Intermediate Code Generation

Page No.:
Date: / /

* What is an Intermediate Code?

- The task of compiler is to convert the source program into machine program. This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass. Then, typically compilers generate an easy to represent form of source language which is called intermediate language.
- The generation of an intermediate language leads to efficient code generation.

* Benefits of intermediate Code Generation -

There are certain benefits of generating machine -independent intermediate code.

1) A compiler for differnt machines can be created by attaching differnt backend to the existing front ends of each machine.
2) A compiler for differnt source languages (on the same machine) can be created by proving differnt front ends for corresponding source languages to existing back end.
3) A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

The role of intermediate code generator in compiler is depicted by

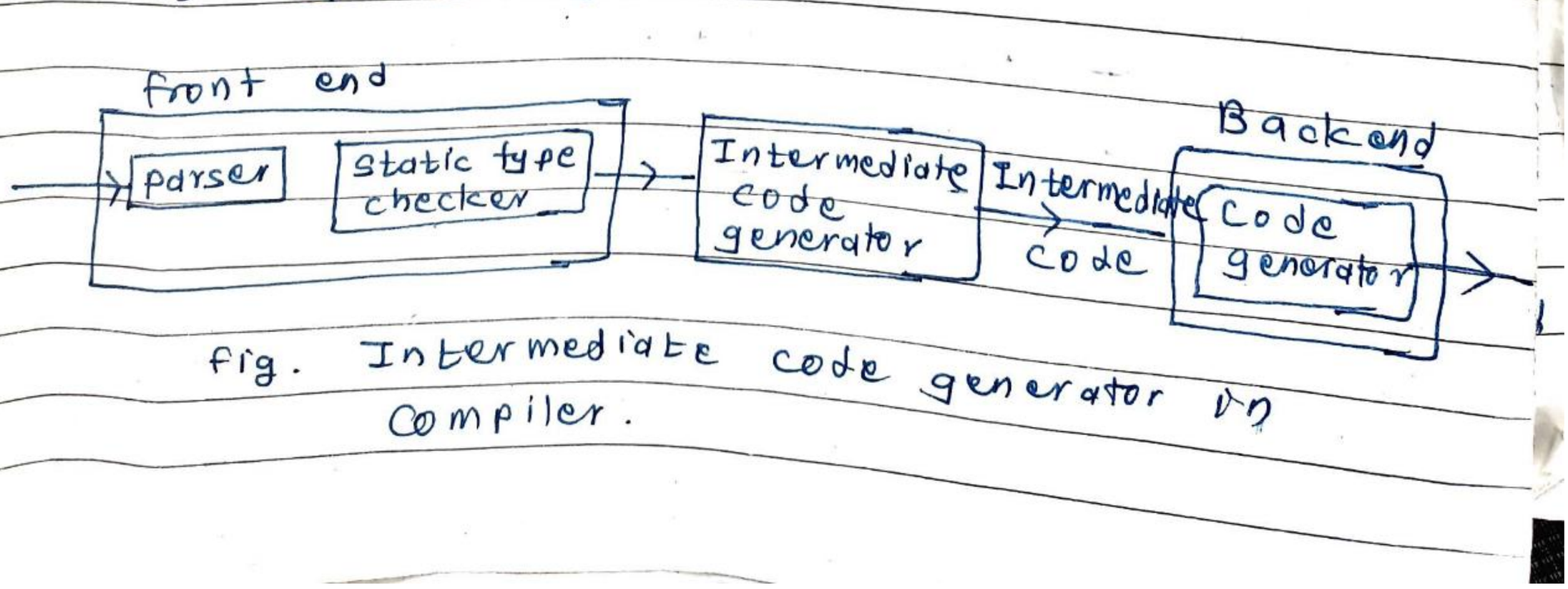


fig. Intermediate code generator in Compiler.

* Intermediate Languages -

- There are mainly three types of intermediate code representations. These three representations are used to represent the intermediate languages.
  i) Syntax tree
  ii) Posix
  iii) Three address code.

1) Syntax tree -

With the help of syntax tree & DAG the code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

ex. The Syntax tree & DAG for -a*b + -a*b postfix is a - b * a - b * +

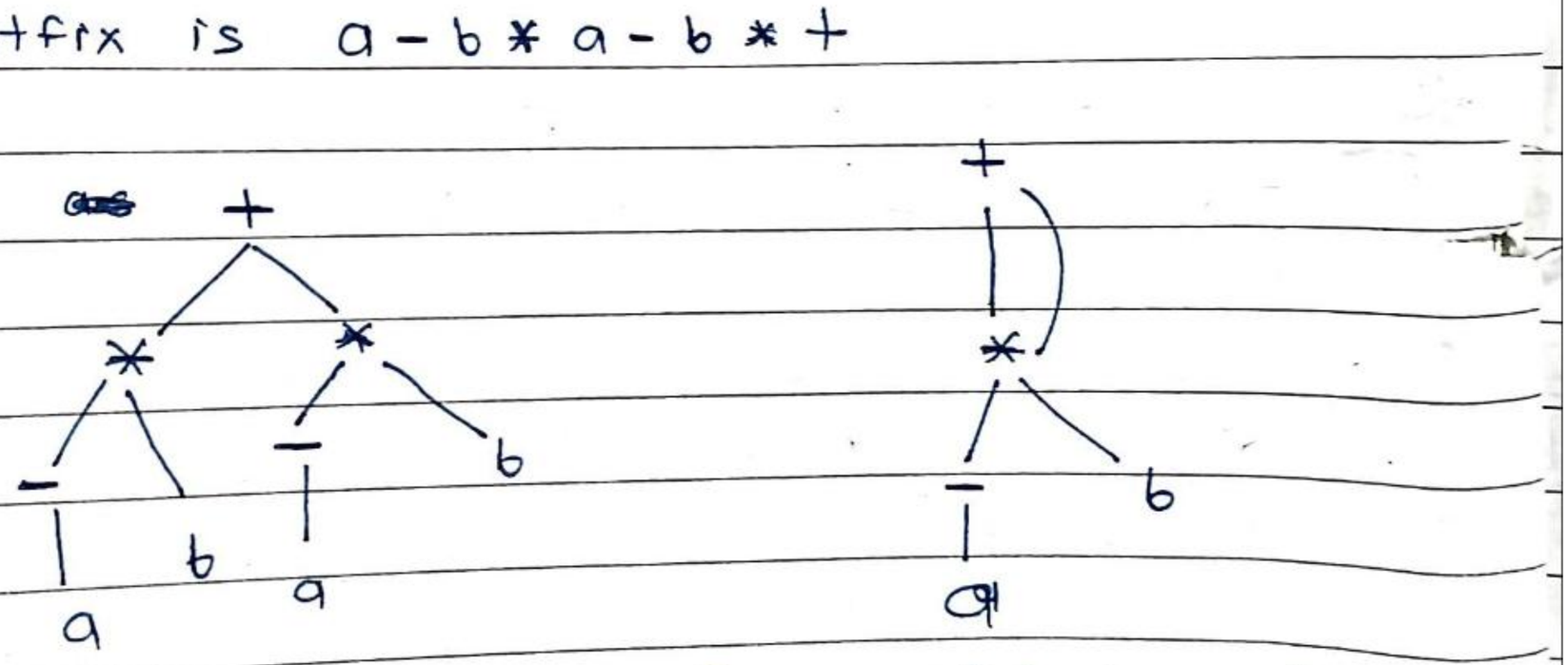


fig. Syntax tree & dag for - a * b + - a * b

2) Posix -

The postfix notation is used using postfix representation. Consider that the input expression X := a * b + - a * b then required posix form is

X := a - b * a - b * +

This is the most natural way of representation in expression evaluation.

SEMP
Page No. :
Date : / /

3) Three address code -

- In three address code form at the most three addresses are used to represent any statement.
- The general form of three address code representation is.

