Intermediate Code Generation

יצירת קוד-ביניים

Why an intermediate code?

- A step in closing a big gap between the high level of programming language and low level of machine language.
- Decompose complex constructions: expressions such as a+b+c*d, or loops, switches, etc. – no such things in machine language.
- Basis for machine-independent optimization. Revealing hidden details may also improve optimization capability. E.g. A[i] = A[i] +5 : seems nothing to improve, but there are hidden address calculations – twice the same (if in loop, may lead to significant loss of efficiency).
- Technological advantage in compilers construction: new language/machine architecture requires a new front-end/back-end, and not an entire new compiler.

Intermediate Code Generation

- Intermediate code generation is done during semantic analysis.
- The front-end of the compiler translates a source program into an intermediate representation from which the back-end of the compiler generates the resulting target code
- There are several intermediate code languages, very similar to one another.

TAC - Three-Address Code

- This is a widely used representation of an intermediate code.
- TAC – Three-Address Code.
- Each statement (command) may use up to three addresses:
  - two for the operands
  - one for the returned result

- TAC statements can be labeled by symbolic labels (A,B,C..).

TAC syntax : operations and commands

- **Assignment statements:** **x := y op z**

where **op** is a binary logical or arithmetic operation
(e.g. + or &).

- **Assignment operation:** **x := op y**

where **op** is a unary logical or arithmetic operation
(e.g. negation, or unary minus)

- **Copy statement:** **x := y**

TAC syntax : operations and commands

- **Unconditional jump:** **goto label**

- **Conditional jump:** **if x relop y goto label**

where **relop** is a relational operation (e.g. < or ==)

TAC syntax : operations and commands

- **Indexed assignment** **x := y[i] and x[i] := y**
- **Address and pointer assignments** **(three command types:)**

  x := &y

  x := *y

  ***x := y**

TAC syntax : operations and commands

- **procedure calls :**

param x

call proc, n

where **n** specifies the number of parameters that procedure **proc** receives.
e.g. :

param x1

param x2

...

param xn

call proc , n

Translation Scheme : Code Generation

- We now consider translation for :

  - Declarations
  - Expressions and assignment statement
  - Control statements

Translation of declaration

- Translation of declarations addresses the need to allocate memory for each declared variable (during execution of the compiled program).
- Amount of memory depends on type of the variable - basic data type (integer, char, real), complex (struct), array, etc.

- Addresses must be relative (given by offset).
- An offset is relative to the start of the scope in which the variable is declared.
- A global variable ***offset*** will hold next available relative address.

Translation of declaration

- $S \rightarrow D$
- $D \rightarrow D;D$
- $D \rightarrow id:T$
- $T \rightarrow integer \mid real$
- $T \rightarrow array[num]\ of\ T1$
- $T \rightarrow *T1$

For the sake of memory allocation and storing appropriate date in symbol table.
For the grammar variable T we define synthesized attributes type and width:
T.type - the type of the declared variable
T.width - the size (measured in bytes) of the variable

$S \rightarrow$ {offset := 0} $\quad D$

$D \rightarrow D \; ; \; D$

$D \rightarrow$ id : T $\qquad$ { entry_ptr = insert(current_table, id.name);
set_type(entry_ptr, T.type);
set_offset(entry_ptr, offset);
offset := offset + T.width; }

$T \rightarrow$ integer { T.type := integer;
T.width := 4}

$T \rightarrow$ real { T.type := real;
T.width := 8}

$T \rightarrow$ array[num] of T1 { T.type := array(num.value,T1.type);
T.width := num.value * T1.width }

$T \rightarrow$ *T1 { T.type := pointer(T1.type);
T.width := 4 }

Translation of expressions and assignments

- In translation of complex expressions of the input language, temporary variables are used to store values of intermediate results.

- E.g.:

a = (b + c) * d + 12;   Boo(a, b, c, d);

Cannot translate into single-line TAC.   Must use temporary variables.

- For example in a rule $\boldsymbol{E} \rightarrow \boldsymbol{E}\ \boldsymbol{op}\ \boldsymbol{E}$, the value of expression E on the left hand side should be computed from the values of its sub-expressions.
- Each of these two values is stored in some variable.
- The result will be stored in a temporary variable.

Translation of expressions and assignments

- $S \rightarrow$ id := E
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow - E$
- $E \rightarrow (E)$
- $E \rightarrow$ id

Translation of expressions and assignments

- To implement the translation, the following attributes are used:

- Attributes (all are synthesized):

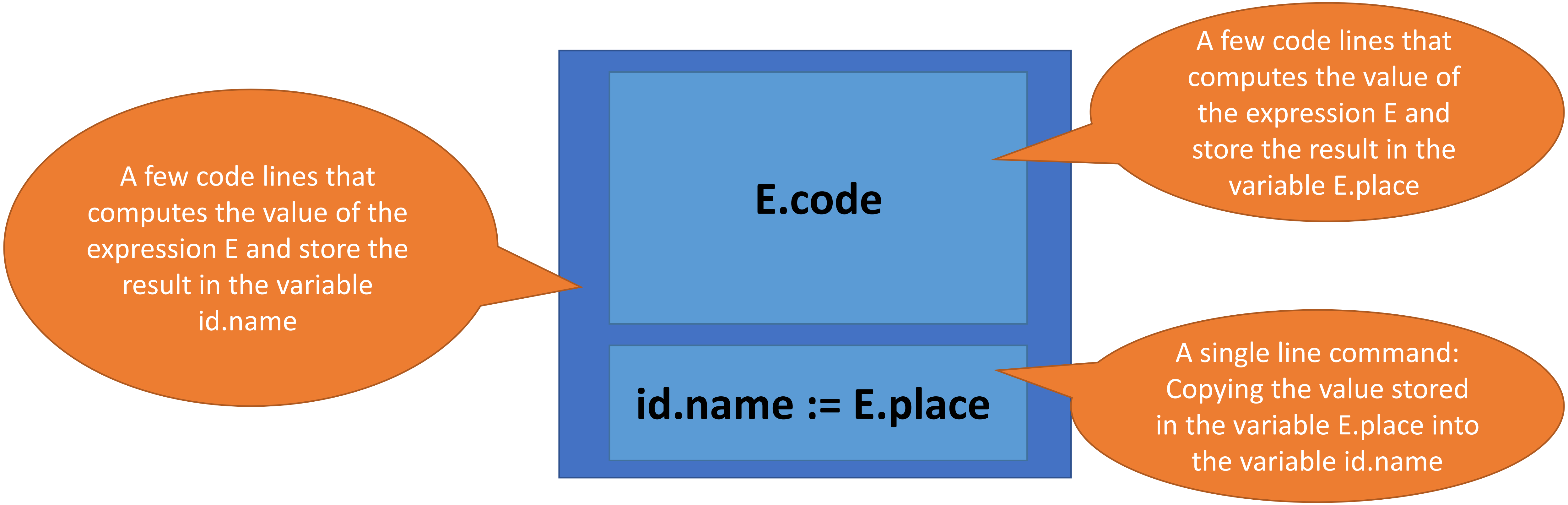
  E.place – name of the variable that holds a value of E
  E.code – holds a TAC for expression E (a series of TAC commands that compute E's value and store it in the variable E.place)
  id.name – name (lexeme) of the identifier, as supplied by the lexical analyzer

- In addition, the following function is used :

function newtemp : creates a new name (for a new temporary variable)

|| - string concatenation

S→id := E { S.code =
E.code || id.name || “ := “ || E.place }



$E \rightarrow id$ {E.place = id.name;
E.code = “”}

- $E \rightarrow E1 + E2$

{ E.place = newtemp;

E.code = E1.code || E2.code ||

E.place || ":=" || E1.place || "+" || E2.place

}

E1.code

E2.code

E.place := E1.place + E2.place

- E→ E1 * E2

{ E.place = newtemp;

E.code = E1.code || E2.code ||

E.place || “:=“ || E1.place || “*” || E2.place

}

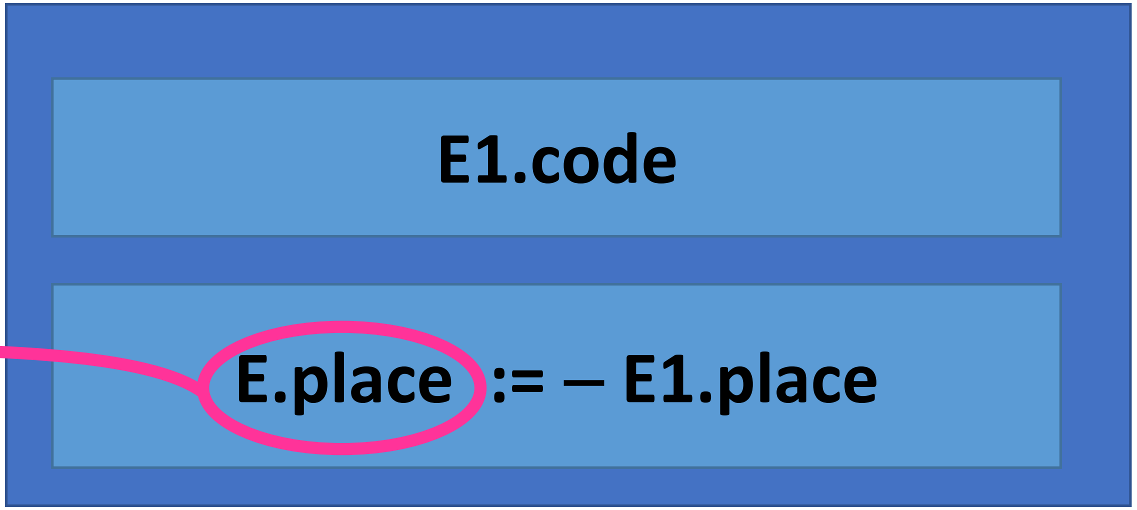
E1.code

E2.code

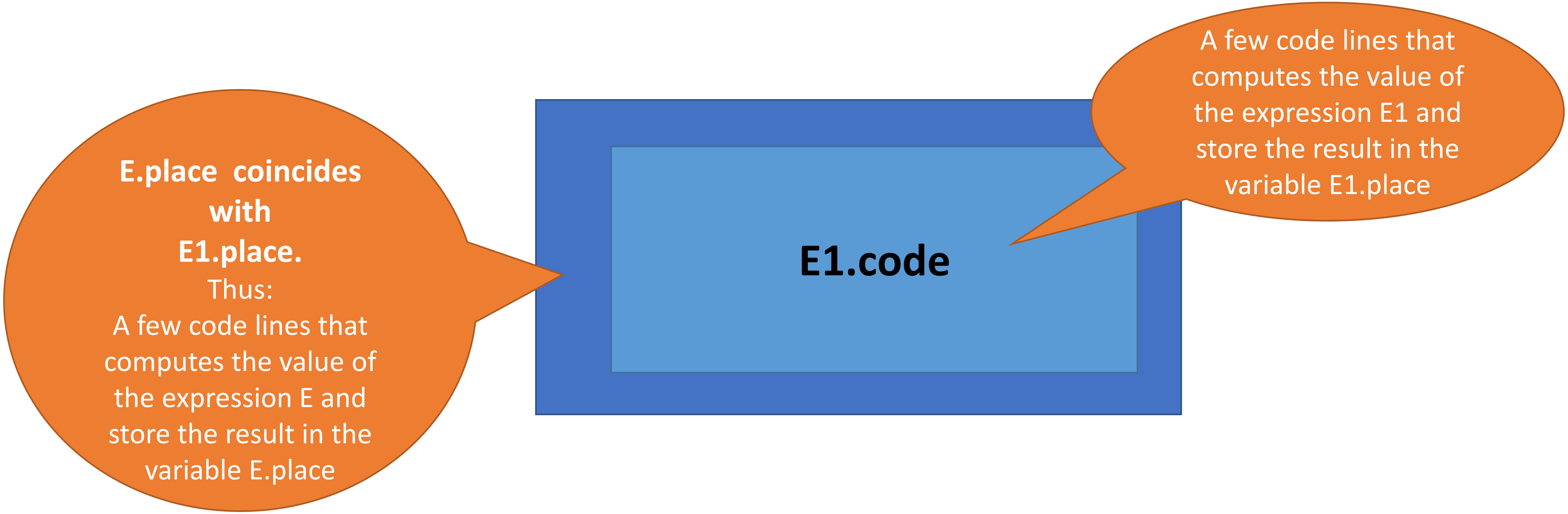
E.place := E1.place * E2.place

- $E \to - E1$

{ E.place = newtemp;
E.code = E1.code ||
E.place || ":= –" || E1.place }



- E$\rightarrow$ (E1)   { E.place = E1.place ;
  E.code = E1.code }



S → id := E	{ S.code = E.code || id.name || " := " || E.place }
E → E1+E2	{E.place = newtemp; E.code = E1.code || E2.code || E.place || ":=" || E1.place || "+" || E2.place }
E → E1*E2	{E.place = newtemp; E.code = E1.code || E2.code || E.place || ":=" || E1.place || "*" || E2.place }
E → -E1	{E.place = newtemp; E.code = E1.code || E.place || ":= -" || E1.place }
E → (E1)	{E.place = E1.place ; E.code = E1.code}
E → id	{E.place = id.name; E.code = ""}

New lines concatenated to existing code

emit - concatenating new code

- For convenience we use function *emit(string)* that creates E.code by appending the string at the end of the already created code, and avoid the concatenating sign.
- E.g.: E → E1+E2 {E.place = newtemp;

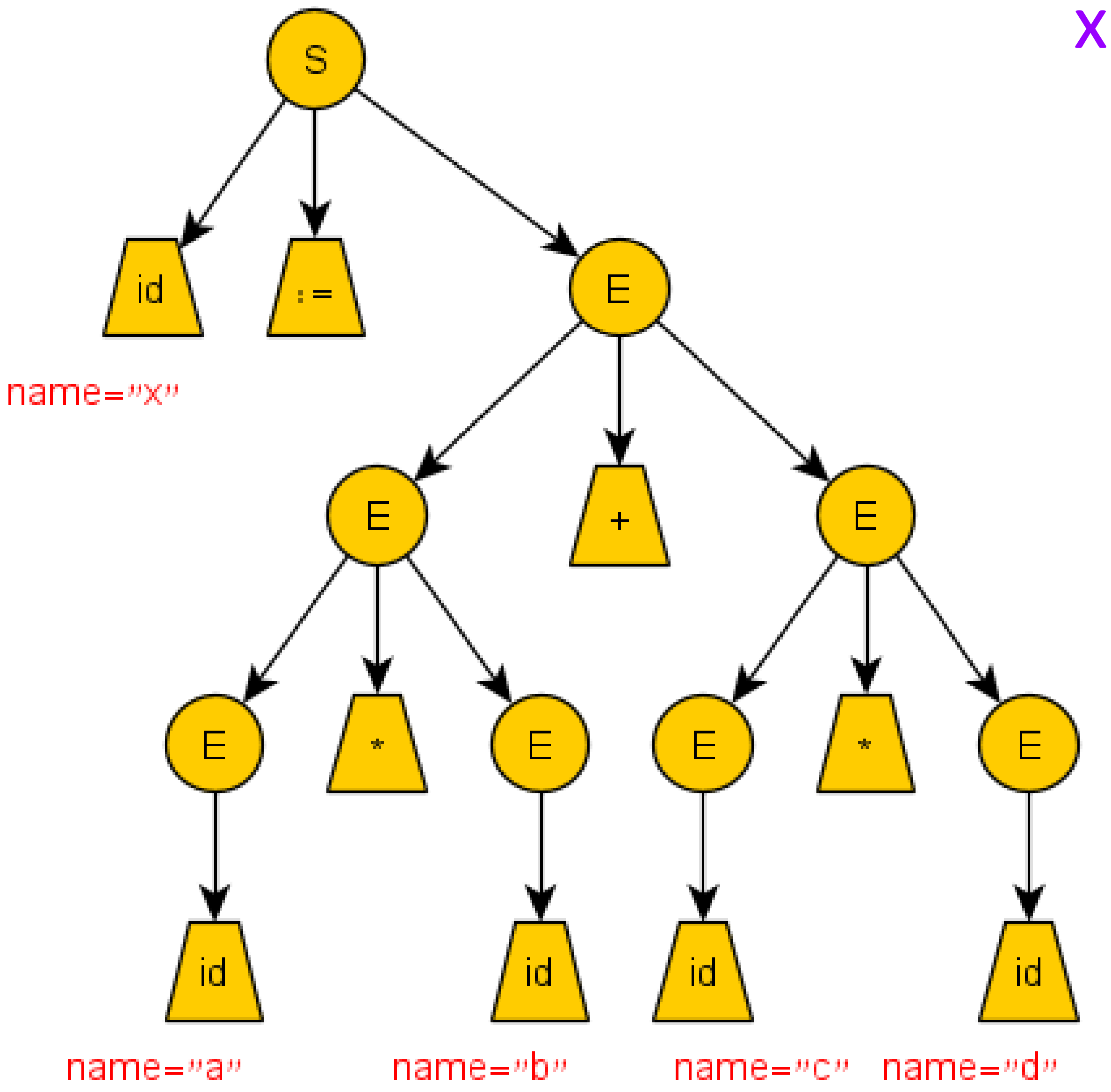
  E.code = E1.code || E2.code ||

  E.place || ":=" || E1.place || "+" || E2.place }
- Becomes: E → E1+E2 {E.place := newtemp;

