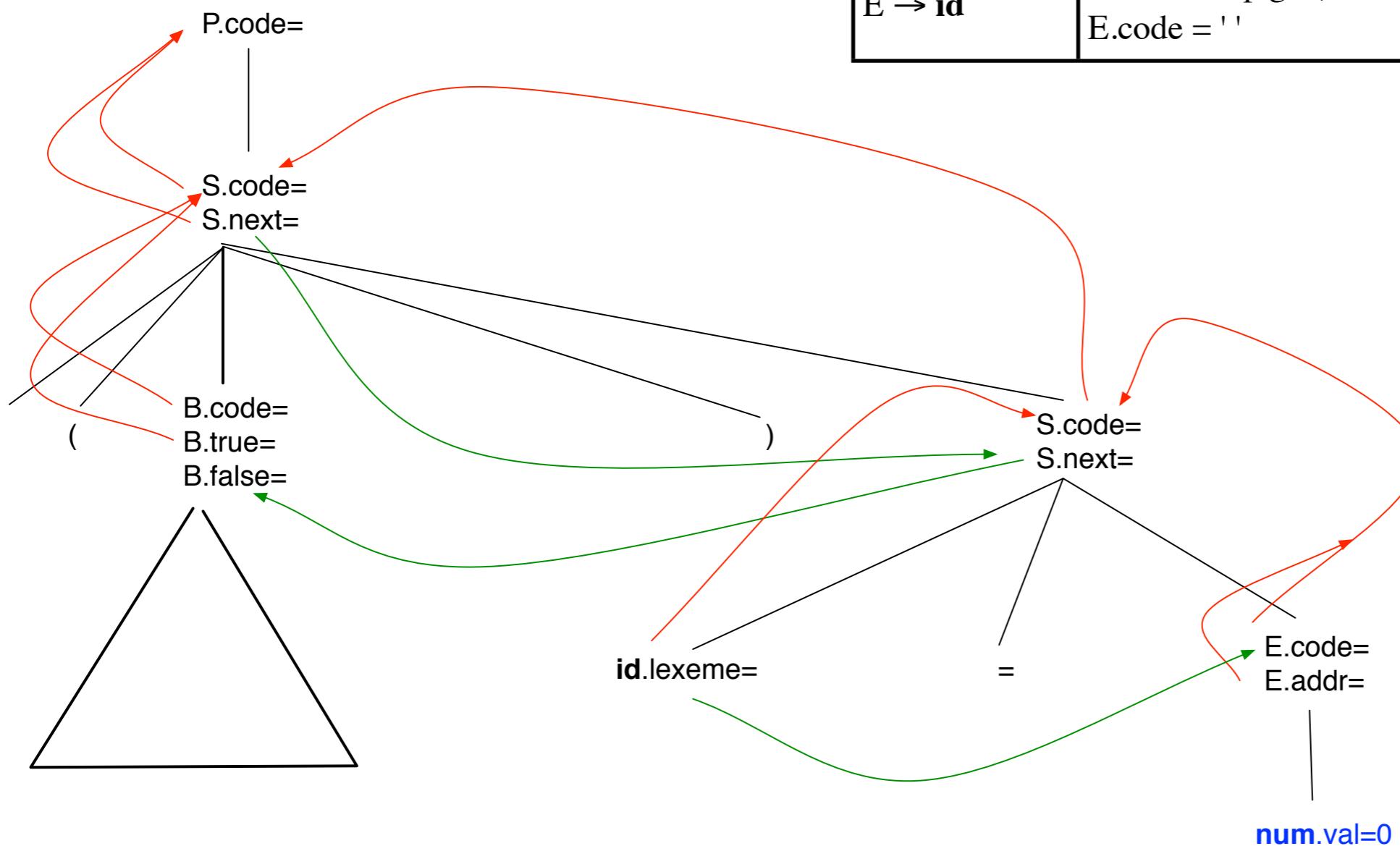
```
if(x<100 || x > 200 && x!=y) x=0;
```

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



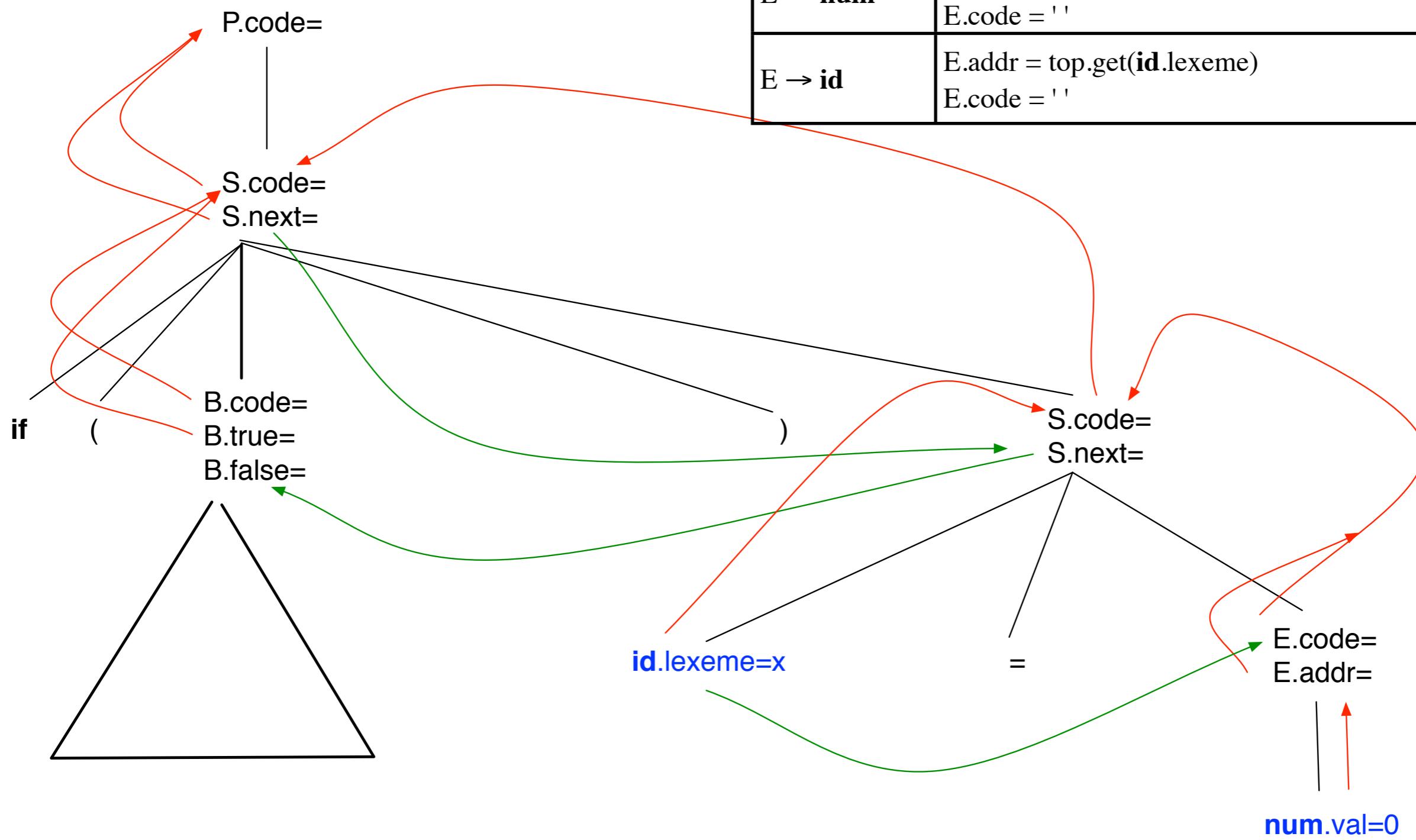
```
if( x<100 || x > 200 && x!=y ) x=0;
```

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



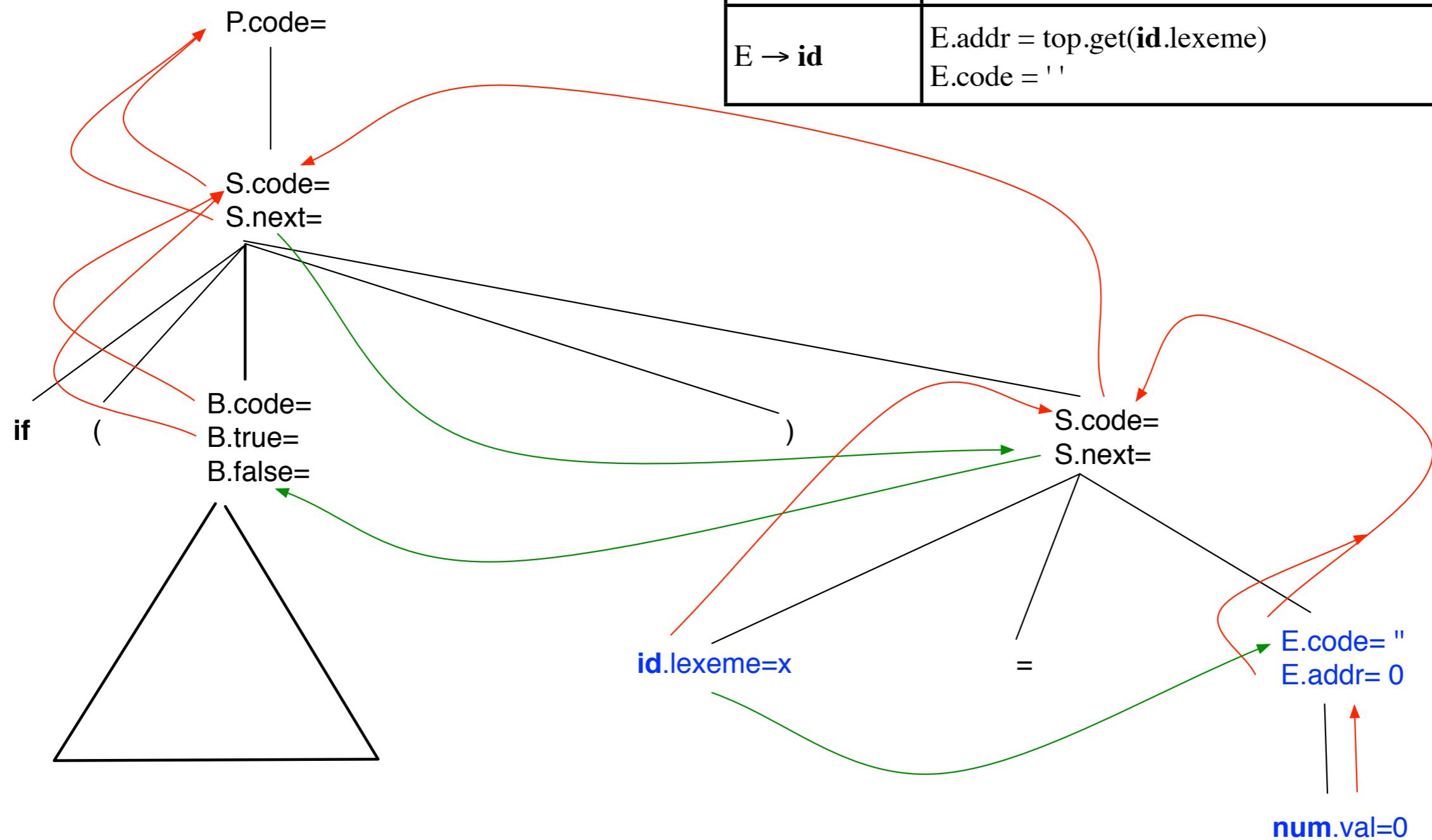
```
if(x<100 || x > 200 && x!=y) x=0;
```

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 = ''