$a := b \ op \ c$

where a, b or c are operands that can be names, constants, compiler generated temporaries & op represents the operator. The operators can be fixed or floating point arithmatic operator or logical operators on boolean valued data. only single operation at right side of the expression is allowed at a time.

ex. the expression like $a = b+c+d$ the three address code will be.

$t1 := b + c$

$t2 := t1 + d$

$a := t2$

here, t1 & t2 are temporary names generated by the compiler. There are at the most three addresses are allowed, hence the name of this representation is three address code.

* Types of three address statements

The form of three address code is very much similar to assembly language. Commonly used three address code for typical language constructs.

Language Construct	Intermediate Code	Meaning
Assignment Statement	$x := y \ op \ z$	here binary operation is performed using operator op
Assignment statement	$x := op \ y$	here unary operation is performed. The operator 'op' is an unary operator

Language construct	Intermediate code	meaning
Copy statement	x := y	here value of y is ass to x.
unconditional jump	goto L	the control flow goe the statement labelled
conditional jump	if x relop y goto L	the relop indicates the relational operators su as <, =, >, ≠
procedure calls	param x_1, param x_2 ---- param x_n call p, n	here the parameters $x_1, x_2, \ldots x_n$ are used parameters to the proce p.
	return y	The return statement indicate the return valu
Array statements	x := y[i]	The value at ith index array y is assigned to
Address & pointer assignments	x := &y	The value of x will be or location of y
	x := *y	They is pointer value
	*x := y	The r-value of object pointed by x is set t the l-value of y.

* Implementation of three address code

- The three address code is an abstract form of intermediate code that can be implemented as a record with address fields.
- There are three representations used for three address code
  a) quadruple
  b) triple
  c) indirect triples.

a) Quadruple representation -

- This is the structure with at most four fields. such as op, arg 1, arg 2, result.
- The op field is used to represent the internal code for operator, the arg 1, & arg 2 representations the two operands used & result field is used to store the result of an expression.
- In quadruple representation using temporary names the enteries in the symbol table against those temporaries can be obtained.
- With quadruple representation is that one can quickly access the value of temporary variables using symbol table.
- Use of temporaries introduces level of indirection for the use of symbol table in quadruple representation.
- This is beneficial for code optimization.

ex. Consider the input statement x := -a*b + -a*b

The three address code is

t1 := unminus a
t2 := t1 * b
t3 := unminus a
t4 := t3 * b
t5 := t2 + t4
x := t5

Quadruple

	Op	Arg 1	Arg 2	Result
(0)	unminus	a		t1
(1)	*	t1	b	t2
(2)	unminus	a		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	:=	t5		x

b) Triples -

- In the triple representation the use of tempora variables is avoided by referring the pointers in the symbol table.
- In Triple representation the pointers are used.
- By using pointers one can access directly the symbol table entry.

ex. X := -a * b + -a * b the triple representat is given below.

Number	OP	Arg1	Arg 2
(0)	unminus	a	
(1)	*	(0)	b
(2)	unminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	4	

c) Indirect triples —

In the indirect triples representation the listing of triples is been done. And listing pointers are used instead of using statements.

- In indirect triple list of all references to computa is made separately & stored.
- Indirect triple saves some amount of space compared with quadruple representation.

Number	Op	Arg1	Arg 2
(0)	unminus	a	
(1)	*	(11)	b
(2)	unminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	25	x

	statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

SEMP
Page No. :
Date : / /

1) List the commonly used ~~rep~~ intermediate code representations. Write the following expression in all types of intermediate representations you know.
(a - b) * (c+d) - (a+b)

→ The commonly used intermediate representations are
a) Syntax tree or dag (Graphical reprensentation)
b) postfix notation.
c) three address code.

a) ~~Graphical representation~~ - Postfix notation
First we convert this expression into postfix.
(a - b) * (c+d) - (a+b)
⇒ a b - * c d + - a b +
⇒ a b - c d + * - a b +
⇒ a b - c d + * a b + -

b) Graphical representation

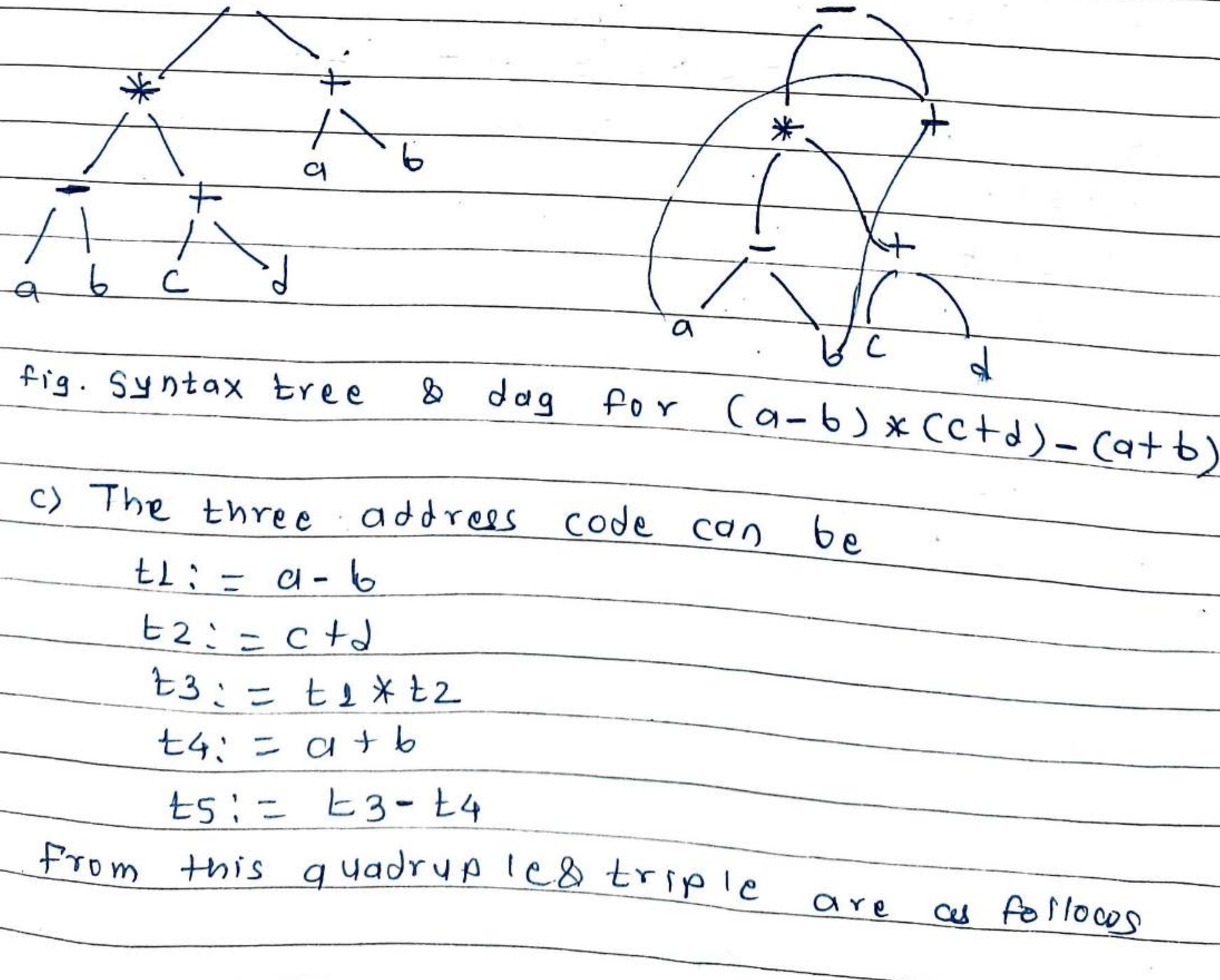


fig. Syntax tree & dag for (a-b) * (c+d) - (a+b)

c) The three address code can be
t1 := a - b
t2 := c + d
t3 := t1 * t2
t4 := a + b
t5 := t3 - t4
From this quadruple & triple are as follows

	op	arg 1	arg 2	result
(0)	-	a	b	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	+	a	b	t4
(4)	-	t3	t4	t5

fig. quadruple

	op	arg 1	arg 2
(0)	-	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	+	a	b
(4)	-	2	3

fig. triple

2) Write Quadruple & Triple representation for follo expression

A = B + C * D / E + - F * - G

→ The three address code.

t1 = B + C

t2 = D / E

t3 = t1 * t2

t4 = - F

t5 = t3 + t4

t6 = - G

t7 = t5 * t6

* Generation of three address Code -

- The translation scheme for various language constr can be generated by using the appropriate semantic actions.
- In this section we will discuss the translatio schemes for declaration statement, array references, boolean expression & so on.