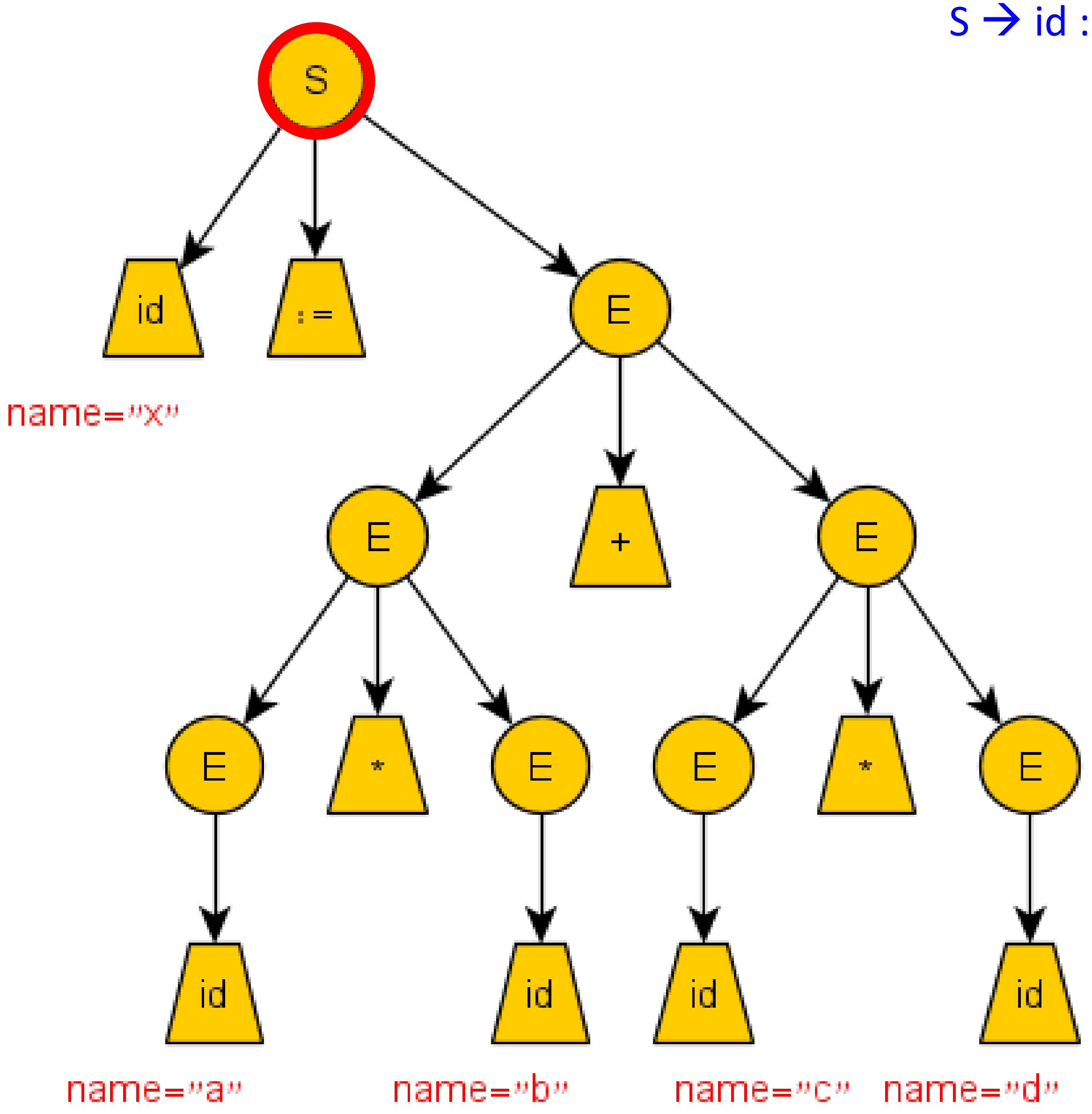
  emit (E.place ":=" E1.place "+" E2.place }

S → id := E	{ emit(id.name ":=" E.place }
E → E1+E2	{E.place = newtemp; emit(E.place ":=" E1.place "+" E2.place) }
E → E1*E2	{E.place = newtemp; emit(E.place ":=" E1.place "*" E2.place) }
E → -E1	{E.place = newtemp; emit(E.place ":= -" E1.place) }
E → (E1)	{E.place = E1.place}
E → id	{E.place = id.name; }