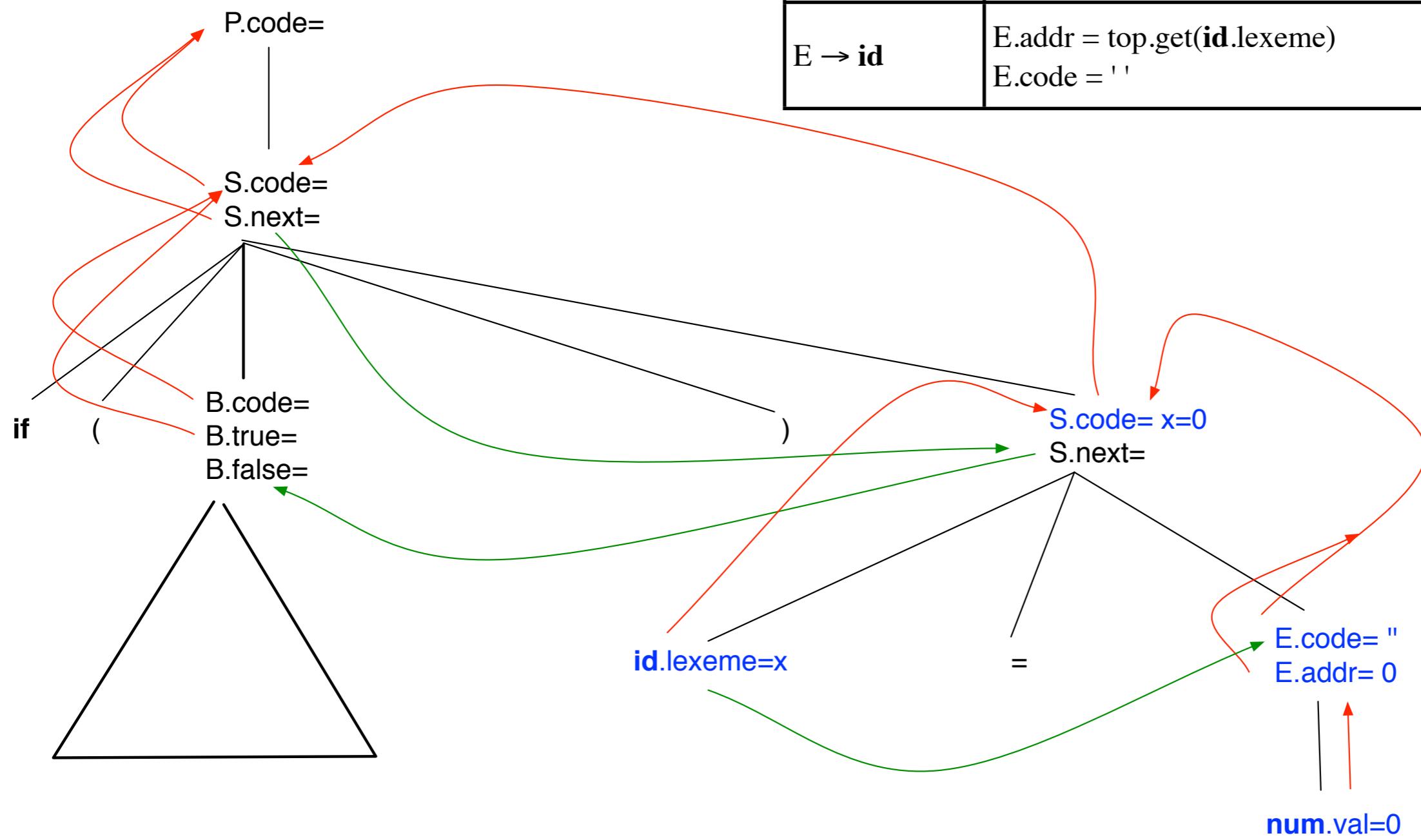
```
if( x<100 || x > 200 && x!=y ) x=0;
```

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



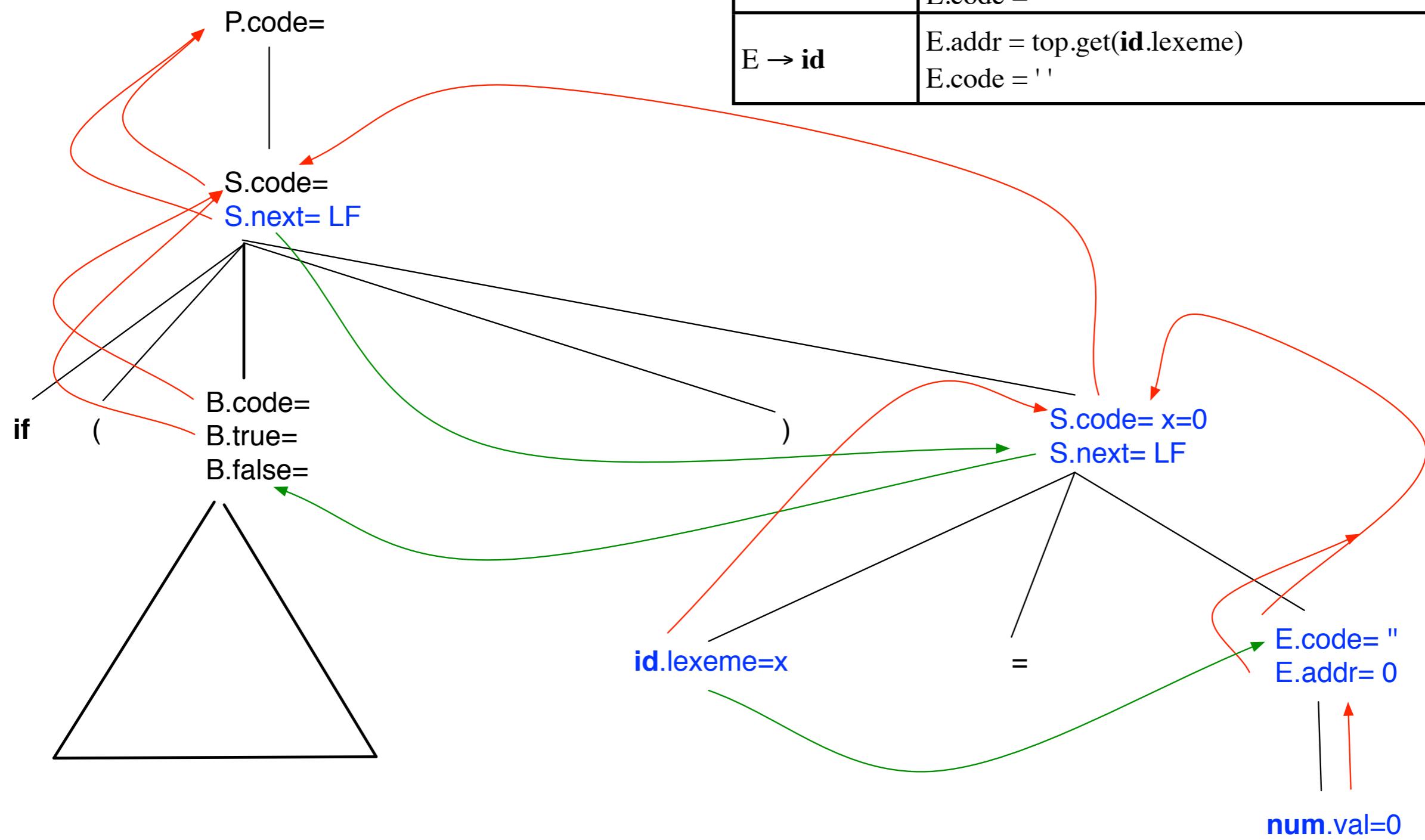
```
if( x<100 || x > 200 && x!=y ) x=0 ;
```

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



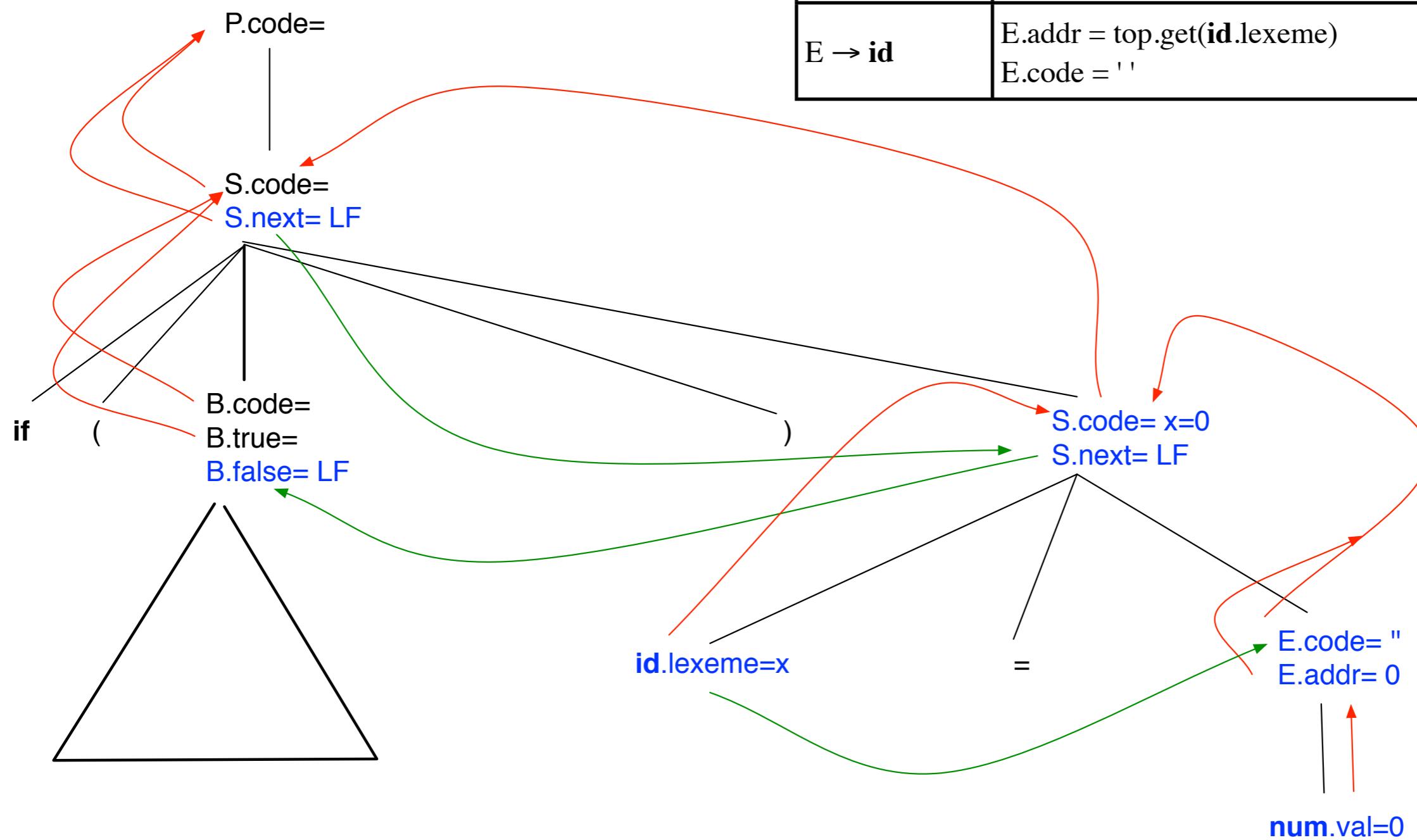
```
if( x<100 || x > 200 && x!=y ) x=0;
```

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

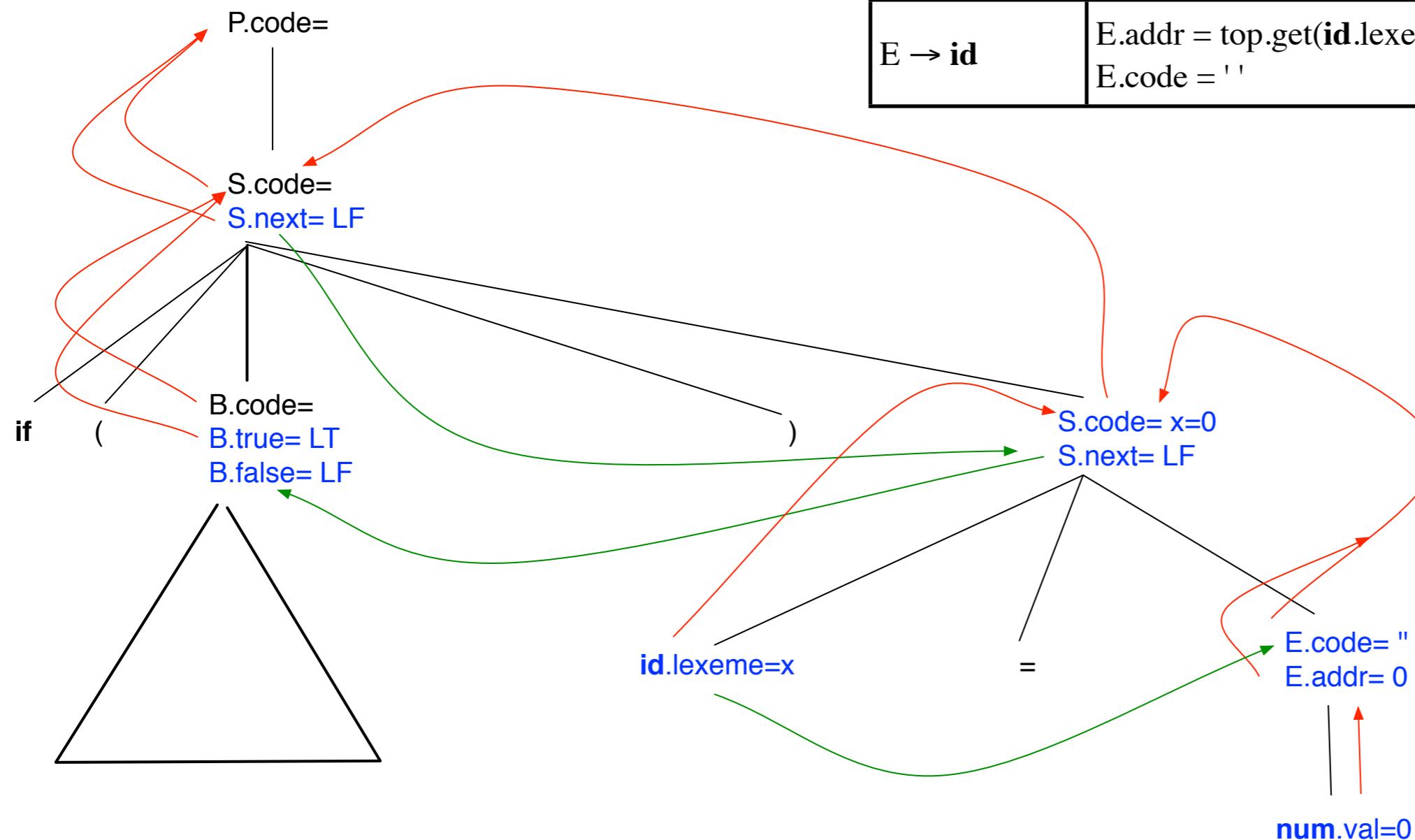


```
if(x<100 || x > 200 && x!=y) x=0;
```