1) Declarations -

In the declarative statements the data items along with their data types are declared.

ex.

S → D	{offset := 0}
D → id : T	{enter_tab(id.name, T.type, off offset := offset + T.width}
T → integer	{T.type := integer ; T.width := 4 }
T → real	{T.type := real ; T.width := 8 }
T → array [num] of T_1	{T.type := array(num.val, T_1.ty T.width := num.val X T_1.widt
T → *T_1	{T.type := pointer(T.type T.width := 4 }

2) Assignment statements -

- The assignment statement mainly deal with the expressions. The expressions can be of type intege real, array & record.
- In this section we will see how to write th syntax directed translation scheme for genera of three address code for assignment state containing arithmetic expressions.

1) Obtain the translation scheme for obtaining the three address code for the grammar.

$S \rightarrow id := E$

$E \rightarrow E1 + E2$

$E \rightarrow E1 * E2$

$E \rightarrow -E1$

$E \rightarrow (E1)$

$E \rightarrow id$

→ Translation scheme for above grammar which in turn will generate the three address code.

Production rule	Semantic actions
$S \rightarrow id := E$	{ id_entry := look_up(id.name); if id.entry ≠ nil then append(id_entry := E.place) else error; /* id not declared */ }
$E \rightarrow E_1 + E_2$	{ E.place := newtemp(); append(E.place := E1.place + E2.place) }
$E \rightarrow E_1 * E_2$	{ E.place := newtemp(); append(E.place := E1.place * E2.place) }
$E \rightarrow -E1$	{ E.place := newtemp(); append(E.place := uminus E1.place) }
$E \rightarrow (E1)$	{ E.place := E1.place }
$E \rightarrow id$	{ id_entry := look_up(id.name); if id_entry ≠ nil then append(id_entry := E.place) else error; /* id not declared */ }

- The look-up returns the entry for id.name in the symbol table if it exists there.
- The function append is for appending the three address code to the output file. Otherwise an error will be reported.
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E.

ex. Consider the assignment statement X: (a+b)*(c+d). We will assume all these identifiers are of the same type. Let us the bottom up parsing method.

Production rule	Semantic action for attribute evaluation	output
E → id	E.place := a	
E → id	E.place := b	
E → E1 + E2	E.place := t1	t1 := a+b
E → id	E.place := c	
E → id	E.place := d	
E → E1 + E2	E.place := t2	t2 := c+d
E → E1 * E2	E.place := t3	t3 := (a+b)*(c
S → id := E		x := t3

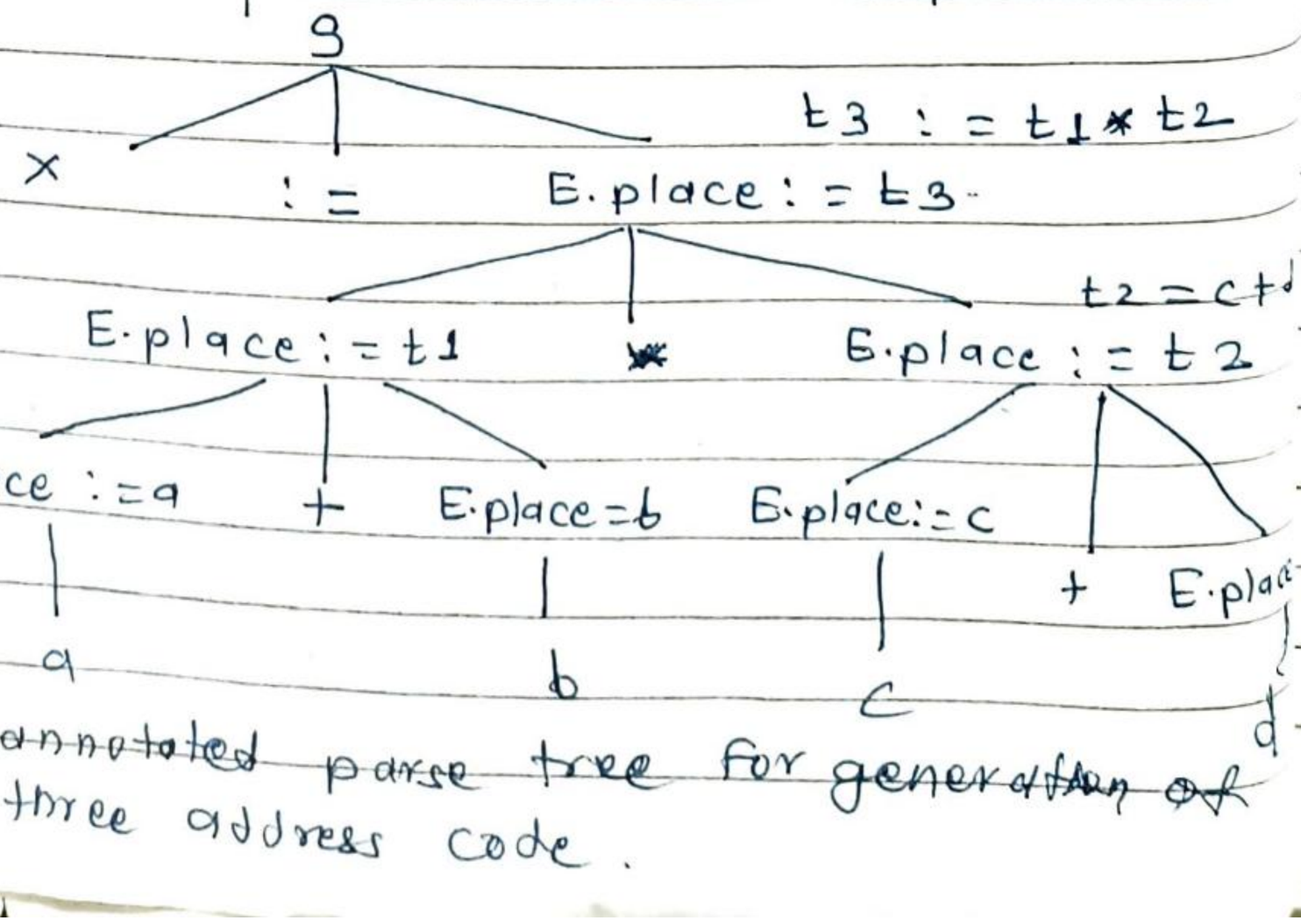


fig. annotated parse tree for generation of three address code.