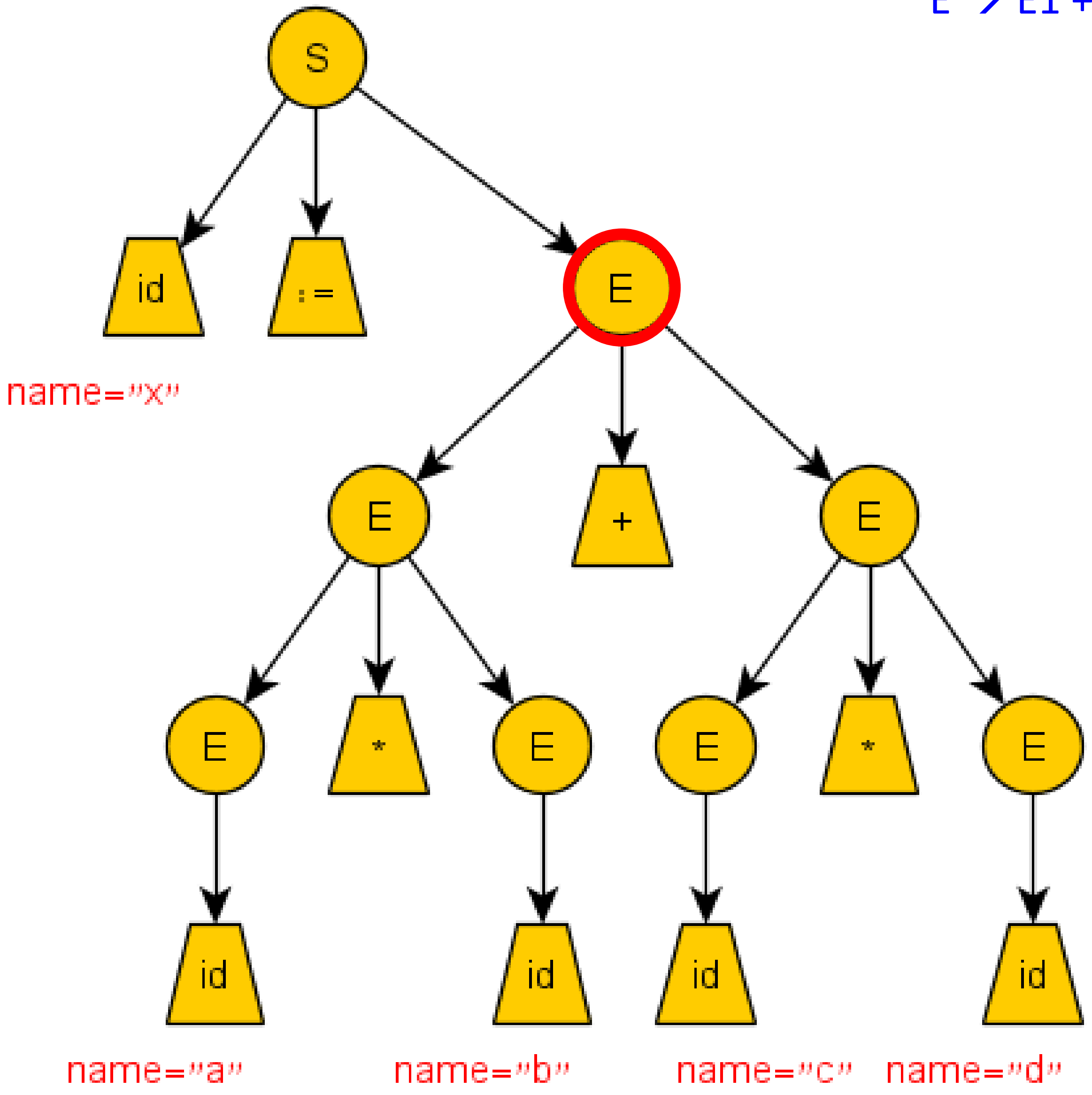
$x \ := a * b + c * d$



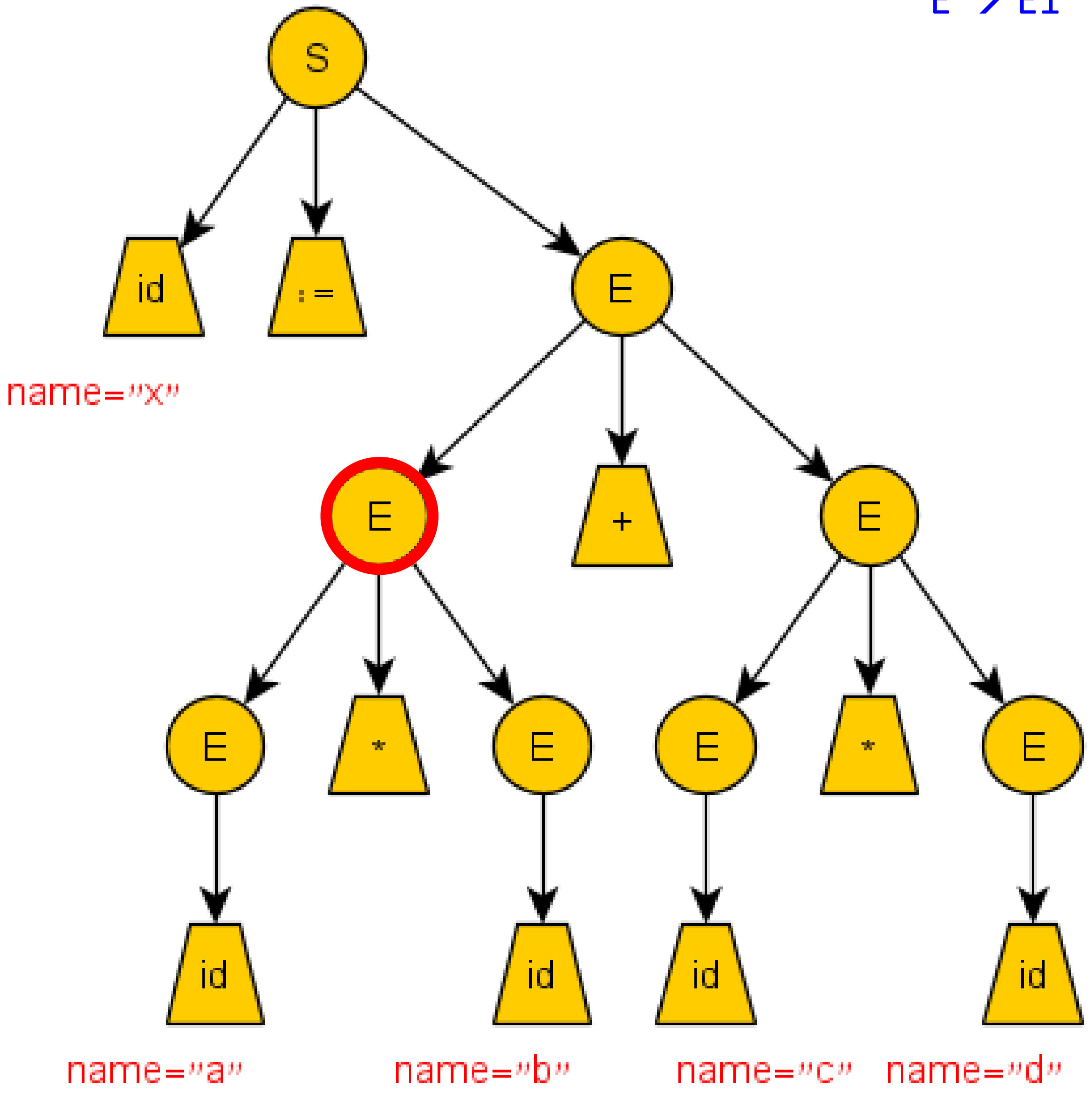
$S \rightarrow id := E$

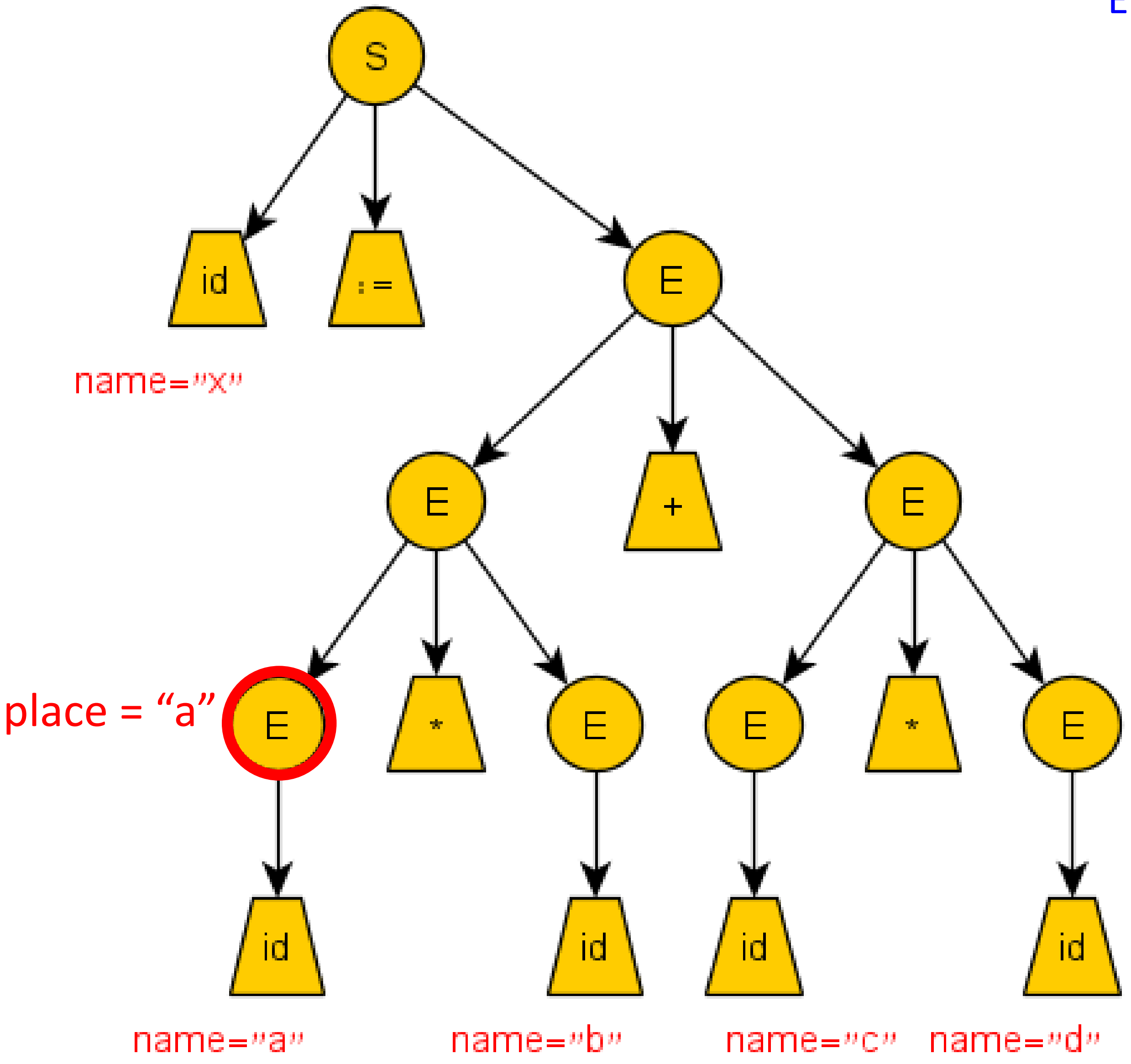


$E \rightarrow E1 + E2$



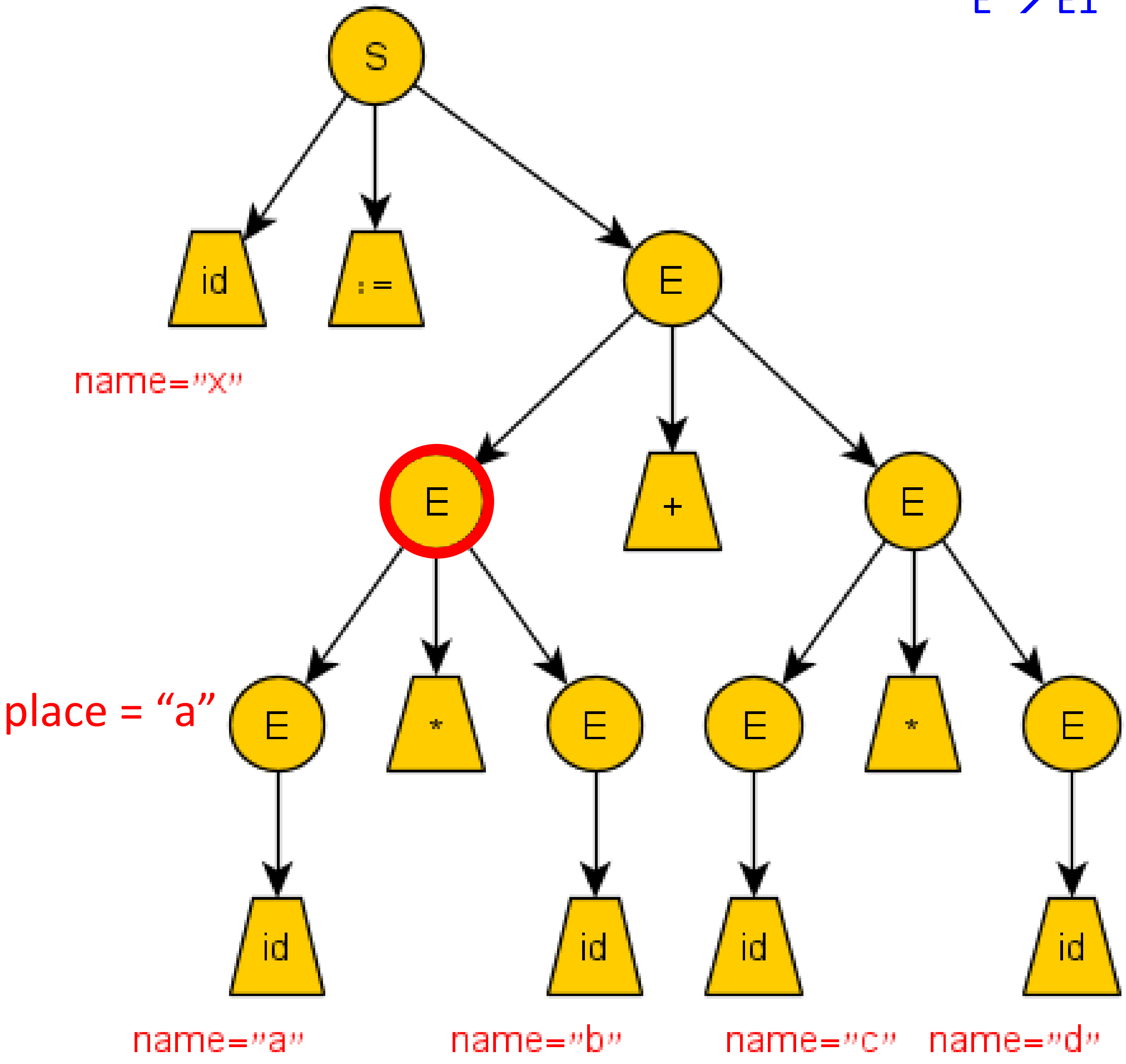
$E \rightarrow E1 * E2$



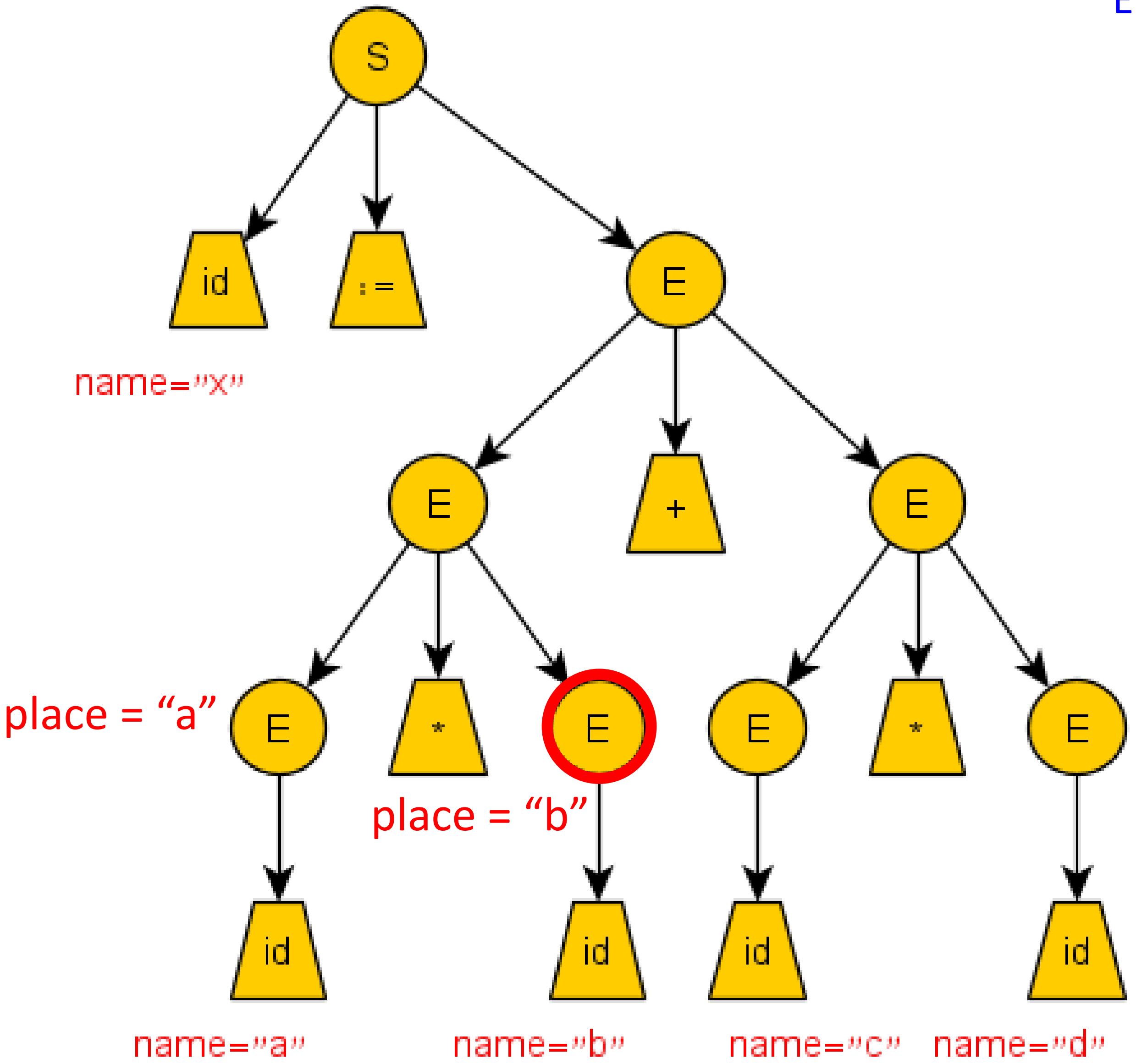


E → id {E.place = id.name; }

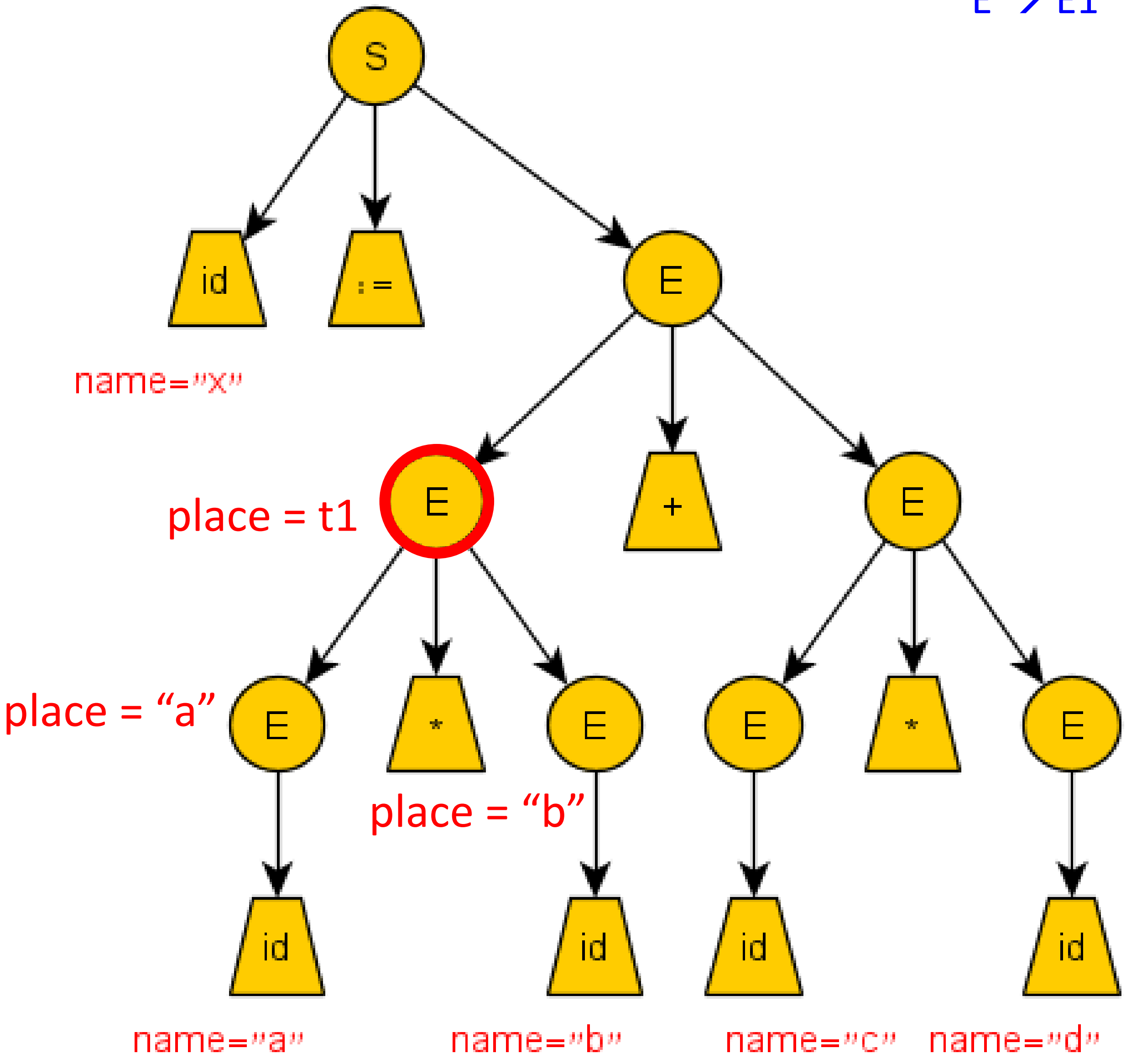
E → E1 * E2



$E \rightarrow id$ {E.place = id.name; }

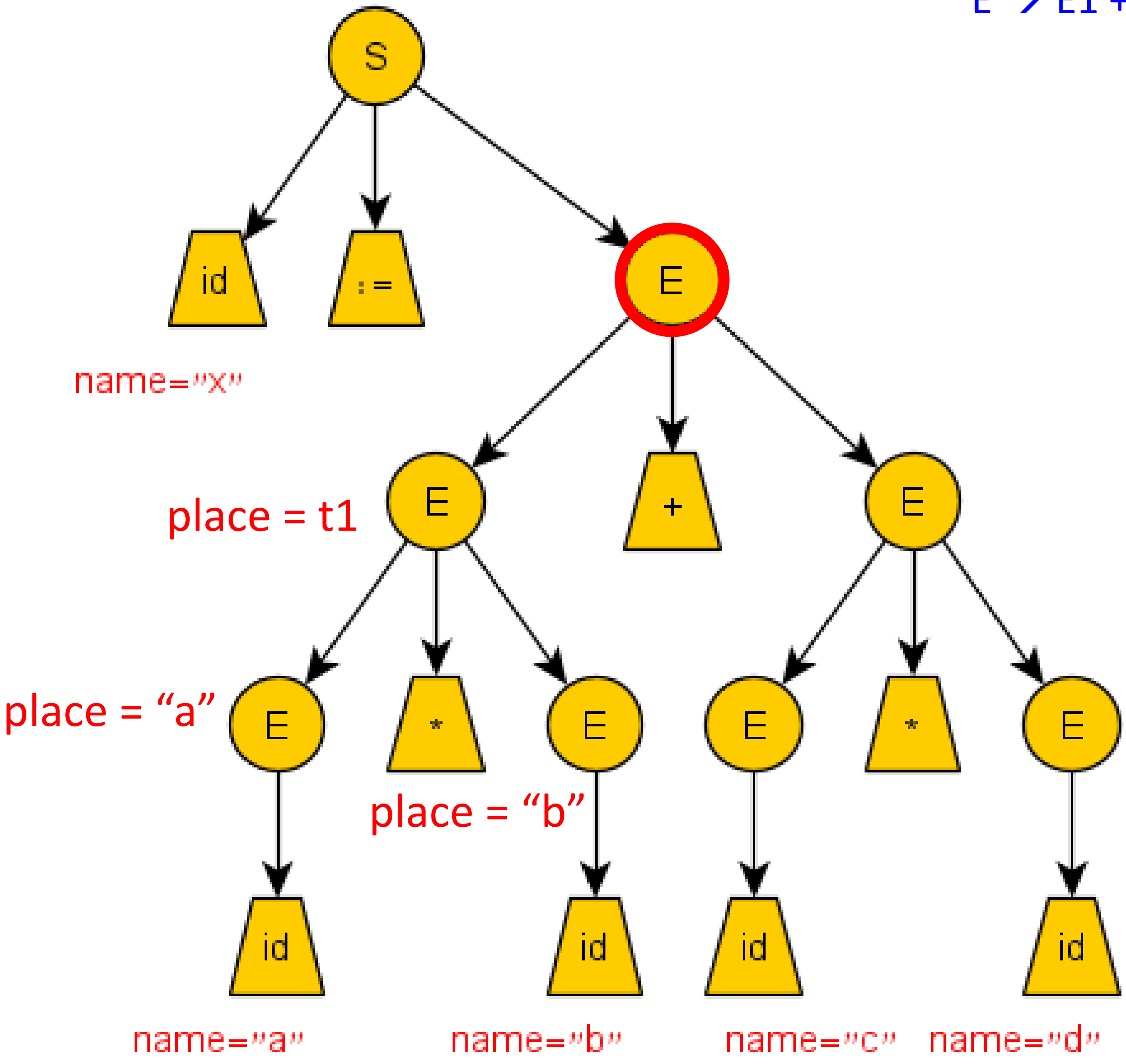


$E \rightarrow E1 * E2$ {E.place = newtemp; emit(E.place ":=" E1.place "*" E2.place) }