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

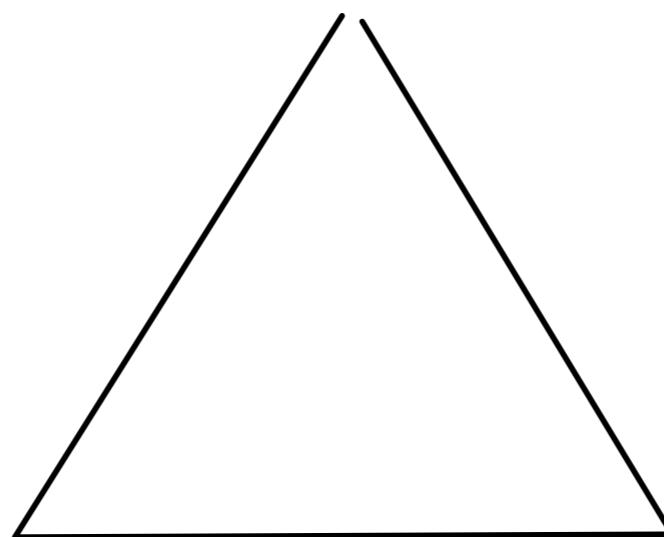


$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{id} = E;$	$S.code = E.code \parallel \text{gen}(\text{top.get(id.lexeme)}'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



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

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



$E \rightarrow \text{num}$

E.addr = num.val
E.code = ''

$E \rightarrow \text{id}$

E.addr = top.get(id.lexeme)
E.code = ''

$B \rightarrow B_1 \parallel B_2$

$B_1.\text{true} = B.\text{true}$
 $B_1.\text{false} = \text{newlabel}()$

$B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$

$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$

$B \rightarrow B_1 \&& B_2$

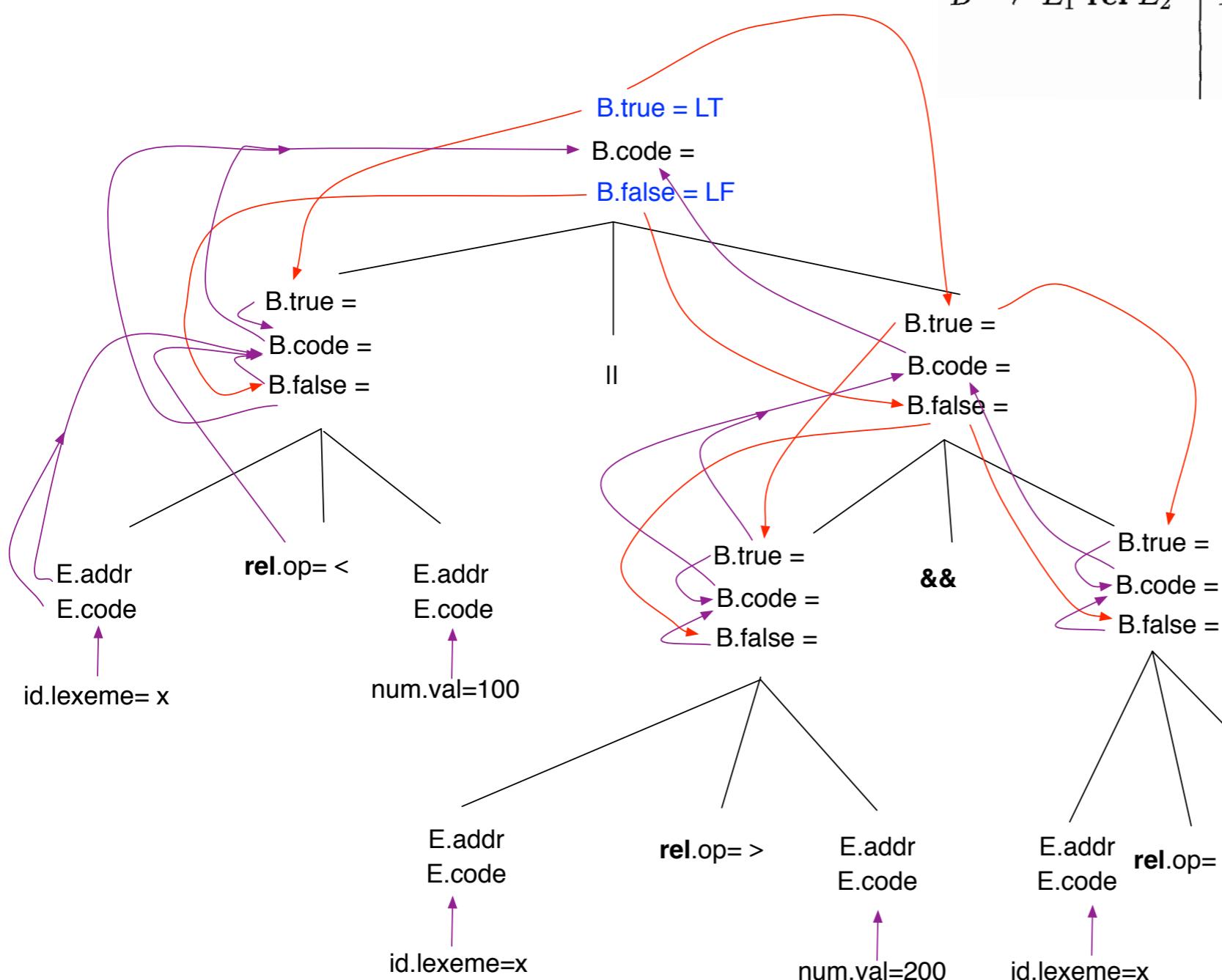
$B_1.\text{true} = \text{newlabel}()$
 $B_1.\text{false} = B.\text{false}$

$B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$

$B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

$B \rightarrow E_1 \text{ rel } E_2$

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



$E \rightarrow \text{num}$ | E.addr = num.val
 E.code = ''

$E \rightarrow \text{id}$ | E.addr = top.get(id.lexeme)
 E.code = ''

$B \rightarrow B_1 \mid\mid B_2$

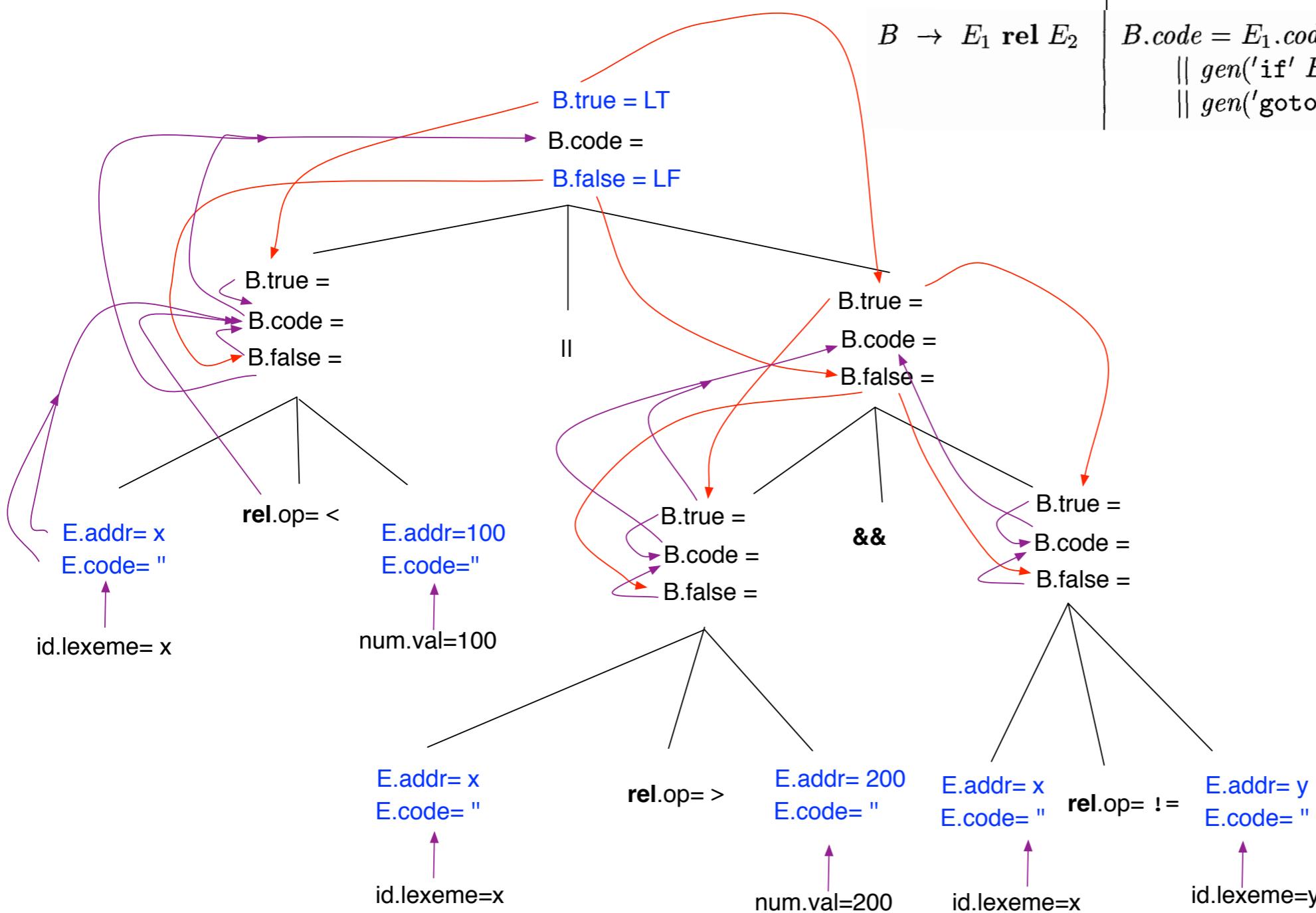
$B_1.\text{true} = B.\text{true}$
 $B_1.\text{false} = \text{newlabel}()$
 $B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$
 $B.\text{code} = B_1.\text{code} \mid\mid \text{label}(B_1.\text{false}) \mid\mid B_2.\text{code}$

$B \rightarrow B_1 \And B_2$

$B_1.\text{true} = \text{newlabel}()$
 $B_1.\text{false} = B.\text{false}$
 $B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$
 $B.\text{code} = B_1.\text{code} \mid\mid \text{label}(B_1.\text{true}) \mid\mid B_2.\text{code}$

$B \rightarrow E_1 \text{ rel } E_2$

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



$E \rightarrow \text{num}$	$E.\text{addr} = \text{num}.val$ $E.\text{code} = ''$	$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get(id.lexeme)}$ $E.\text{code} = ''$	$B \rightarrow B_1 \& B_2$	$B_1.\text{true} = \text{newlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$
		$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$ $\parallel \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\parallel \text{gen('goto' } B.\text{false})$
			$C1 =$ $\text{if } x < 100 \text{ goto LT}$ goto L_1