SEMP
Page No. :
Date : / /

Type conversion

While generating the three address code for mixed mode expressions it becomes necessary to achieve the type compatibility betn the sub-expressions. Consider the grammar $E \rightarrow E+E$ & $E \rightarrow E*E$

Production rule	Semantic actions

$E \rightarrow E_1 + E_2$	E.place := newtemp()
	if E_1.type = integer & E_2.type = integer then
	{
	append (E.place := E_1.place 'int +' E_2.place)
	E.type := integer
	}
	else if E_1.type = real & E_2.type = real then
	{
	append(E.place := E_1.place 'real +' E_2.place)
	E.type : real
	}
	else if E_1.type = integer & E_2.type = real then
	{
	t := newtemp();
	append (t := 'inttoreal' E_1.place);
	append(E.place := t 'real +' E_2.place);
	E.type := real
	}
	else if E_1.type = real & E_2.type = integer then
	{
	t := newtemp();
	append(t := 'inttoreal' E_2.place);
	append(E.place := E_1.place 'real +' t);
	E.type := real
	}
	else
	E.type := type_error();
	}

(15)

$E \rightarrow E_1 * E_2$

E.place := newtemp()

if E1.type = integer & E2.type = integer

{

append(E.place := E1.place 'int *' E2.place)

E.type := integer

}

else if E1.type = real & E2.type = real

{

append(E.place := E1.place 'real *' E2.place)

E.type := real

}

else if E1.type = integer & E2.type = real

{

t := newtemp();

append(t := 'inttoreal' E1.place);

append(E.place := t real * E2.place);

E.type := real

}

else if E1.type = real & E2.type = integer

{

t := newtemp();

append(t := 'inttoreal' E2.place);

append(E.place := E1.place 'real *' t);

E.type := real

}

else

E.type := type_error();

fig. translation scheme for generating 3-address co
for expressions wing type expression.

SEMP
Page No :
Date : / /

ex. 1) Generate the three address code for an expression
x := a+b*c+d ; Assuming x & a as real & b,c & d are integer.

→ t1 := b int * c
t2 := t1 int + d
t3 := inttoreal t2
t4 := a real + t3
x := t4

* Arrays
- Array is a collection of contingeous storage of elements, for accessing elements of an array what we need is its address.
- For statically declared arrays it is possible to compute the relative address of each element.
- Typically there are two representations of arrays.
1) Row major representation
2) Column major representation.

The presentations are as follows.

Row major representation

A[1,1]	A[1,2]	A[1,3]	A[2,1]	A[2,2]	A[2,3]

← row 1 → ← row 2 →

column major representation

A[1,1]	A[2,1]	A[1,2]	A[2,2]	A[1,3]	A[2,3]

← col 1 → ← col 2 → ← col 3 →

To compute the address of any element;
The base is the address at a[] & w is the width of the element then to compute ith address of a[]
base + (i - low) x w
where low is lower bound on subscript
here a[low] = base.

base + (i - low) w = base + i x w - low x w
= i x w + (base - low x w)

Let c = base - low x w is computed at compile time. Then the relative address can be computed as

c + (i x w)

Similarly, for calculation of relative address of two dimensional array we need to consider i & j subscripts. Considering row major representation we will compute relative address for a[i,j] using following formula.

$$a[i,j] = base + ((i - low1) \times n2 + (j - low2)) \times w$$

where low1 & low2 are the two lower bounds on values of i & j & n2 is number of values that j can take. In other words.

$$n_2 = high2 - low2 + 1$$

where, high2 is the upper bound on j.
Assuming that i & j are not known at compile time we can rewrite the formula as

$$a[i,j] = ((i \times n_2) + j) \times w + (base - ((low1 \times n_2) + low2) \times w)$$

The term (base - ((low1 x n2) + low2) x w) can be computed at compile time.

ex. Consider an array A of size 10 x 20 assuming low1 = 1 & low2 = 1. The computation of A[i,j] is possible by assuming that w = 4 as

$$A[i,j] = ((i \times n_2) + j) \times w + (base - ((low1 \times n_2) + low_2) \times$$

Given $n_2 = 20$

$w = 4$

$low_1 = 1$

$low_2 = 1$

$A[i,j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4$

$A[i,j] = 4 \times (20i + j] + (base - 84)$

The value base - 84 can be computed at compile time.

* Generate the three address code for the expression $x := A[i,j]$ for an array 10×20. Assume $low_1 = 1$ & $low_2 = 1$.

→ Given that, $low_1 = 1$ & $low_2 = 1$, $n_1 = 10$, $n_2 = 20$

$A[i,j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$

$= ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4)$

$= 4 \times (20i + j) + \underbrace{(base - 84)}_{c}$

The three address code for this expression can be

```
t1 := i * 20
t1 := t1 + j
t2 := c          /* computation of c = base-84 */
t3 := 4 * t1
t4 := t2 [t3]
x := t4
```

The annotated parse tree can be drawn as follows

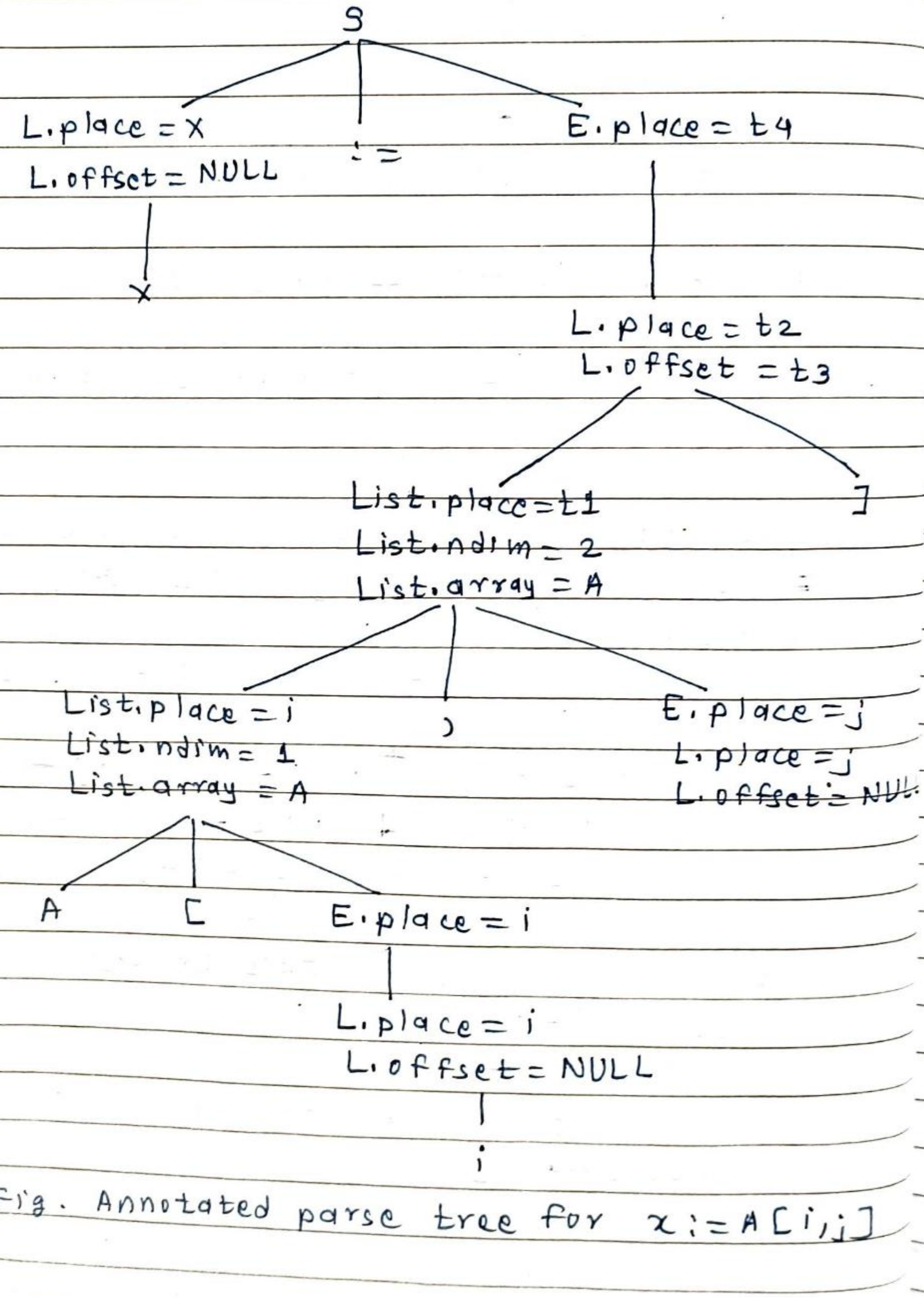


Fig. Annotated parse tree for x := A[i,j]

SEMP
Page No. :
Date : / /

* Boolean Expressions —

Normally there are two types of boolean expressions used.