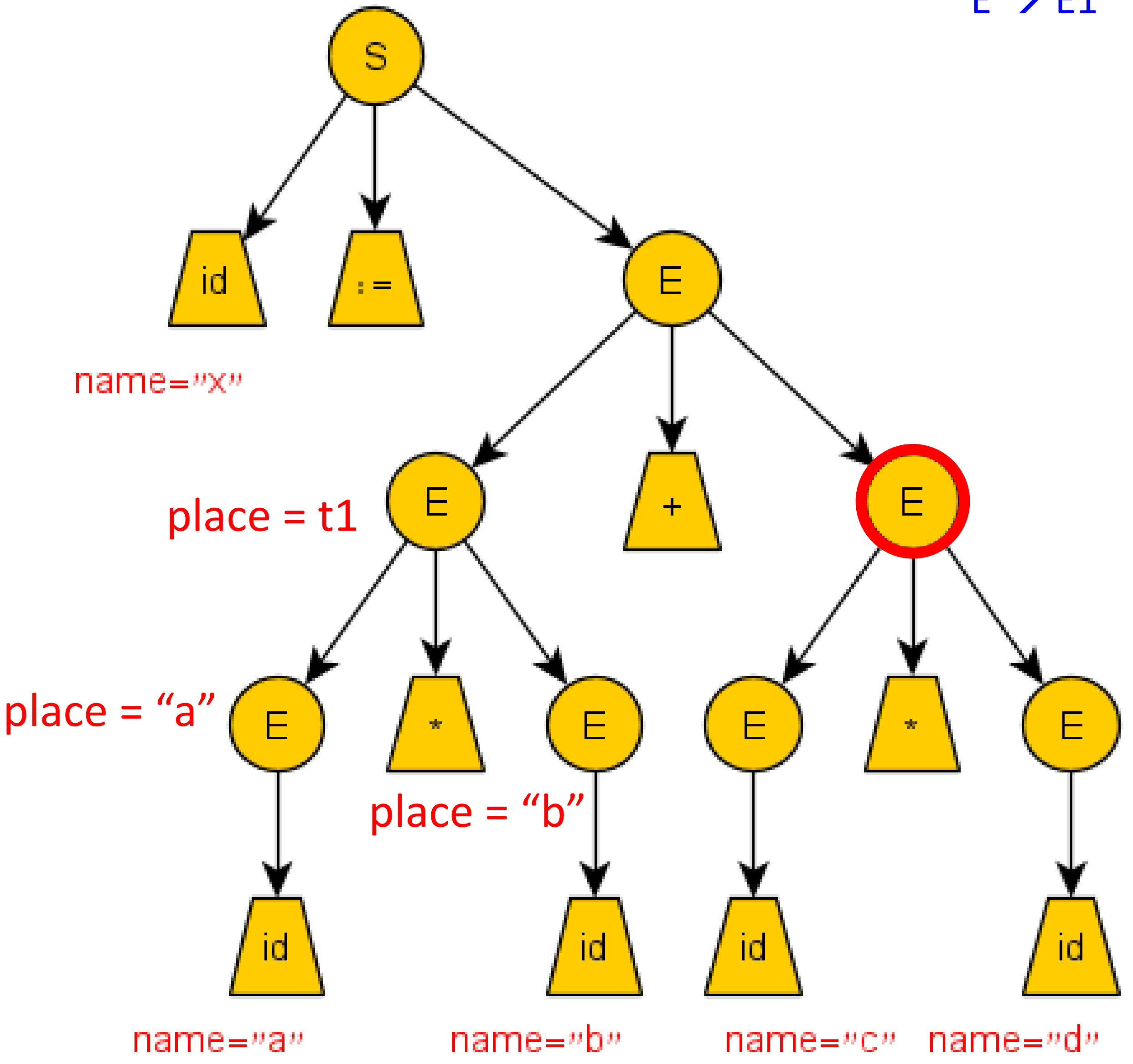
t1 := a * b

E → E1 + E2


S
id
:=
E
name="x"
place = t1
E
+
E
place = "a"
E
*
E
E
*
E
place = "b"
id
id
id
id
name="a"
name="b"
name="c"
name="d"

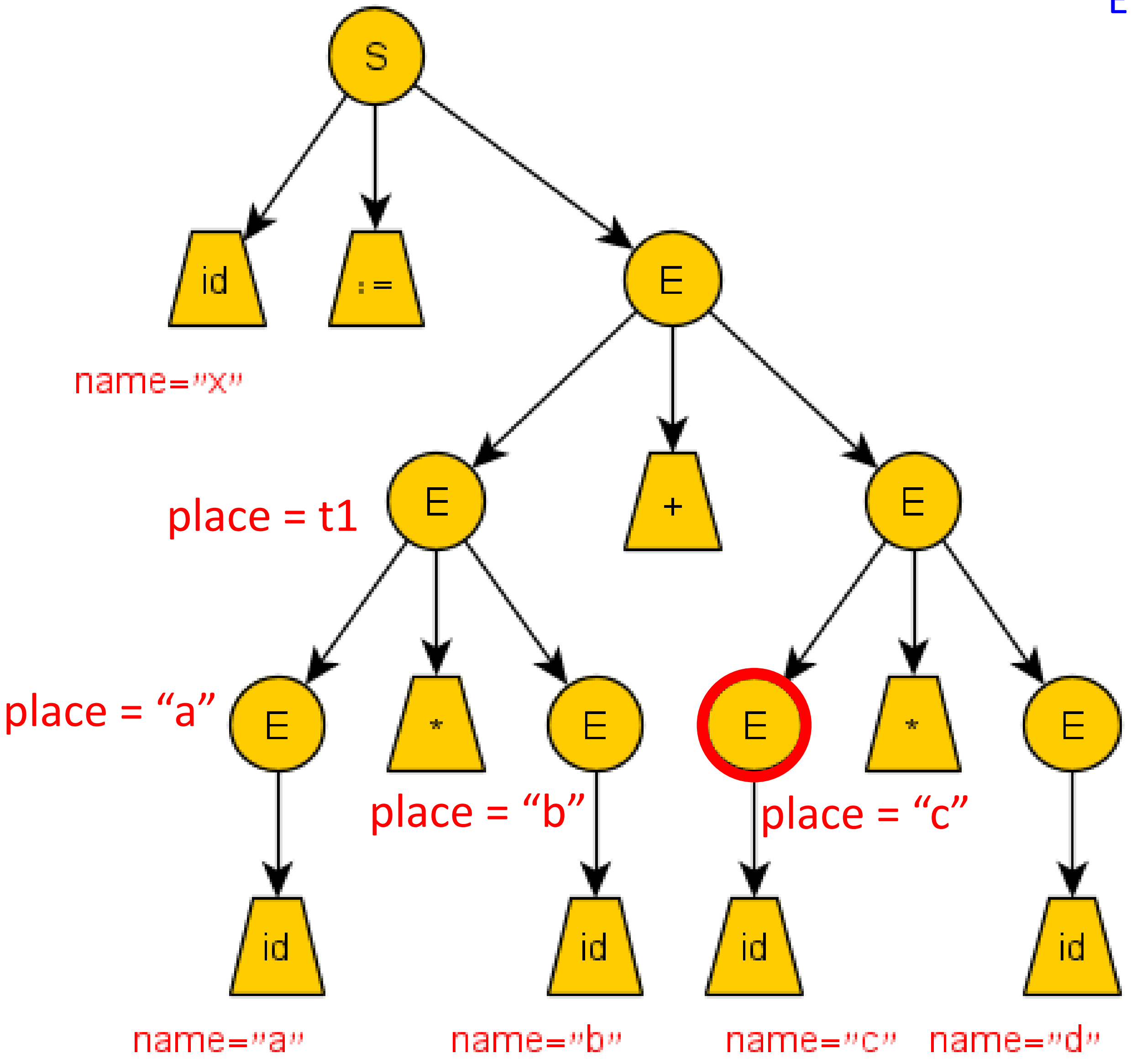
t1 := a * b

E → E1 * E2



S
id
:=
E
name="x"
E
+
E
place = t1
place = "a"
E
*
E
E
*
E
place = "b"
id
id
id
id
name="a"
name="b"
name="c"
name="d"

