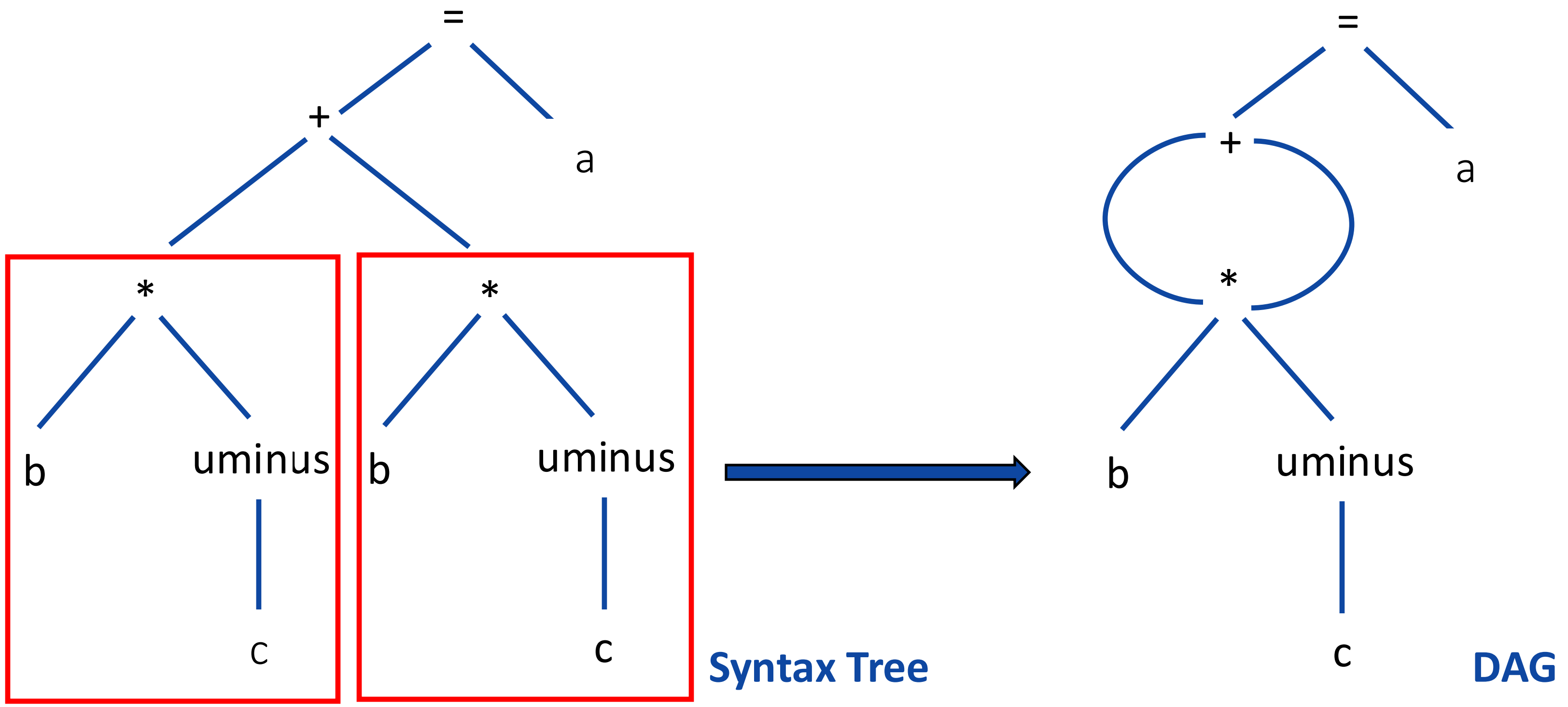
Module 4 – Intermediate Code Generation

Different Intermediate Forms

- Abstract syntax tree
- Postfix notation
- Three address code

Abstract syntax tree & DAG

- A syntax tree depicts the natural hierarchical structure of a source program.
- A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Ex: a=b*-c+b*-c



Postfix Notation

- Postfix notation is a linearization of a syntax tree.
- In postfix notation the operands occurs first and then operators are arranged.
- Ex: **(A + B) * (C + D)**

  Postfix notation: A B + C D + *

- Ex: **(A + B) * C**

  Postfix notation: A B + C *

- Ex: **(A * B) + (C * D)**

  Postfix notation: A B * C D * +

Three address code

- Three address code is a sequence of statements of the general form,

  a:= b op c

- Where a, b or c are the operands that can be names or constants and op stands for any operator.
- Example: a = b + c + d

  t_1=b+c

  t_2=t_1+d

  a= t_2

- Here t_1 and t_2 are the temporary names generated by the compiler.
- There are at most three addresses allowed (two for operands and one for result). Hence, this representation is called three-address code.

Different Representation of Three Address Code

- There are three types of representation used for three address code:
  1. Quadruples
  2. Triples
  3. Indirect triples
- Ex: $x = -a*b + -a*b$

  $t_1 = -a$

  $t_2 = t_1 * b$

  $t_3 = -a$

  $t_4 = t_3 * b$

  $t_5 = t_2 + t_4$

  $x = t_5$

Quadruple

- The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- The op field is used to represent the internal code for operator.
- The arg1 and arg2 represent the two operands.
- And result field is used to store the result of an expression.

Quadruple

$x = -a*b + -a*b$

$t_1 = -a$

$t_2 = t_1 * b$

$t_3 = -a$

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

$x = t_5$

No.	Operator	Arg1	Arg2	Result
(0)				
(1)				
(2)				
(3)				
(4)				
(5)				

Triple

- To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

Quadruple

No.	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Triple

No.	Operator	Arg1	Arg2
(0)			
(1)			
(2)			
(3)			
(4)			
(5)			

Indirect Triple

- In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- This implementation is called indirect triples.

Triple

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect Triple

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*		b
(2)	uminus	a	
(3)	*		b
(4)	+		
(5)	=	x	

Exercise

Write quadruple, triple and indirect triple for following:

1. -(a*b)+(c+d)
2. a*-(b+c)
3. x=(a+b*c)^(d*e)+f*g^h
4. z=g+a*(b-c)+(x-y)*d

Types of three address statements

1. Binary Assignment statements
2. Unary Assignment statements
3. Copy statements
4. Unconditional jump
5. Conditional jump
6. Procedural call
7. Indexed assignments
8. Address and pointer assignments

1. & 2. Assignment Statements

1. Binary Assignment statements
    - Syn: X = Y op Z where X,Y,Z are compiler generated statements
    - Ex: a = b+c*d

   Three address statement

   t1 = c * d

   t2 = b + t1

   a = t2

2. Unary Assignment statements
    - Syn: X = op Y
    - Ex: a = b*-c

   Three address statement

   t1 = -c

   t2 = b * t1

   a = t2

3. Copy statements

Syn: X = Y

Ex: a = t2

Three address statement

a = t2

4. Unconditional jump & 5. Conditional jump

- Unconditional jump
  - Syn: goto L [the control will be transferred to the three address statement labelled with L]

- Conditional jump
  - Syn: if x relop y goto L
  - Ex: if x > y goto L
    - True → the control will be transferred to statement labelled with L
    - False → next statement after conditional jump statement

6. Procedure call

- Syn: P (x1, x2, …, xn)
- Three address statement

  Param x1
  Param x2
  …
  Param xn
  Call P, n

- Syn: return y
- Three address statement

  return y

7. Indexed Assignments

- Syn: x = y[index] & y[index] = x
- Ex: a[i]=b[i]

Three address statement

t=b[i]

a[i]=t

8. Address and pointer assignments

- Syn: x = & y, x = *y, *x = y
- Ex: *x = *y

Three address statement

t = *y

*x = t

Boolean Expression (True / False)

1. Compute logical values (T / F)
2. Conditional expressions in flow of control statements
   - Ex: if BE then S
     if BE then S1 else S2
     while BE do S

- Boolean operators: and, or, not

Operators	Precedence	Associativity
or	3 [Low]	Left
and	2	Left
not	1 [High]	Right

1. SDT to construct 3AC for Booleans

- $E \rightarrow E_1 \;\; or \;\; E_2$
- $E \rightarrow E_1 \;\; and \;\; E_2$
- $E \rightarrow not \;\; E_1$
- $E \rightarrow \;\; (E_1)$
- $E \rightarrow id_1 \;\; rel\_op \;\; id_2$
- $E \rightarrow true$
- $E \rightarrow false$

$E \rightarrow E_1 \ or \ E_2$

- $\{ E.place = newtemp,$
- $emit(E.place = \ E_1.place \ 'or' \ E_2.place\}$

$E \rightarrow E_1 \ and \ E_2$

- $\{ E.place = newtemp,$
- $emit(E.place = \ E_1.place \ 'and' \ E_2.place\}$

$E \rightarrow not\ \ E_1$

- $\{\ E.place = newtemp,$
- $emit(E.place =\ not\ E_1.place\}$

$E \rightarrow (E_1)$

- $\{\ E.place = newtemp,$
- $emit(E.place =\ E_1.place\}$

$E \rightarrow id_1 \;\; rel\_op \;\; id_2$

- $\{\, E.place = newtemp,$
- $emit\ (if\ id_1.place\ relop\ id_2.place\ goto\ address + 3);$
- $emit(E.place = 0);$
- $emit(goto\ address + 2);$
- $emit(E.place = 1);\}$

$E \rightarrow true$

- $\{E.place = newtemp$
- $emit(E.place = 1)$

$E \rightarrow false$

- $\{E.place = newtemp$
- $emit(E.place = 0)$

Example

1. a or b and not c

Three address representation

t1 = not c

t2 = b and t1

t3 = a or t2

2. a < b

if a < b then 1 else 0

Three address representation

100: if a < b then goto 103 [*100 + 3*]