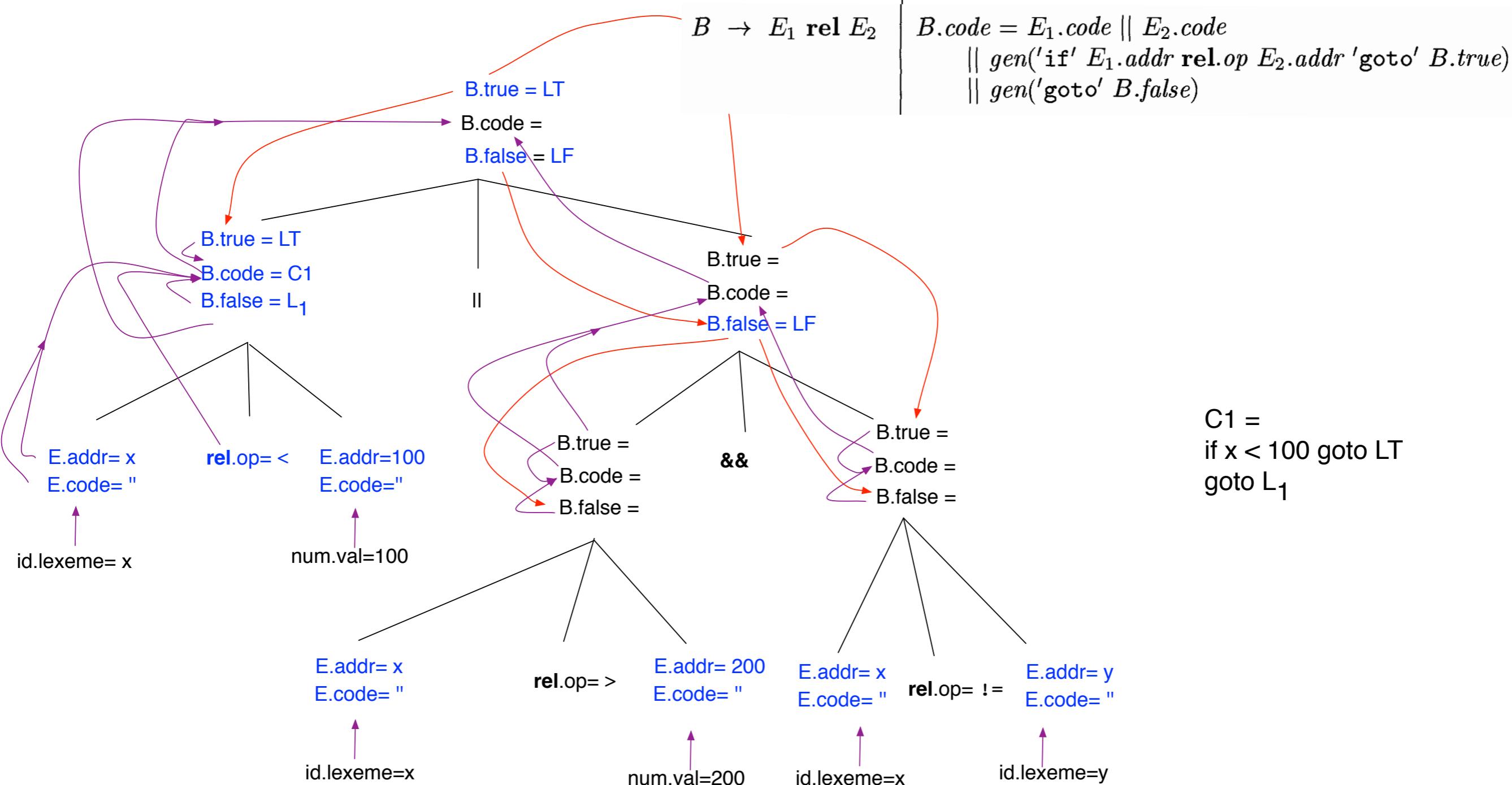
<img alt="A parse tree diagram for a relational expression. The root node is B → E1 rel E2. It branches into E1 and rel E2. E1 further branches into E2 and rel op E2. E2 branches into E3 and E4. E3 branches into E5 and E6. E5 branches into E7 and E8. E6 branches into E9 and E10. E7 branches into E11 and E12. E8 branches into E13 and E14. E11 branches into E15 and E16. E12 branches into E17 and E18. E15 branches into E19 and E20. E16 branches into E21 and E22. E19 branches into E23 and E24. E20 branches into E25 and E26. E23 branches into E27 and E28. E24 branches into E29 and E30. E27 branches into E31 and E32. E28 branches into E33 and E34. E29 branches into E35 and E36. E30 branches into E37 and E38. E31 branches into E39 and E40. E32 branches into E41 and E42. E33 branches into E43 and E44. E34 branches into E45 and E46. E35 branches into E47 and E48. E36 branches into E49 and E50. E37 branches into E51 and E52. E38 branches into E53 and E54. E39 branches into E55 and E56. E40 branches into E57 and E58. E41 branches into E59 and E60. E42 branches into E61 and E62. E43 branches into E63 and E64. E44 branches into E65 and E66. E45 branches into E67 and E68. E46 branches into E69 and E70. E47 branches into E71 and E72. E48 branches into E73 and E74. E49 branches into E75 and E76. E50 branches into E77 and E78. E43 branches into E79 and E80. E44 branches into E81 and E82. E45 branches into E83 and E84. E46 branches into E85 and E86. E47 branches into E87 and E88. E48 branches into E89 and E90. E49 branches into E91 and E92. E50 branches into E93 and E94. E51 branches into E95 and E96. E52 branches into E97 and E98. E53 branches into E99 and E100. E54 branches into E101 and E102. E55 branches into E103 and E104. E56 branches into E105 and E106. E57 branches into E107 and E108. E58 branches into E109 and E110. E59 branches into E111 and E112. E60 branches into E113 and E114. E61 branches into E115 and E116. E62 branches into E117 and E118. E63 branches into E119 and E120. E64 branches into E121 and E122. E65 branches into E123 and E124. E66 branches into E125 and E126. E67 branches into E127 and E128. E68 branches into E129 and E130. E69 branches into E131 and E132. E70 branches into E133 and E134. E63 branches into E135 and E136. E64 branches into E137 and E138. E65 branches into E139 and E140. E66 branches into E141 and E142. E67 branches into E143 and E144. E68 branches into E145 and E146. E69 branches into E147 and E148. E70 branches into E149 and E150. E63 branches into E151 and E152. E64 branches into E153 and E154. E65 branches into E155 and E156. E66 branches into E157 and E158. E67 branches into E159 and E160. E68 branches into E161 and E162. E69 branches into E163 and E164. E70 branches into E165 and E166. E63 branches into E167 and E168. E64 branches into E169 and E170. E65 branches into E171 and E172. E66 branches into E173 and E174. E67 branches into E175 and E176. E68 branches into E177 and E178. E69 branches into E179 and E180. E70 branches into E181 and E182. E63 branches into E183 and E184. E64 branches into E185 and E186. E65 branches into E187 and E188. E66 branches into E189 and E190. E67 branches into E191 and E192. E68 branches into E193 and E194. E69 branches into E195 and E196. E70 branches into E197 and E198. E63 branches into E199 and E200. E64 branches into E201 and E202. E65 branches into E203 and E204. E66 branches into E205 and E206. E67 branches into E207 and E208. E68 branches into E209 and E210. E69 branches into E211 and E212. E70 branches into E213 and E214. E63 branches into E215 and E216. E64 branches into E217 and E218. E65 branches into E219 and E220. E66 branches into E221 and E222. E67 branches into E223 and E224. E68 branches into E225 and E226. E69 branches into E227 and E228. E70 branches into E229 and E230. E63 branches into E231 and E232. E64 branches into E233 and E234. E65 branches into E235 and E236. E66 branches into E237 and E238. E67 branches into E239 and E240. E68 branches into E241 and E242. E69 branches into E243 and E244. E70 branches into E245 and E246. E63 branches into E247 and E248. E64 branches into E249 and E250. E65 branches into E251 and E252. E66 branches into E253 and E254. E67 branches into E255 and E256. E68 branches into E257 and E258. E69 branches into E259 and E260. E70 branches into E261 and E262. E63 branches into E263 and E264. E64 branches into E265 and E266. E65 branches into E267 and E268. E66 branches into E269 and E270. E67 branches into E271 and E272. E68 branches into E273 and E274. E69 branches into E275 and E276. E70 branches into E277 and E278. E63 branches into E279 and E280. E64 branches into E281 and E282. E65 branches into E283 and E284. E66 branches into E285 and E286. E67 branches into E287 and E288. E68 branches into E289 and E290. E69 branches into E291 and E292. E70 branches into E293 and E294. E63 branches into E295 and E296. E64 branches into E297 and E298. E65 branches into E299 and E300. E66 branches into E301 and E302. E67 branches into E303 and E304. E68 branches into E305 and E306. E69 branches into E307 and E308. E70 branches into E309 and E310. E63 branches into E311 and E312. E64 branches into E313 and E314. E65 branches into E315 and E316. E66 branches into E317 and E318. E67 branches into E319 and E320. E68 branches into E321 and E322. E69 branches into E323 and E324. E70 branches into E325 and E326. E63 branches into E327 and E328. E64 branches into E329 and E330. E65 branches into E331 and E332. E66 branches into E333 and E334. E67 branches into E335 and E336. E68 branches into E337 and E338. E69 branches into E339 and E340. E70 branches into E341 and E342. E63 branches into E343 and E344. E64 branches into E345 and E346. E65 branches into E347 and E348. E66 branches into E349 and E350. E67 branches into E351 and E352. E68 branches into E353 and E354. E69 branches into E355 and E356. E70 branches into E357 and E358. E63 branches into E359 and E360. E64 branches into E361 and E362. E65 branches into E363 and E364. E66 branches into E365 and E366. E67 branches into E367 and E368. E68 branches into E369 and E370. E69 branches into E371 and E372. E70 branches into E373 and E374. E63 branches into E375 and E376. E64 branches into E377 and E378. E65 branches into E379 and E380. E66 branches into E381 and E382. E67 branches into E383 and E384. E68 branches into E385 and E386. E69 branches into E387 and E388. E70 branches into E389 and E390. E63 branches into E391 and E392. E64 branches into E393 and E394. E65 branches into E395 and E396. E66 branches into E397 and E398. E67 branches into E399 and E400. E68 branches into E401 and E402. E69 branches into E403 and E404. E70 branches into E405 and E406. E63 branches into E407 and E408. E64 branches into E409 and E4010. E65 branches into E4011 and E4012. E66 branches into E4013 and E4014. E67 branches into E4015 and E4016. E68 branches into E4017 and E4018. E69 branches into E4019 and E4020. E70 branches into E4021 and E4022. E63 branches into E4023 and E4024. E64 branches into E4025 and E4026. E65 branches into E4027 and E4028. E66 branches into E4029 and E4030. E67 branches into E4031 and E4032. E68 branches into E4033 and E4034. E69 branches into E4035 and E4036. E70 branches into E4037 and E4038. E63 branches into E4039 and E4040. E64 branches into E4041 and E4042. E65 branches into E4043 and E4044. E66 branches into E4045 and E4046. E67 branches into E4047 and E4048. E68 branches into E4049 and E4050. E69 branches into E4051 and E4052. E70 branches into E4053 and E4054. E63 branches into E4055 and E4056. E64 branches into E4057 and E4058. E65 branches into E4059 and E4060. E66 branches into E4061 and E4062. E67 branches into E4063 and E4064. E68 branches into E4065 and E4066. E69 branches into E4067 and E4068. E70 branches into E4069 and E4070. E63 branches into E4071 and E4072. E64 branches into E4073 and E4074. E65 branches into E4075 and E4076. E66 branches into E4077 and E4078. E67 branches into E4079 and E4080. E68 branches into E4081 and E4082. E69 branches into E4083 and E4084. E70 branches into E4085 and E4086. E63 branches into E4087 and E4088. E64 branches into E4089 and E4090. E65 branches into E4091 and E4092. E66 branches into E4093 and E4094. E67 branches into E4095 and E4096. E68 branches into E4097 and E4098. E69 branches into E4099 and E4100. E70 branches into E4101 and E4102. E63 branches into E4103 and E4104. E64 branches into E4105 and E4106. E65 branches into E4107 and E4108. E66 branches into E4109 and E4110. E67 branches into E4111 and E4112. E68 branches into E4113 and E4114. E69 branches into E4115 and E4116. E70 branches into E4117 and E4118. E63 branches into E4119 and E4120. E64 branches into E4121 and E4122. E65 branches into E4123 and E4124. E66 branches into E4125 and E4126. E67 branches into E4127 and E4128. E68 branches into E4129 and E4130. E69 branches into E4131 and E4132. E70 branches into E4133 and E4134. E63 branches into E4135 and E4136. E64 branches into E4137 and E4138. E65 branches into E4139 and E4140. E66 branches into E4141 and E4142. E67 branches into E4143 and E4144. E68 branches into E4145 and E4146. E69 branches into E4147 and E4148. E70 branches into E4149 and E4150. E63 branches into E4151 and E4152. E64 branches into E4153 and E4154. E65 branches into E4155 and E4156. E66 branches into E4157 and E4158. E67 branches into E4159 and E4160. E68 branches into E4161 and E4162. E69 branches into E4163 and E4164. E70 branches into E4165 and E4166. E63 branches into E4167 and E4168. E64 branches into E4169 and E4170. E65 branches into E4171 and E4172. E66 branches into E4173 and E4174. E67 branches into E4175 and E4176. E68 branches into E4177 and E4178. E69 branches into E4179 and E4180. E70 branches into E4181 and E4182. E63 branches into E4183 and E4184. E64 branches into E4185 and E4186. E65 branches into E4187 and E4188. E66 branches into E4189 and E4190. E67 branches into E4191 and E4192. E68 branches into E4193 and E4194. E69 branches into E4195 and E4196. E70 branches into E4197 and E4198. E63 branches into E4199 and E4200. E64 branches into E4201 and E4202. E65 branches into E4203 and E4204. E66 branches into E4205 and E4206. E67 branches into E4207 and E4208. E68 branches into E4209 and E4210. E69 branches into E4211 and E4212. E70 branches into E4213 and E4214. E63 branches into E4215 and E4216. E64 branches into E4217 and E4218. E65 branches into E4219 and E4220. E66 branches into E4221 and E4222. E67 branches into E4223 and E4224. E68 branches into E4225 and E4226. E69 branches into E4227 and E4228. E70 branches into E4229 and E4230. E63 branches into E4231 and E4232. E64 branches into E4233 and E4234. E65 branches into E4235 and E4236. E66 branches into E4237 and E4238. E67 branches into E4239 and E4240. E68 branches into E4241 and E4242. E69 branches into E4243 and E4244. E70 branches into E4245 and E4246. E63 branches into E4247 and E4248. E64 branches into E4249 and E4250. E65 branches into E4251 and E4252. E66 branches into E4253 and E4254. E67 branches into E4255 and E4256. E68 branches into E4257 and E4258. E69 branches into E4259 and E4260. E70 branches into E4261 and E4262. E63 branches into E4263 and E4264. E64 branches into E4265 and E4266. E65 branches into E4267 and E4268. E66 branches into E4269 and E4270. E67 branches into E4271 and E4272. E68 branches into E4273 and E4274. E69 branches into E4275 and E4276. E70 branches into E4277 and E4278. E63 branches into E4279 and E4280. E64 branches into E4281 and E4282. E65 branches into E4283 and E4284. E66 branches into E4285 and E4286. E67 branches into E4287 and E4288. E68 branches into E4289 and E4290. E69 branches into E4291 and E4292. E70 branches into E4293 and E4294. E63 branches into E4295 and E4296. E64 branches into E4297 and E4298. E65 branches into E4299 and E4300. E66 branches into E4301 and E4302. E67 branches into E4303 and E4304. E68 branches into E4305 and E4306. E69 branches into E4307 and E4308. E70 branches into E4309 and E4310. E63 branches into E4311 and E4312. E64 branches into E4313 and E4314. E65 branches into E4315 and E4316. E66 branches into E4317 and E4318. E67 branches into E4319 and E4320. E68 branches into E4321 and E4322. E69 branches into E4323 and E4324. E70 branches into E4325 and E4326. E63 branches into E4327 and E4328. E64 branches into E4329 and E4330. E65 branches into E4331 and E4332. E66 branches into E4333 and E4334. E67 branches into E4335 and E4336. E68 branches into E4337 and E4338. E69 branches into E4339 and E4340. E70 branches into E4341 and E4342. E63 branches into E4343 and E4344. E64 branches into E4345 and E4346. E65 branches into E4347 and E4348. E66 branches into E4349 and E4350. E67 branches into E4351 and E4352. E68 branches into E4353 and E4354. E69 branches into E4355 and E4356. E70 branches into E4357 and E4358. E63 branches into E4359 and E4360. E64 branches into E4361 and E4362. E65 branches into E4363 and E4364. E66 branches into E4365 and E4366. E67 branches into E4367 and E4368. E68 branches into E4369 and E4370. E69 branches into E4371 and E4372. E70 branches into E4373 and E4374. E63 branches into E4375 and E4376. E64 branches into E4377 and E4378. E65 branches into E4379 and E4380. E66 branches into E4381 and E4382. E67 branches into E4383 and E4384. E68 branches into E4385 and E4386. E69 branches into E4387 and E4388. E70 branches into E4389 and E4390. E63 branches into E4391 and E4392. E64 branches into E4393 and E4394. E65 branches into E4395 and E4396. E66 branches into E4397 and E4398. E67 branches into E4399 and E4400. E68 branches into E4401 and E4402. E69 branches into E4403 and E4404. E70 branches into E4405 and E4406. E63 branches into E4407 and E4408. E64 branches into E4409 and E4410. E65 branches into E4411 and E4412. E66 branches into E4413 and E4414. E67 branches into E4415 and E4416. E68 branches into E4417 and E4418. E69 branches into E4419 and E4420. E70 branches into E4421 and E4422. E63 branches into E4423 and E4424. E64 branches into E4425 and E4426. E65 branches into E4427 and E4428. E66 branches into E4429 and E4430. E67 branches into E4431 and E4432. E68 branches into E4433 and E4434. E69 branches into E4435 and E4436. E70 branches into E4437 and E4438. E63 branches into E4439 and E4440. E64 branches into E4441 and E4442. E65 branches into E4443 and E4444. E66 branches into E4445 and E4446. E67 branches into E4447 and E4448. E68 branches into E4449 and E4450. E69 branches into E4451 and E4452. E70 branches into E4453 and E4454. E63 branches into E4455 and E4456. E64 branches into E4457 and E4458. E65 branches into E4459 and E4460. E66 branches into E4461 and E4462. E67 branches into E4463 and E4464. E68 branches into E4465 and E4466. E69 branches into E4467 and E4468. E70 branches into E4469 and E4470. E63 branches into E4471 and E4472. E64 branches into E4473 and E4474. E65 branches into E4475 and E4476. E66 branches into E4477 and E4478. E67 branches into E4479 and E4480. E68 branches into E4481 and E4482. E69 branches into E4483 and E4484. E70 branches into E4485 and E4486. E63 branches into E4487 and E4488. E64 branches into E4489 and E4490. E65 branches into E4491 and E4492. E66 branches into E4493 and E4494. E67 branches into E4495 and E4496. E68 branches into E4497 and E4498. E69 branches into E4499 and E4500. E70 branches into E4501 and E4502. E63 branches into E4503 and E4504. E64 branches into E4505 and E4506. E65 branches into E4507 and E4508. E66 branches into E4509 and E4510. E67 branches into E4511 and E4512. E68 branches into E4513 and E4514. E69 branches into E4515 and E4516. E70 branches into E4517 and E4518. E63 branches into E4519 and E4520. E64 branches into E4521 and E4522. E65 branches into E4523 and E4524. E66 branches into E4525 and E4526. E67 branches into E4527 and E4528. E68 branches into E4529 and E4530. E69 branches into E4531 and E4532. E70 branches into E4533 and E4534. E63 branches into E4535 and E4536. E64 branches into E4537 and E4538. E65 branches into E4539 and E4540. E66 branches into E4541 and E4542. E67 branches into E4543 and E4544. E68 branches into E4545 and E