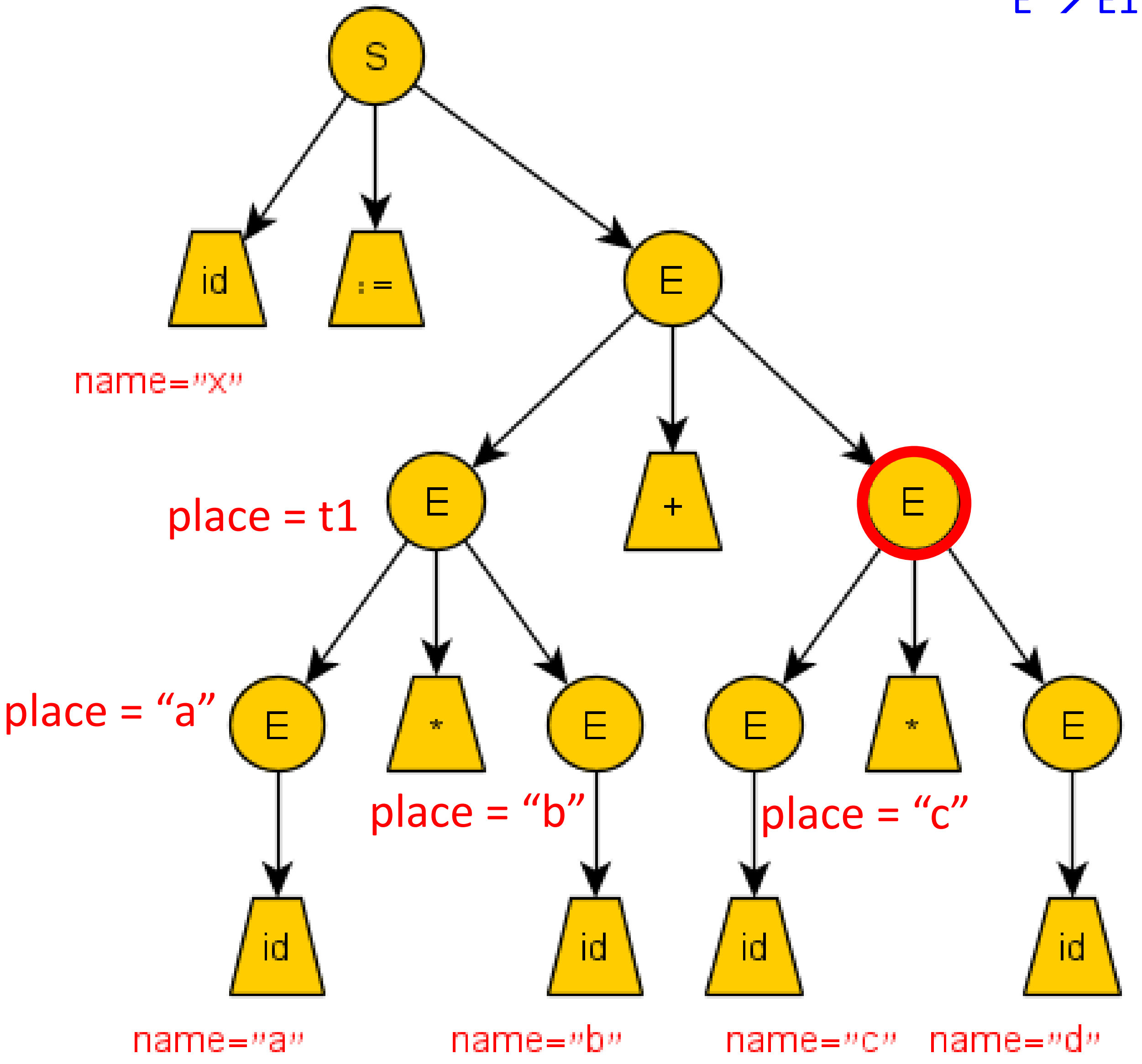
t1 := a * b



E → id {E.place = id.name; }

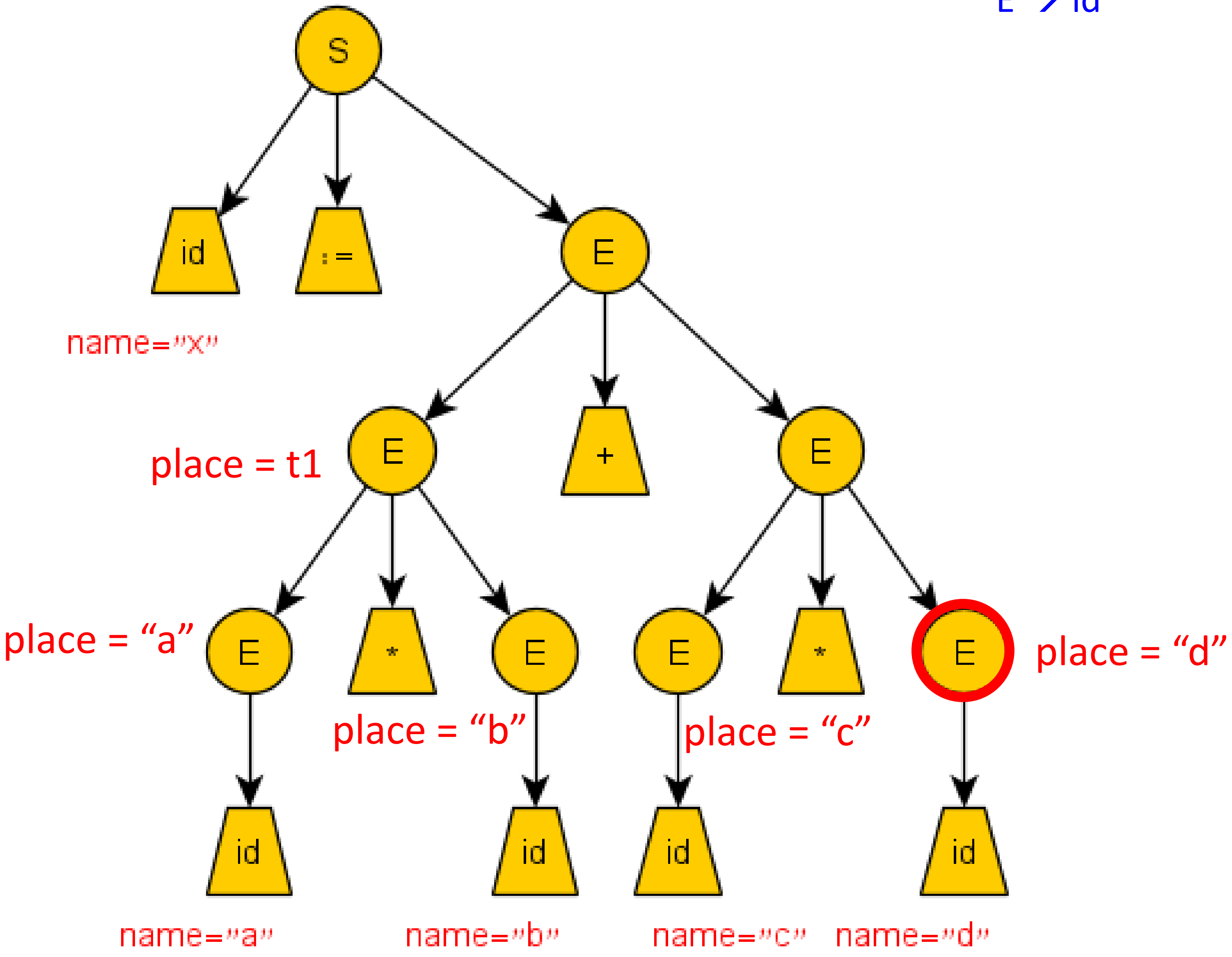
```
t1 := a * b
```

E → E1 * E2

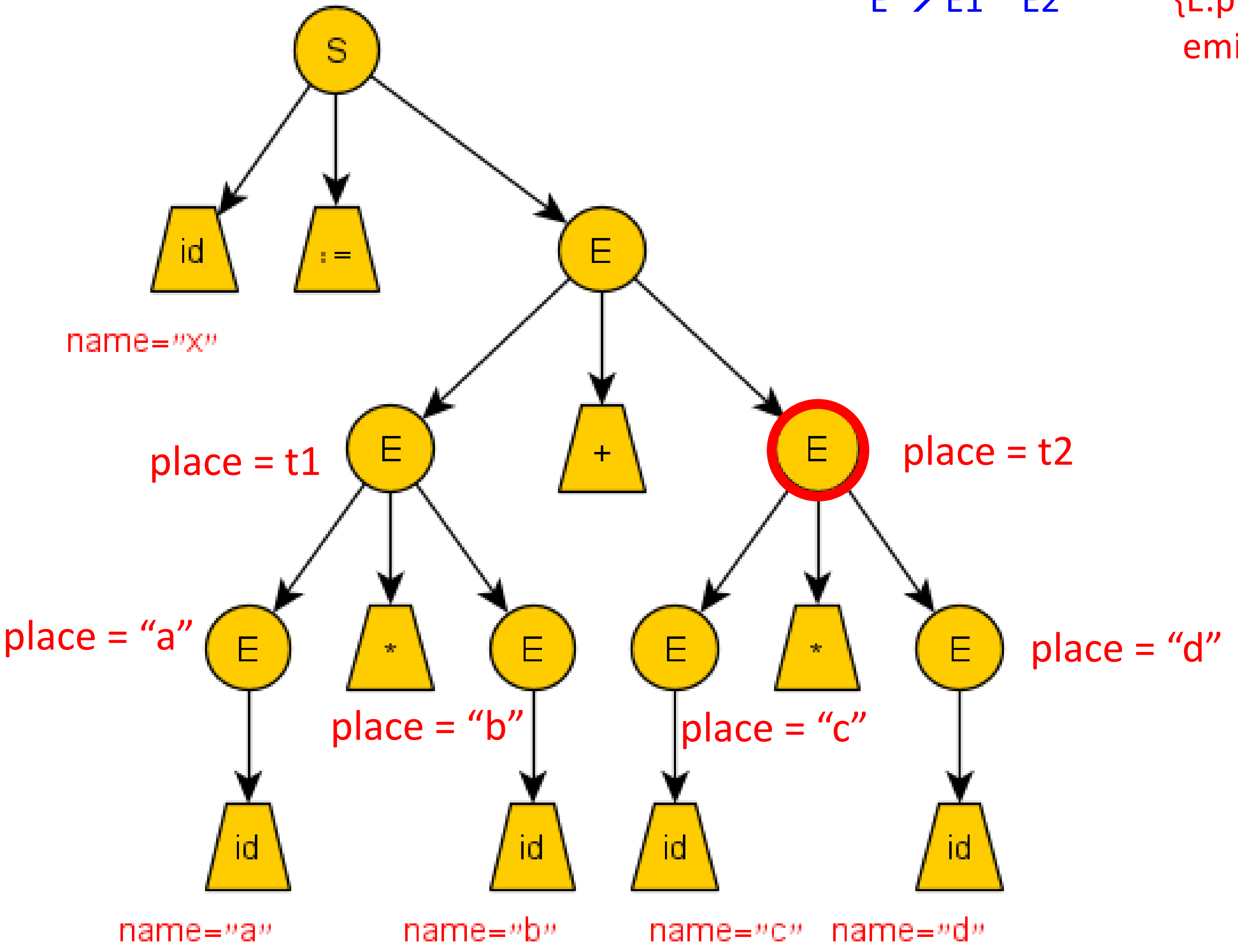

S
id
:=
E
name="x"
place = t1
E
+
E
place = "a"
E
*
E
E
*
E
place = "b"
place = "c"
id
id
id
id
name="a"
name="b"
name="c"
name="d"

t1 := a * b

E → id {E.place = id.name; }



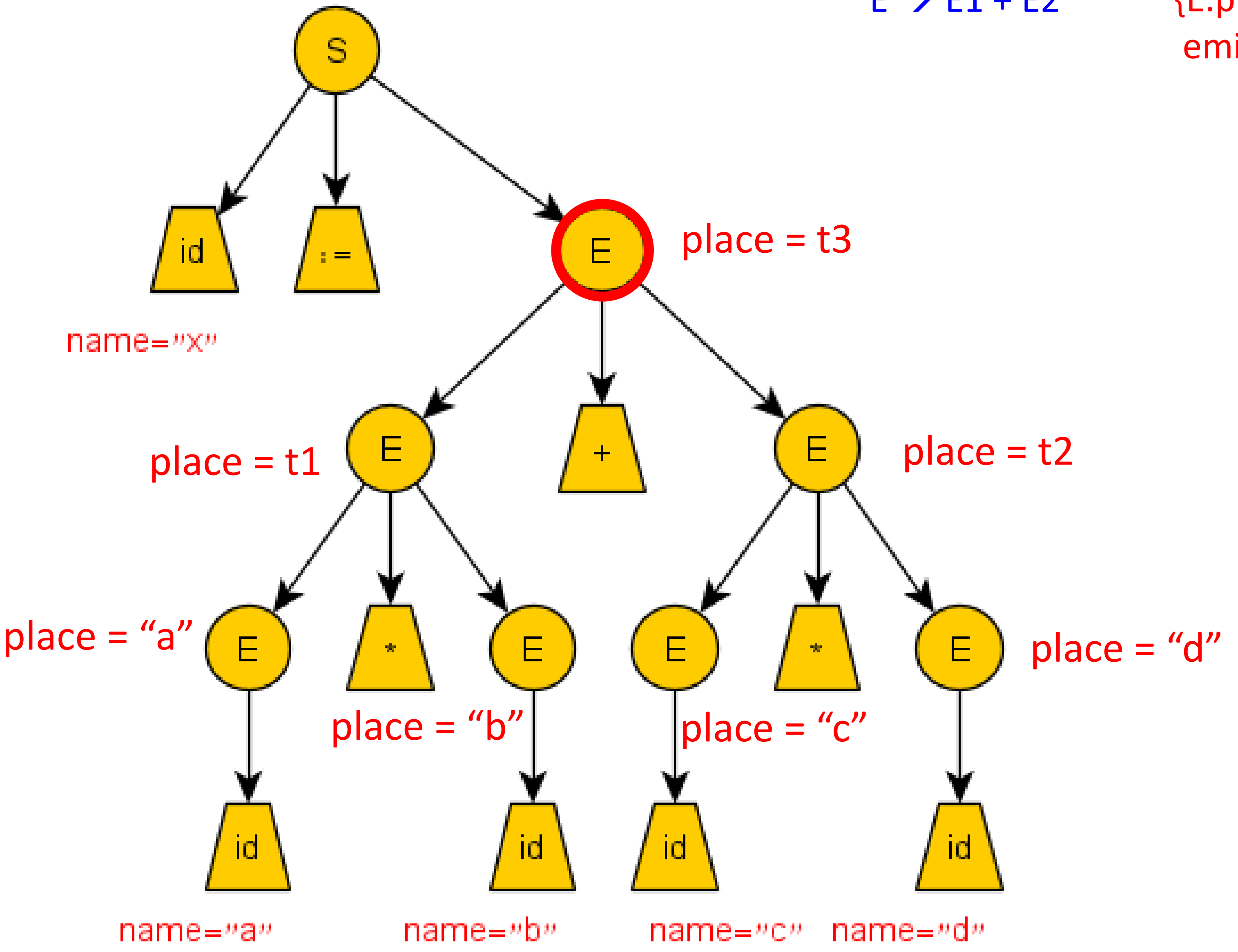

```
t1 := a * b
```



E → E1 * E2 {E.place = newtemp;
emit(E.place ":=" E1.place "*" E2.place) }

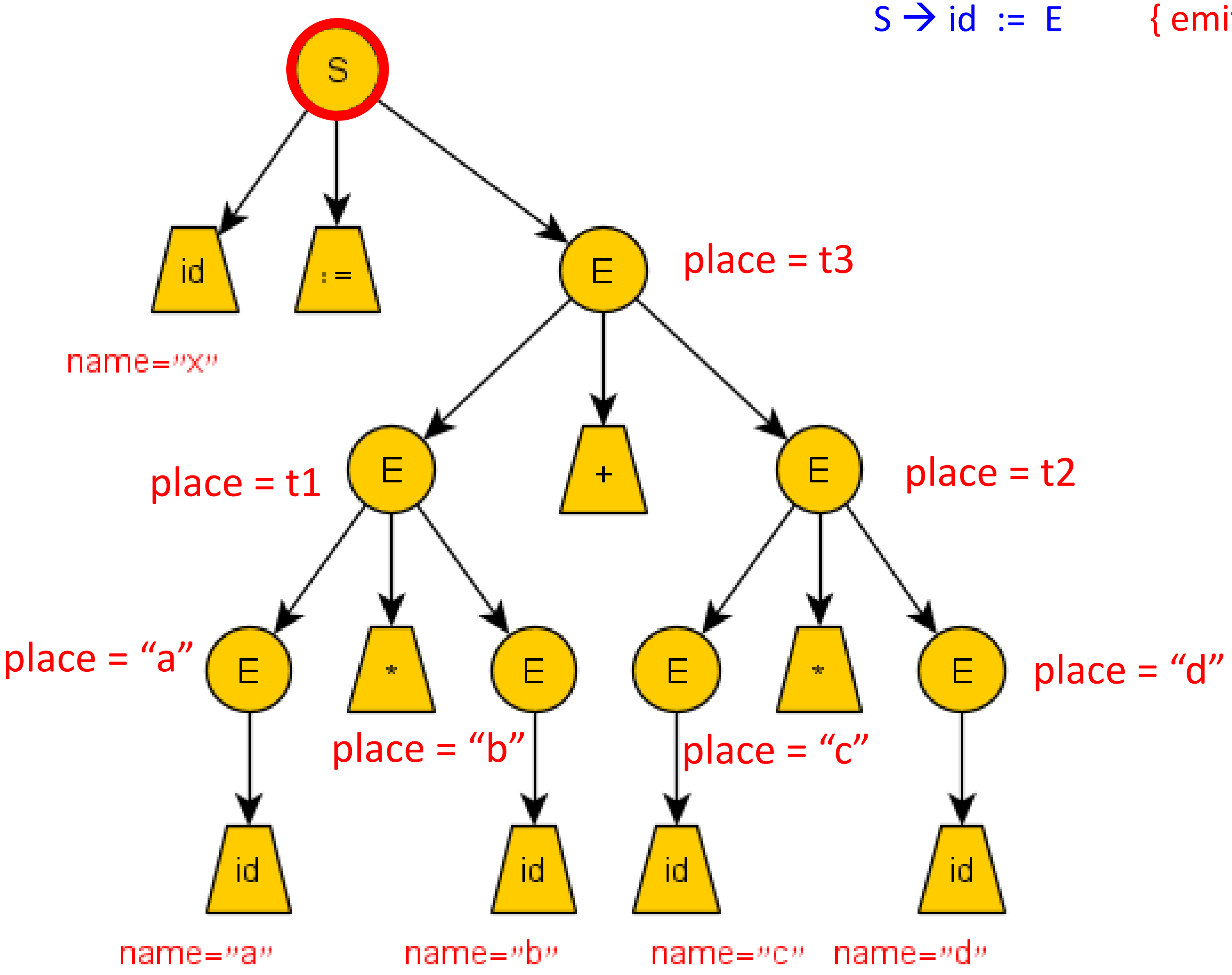
```
t1 := a * b
t2 := c * d
```

$E \rightarrow E1 + E2$ {E.place = newtemp;
emit(E.place ":=" E1.place "+" E2.place) }




```
t1  :=  a * b
t2  :=  c * d
t3  :=  t1 + t2
```

$S \rightarrow id := E$ { emit(id.name ":=" E.place }




```
t1  :=  a * b
t2  :=  c * d
t3  :=  t1 + t2
x   :=  t3
```

Type-sensitive translation

- The above translation scheme is type-insensitive.
- In order to compute the sum of, e.g., integer and real, the integer must be converted to real.

- We assume the existence of TAC command *inttoreal*.
- E.g., for $E \rightarrow E1+E2$:
- We assume:
  - Operator “int+” : integers addition
  - Operator “real+” : reals addition.

E → E1+E2

```
{E.place := newtemp;
if   E1.type==integer and E2.type ==integer
     then {   emit( E.place ":=" E1.place 'int+' E2.place);
              E.type:=integer    }
else if E1.type==real and E2.type ==real
       then { emit(E.place ':=' E1.place 'real +' E2.place);
                E.type:=real  }
else if E1.type==integer and E2.type ==real
       then { u = newtemp;
                emit(u ":=  inttoreal"  E1.place);
                emit(E.place ":="   u  "real+"  E2.place);
                E.type=real  }
else if E1.type== real and E2.type == integer
        then { u = newtemp;
        emit(u " :=  inttoreal"  E2.place);
         emit(E.place ":="  E1.place " real+"  u);
         E.type:=real  }
else  E.type:=type_error;    }
```

Translation of Boolean expressions

- E $\rightarrow$ E or E
- E $\rightarrow$ E and E
- E $\rightarrow$ not E
- E $\rightarrow$ (E)
- E $\rightarrow$ id relop id
- E $\rightarrow$ True
- E $\rightarrow$ False

$E \rightarrow E1$ or $E2$	{ E.place = newtemp; emit(E.place ":=" E1.place "or" E2.place) }
$E \rightarrow E1$ and $E2$	{ E.place = newtemp; emit(E.place ":=" E1.place "and" E2.place) }
$E \rightarrow$ not $E1$	{ E.place = newtemp; emit(E.place ":= not" E1.place) }
$E \rightarrow (E1)$	{ E.place = E1.place}
$E \rightarrow$ true	{ E.place = newtemp; emit(E.place ":=1") }
$E \rightarrow$ false	{ E.place = newtemp; emit(E.place ":=0") }

Assume:

- variable **nextstat** holds the index (label) of the next TAC statement
- each call to emit increments the value of nexstat

E $\rightarrow$ id1 relop id2 { E.place = newtemp;
emit("if" id1.place relop id2.place "goto" nextstat+3);
emit(E.place ':=0');
emit('goto' nextstat+2);
emit(E.place ':=1') }

Assume:
- variable **nextstat** holds the index (label) of the next TAC statement
- each call to emit increments the value of nexstat