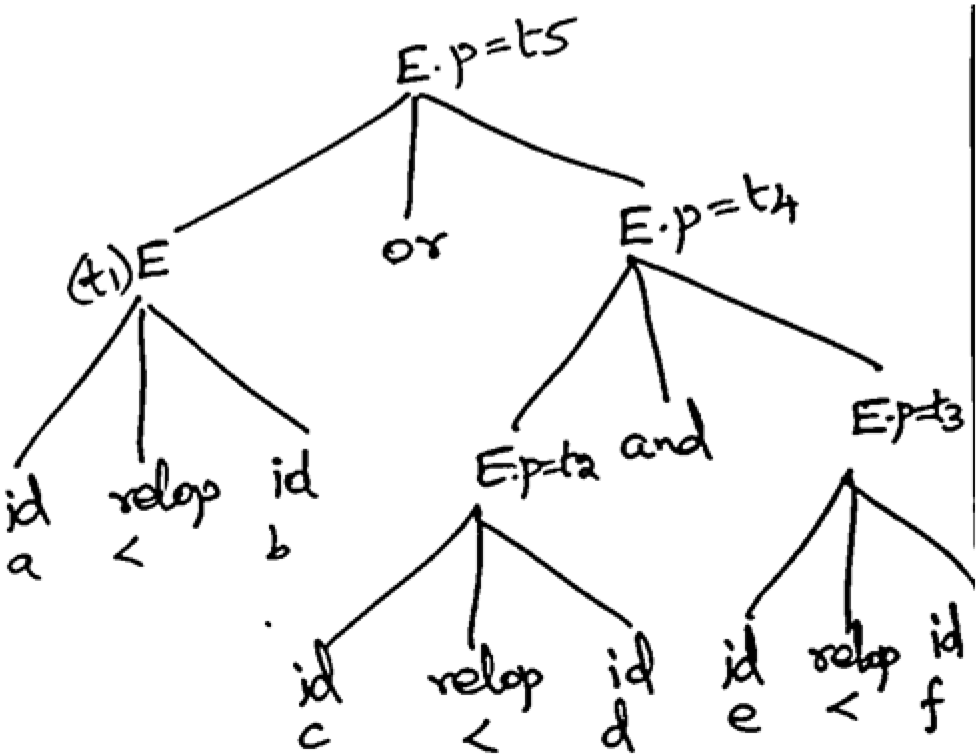
101: t = 0

102: goto 104 [*102 + 2*]

103: t = 1

Example

- a<b or c<d and e<f



100: if a<b then goto 103
101: t1=0
102: goto 104
103: t1=1
104: if c<d then goto 107
105: t2=0
106: goto 108
107: t2=1
108: if e<f then goto 111
109: t3=0
110: goto 112
111: t3=1
112: t4 = t2 and t3
113: t5 = t1 or t4

2. Control Flow Translation of Boolean Expressions [Short circuit code or jump code]

- Generate code without generating code for Boolean operators
- Generate code without evaluating the entire expression
- Ex:

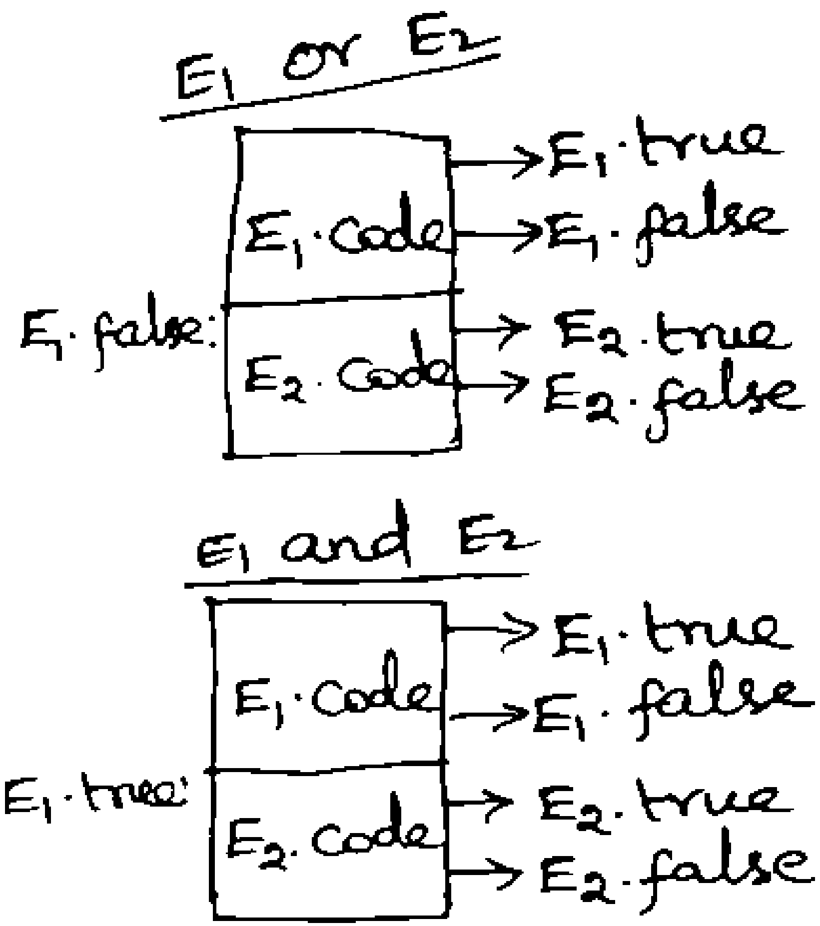
  A < B

  If A<B goto E.true

  Goto E.false

SDT to construct 3AC for Booleans

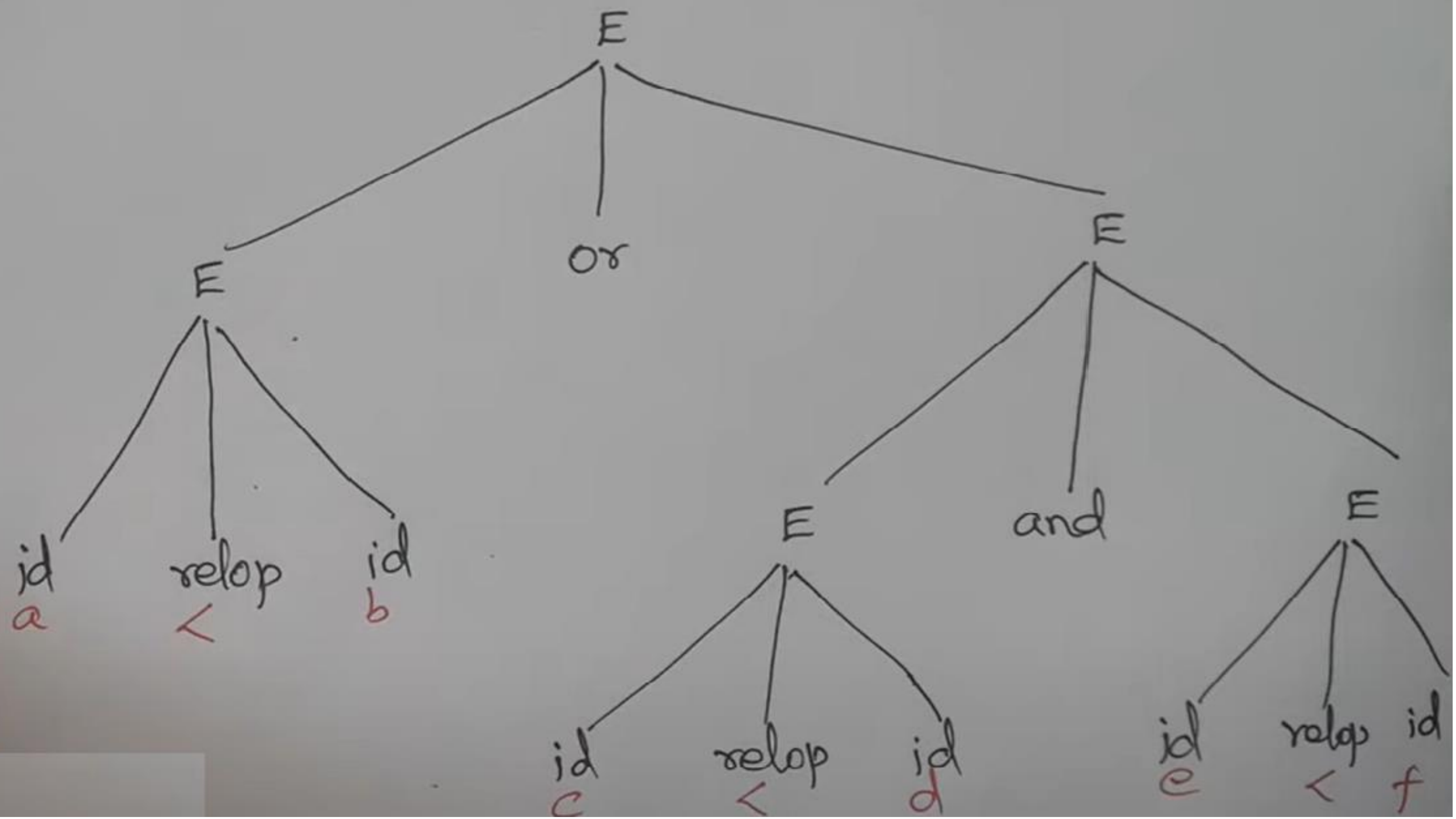
Production	Semantic Rules
$E \rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code
$E \rightarrow E_1$ and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code