1) for computing the logical values.
2) In conditional expressions using if-then-else or while-do

Consider, the boolean expression generated by following grammar.

E → E OR E
E → E AND E
E → NOT E
E → (E)
E → id relop id
E → TRUE
E → FALSE

here, the relop is denoted by ≤, ≥, ≠, <, >. The OR & AND are left associate. The highest precedence is to NOT then AND & lastly OR.

The translation scheme for Boolean expressions having numerical representation given below.

Production Rule	Semantic Rule.
E → E1 OR E2	{ E.place := newtemp() append(E.place := E1.place 'OR' E2.place) }
E → E1 AND E2	{ E.place := newtemp() append(E.place := E1.place 'AND' E2.place) }
E → NOT E1	{ E.place := newtemp() append(E.place := 'NOT' E1.place) }

Production rule	Semantic rule
$E \rightarrow (E1)$	{ E.place := E1.place }
$E \rightarrow id1\ relop\ id2$	{ E.place := newtemp() append('if' id1.place relop op id2.plac 'goto' next_state+3); append(E.place := '0'); append('goto' next_state +2); append(E.place := '1') }
$E \rightarrow TRUE$	{ E.place := newtemp(); append(E.place := 1) }
$E \rightarrow FALSE$	{ E.place := new temp() append (E.place := '0') }

fig. translation scheme for generation of 3 address code for

The function append generates the three address cod & newtemp() is for generation of temporary variab For the semantic action for the rule $E \rightarrow id1\ relop\ id$ contains next_state which gives the index of ne three address statement in the output seqn. Let us take an example the three address code using above translation scheme.

SEMP
Page No. :
Date : / /

P > q AND r < s OR u > v

100 : if P > q goto 103
101 : t1 := 0
102 : goto 104
103 : t1 := 1
104 : if r < s goto 107
105 : t2 := 0
106 : goto 108
107 : t2 := 1
108 : if u > v goto 111
109 : t3 := 0
110 : goto 112
111 : t3 := 1
112 : t4 := t1 AND t2
113 : t5 := t4 OR t3

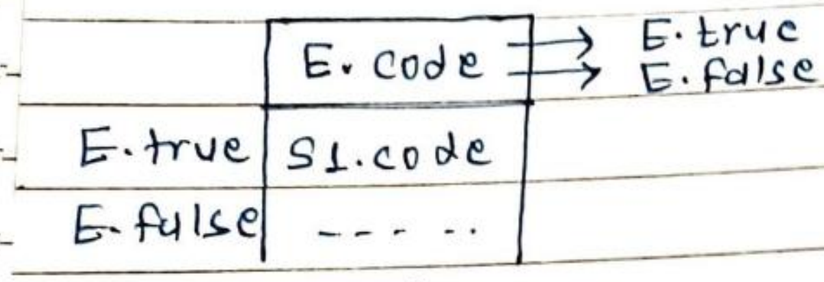
The AND has higher precedence over OR hence at location 112 the ANDing is performed & then in the intermediate next statement ORing is performed.

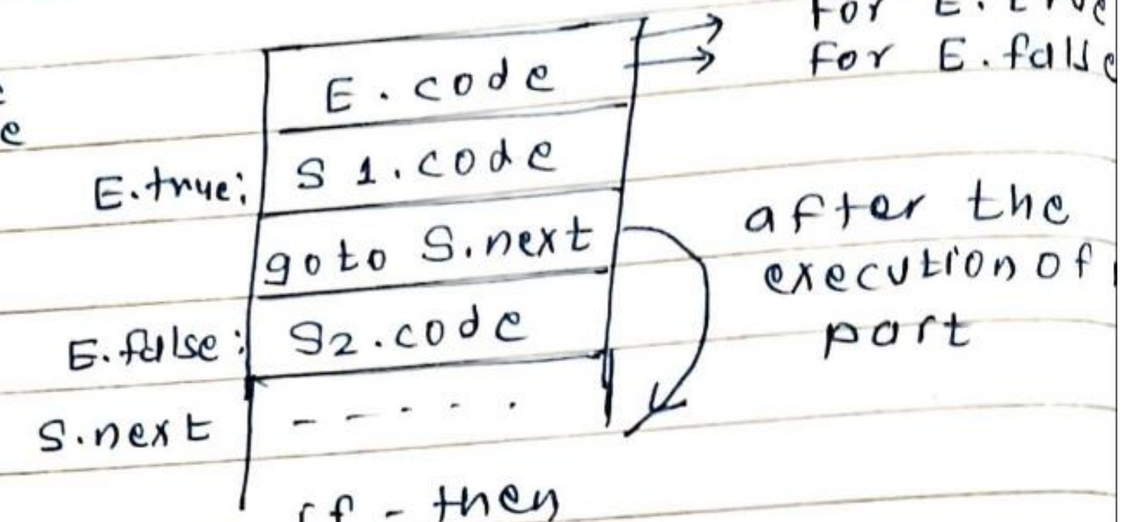
* Flow of control statements -

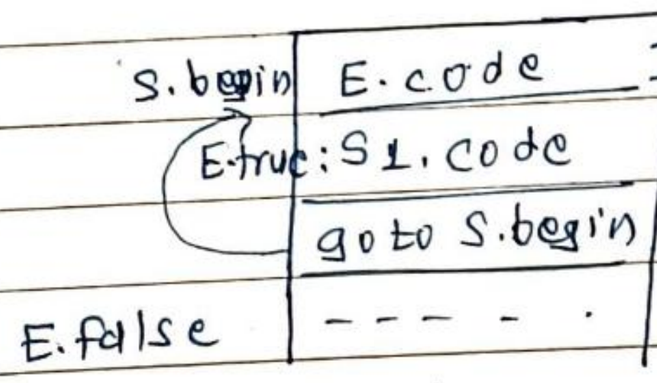
The control statements are if - then - else & while - do. The grammar for such statement is as shown below.

S -> if E then S1
| if E then S1 else S2
| while E do S1

while generating three address code.
- To generate new symbolic label the function new_label() is used
- With the expression E.true & E.false are the labels associated
- S.code & E.code is for generating three address code







S → if E then S1

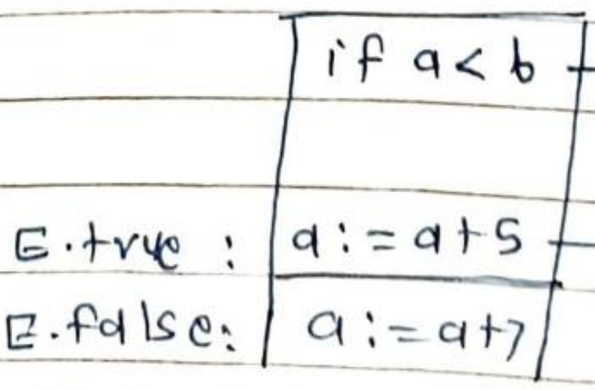
E.true := new_label ()

E.false := S.next

S1.next := S.next

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

In the above translation scheme || is used to concat the strings. The function gen_code is used to eval the non quoted arguments passed to it & to concate complete string. The S.code is the important rule which ultimately generates the three address code