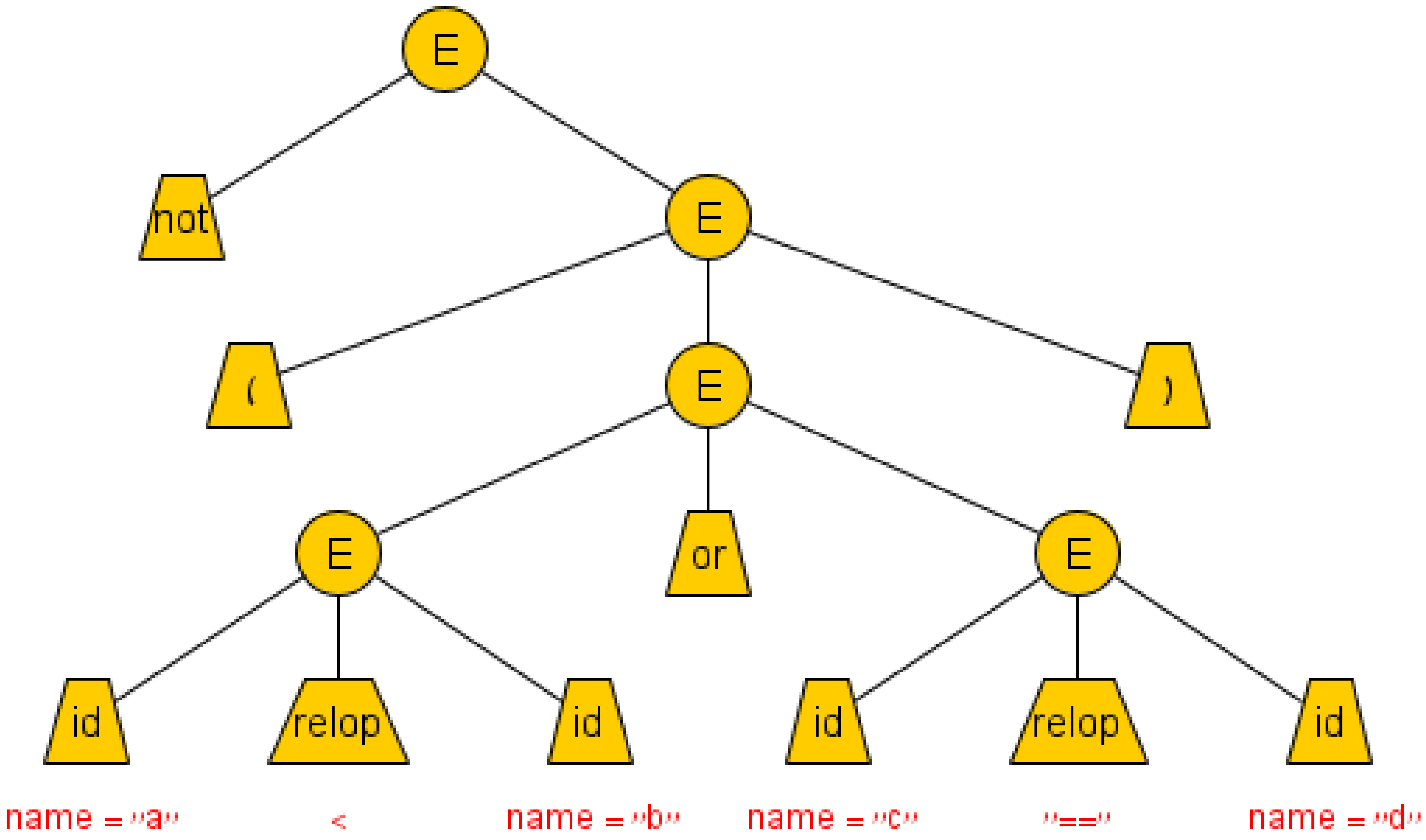
$E \rightarrow$ id1 relop id2 { E.place := newtemp;
emit(“if” id1.place relop id2.place “goto” nextstat+3);
emit(E.place ’:=0’);
emit(‘goto’ nextstat+2);
emit(E.place ’:=1’) }

Assume:
id1.place = a
Id2.place = b
E1.place = t17
E.place = t22
nextstat = 125 (at first)

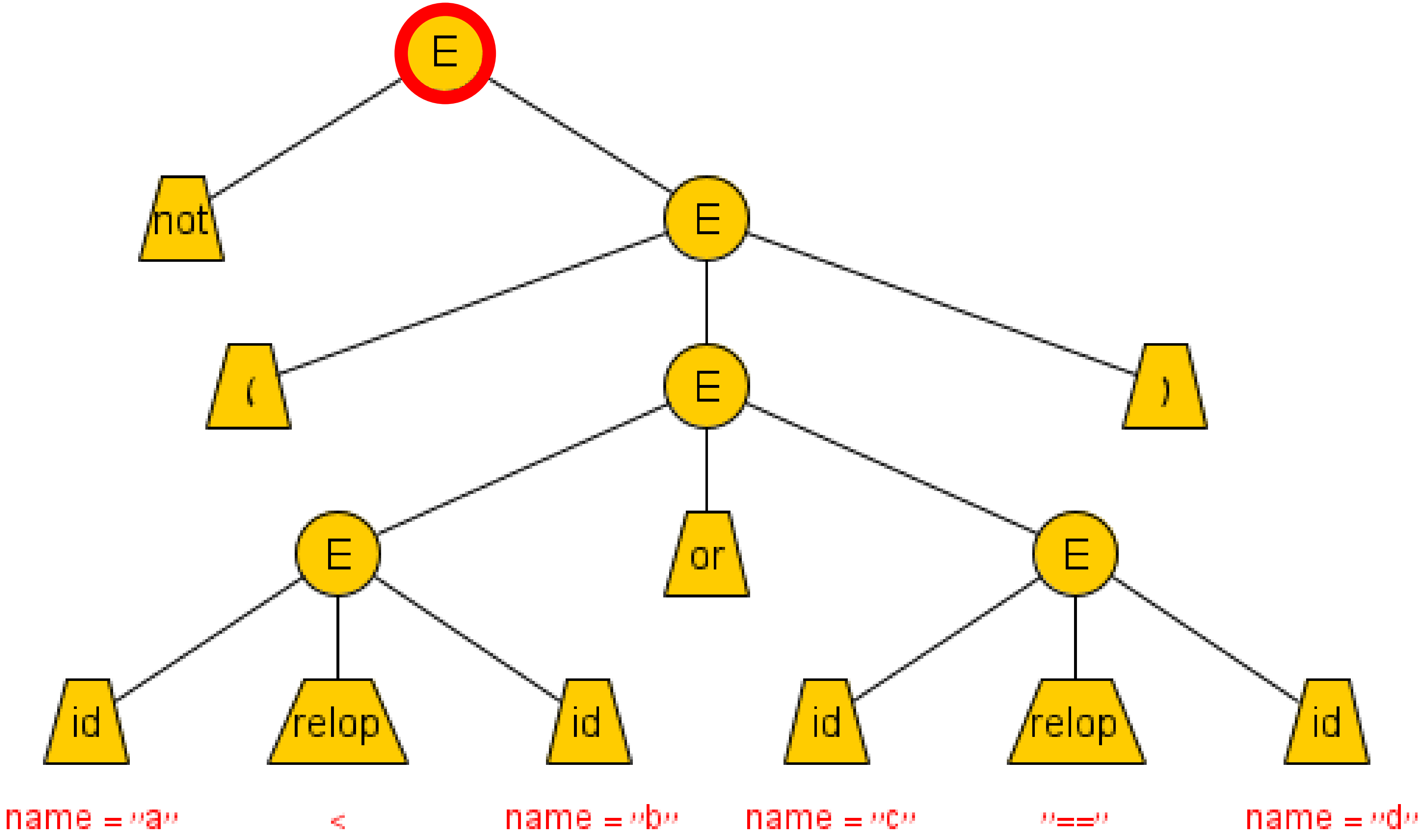
nextstat = 129

125:	**if a <= b goto 128**
126:	**t22 := 0**
127:	**goto 129**
128:	**t22 := 1**
129:	

Example: not$(a < b\ or\ c == d)$



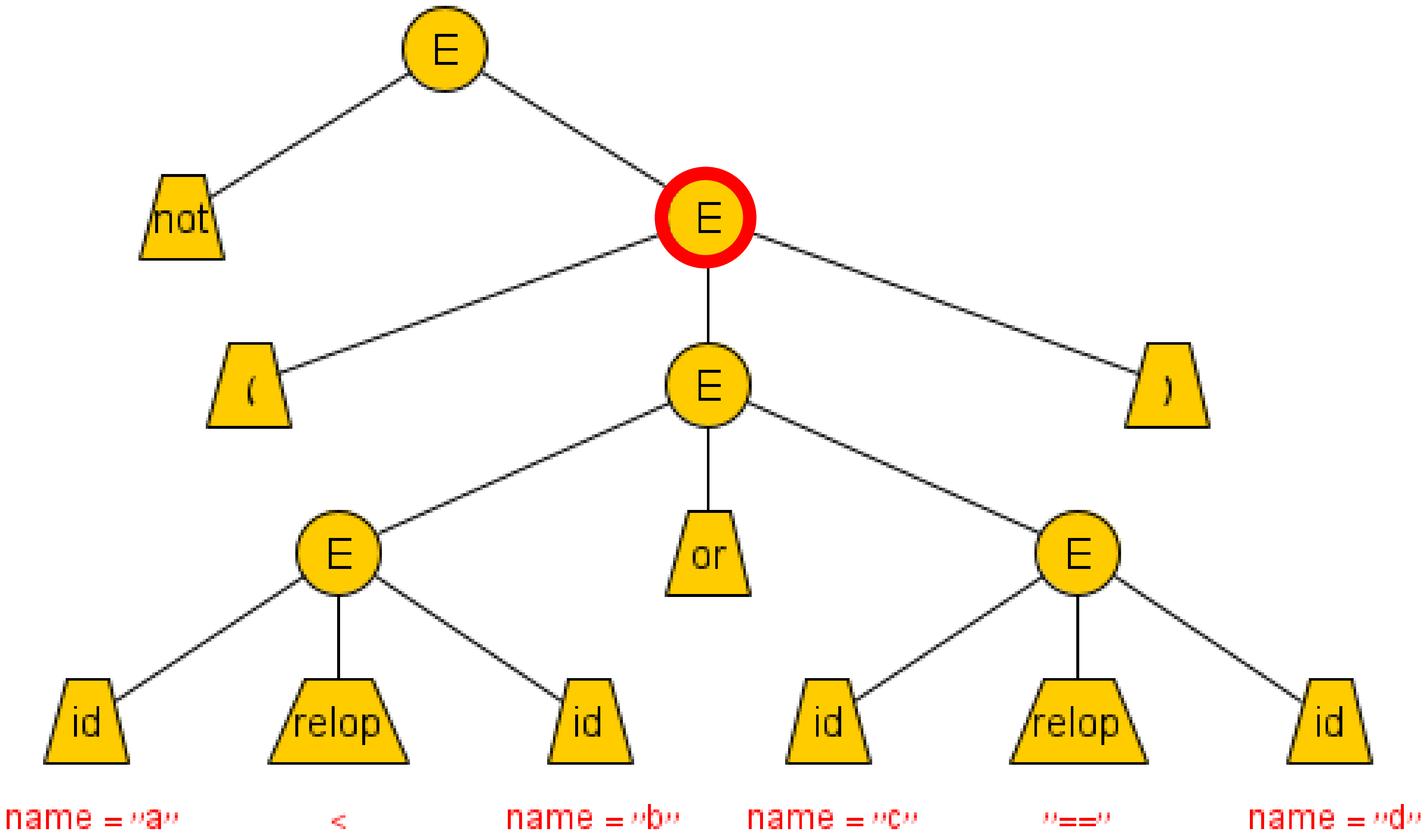
$E \rightarrow$ not E1



nextstat = 1

nextemp = 1

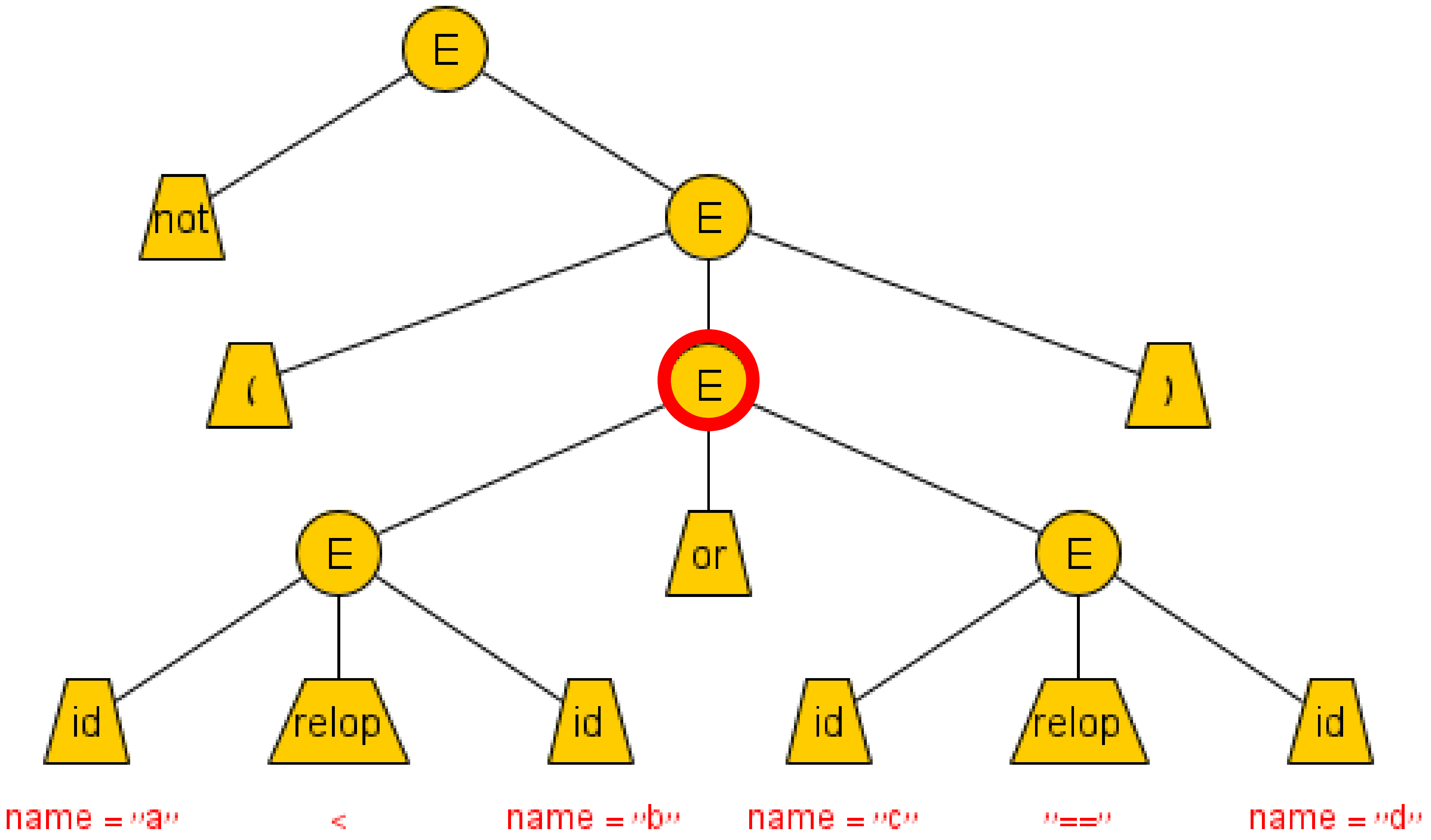
$E \rightarrow (E1)$



nextstat = 1

nextemp = 1

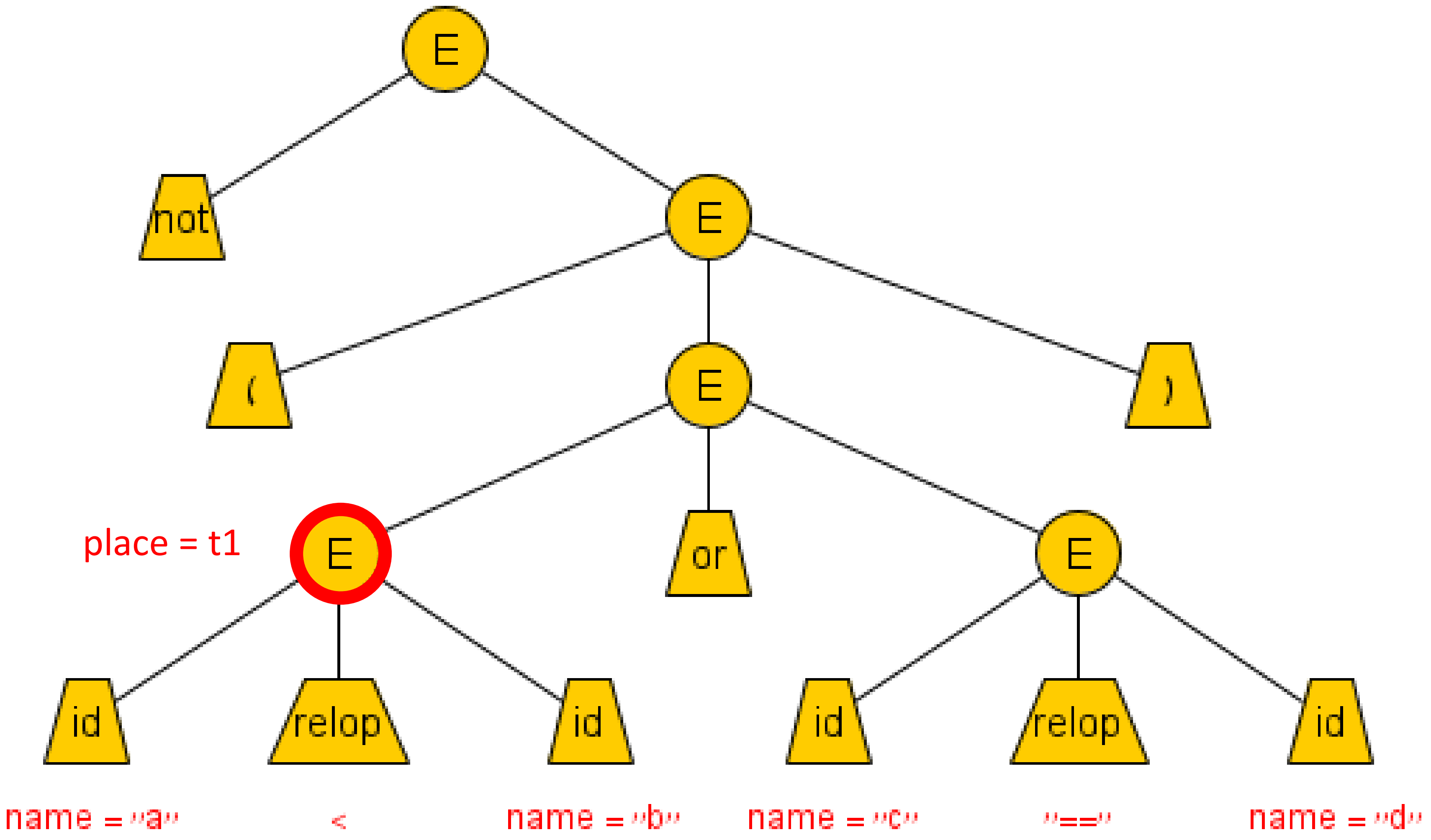
E $\rightarrow$ E1 or E2



nextstat = 1

nextemp = 1

E → id1 relop id2 { E.place = newtemp;
emit(“if” id1.place relop id2.place “goto” nextstat+3);
emit(E.place ’:=0’);
emit(‘goto’ nextstat+2);
emit(E.place ’:=1’) }