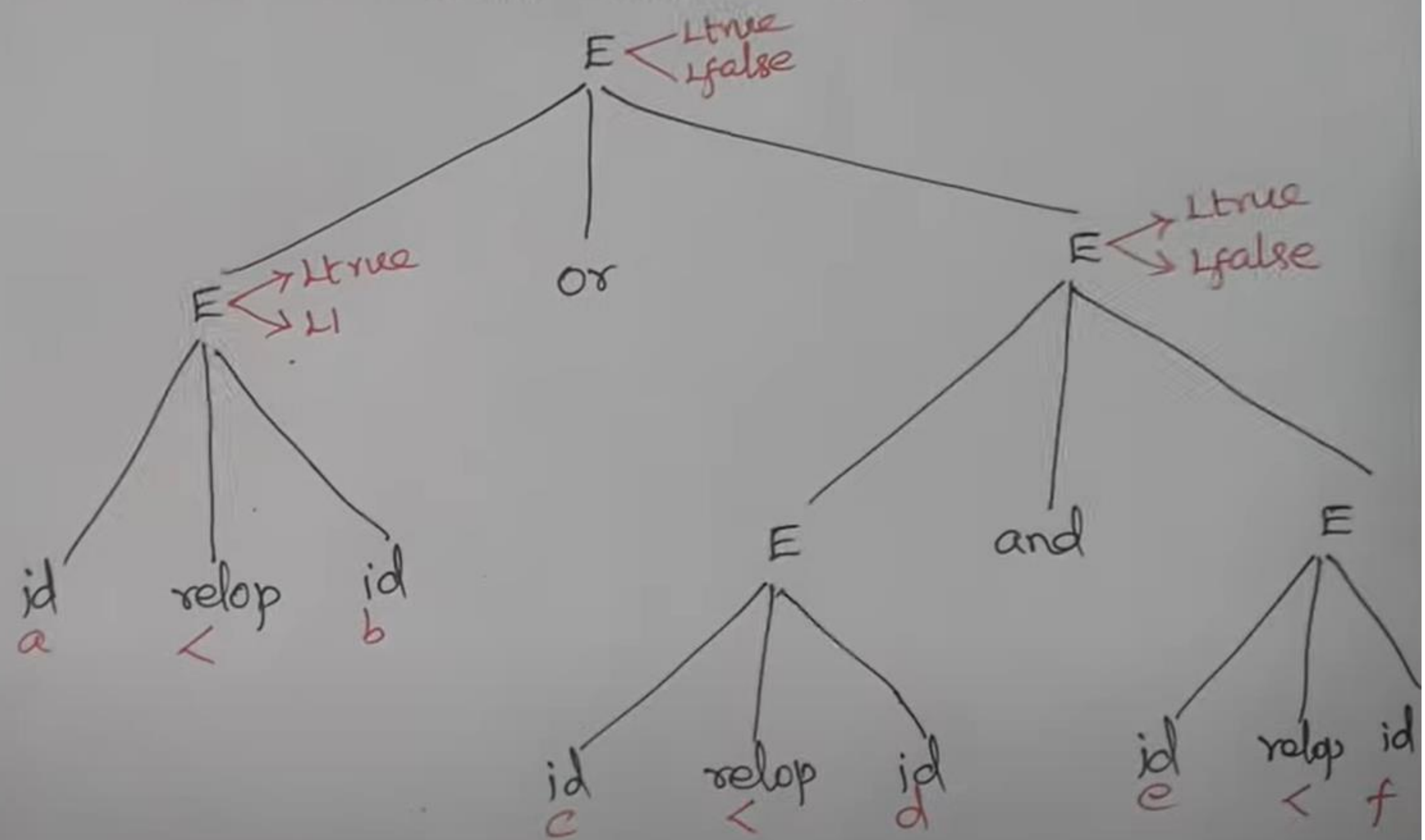
SDT to construct 3AC for Booleans – cont…

Production	Semantic Rules
$E \rightarrow$ not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E1.code
$E \rightarrow (E_1)$	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code
$E \rightarrow id_1$ relop id_2	E.code := gen('if' id.place relop.op id2.place 'goto' E.true) || gen('goto' E.false)
$E \rightarrow$ true	E.code := gen('goto' E.true)
$E \rightarrow$ false	E.code := gen('goto' E.false)

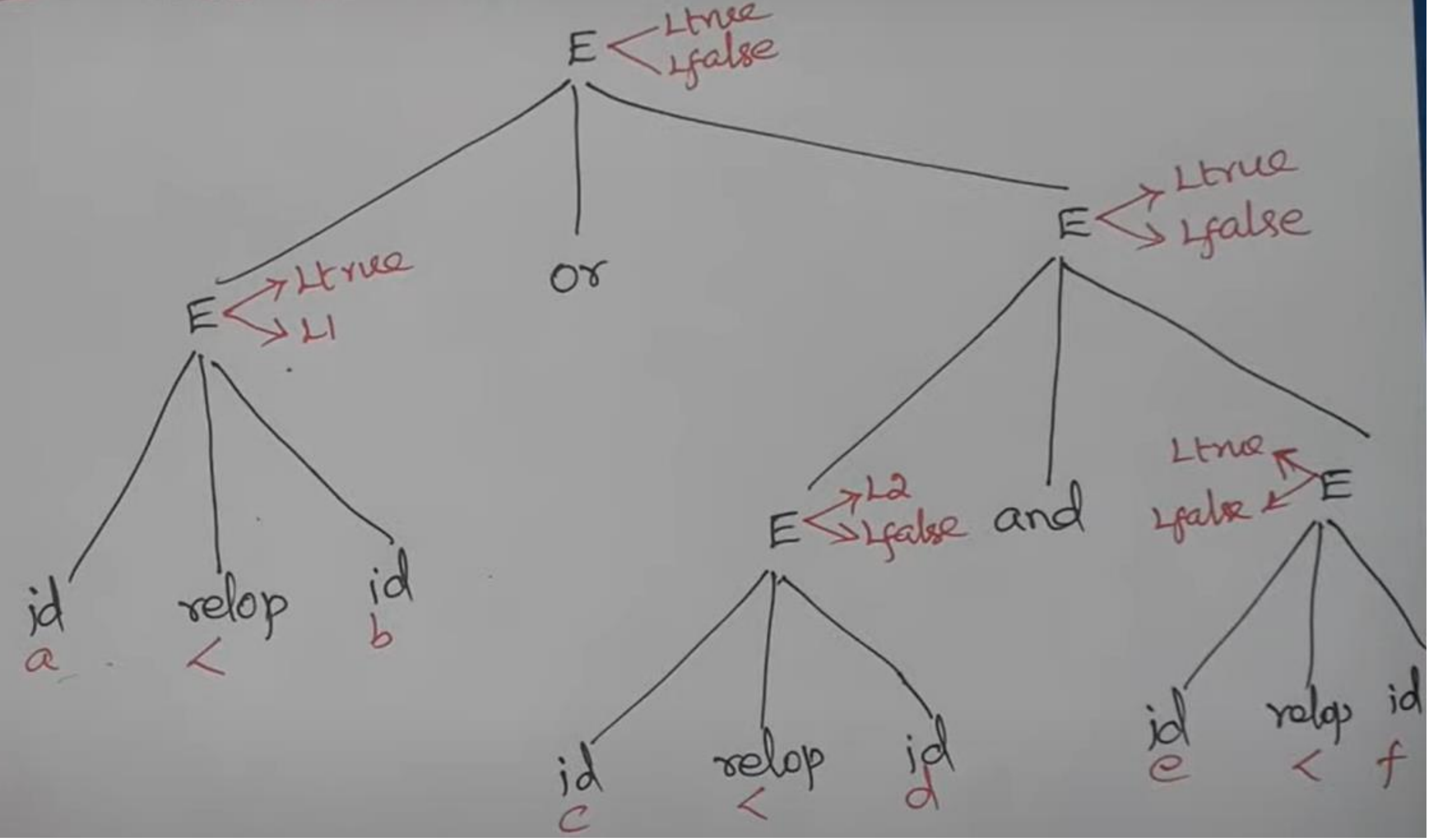
Ex: a<b or c<d and e<f



Ex: a<b or c<d and e<f



EX: a<b or c<d and e<f



Three Address Code

```
if a<b goto Ltrue
goto L1
L1: if c<d goto L2
    goto Lfalse
L2: if e<f goto Ltrue
    goto Lfalse
```

Assignment Statement

Production rule	Semantic actions

S → id :=E	{p = look_up(id.name);
	 If p ≠ nil then
	 Emit (p = E.place)
	 Else
	 Error;
	}

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 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → E1 / E2	{E.place = newtemp(); Emit (E.place = E1.place '/' E2.place) }