$E \rightarrow \text{num}$	$E.\text{addr} = \text{num}.val$ $E.\text{code} = ''$
$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get}(\text{id}.lexeme)$ $E.\text{code} = ''$

$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$B \rightarrow B_1 \&& B_2$	$B_1.\text{true} = \text{newlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

$B \rightarrow E_1 \text{ rel } E_2$

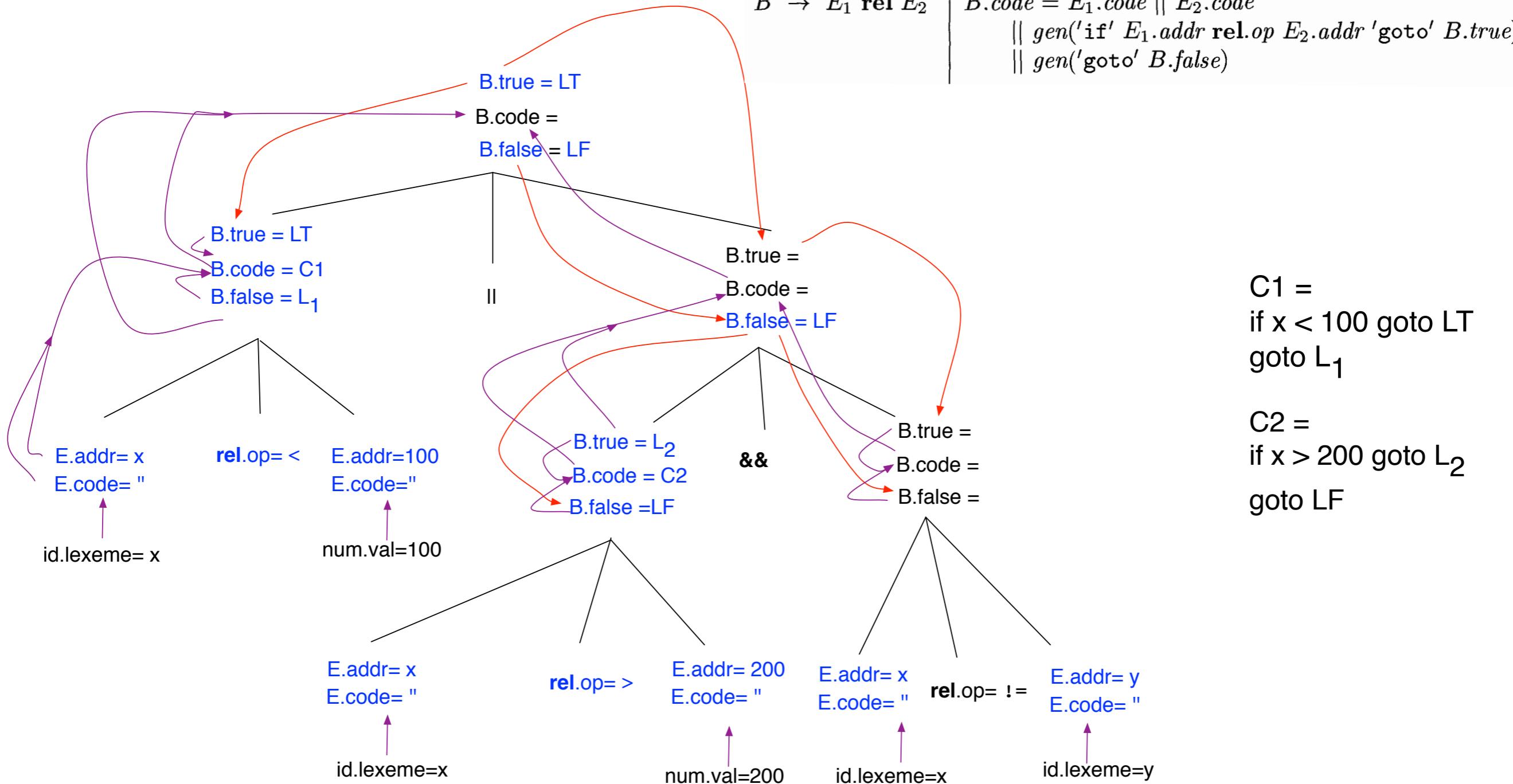
$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$
 $\parallel \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$
 $\parallel \text{gen('goto' } B.\text{false})$



$E \rightarrow \text{num}$	E.addr = num.val E.code = ''
$E \rightarrow \text{id}$	E.addr = top.get(id.lexeme) E.code = ''

$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$B \rightarrow B_1 \&& B_2$	$B_1.\text{true} = \text{newlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