Consider, the statement if a<b then a=a+5 else a=

The three address code for if-then is

100 : if a<b goto L1

101 : goto 103

102 : L1 : a = a+5

103 : a := a+7

here E.code is "if 'a < b" L1 denotes E.true & E.false is shown by jumping to line 103 (i.e. S.next)

similarly, S → if E then S1 else S2

E.true := new_label()
E.false := new_label()
S1.next := S.next
S2.next := S.next
S.code := E.code || gen_code(E.true)
S1.code || gen_code('goto', S.next)
|| gen_code(E.false ':') || S2.code

S → while E do S1

S.begin := new_label()
E.true := new_label()
E.false := S.next
S1.next := S.begin
S.code := gen_code(S.begin ':') || E.code
|| gen_code(E.true ':') || S1.code || gen_code('goto' S.begin)

ex. Translate executable sentences of the following C program

```
main()
{
  int i=1;
  int a[10];
  while(i<=10)
  {
    a[i] = 0;
    i = i+1;
  }
}
```

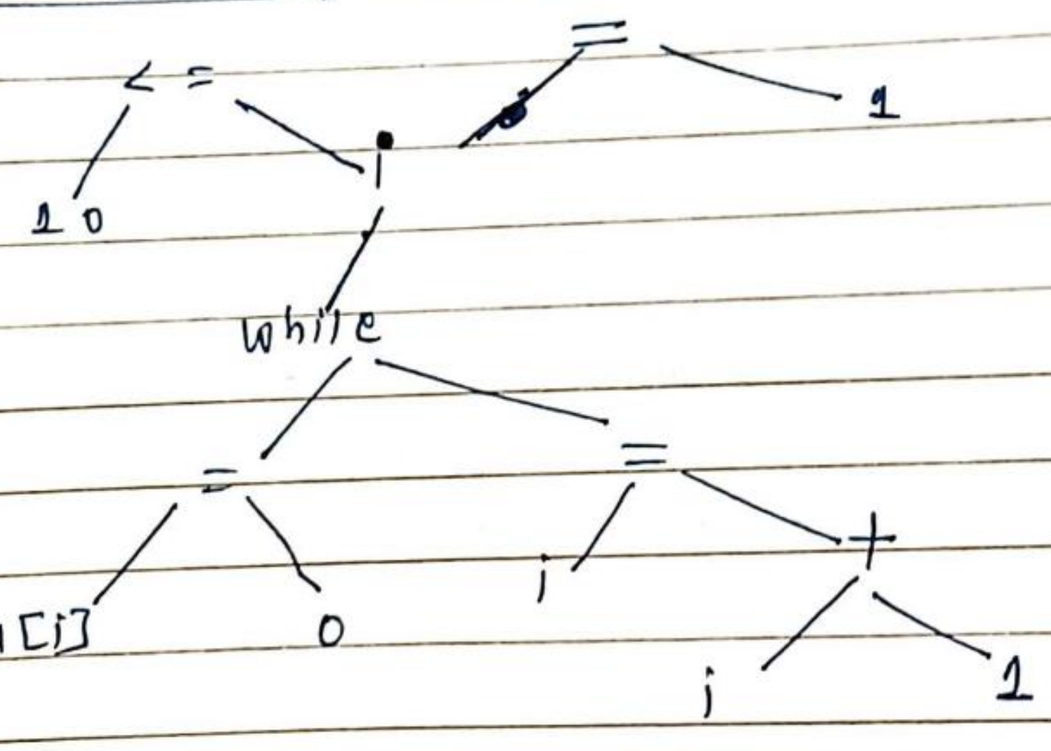
into

a) syntax tree
b) postfix notation
c) three-address code.

→ Postfix notation

```
i 1 =
while i 10 < =
a [i] 0 =
i i 1 + =
```

Syntax tree



Three address code -

```
i := 1
t1 := i
L1 : if  t1 < = 10 goto .
     goto L2
     if i * 4 ← width of variables
     t1 := i
     a [t1] := 0
     t1 := t1 + 1
     goto L1.
L2 : stop
```

SEMP
Page No. :
Date : / /

* Case statements -

The syntax for switch case statement can be as shown below

```
switch expression
{
  case value : statement
  - - - - - -
  case value : statement
  default : statement
}
```

Let us, write the translation scheme for case statement

Production rule	Semantic action.

switch E
{
case Y_1 : S_1
- - - - - -
case Y_{n-1} : S_{n-1}
default : S_n
}
	Evaluate E into t such that t = E
	goto check.
	L_1 : code for S_1 goto last
	- - - - - - - - - - - -
	L_n : code for S_n goto last
	last :

1) Generate the three address code for

```
switch (ch)
{
case 1 : c = a+b;
         break;
case 2 : c = a-b;
         break;
}
```

The three address code can be

100 : if ch = 1 goto L_1
101 : if ch = 2 goto L_2
102 : L_1 : t1 := a+b
103 : c := t1
104 : goto Last
105 : L_2 : t1 := a-b
106 : c := t1
107 : goto Last
108 : Last :

(28)

* Backpatching -

- Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.
- To generate code using backpatching in the semantic actions following functions are used.

a) mklist(i) - ~~create~~
create the new list. The index i is passed as argument to this function where I is an index the array of quadruple.

b) merge_list(p1, p2)
function concatenates two lists pointed by p1 p2. It returns the pointer to the concatenated

c) backpatch(p, i)
Inserts i as target label for the statement pointed by pointer p.

SEMP
Page No. :
Date : / /

* Backpatching using boolean expressions

Consider the grammar for boolean expressions.

$E \rightarrow E_1$ OR M E_2

$E \rightarrow E_1$ AND M E_2

$E \rightarrow$ NOT E_1

$E \rightarrow (E_1)$

$E \rightarrow id_1$ relop id_2

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

$M \rightarrow \epsilon$

→ here, a new nonterminal M is inserted as a marker nonterminal. The purpose of M is to mark the exact point where the semantic action is picked up.

- The synthesized attributes Tlist & Flist are used to generate the jumping code for Boolean expressions. for the true statement E.Tlist will be generated & for the false statement E.Flist is generated.

- The attribute 'state' will be associated with the M & that is used to record the number (address) of the statement. It is denoted as M.state. The 'nextstate' will point to the next statement.

Production rule	Semantic action
$E \rightarrow E_1$ OR M E_2	{ backpatch(E1.Flist, M.state); E.Tlist := merge(E1.Tlist, E2.Tlist); E.Flist := E2.Flist }
$E \rightarrow E_1$ AND M E_2	{ backpatch(E1.Tlist, M.state); E.Tlist := E2.Tlist; E.Flist := merge(E1.Flist, E2.Flist) }

30/

SEMP
Page No. :
Date : / /

E → NOT E1 {
E.Tlist := E1.Flist;
E.Flist := E1.Tlist;
}

E → id1 · (E1) {
E.Tlist := E1.Tlist;
E.Flist := E1.First;
}

E → id1 relop id2 {
E.Tlist := mklist(nextstate);
E.Flist := mklist(nextstate+1)
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.state := nextstate
}

SEMP
Page No. :
Date : / /

Q.1. Generate an intermediate code for following expression.
A < B OR C < D AND P < Q

→ For the given expression A < B or C < D AND P < Q
We will scan it from left to right.
A < B matches with the rule E → id1 relop id2
In response to this statement the three address code will be generated as,

100 if A < B goto _ } as semantic actions are
101 goto _ | append(if id1.place relope id2.place 'goto_')
append('goto_')

Similarly for remaining part of expression the three address code will be,

102 if C < D goto _
103 goto _
104 if P < Q goto _
105 goto _

Now, for the expression if A < B OR C < D AND P < Q we will solve AND operation first then OR, as AND has higher precedence over OR.