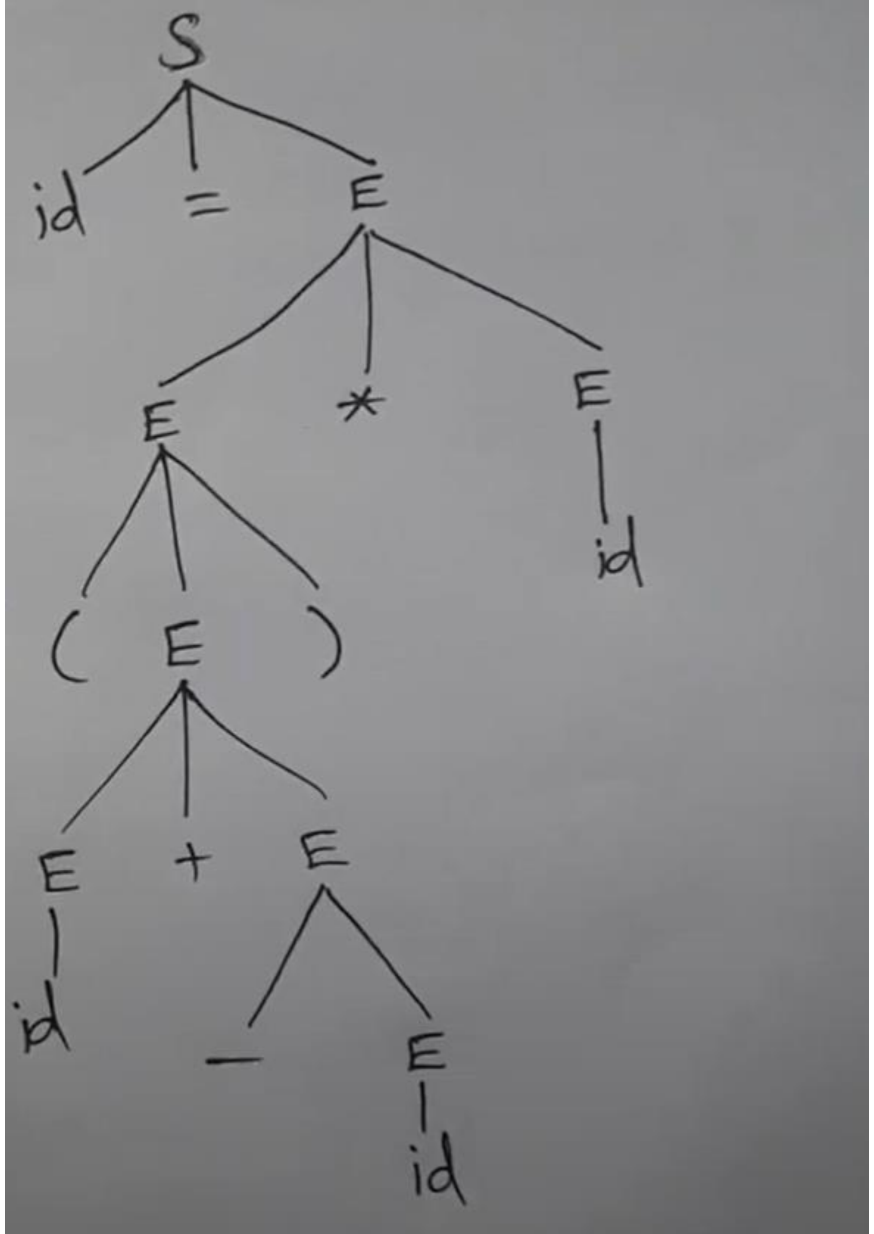
Assignment Statement

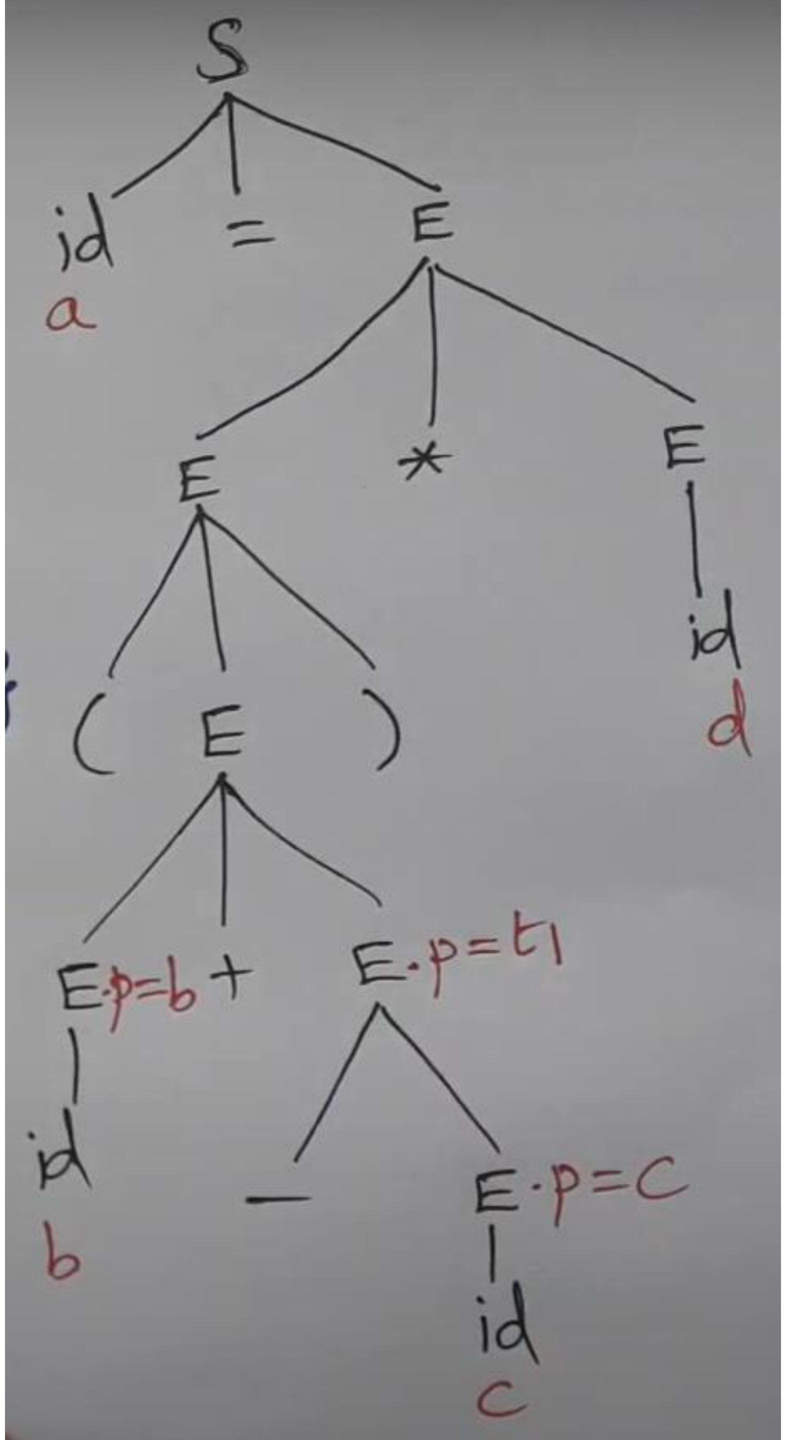
Production rule	Semantic actions
E → - E1	{E.place = newtemp(); Emit (E.place = 'uminus' E1.place) }
E → (E1)	{E.place = E1.place}

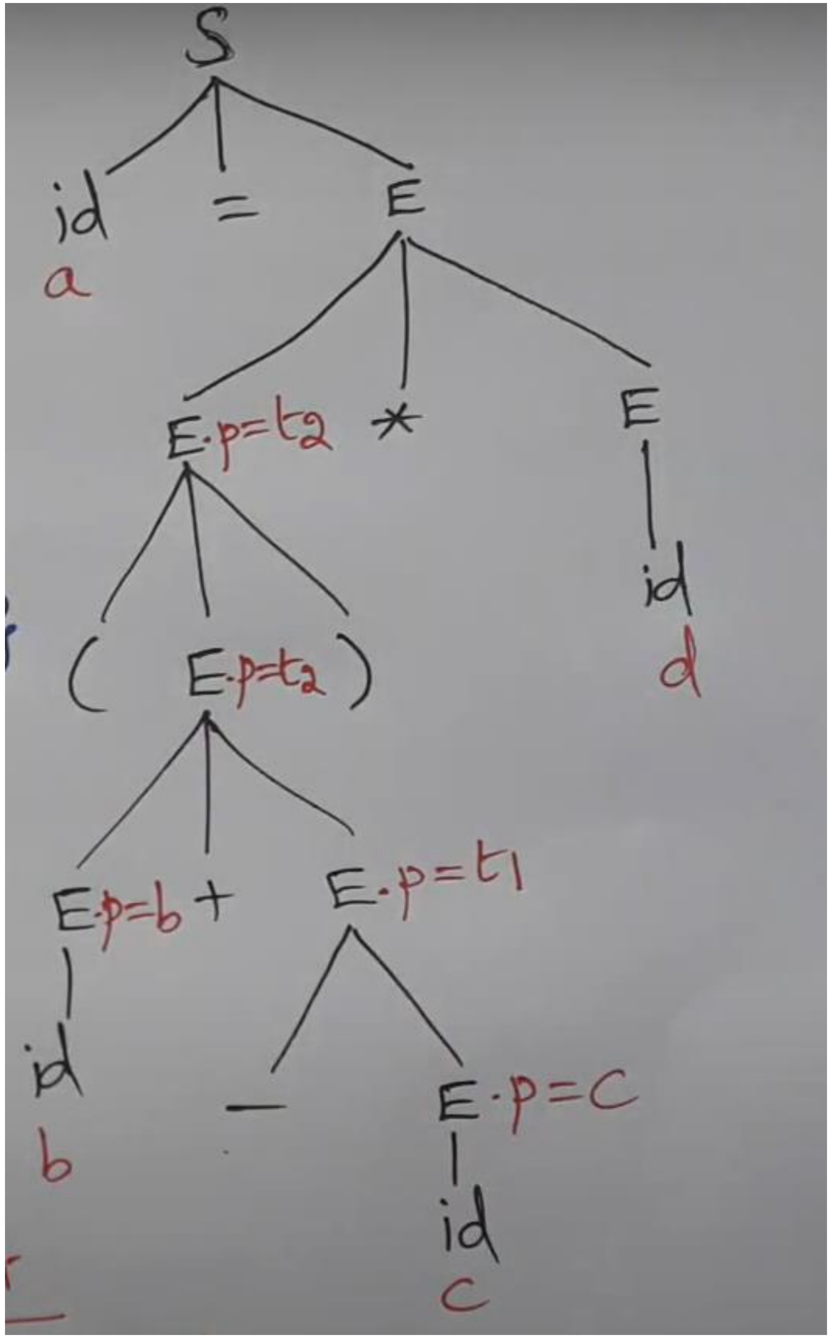
E → id	{p = look_up(id.name);
	If p ≠ nil then
	p = E.place
	Else
	Error;
	}