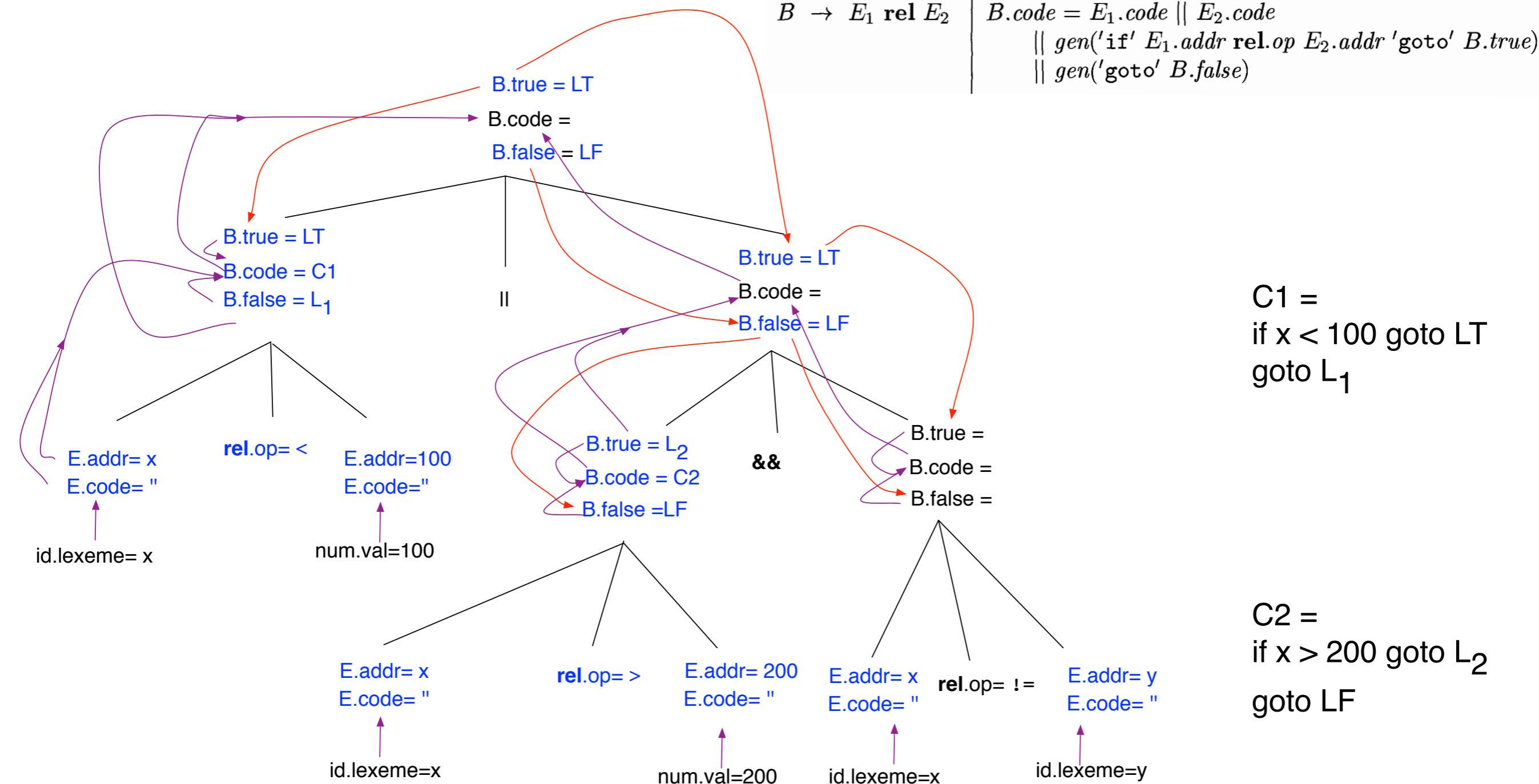
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$ $\parallel \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\parallel \text{gen('goto' } B.\text{false})$
--------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



$E \rightarrow \text{num}$	$E.\text{addr} = \text{num}.val$ $E.\text{code} = ''$
$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get}(\text{id}.lexeme)$ $E.\text{code} = ''$

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

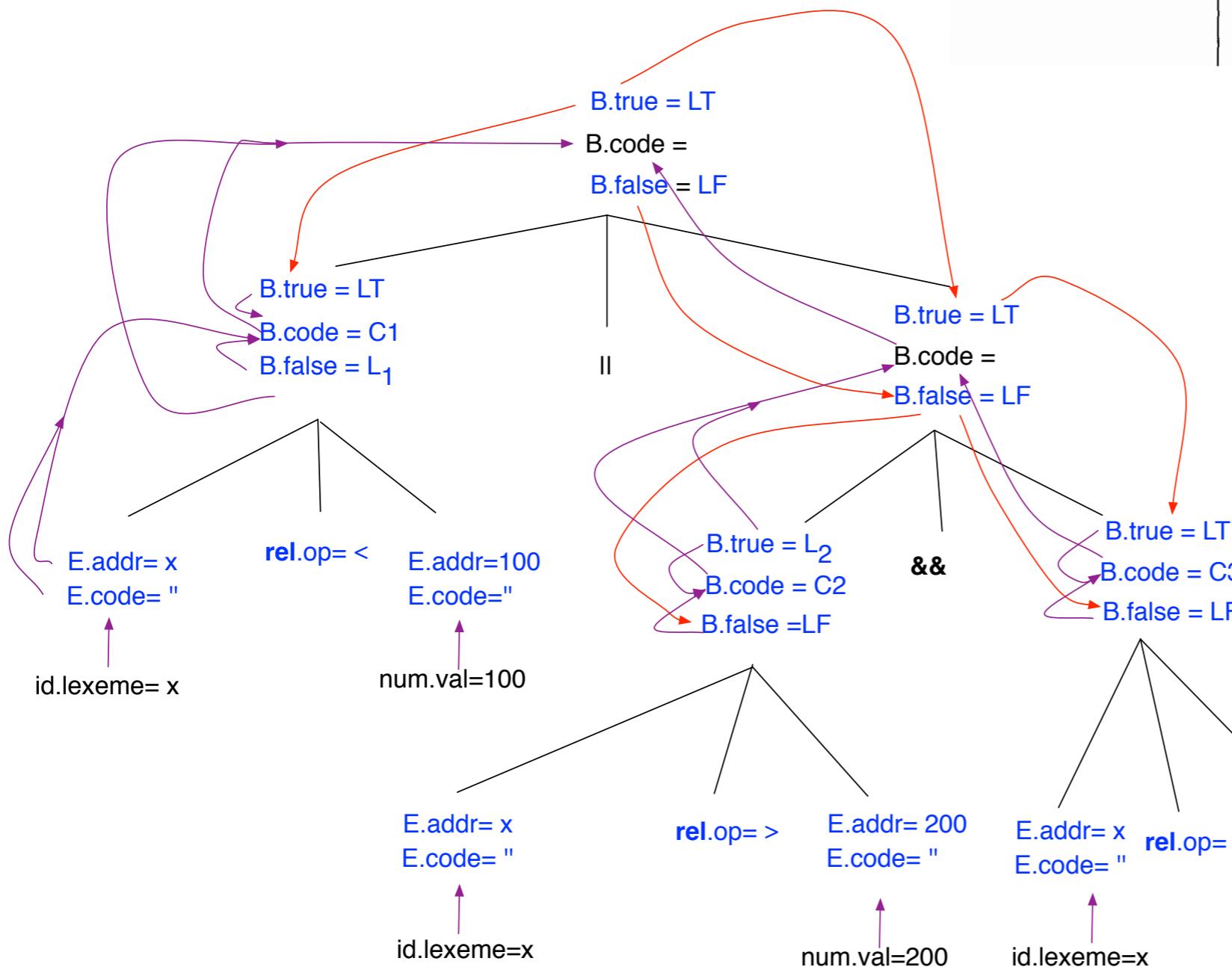
$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code} \mid\mid E_2.\text{code}$ $\mid\mid \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\mid\mid \text{gen('goto' } B.\text{false})$
--------------------------------------	--------------------------------------------------------------------------------------------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



$E \rightarrow \text{num}$	$E.\text{addr} = \text{num}.val$ $E.\text{code} = ''$
$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get}(\text{id}.lexeme)$ $E.\text{code} = ''$

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

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



$C_1 =$
if $x < 100$ goto L_1
goto LF

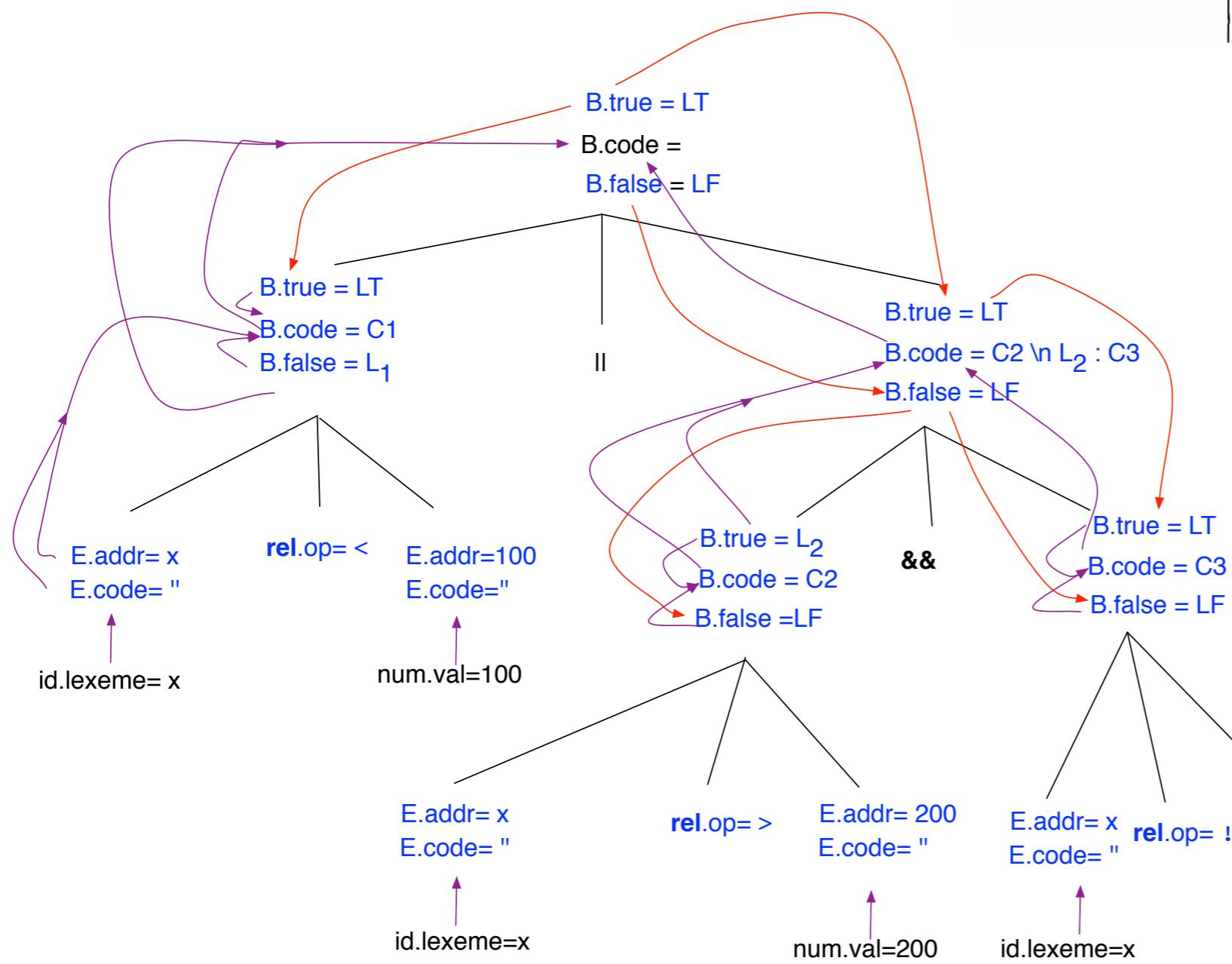
$C_2 =$
if $x > 200$ goto L_2
goto LF

$C_3 =$
if $x \neq y$ goto L_3
goto LF

$E \rightarrow \text{num}$	E.addr = num.val E.code = ''
$E \rightarrow \text{id}$	E.addr = top.get(id.lexeme) E.code = ''

$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
$B \rightarrow B_1 \&& B_2$	$B_1.\text{true} = \text{newlabel}()$ $B_1.\text{false} = B.\text{false}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

$B \rightarrow E_1 \text{ rel } E_2$	$B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$ $\parallel \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\parallel \text{gen('goto' } B.\text{false})$
--------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



$C_1 =$
if $x < 100$ goto LT
goto L_1

$C_2 =$
if $x > 200$ goto L_2
goto LF

$C_3 =$
if $x \neq y$ goto LT
goto LF

$E \rightarrow \text{num}$ | E.addr = num.val
 E.code = ''

$E \rightarrow \text{id}$ | E.addr = top.get(id.lexeme)
 E.code = ''