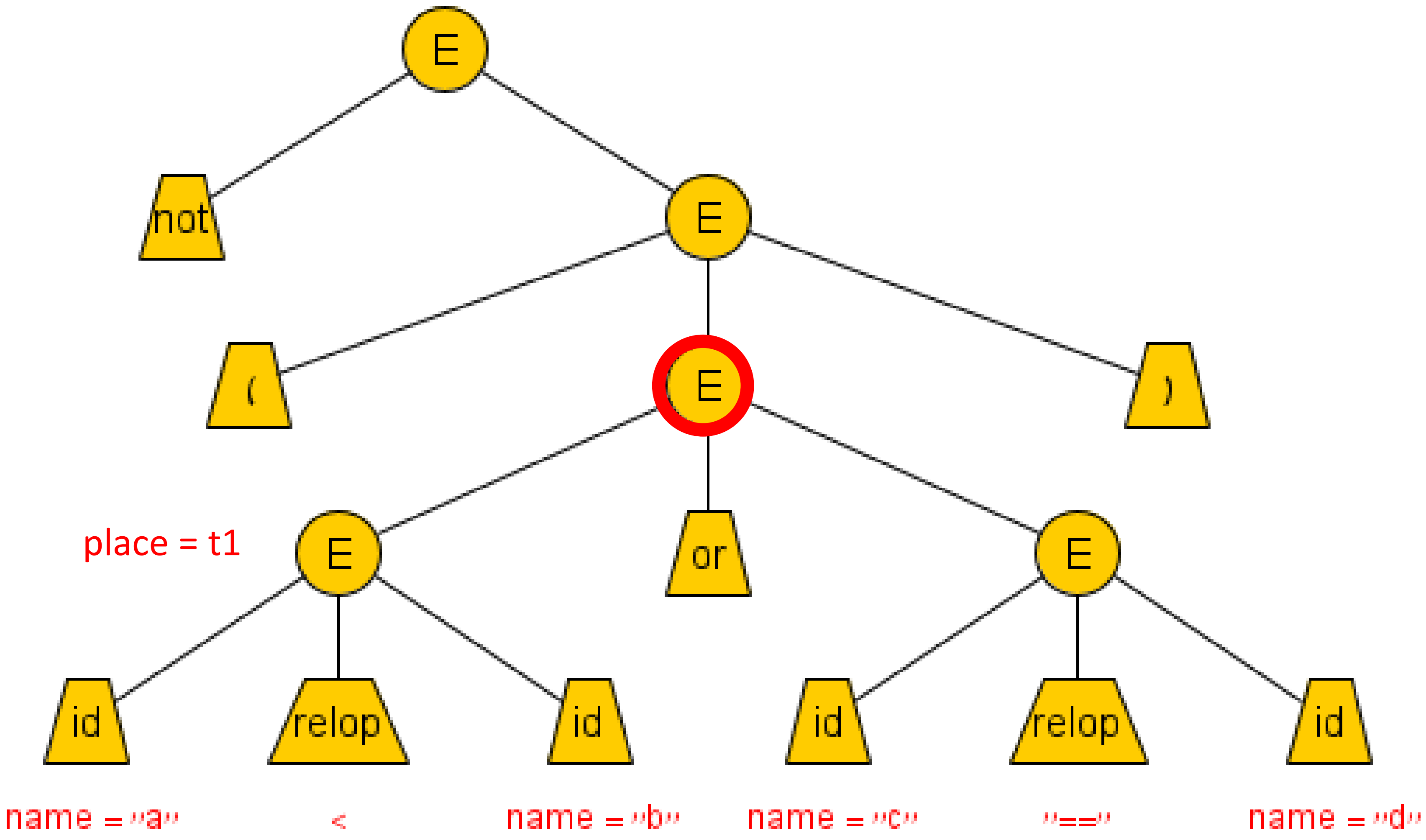


nextstat = 5

nextemp = 2

1: if a < b goto 4
2: t1 := 0
3: goto 5
4: t1 := 1

$E \rightarrow E1$ or $E2$



nextstat = 5

nextemp = 2

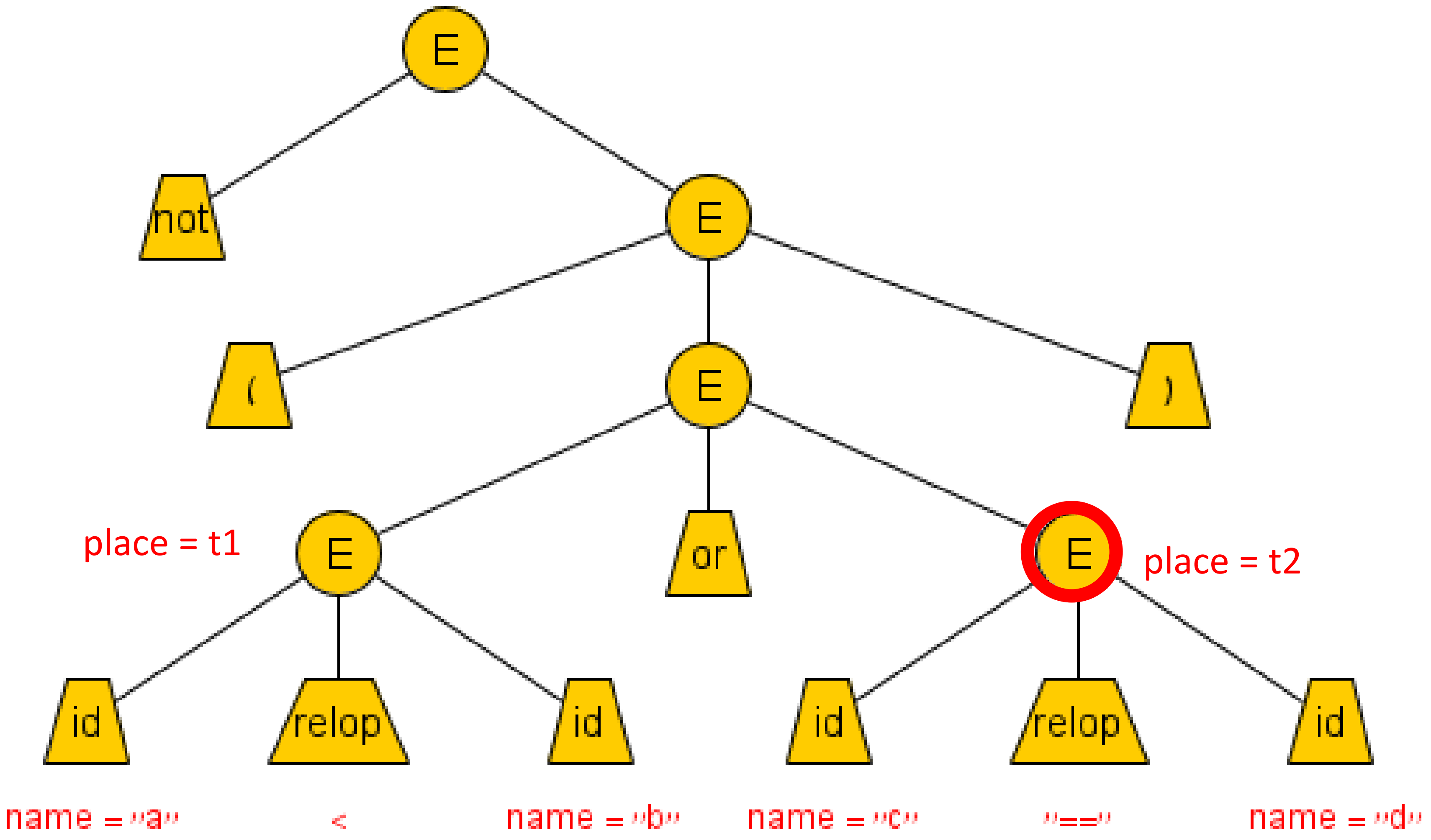
1: if a < b goto 4

2: t1 := 0

3: goto 5

4: t1 := 1

$E \rightarrow$ id1 relop id2 { E.place = newtemp;
emit("if" id1.place relop id2.place "goto" nextstat+3);
emit(E.place ':=0');
emit('goto' nextstat+2);
emit(E.place ':=1') }



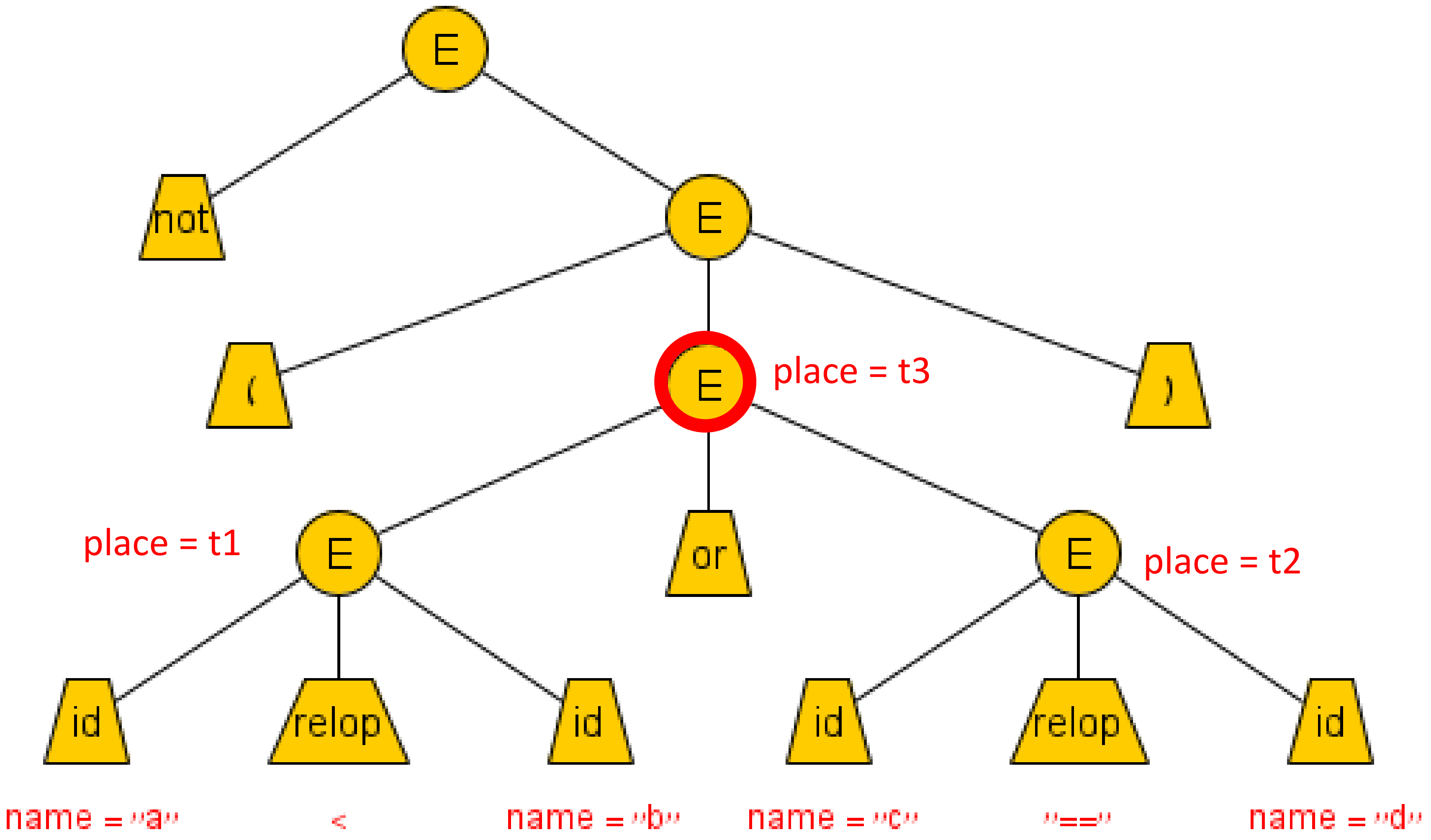
nextstat = 9

nextemp = 3

1: if a < b goto 4
2: t1 := 0
3: goto 5
4: t1 := 1
5: if c == d goto 8
6: t2 := 0
7: goto 9
8: t2 := 1

$E \rightarrow E1$ or $E2$

{ E.place = newtemp;
emit(E.place ":=" E1.place "or" E2.place) }

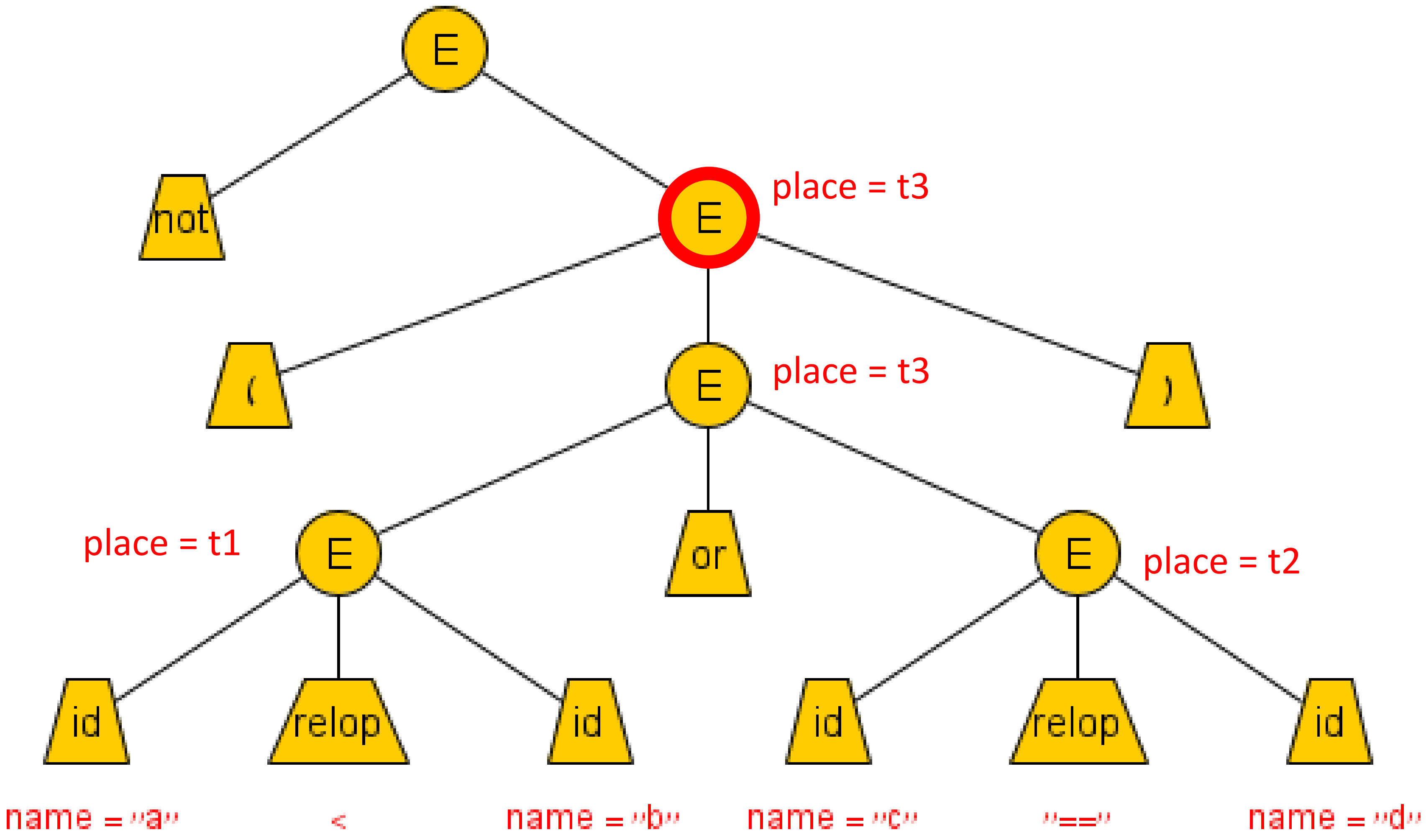


nextstat = 10

nextemp = 4

```
1:   if a < b goto 4
2:   t1 := 0
3:   goto 5
4:   t1 := 1
5:   if c == d goto 8
6:   t2 := 0
7:   goto 9
8:   t2 := 1
9:   t3 := t1 or t2
```