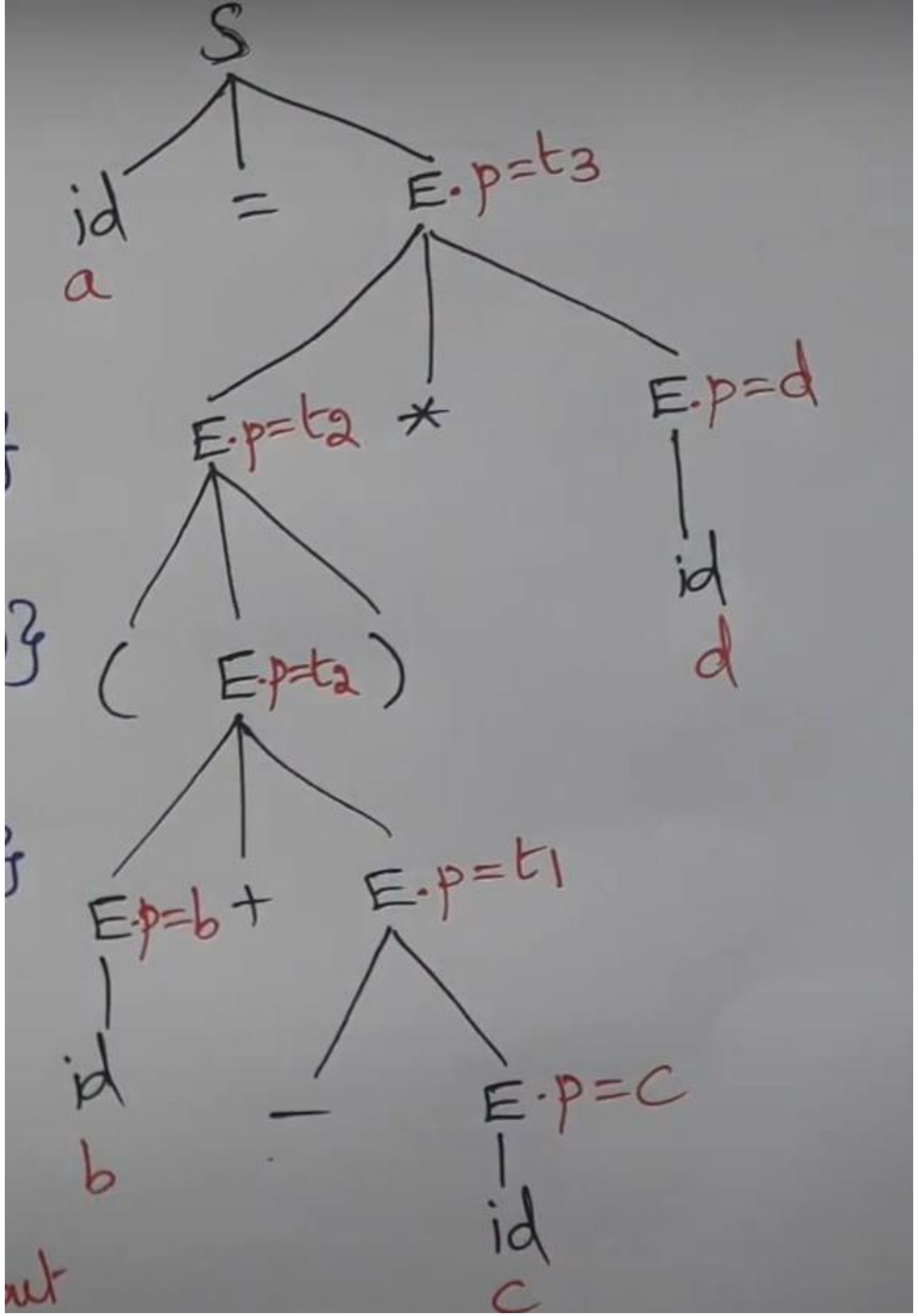
	The p returns the entry for id.name in the symbol table. The Emit function is used for appending the three address code to the output file. Otherwise it will report an error. The newtemp() is a function used to generate new temporary variables. E.place holds the value of E.

Ex: $a=(b+-c)*d$






S
id
=
E
a
E.p=t2
*
E
(
E.p=t2
)
id
d
E.p=b
+
E.p=t1
id
—
E.p=c
b
id
c


S
id
=
E.p=t3
a
E.p=t2
*
E.p=d
(
E.p=t2
)
id
d
E.p=b
+
E.p=t1
id
—
E.p=c
b
id
c

Output

$t_1 = \text{uminus } c$

$t_2 = b + t_1$

$t_3 = t_2 * d$

$a = t_3$

Flow of control statements

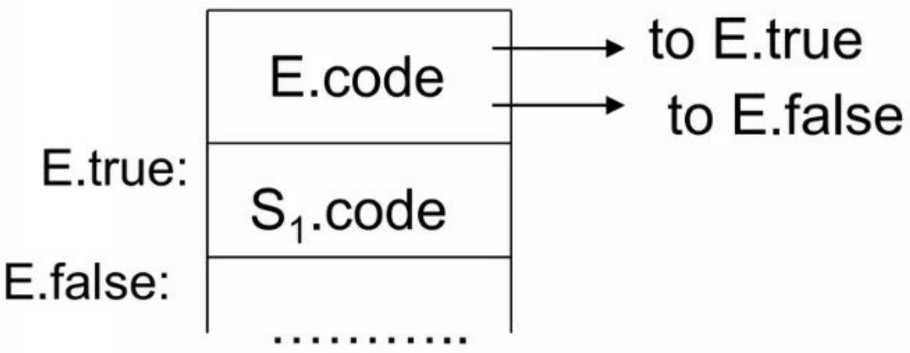
- **S $\rightarrow$ if E then S_1 |**

  if E then S_1 else S_2|while E do S_1

- Here E is the boolean expn. to be translated
- We assume that 3-address code can be labeled
- ***newlabel*** returns a symbolic label each time its called.
- E is associated with 2 labels
  1. **E.true – label which controls flow if E is true**
  2. **E.false – label which controls flow if E is false**
- **S.next** – is a label that is attached to the first 3 address instruction to be executed after the code for S

1. Code for if-then

$S \rightarrow$ **If E then** S_1



Semantic rules

E.true := newlabel;

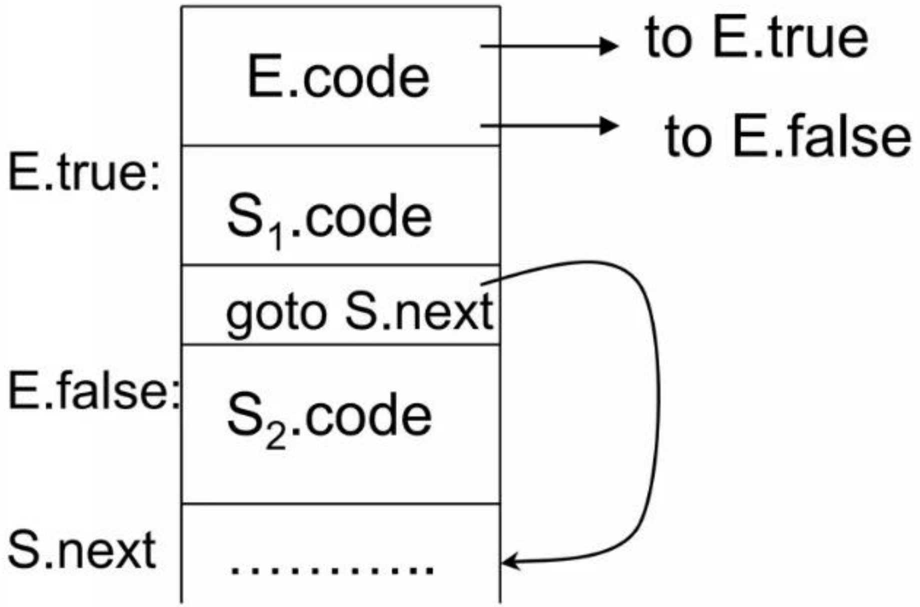
E.false := S.next;

S1.next := S.next;

S.code := E.code ||
gen(E.true ':') ||
S1.code

2. Code for if-then-else

$S \rightarrow$ if E then S_1 else S_2

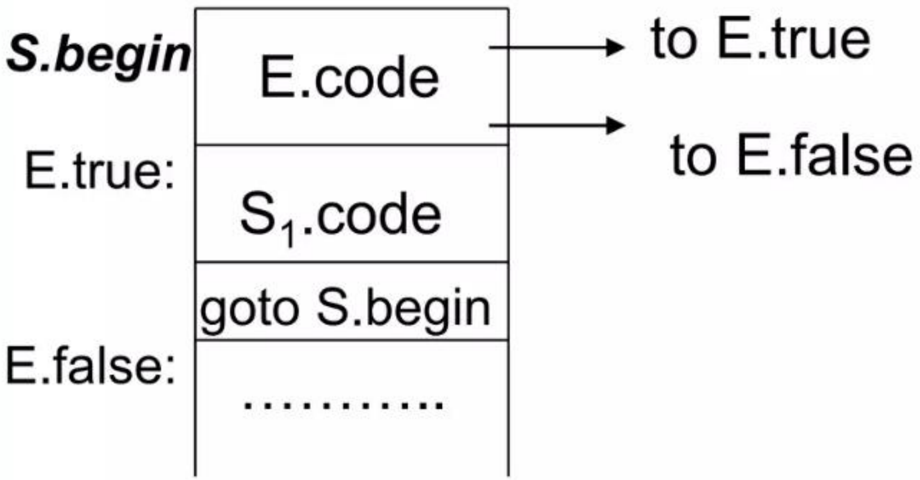


Semantic rules

E.true := newlabel;
E.false := newlabel;
S1.next := S.next;
S2.next := S.next;
S.code := E.code ||
gen(E.true ':') ||
S1.code ||
gen(' goto' S.next) ||
gen (E.false ':') ||
S2.code