$B \rightarrow B_1 \parallel B_2$

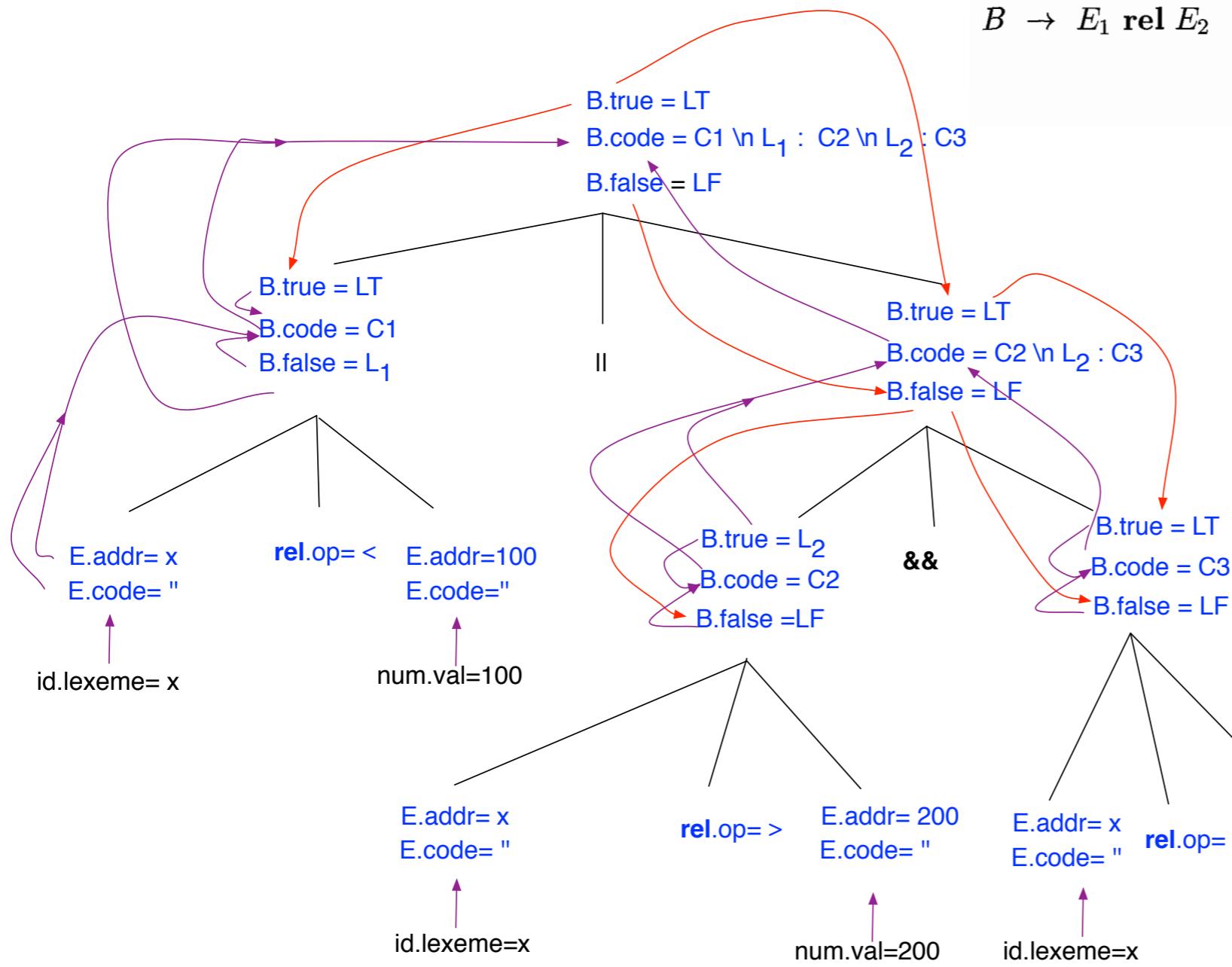
$B_1.\text{true} = B.\text{true}$
 $B_1.\text{false} = \text{newlabel}()$
 $B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$
 $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$

$B \rightarrow B_1 \&& B_2$

$B_1.\text{true} = \text{newlabel}()$
 $B_1.\text{false} = B.\text{false}$
 $B_2.\text{true} = B.\text{true}$
 $B_2.\text{false} = B.\text{false}$
 $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{true}) \parallel B_2.\text{code}$

$B \rightarrow E_1 \text{ rel } E_2$

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



if $x < 100$ goto LT
goto L₁

L₁ : if $x > 200$ goto L₂
goto LF

L₂ : if $x \neq y$ goto LT
goto LF

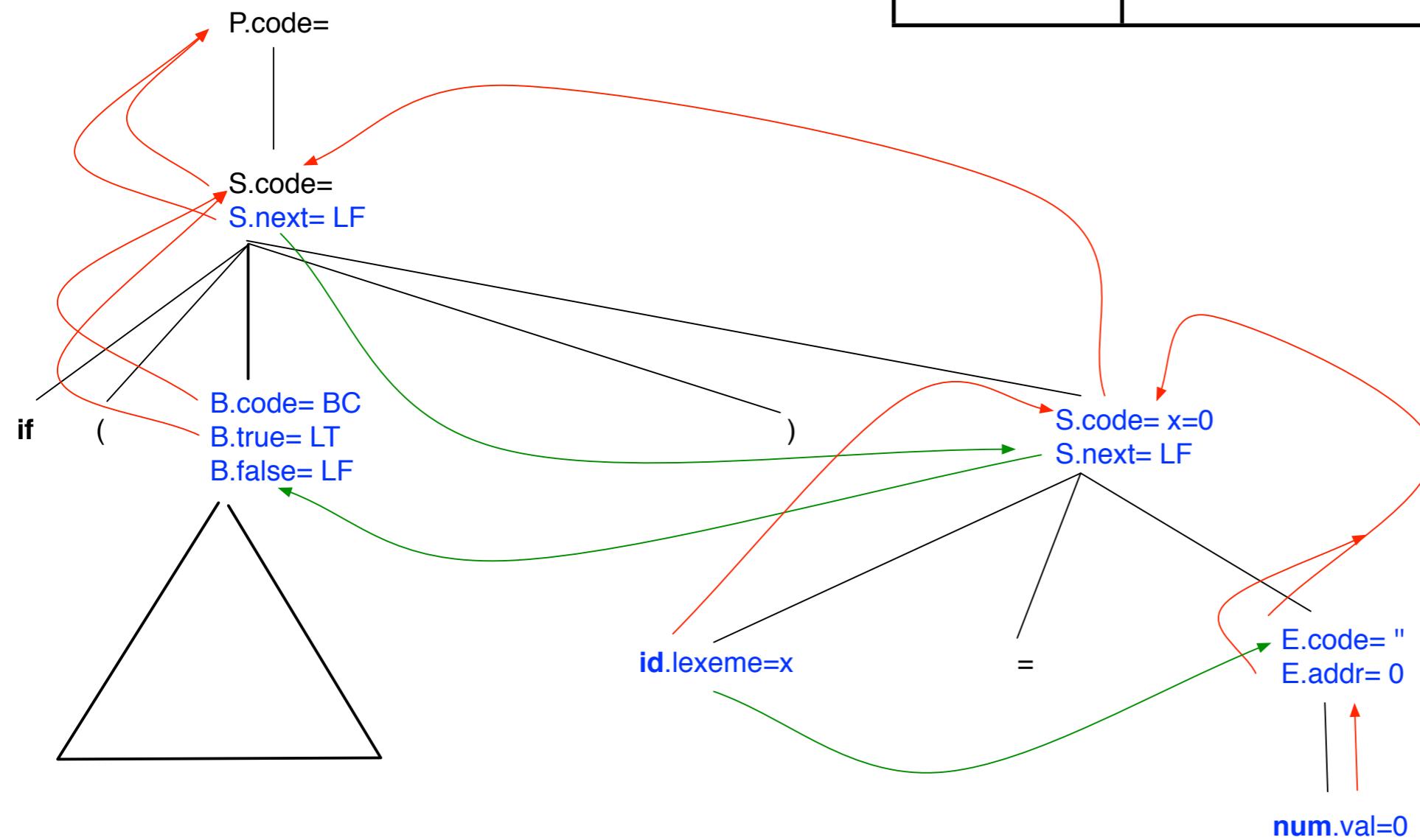
C₁ =
if $x < 100$ goto L₁
goto LF

C₂ =
if $x > 200$ goto L₂
goto LF

C₃ =
if $x \neq y$ goto LT
goto LF

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

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

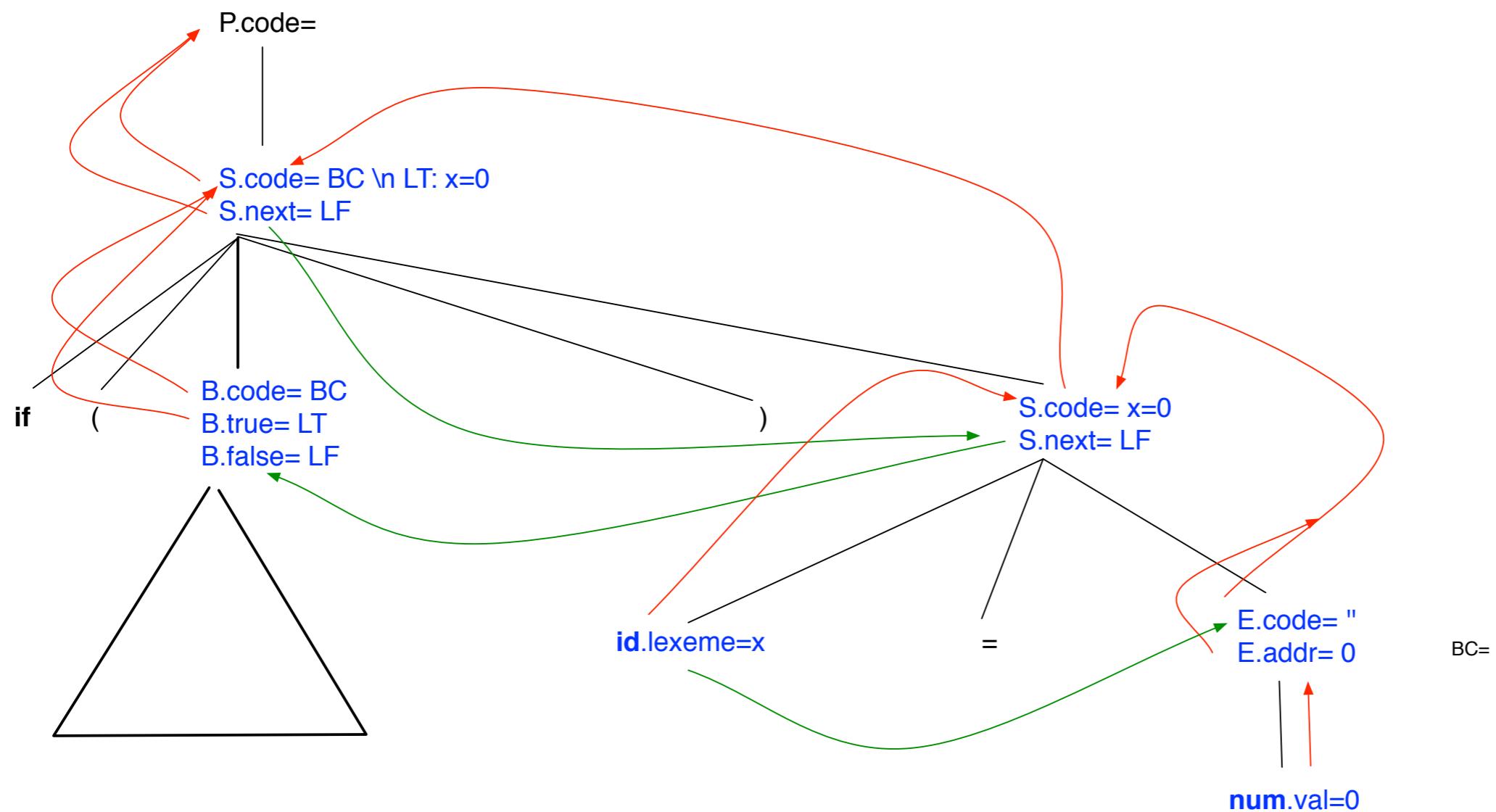


$if x < 100 \text{ goto } LT$
 $goto L_1$
 $BC = L_1 : if x > 200 \text{ goto } L_2$
 $goto LF$
 $L_2 : if x \neq y \text{ goto } LT$
 $goto LF$