E → (E1) { E.place = E1.place}

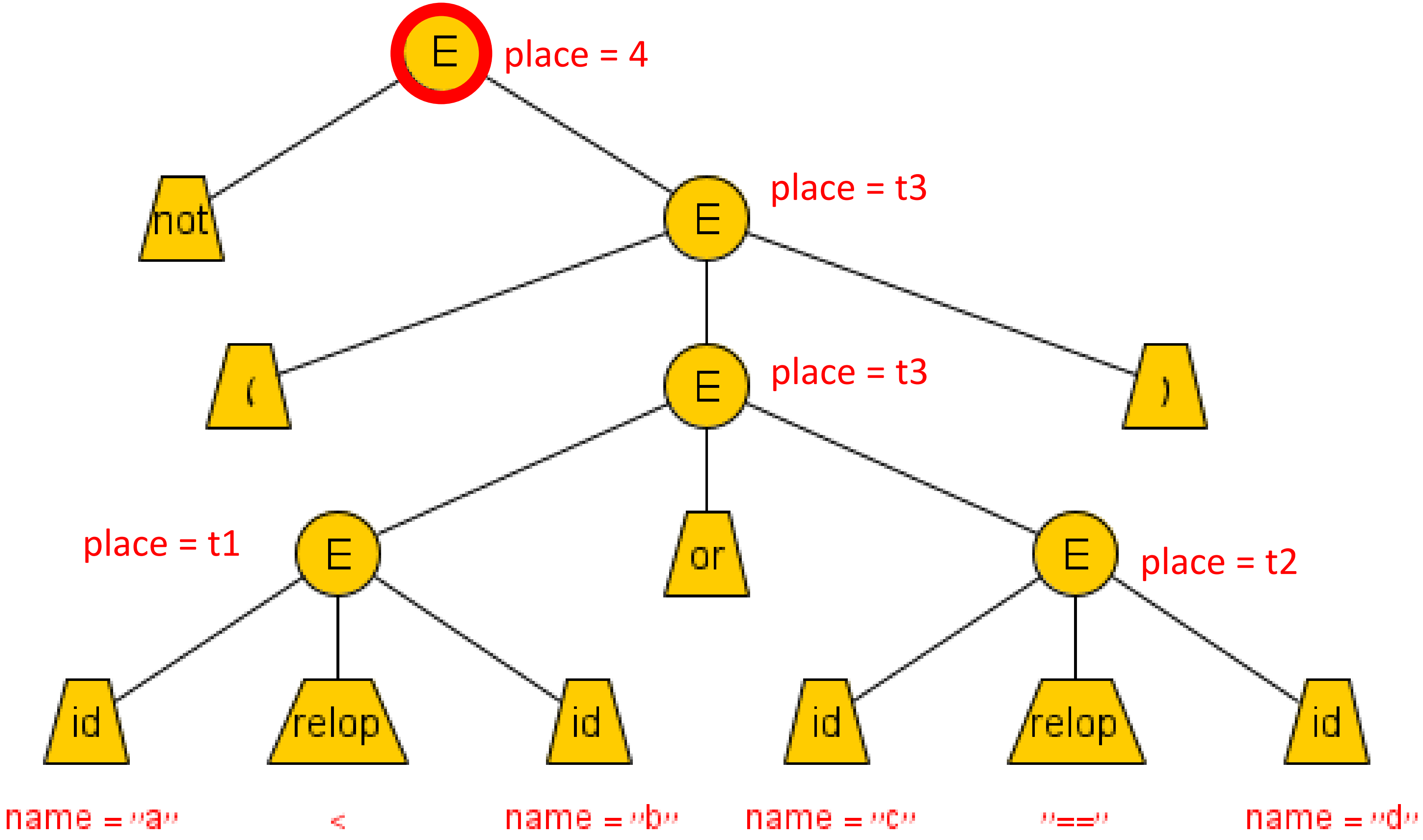


nextstat = 10

nextemp = 4

1: if a < b goto 4
2: t1 := 0
3: goto 5
4: t1 := 1
5: if c == d goto 8
6: t2 := 0
7: goto 9
8: t2 := 1
9: t3 := t1 or t2

E $\rightarrow$ not E1 { E.place = newtemp;
emit(E.place ":= not" E1.place) }



nextstat = 11

nextemp = 5

1: if a < b goto 4
2: t1 := 0
3: goto 5
4: t1 := 1
5: if c == d goto 8
6: t2 := 0
7: goto 9
8: t2 := 1
9: t3 := t1 or t2
10: t4 := not t3

Translation of Control statements

- Control statements : conditional and loops.

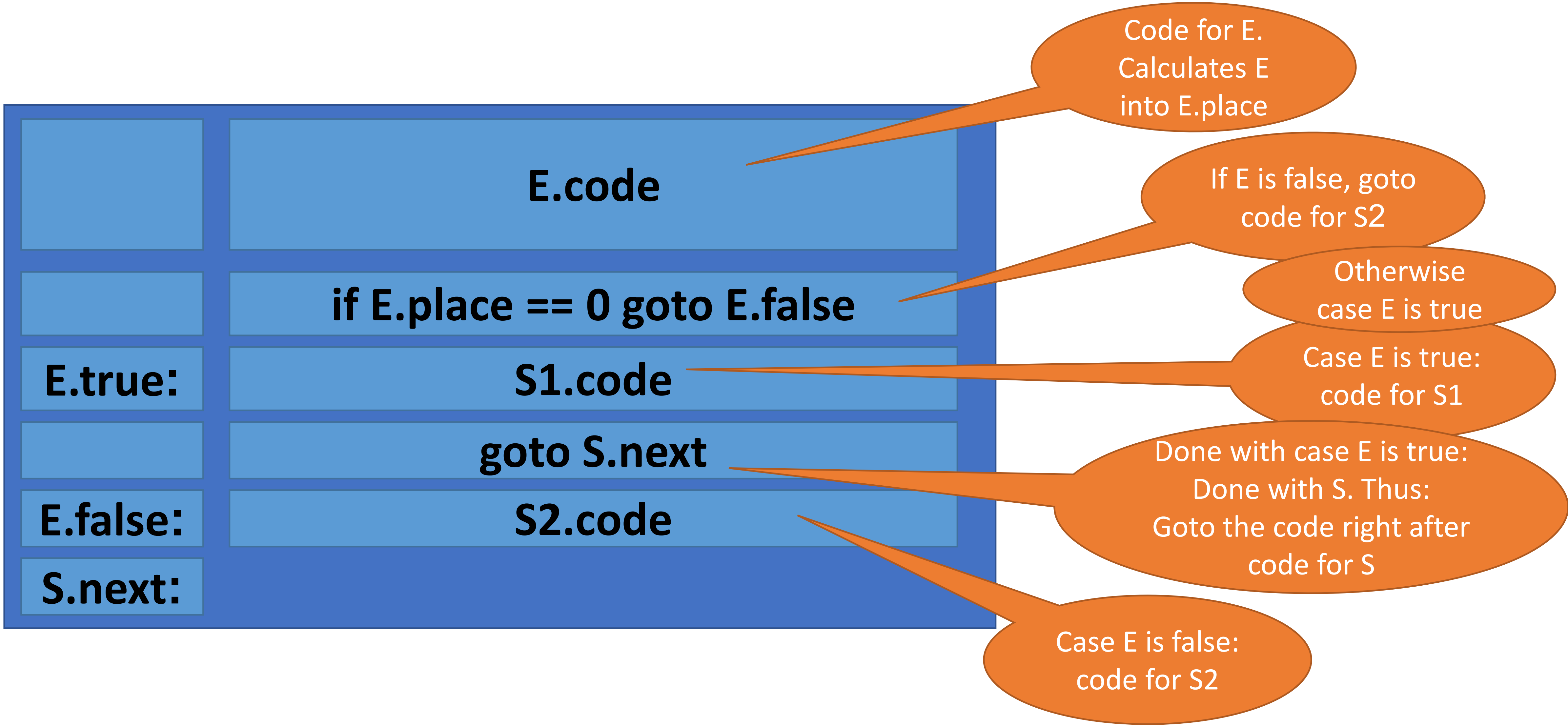
  S $\rightarrow$ if E then S1 else S2

  S $\rightarrow$ if E then S1

  S $\rightarrow$ while E do S1

- switch – case statement

if-then-else : S $\rightarrow$ if E then S1 else S2



Required for if-then-else

- E.place
- E.code
- S.code

As before

- E.true – label for branching in case E is true.
- E.false – label for branching in case E is false.
- S.next – label of first command right after S.code. **Inherited attribute**.

- *Newlabel* – function. Generates a new label

$S \rightarrow$ if E then S1 else S2

$S \rightarrow$ if E then {S1.next = S.next;} S1
else {S2.next = S.next;} S2
{ E.true = newlabel;
E.false = newlabel;
S.code =
E.code ||
"if" E.place " == 0 goto" E.false ||
E.true || ":" || S1.code ||
"goto" S.next ||
E.false || ":" || S2.code ||
S.next || ":"
}

S → if E then S1

S → if E then {S1.next = S.next;} S1

{ E.true = newlabel;

E.false = S.next;

S.code =

E.code ||

"if" E.place " == 0 goto" E.false ||

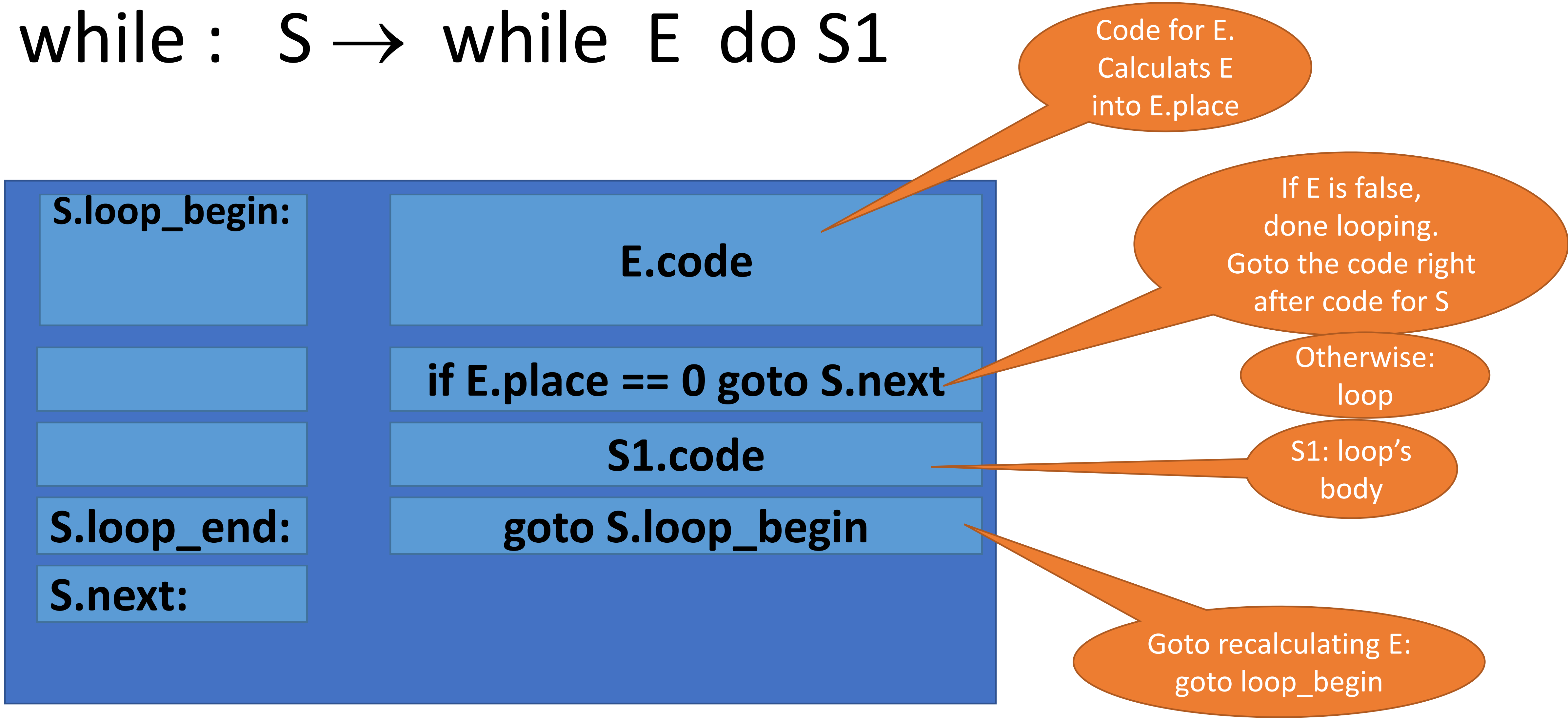
E.true || ":" || S1.code ||

S.next || ":"

}

Case E is false:
Done with if statement.
Goto the code right after
code for S

while : S → while E do S1



Required for while-do

- E.place
- E.code
- S.code
- S.next
- *Newlabel*

As before

- S.loop_begin – label of first command of the loop (computation of E)
- S.loop_end – label of last command of the loop (goto back to loop's start)

$S \rightarrow$ while E do S1

$S \rightarrow$ while E do { S.loop_begin = newlabel;
S.loop_end = newlabel;
S1.next = S.loop_end }

S1

{ S.code =
S.loop_begin ':' || E.code ||
"if" E.place "== 0 goto" S.next ||
S1.code ||
S.loop_end ": goto" || S.loop_begin ||
S.next || ":" }

switch – case statement

The basic switch statement is:

“int_num”s

```
switch E
{
    case V1: S1;
    case V2: S2;
    ...
    case Vn-1: Sn-1;
    default: Sn;
}
```

Solution 1

	E.Code
	if E.place != V1 goto L1
	S1.code
	goto S.next
L1:	if E.place != V2 goto L2
	S2.code
	goto S.next
L2:	if E.place != V3 goto L3
	S3.code
	goto S.next
Ln-1:	Sn.code
S.next:	

Solution 2

	E.Code
	goto TEST
L1:	S1.code
	goto S.next
L2:	S2.code
	goto S.next
Ln:	Sn.code
	Goto S.next
TEST:	if E.place==V1 goto L1
	if E.place==V2 goto L2
S.next:	