3. Code for while-do

S → while E do S_1

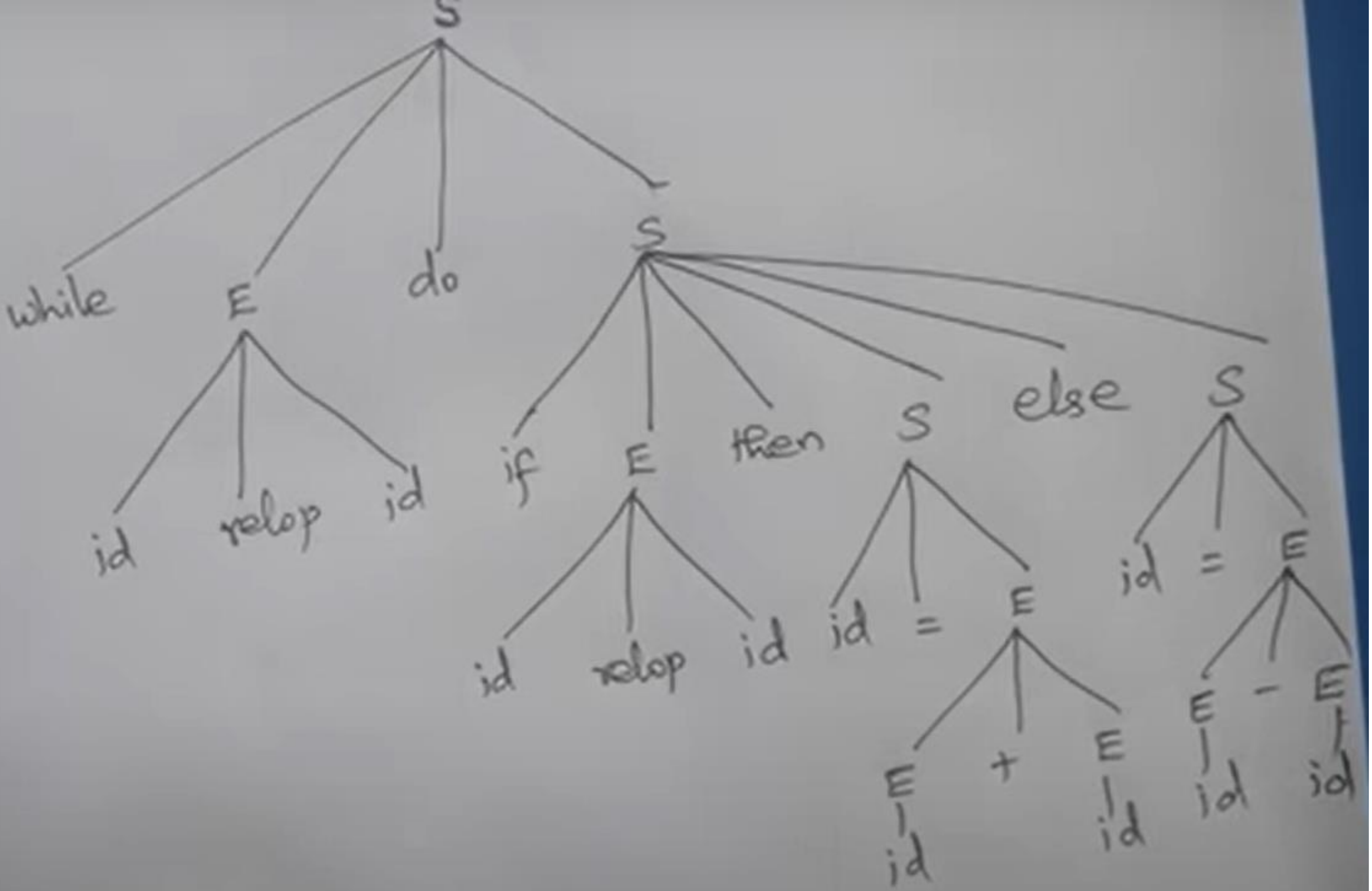


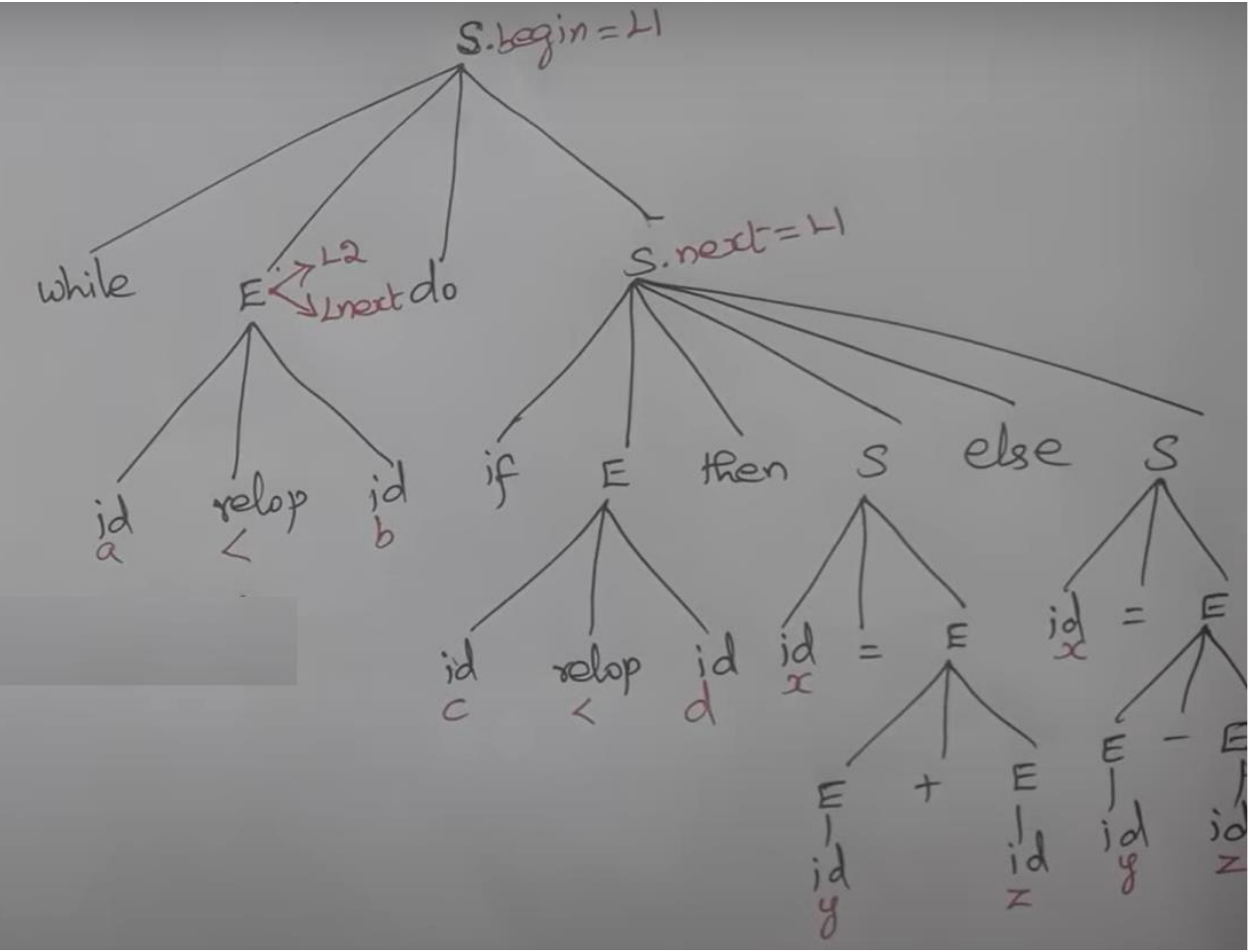
Semantic rules

S.begin := newlabel;
E.true := newlabel;
E.false := S.next;
S1.next := S.begin;
S.code := gen(S.begin ':') ||
E.code ||
gen(E.true ':') ||
S1.code ||
gen('goto' S.begin)

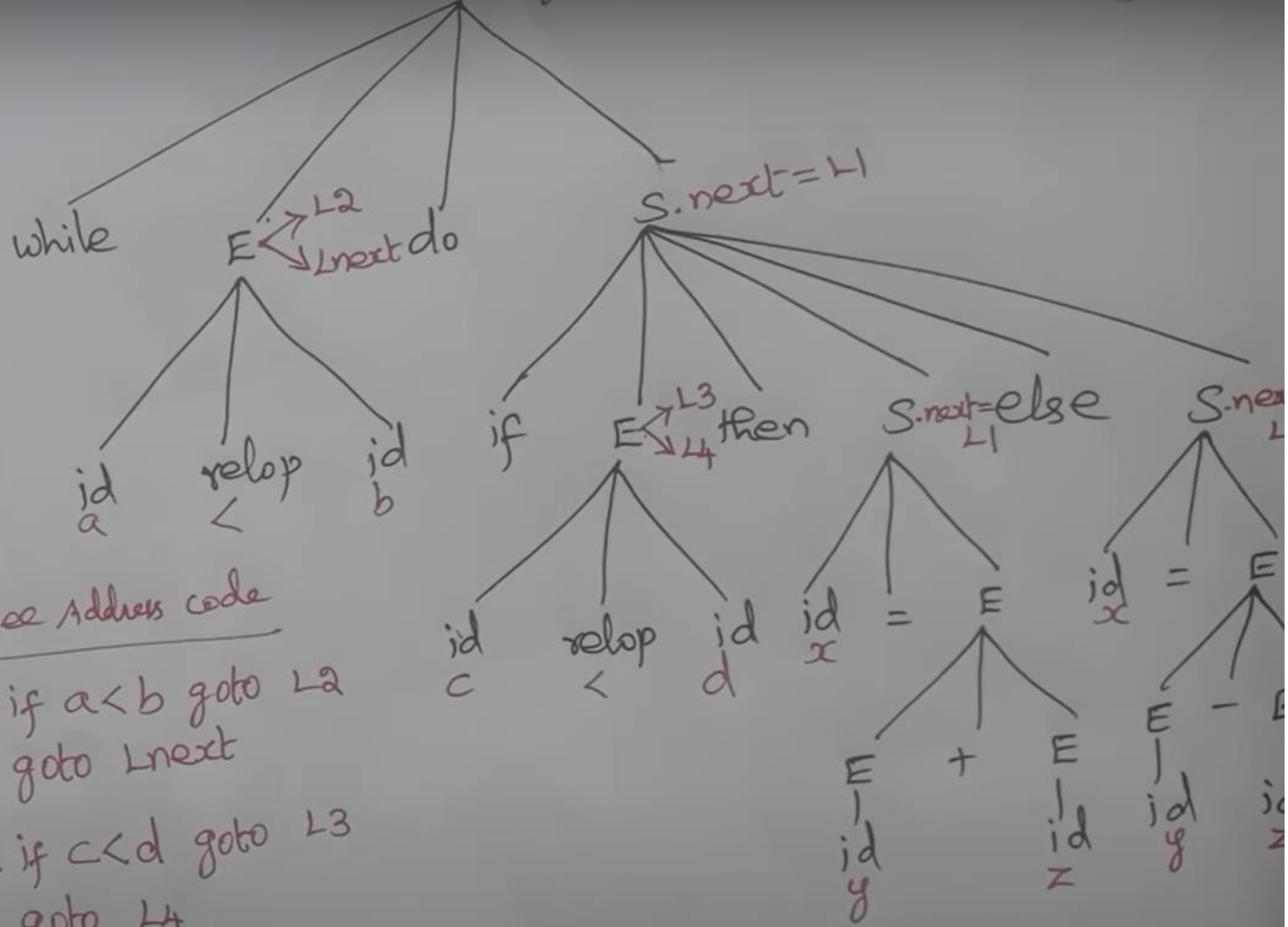
Example

```
While a<b do
    If c<d then
        X=y+z
    Else
        X=y-z
```