$num.val=0$

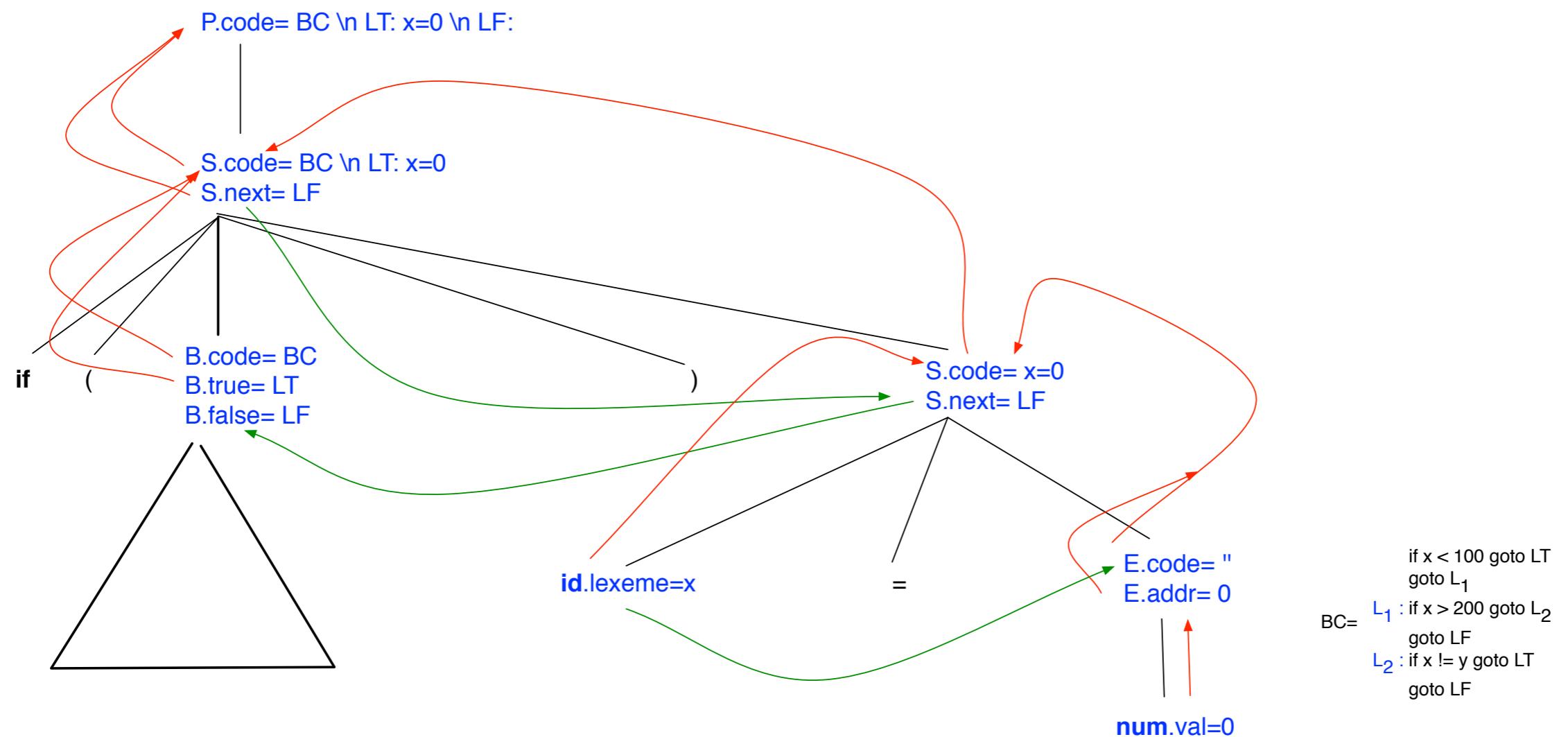
```
if( x<100 || x > 200 && x!=y ) x=0 ;
```

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



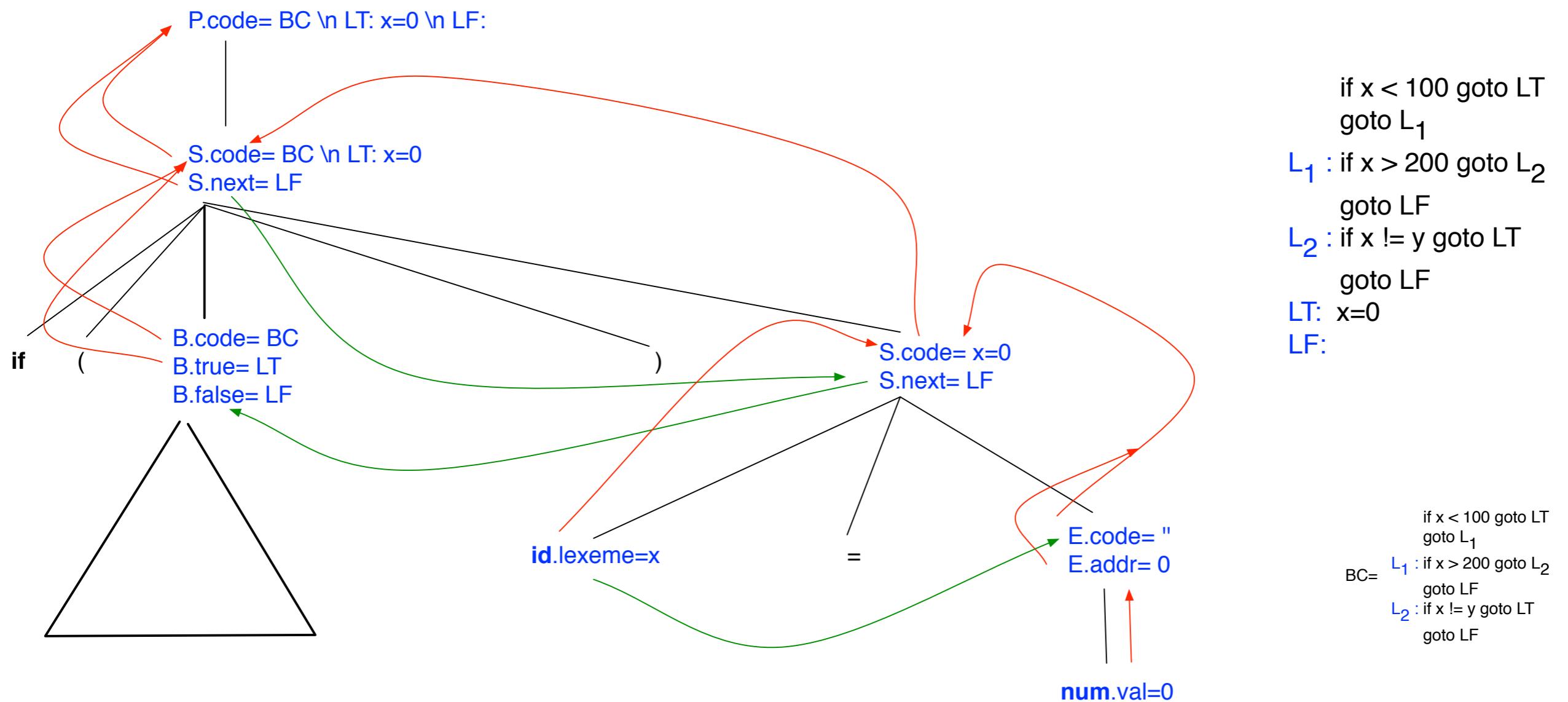
```
if(x<100 || x > 200 && x!=y) x=0;
```

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



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

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



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;
```

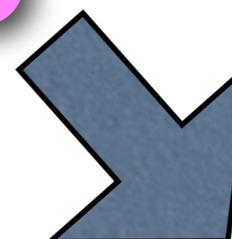
AVOIDING REDUNDANT GOTOS

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
$S \rightarrow \text{if}(B) S_1$	<p>B. true = fall B.false = $S_1.\text{next} = S.\text{next}$ $S.\text{code} = B.\text{code} \parallel S_1.\text{code}$</p> <div style="border: 1px solid black; padding: 10px; background-color: #fffacd;"> $B.\text{true} = \text{newlabel}()$ $B.\text{false} = S_1.\text{next} = S.\text{next}$ $S.\text{code} = B.\text{code} \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code}$ </div>
$B \rightarrow E_1 \text{ rel } E_2$	<p>test = $E_1.\text{addr} \text{ rel.op } E_2.\text{addr}$ $s = \text{if } B.\text{true} \neq \text{fall and } B.\text{false} \neq \text{fall then}$ $\quad \text{gen('if' test 'goto' } B.\text{true}) \parallel \text{gen('goto' } B.\text{false})$ $\quad \text{else if } B.\text{true} \neq \text{fall then gen('if' test 'goto' } B.\text{true})$ $\quad \text{else if } B.\text{false} \neq \text{fall then gen('iffalse' test 'goto' } B.\text{false})$ $\quad \text{else ''}$ $B.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel s$</p> <div style="border: 1px solid black; padding: 10px; background-color: #fffacd;"> $B.\text{code} = E_1.\text{code} \parallel E_2.\text{code}$ $\parallel \text{gen('if' } E_1.\text{addr rel.op } E_2.\text{addr 'goto' } B.\text{true})$ $\parallel \text{gen('goto' } B.\text{false})$ </div>

$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = B.\text{true}$ $B_1.\text{false} = \text{newlabel}()$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = B_1.\text{code} \parallel \text{label}(B_1.\text{false}) \parallel B_2.\text{code}$
-----------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$B \rightarrow B_1 \parallel B_2$	$B_1.\text{true} = \mathbf{if } B.\text{true} \neq \text{fall} \mathbf{then } B.\text{true} \mathbf{else } \text{newlabel}()$ $B_1.\text{false} = \text{fall}$ $B_2.\text{true} = B.\text{true}$ $B_2.\text{false} = B.\text{false}$ $B.\text{code} = \mathbf{if } B.\text{true} \neq \text{fall} \mathbf{then } B_1.\text{code} \parallel B_2.\text{code}$ $\mathbf{else } B_1.\text{code} \parallel B_2.\text{code} \parallel \text{label}(B_1.\text{true})$
-----------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

Suppose **$B.\text{true is fall}$** ; i.e, control falls through B , if B evaluates to true. Although B evaluates to true if B_1 does, $B_1.\text{true}$ must ensure that control jumps over the code for B_2 to get to the next instruction after B .

On the other hand, if B_1 evaluates to false, the truth-value of B is determined by the value of B_2 , so the rules ensure that $B_1.\text{false}$ corresponds to control falling through from B_1 to the code for B_2 .

$S \rightarrow \mathbf{id} = B$



$x < 100 \quad || \quad x > 200 \quad \&\& \quad x \neq y$

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

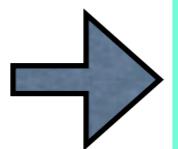
$p = x < 100 \quad || \quad x > 200 \quad \&\& \quad x \neq y$

```
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

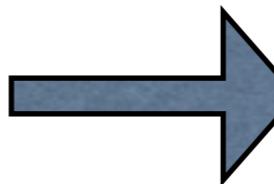
```
S.addr = new Temp()
S.lab = newlabel()
B.true = newlabel()
B.false = newlabel()
S.code = B.code ||
    label(B.true) || gen(S.addr '= true') ||
    gen('goto' S.lab) ||
    label(B.false) || gen(S.addr '= false') ||
    label(S.lab) || gen(top.get(id.lexeme) '=' S.addr)
```

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



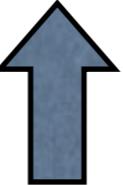
```
if x < 100 goto LT
goto L1
L1 : if x > 200 goto L2
      goto LF
L2 : if x != y goto LT
      goto LF
```

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

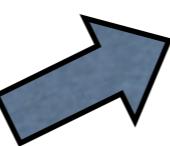


```
if x < 100 goto LT
goto L1
L1: if x > 200 goto L2
      goto LF
L2: if x != y goto LT
      goto LF
LT: t = true
      goto L
LF: t = false
L: p = t
```

$$S \rightarrow id = E; \mid if (E) S \mid while (E) S \mid S S$$

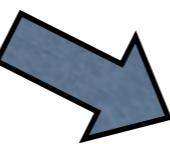
$$E \rightarrow E \parallel E \mid E \&\& E \mid E \text{ rel } E \mid E + E \mid (E) \mid id \mid true \mid false$$


a single nonterminal **E** for expressions



When E appears in $S \rightarrow \text{while} (E) S_1$, 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 $S.next$.

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 $S \rightarrow id = E;$, 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.

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$	$P \rightarrow D$	$\{ offset = 0; \}$
C	$\{ T.type = C.type; T.width = C.width; \}$		
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$	$D \rightarrow T \text{ id } ;$	$\{ top.put(\text{id}.lexeme, T.type, offset);$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$		$offset = offset + T.width; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$	$D \xrightarrow{D_1} \epsilon$	
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$		

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

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

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

$E \rightarrow E_1[E_2]$

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$	$P \rightarrow D$	$\{ offset = 0; \}$
C	$\{ T.type = C.type; T.width = C.width; \}$		
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$	$D \rightarrow T \text{ id} ;$	$\{ top.put(\text{id}.lexeme, T.type, offset);$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$		$offset = offset + T.width; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$	$D \xrightarrow{D_1} \epsilon$	
$C \rightarrow [\text{num}] C_1$	$\{ C.type = \text{array}(\text{num.value}, C_1.type);$ $C.width = \text{num.value} \times C_1.width; \}$		

$E \rightarrow \text{num}$	$\{ E.type := \text{integer} \}$
$E \rightarrow \text{id}$	$\{ E.type := \text{lookup} (\text{id}.entry) \}$
$E \rightarrow E_1 \text{ mod } E_2$	$\{ E.type := \text{if } E_1.type = \text{integer} \text{ and}$ $\quad \quad \quad \text{sequiv}(E_1.type, E_2.type)$ $\quad \quad \quad \text{then integer else type_error} \}$
$E \rightarrow E_1 \text{ binop } E_2$	$\{ E.type := \text{if sequiv}(E_1.type, E_2.type)$ $\quad \quad \quad \text{then } E_1.type \text{ else type_error} \}$
$E \rightarrow E_1[E_2]$	$\{ E.type := \text{if } E_2.type = \text{integer} \text{ and}$ $\quad \quad \quad E_1.type = \text{array}(s,t)$ $\quad \quad \quad \text{then } t \text{ else type_error} \}$

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

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

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

$S \rightarrow S_1; S_2$

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

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

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

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