S
while
E
do
S
id
relop
id
if
E
then
S
else
S
id
relop
id
id
=
E
id
=
E
E
+
E
E
-
E
id
id
id
id



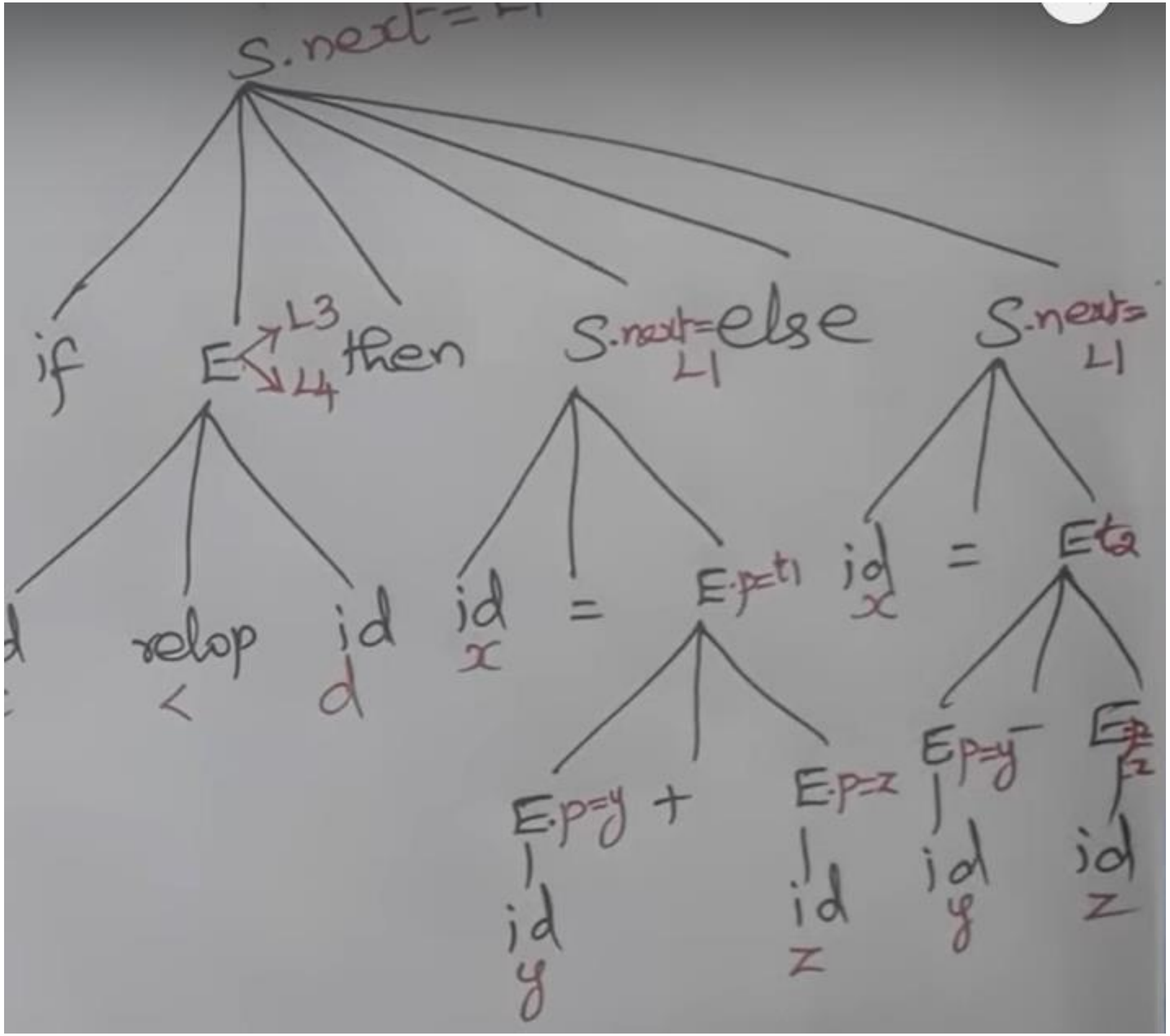
L1: if a<b goto L2
goto Lnext



Three Address code

L1: if a<b goto L2
goto Lnext

L2: if c<d goto L3
goto L4



S.next = L1
if
E
L3
L4
then
S.next= L1
else
S.next= L1
relop
<
id
d
id
x
=
E.p=t1
id
x
=
Et2
E.p=y
+
E.p=z
id
y
id
z
E.p=y
id
y
id
z

Three Address code

L1: if a<b goto L2
    goto Lnext

L2: if c<d goto L3
    goto L4

L3: $t_1 = y + z$
    $x = t_1$
    goto L1

L4: $t_2 = y - z$
    $x = t_2$
    goto L1

Lnext:

Case Statements

```
Switch  <expression>
  begin
    case value   :    statement
    case value   :    statement
              ........
    case value   : statement
    default      : statement
  end
```

Translation of a case statement

```
code to evaluate E into t
        goto test
  L1:   code for S1
        goto next
         ...
Ln-1:   code for Sn-1
        goto next
  Ln:   code for Sn
        goto next
```

```
test:  if t = V1 goto L1
         ...
       if t = Vn-1 goto Ln-1
       goto Ln
next:
```

Procedure calls

Function Calling: $P(X_1,X_2,\ldots,X_n)$

Three address code:

Param x_1
Param x_2
.
.
.
Param x_n
Call P, n

```
main()
{
.
.
add(x,y)
.
.
}
void add(int a, int b)
{
.
.
}
```

SDT for procedure calls

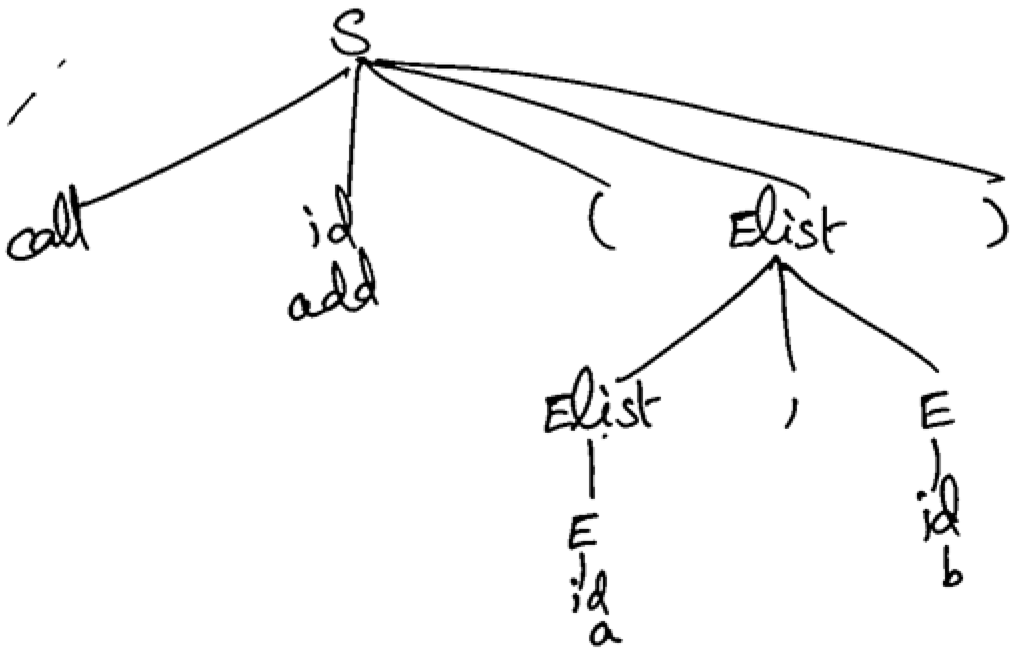
S → call id (Elist) {for each item p on queue do emit ('param' p);
emit('call' id.place)}

Elist → Elist, E {append E.place to the end of queue}

Elist → E {initialize queue to contain E.place}

Example

add (a, b)



QUEUE

a	b			
p	p			

Three address statement

param a

param b

call add

NOTE:
Elist.nd = for storing the number of actual parameters

Backpatching

- Problem generated by short-circuit code in generation of three address code
- Process of delaying the address generation
  - Leaves the label unspecified and fill it later
- Ex:

  a<b or c>d and e<f

100: if a<b goto 106
101: goto 102
102: if c>d goto 104
103: goto 107
104: if e<f goto 106
105: goto 107
106: (true)
107: (false)

Functions in backpatching

- Makelist(i) – creates a new list with index, i
- Merge(L1, L2) – concatenate List, L1 with list, L2
- Backpatch(L, label) – fills all 3ac in list, L with the target, Label
- Nextquad – gives index of next quadruple

SDT for Boolean expression using backpatching

A non-terminal marker M in the grammar generate a semantic action to pick up, at suitable times, the index of the next instruction to be created.

m.quad should be filled with all statements in E1.falselist

Production Rule		Semantic action
E	E_1 OR M E_2	{backpatch (E_1.flist, M.quad); E.Tlist:=merge(E_1.Tlist, E_2.Tlist) E.Flist:= E_2.Flist}
E	E_1 AND M E_2	{backpatch (E_1.Tlist, M.quad); E.Tlist:=E_2.Tlist; E.Flist:=merge(E_1.Flist,E_2.Flist);}
E	NOT E_1	{E.Tlist:=E_1.Flist; E.Flist:=E_1.Tlist;}
E	(E_1)	{E.Tlist:=E_1.Tlist; E.Flist:=E_1.Flist;}
E	id1 relop id2	{E.Tlist:=mklist(nextstate); E.Flist:=mklist(nextstate+); Append('if' id1.place relop.op id2.place 'goto_'); Append('goto_')}
E	true	{E.Tlist:=mklist(nextstate); Append('goto_');}
E	false	{E.Flist:=mklist(nextstate); Append('goto_');}
M	ε	{m.quad:=nextquad;}

SDT for Boolean expression using backpatching

M→ index of E2

M.quad should be filled with all statements in E1.falselist

$E \rightarrow E_1$ or $M E_2$	{backpatch(E_1.falselist, M.quad); E.truelist = merge(E_1.truelist, E_2.truelist); E.falselist = E_2.falselist }
$E \rightarrow E_1$ and $M E_2$	{backpatch(E_1.truelist, M.quad); E.truelist = E_2.truelist; E.falselist = merge(E_1.falselist, E_2.falselist)}
$E \rightarrow$ not E_1	{E.truelist = E_1.falselist; E.falselist = E_1.truelist }
$E \rightarrow (E_1)$	{ E.truelist = E_1.truelist; E.falselist = E_1.falselist }
$E \rightarrow id_1$ relop id_2	{E.truelist = makelist (nextquad); E.falselist = makelist(nextquad+1); gen('if' id_1 relop id_2 goto —') gen ('goto —') }
$M \rightarrow \varepsilon$	M.quad = nextquad

A non-terminal marker M in the grammar generate a semantic action to pick up, at suitable times, the index of the next instruction to be created.

a<b or c>d and e<f

100: if a<b goto --------------

101: goto --------------

102: if c>d goto --------------

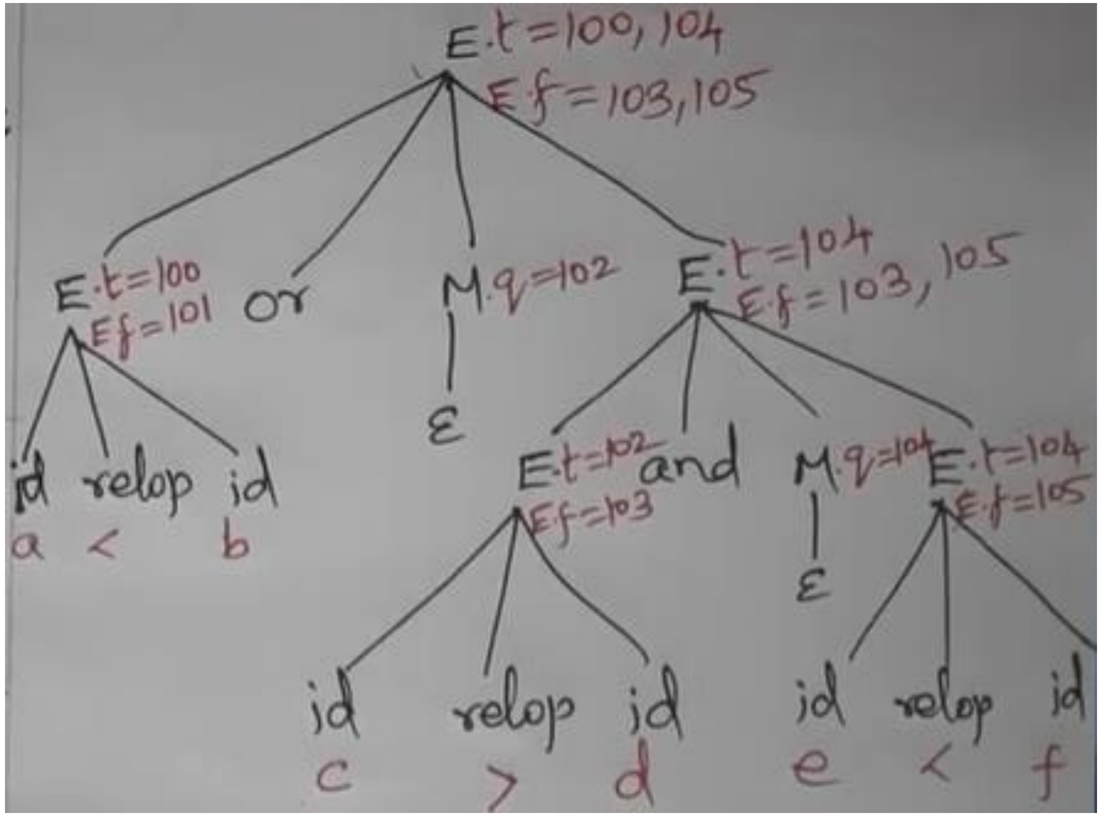
103: goto --------------

104: if e<f goto --------------

105: goto --------------

106: (true part - code)

107: (false part – code)



a<b or c>d and e<f

100: if a<b goto 106

101: goto 102

102: if c>d goto 104

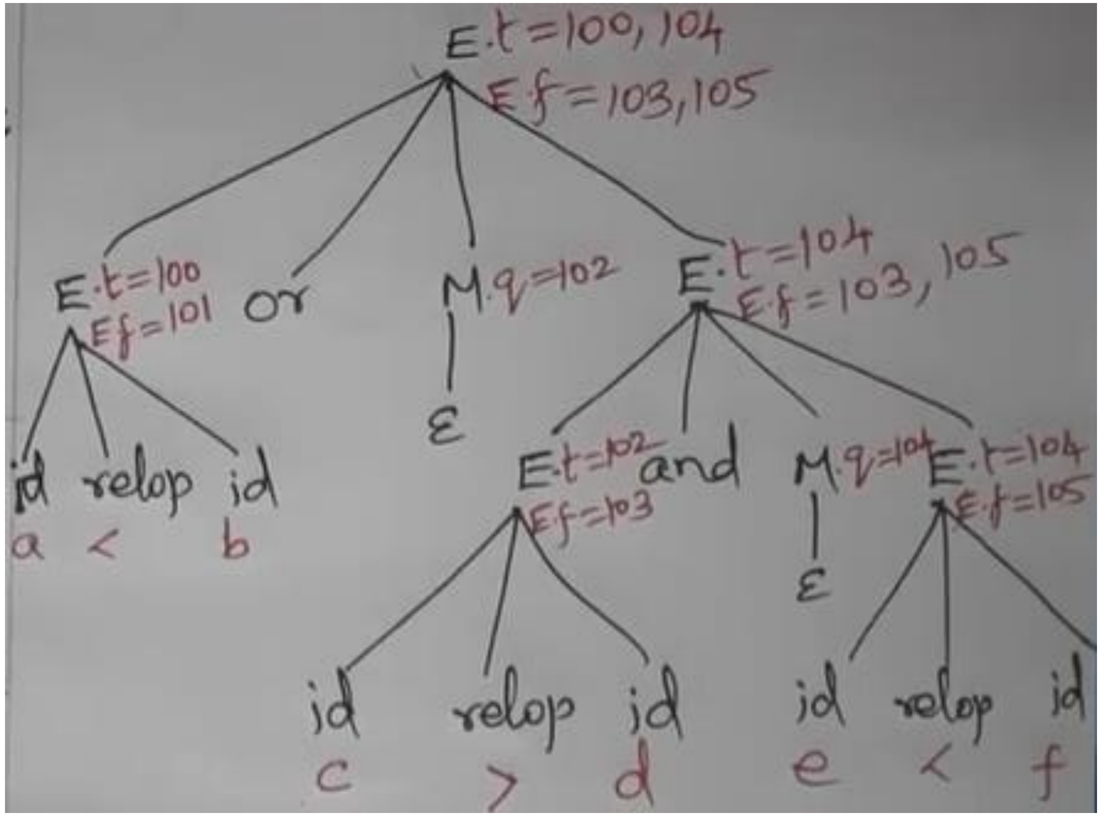
103: goto 107

104: if e<f goto 106

105: goto 107

106: (true part - code)

107: (false part – code)



Flow-of-Control Statements - Backpatching

$$S \rightarrow \textbf{if}\,(\,B\,)\,S \;\mid\; \textbf{if}\,(\,B\,)\,S\;\textbf{else}\;S \;\mid\; \textbf{while}\,(\,B\,)\,S$$

SDTs for Control Flow statements

1) $S \rightarrow \textbf{if}\,(\,B\,)\,M\,S_1$ { $backpatch(B.truelist,\ M.instr);$
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist);$ }

2) $S \rightarrow \textbf{if}\,(\,B\,)\,M_1\,S_1\,N\,\textbf{else}\,M_2\,S_2$
{ $backpatch(B.truelist,\ M_1.instr);$
$backpatch(B.falselist,\ M_2.instr);$
$temp\ =\ merge(S_1.nextlist,\ N.nextlist);$
$S.nextlist\ =\ merge(temp,\ S_2.nextlist);$ }

3) $S \rightarrow \textbf{while}\ M_1\,(\,B\,)\,M_2\,S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist\ =\ B.falselist;$
$gen($'goto' $M_1.instr);$ }

4) $S \rightarrow \{\,L\,\}$ { $S.nextlist\ =\ L.nextlist;$ }

5) $S \rightarrow A\ ;$ { $S.nextlist\ =\ \textbf{null};$ }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr;$ }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr);$
$gen($'goto _'$);$ }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr);$
$L.nextlist\ =\ S.nextlist;$ }

9) $L \rightarrow S$ { $L.nextlist\ =\ S.nextlist;$ }