E1.Tlist

102 if C < D goto _ ← backpatch(E1.Tlist, 104)
103 goto _ M.state = nextstate = {104}
104 if P < Q goto _
105 goto _

⇒ E.Tlist := {104}
E.Flist = {103, 105}

hence,
102 if C < D goto _104
103 goto _105
104 if P < Q goto _106
105 goto _

Now, consider

A<B OR C<D AND P<Q

100 if A<B goto _

101 goto _ ← backpatch
backpatch(E1.Tlist)

E1.Tlist

102 if C<D goto _ nextstate

103 goto _

104 if P<Q goto _

105 goto _

E2.Tlist

At this line expression C<D AND P<Q is decided to be true & if P<Q is also true then C<D AND P<Q = TRUE, hence E2.Tlist = {104}

E.Tlist = {100,1

Finally three address code with backpatching will be

100 if A<B goto 102

101 goto 102

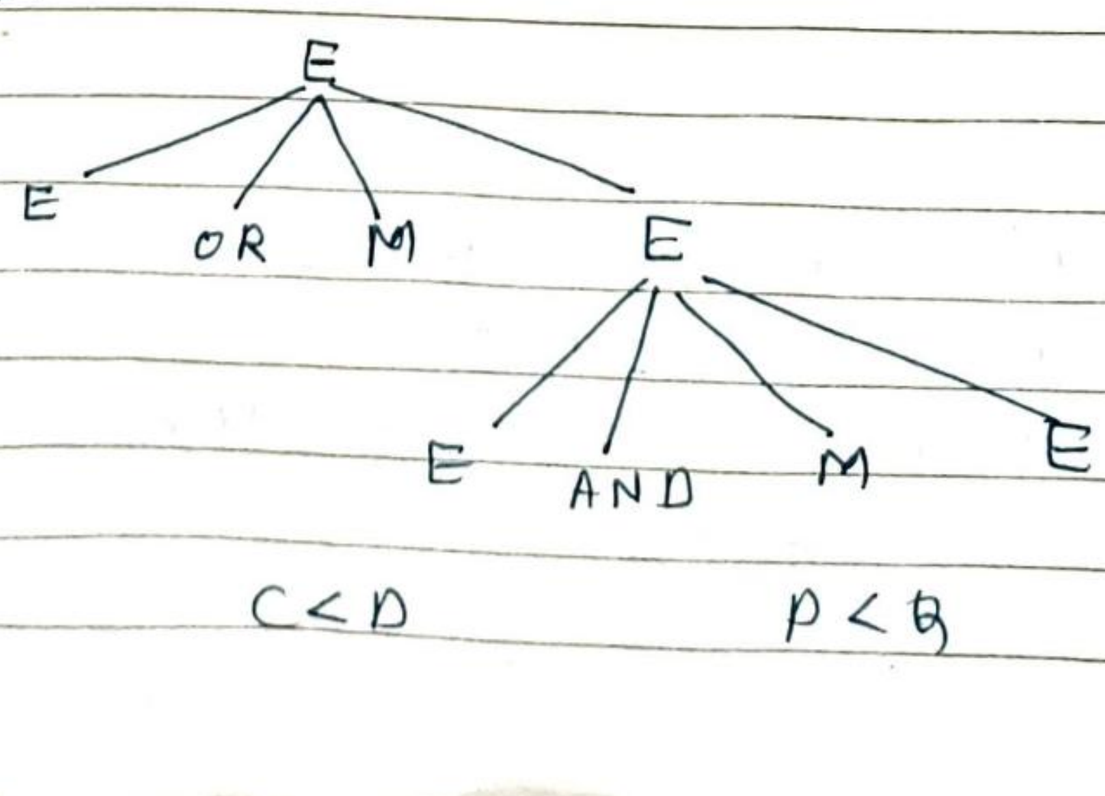
102 if C<D goto 104

103 goto 106

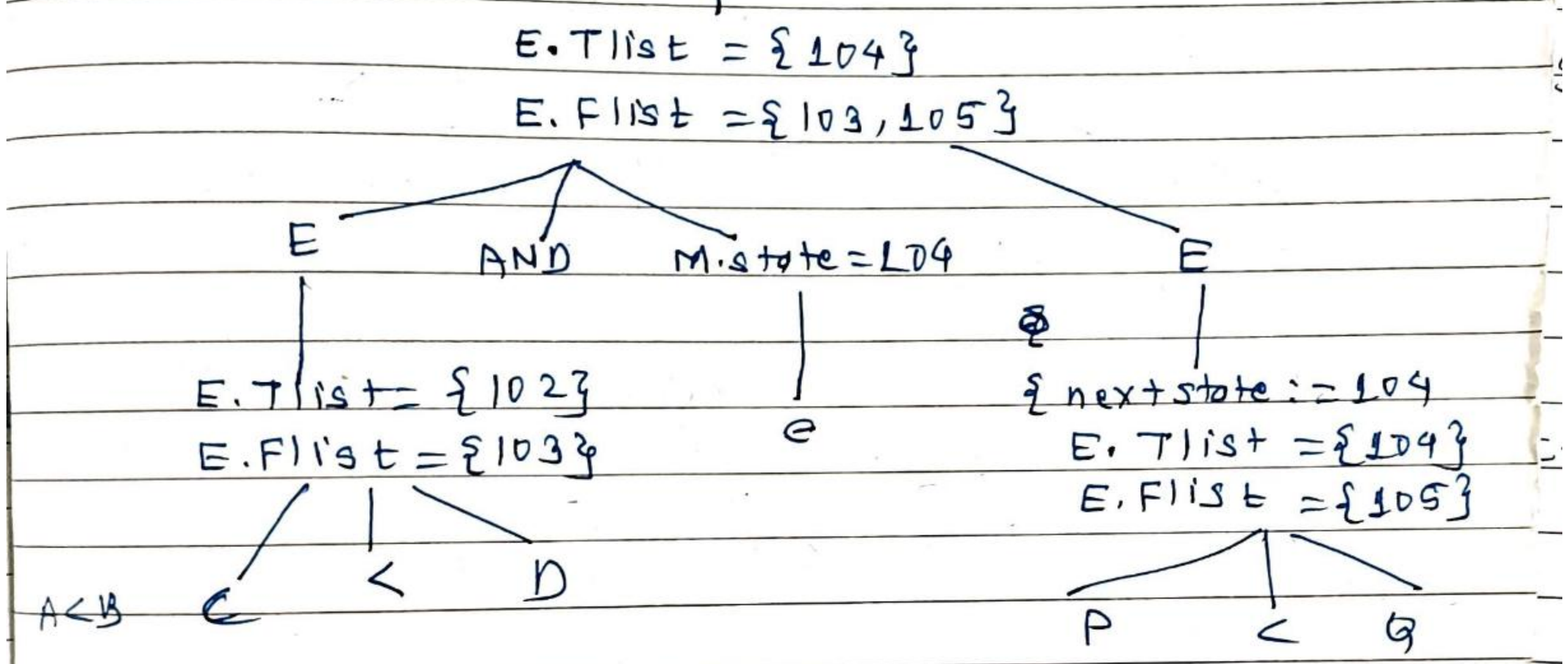
104 if P<Q goto 106

105 goto 106

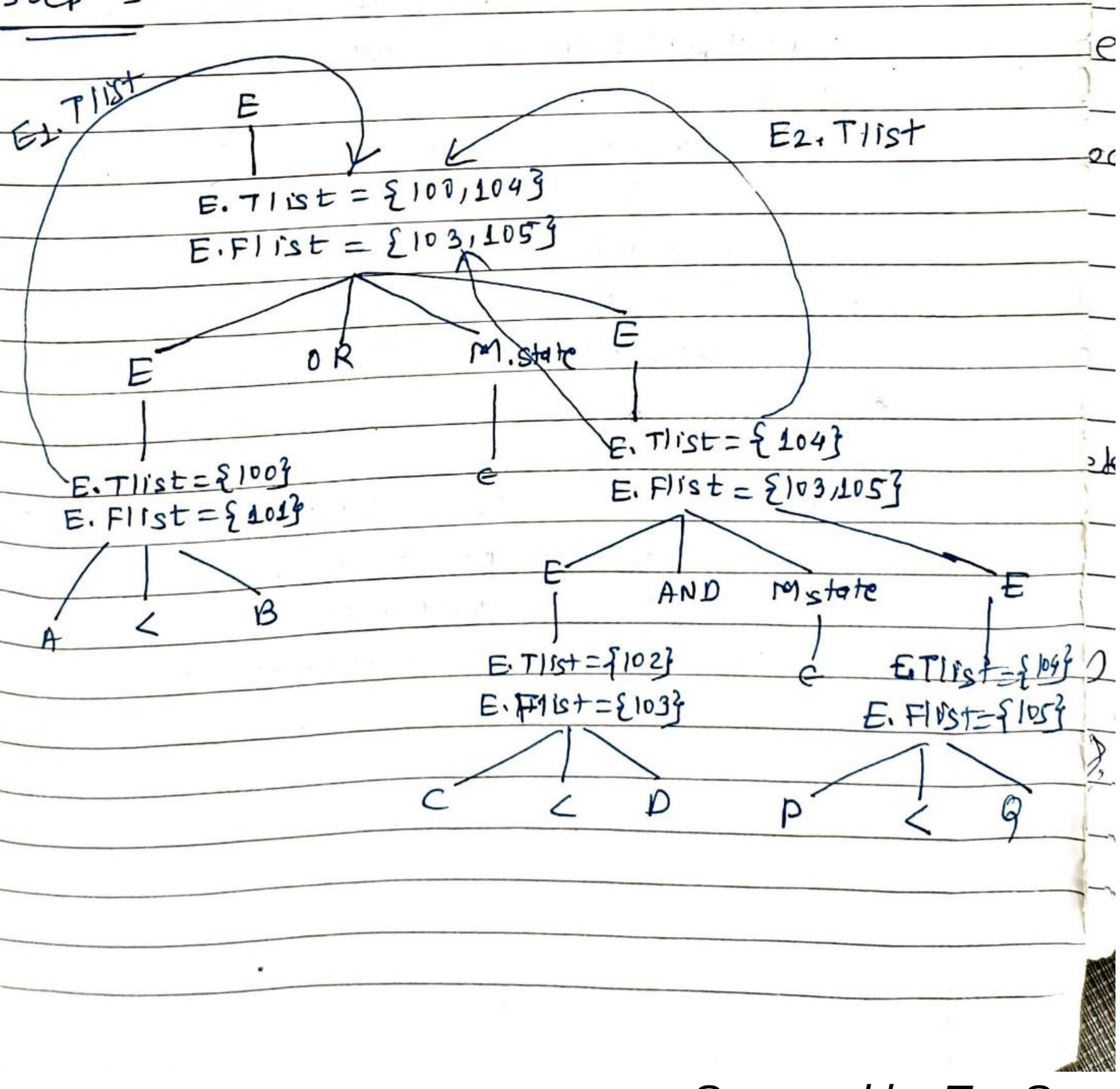
The annotated parse tree can be constructed as follows.

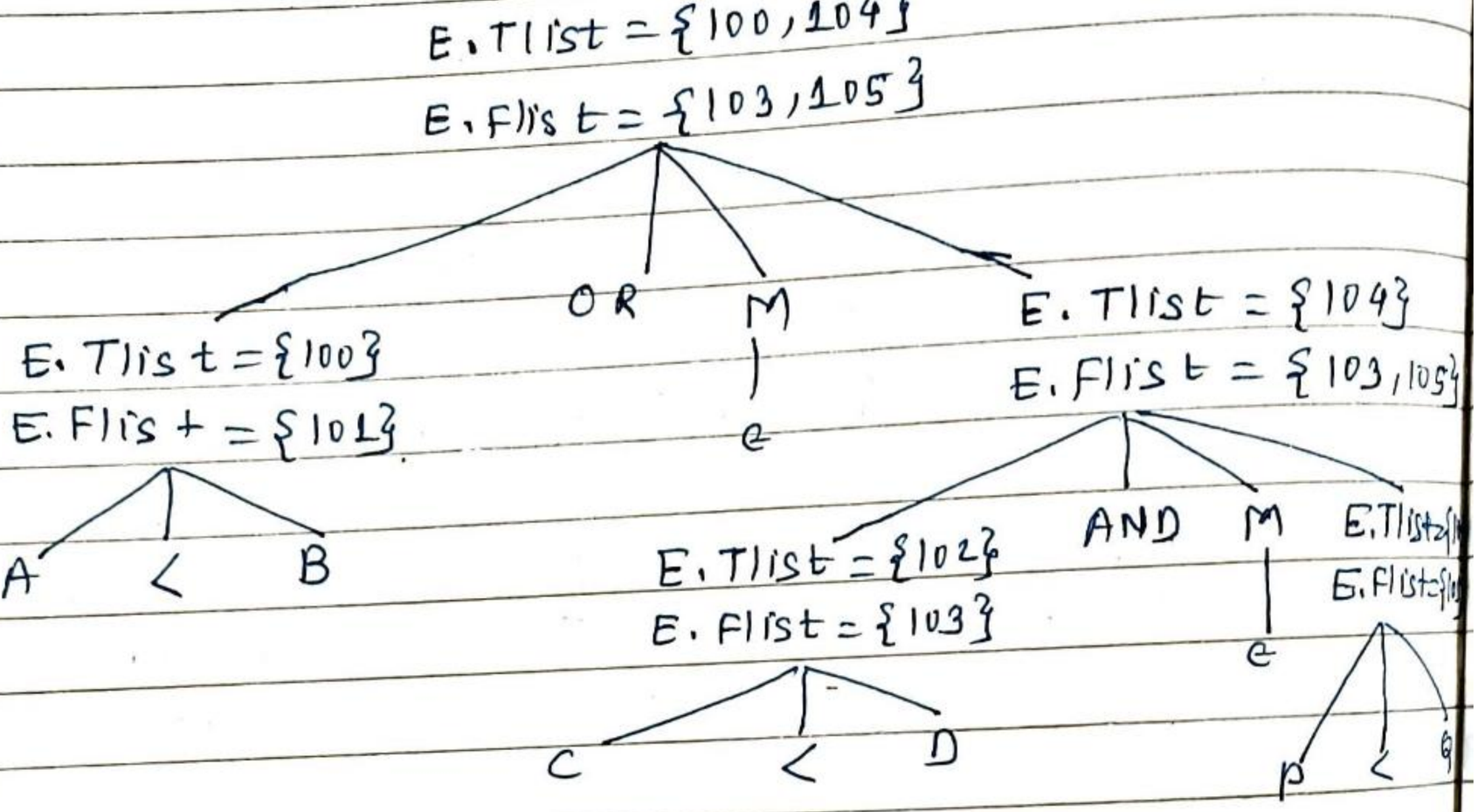


step 2



step 3





Backpatching using flow of control statements

The flow of control statements can be handled using backpatching techniques for the following grammar.

S → if B then S

S → if B then S else S

S → while B then do S

S → begin L end

S → A

L → L;S

L → S

where S stands for statement

B for Boolean expression

L Stands for statement list

A stands for assignment statement

SEMP
Page No.:
Date: / /

* Procedure Calls -

procedure or function is an important programming construct which is used to obtain the modularity in the user program.

Consider, the grammar for a simple procedure call

S → call id (L)

L → L, E

L → E

here the non terminal S denotes the statement & non terminal. L denotes the list of parameters. And E denotes the expression it could be id as well.

The translation scheme can be as given below.

Production rule	Semantic action
S → call id(L)	{ for each item p in queue do append ('param', p); append ('call' id.place) }
L → L, E	{ insert E.place in the queue}
L → E	{ initialize queue & insert E.place in the queue}

The data structure queue is used to hold the various parameters of the procedure. The keyword param is used to denote the list of parameters passed to the procedure. The call to the procedure is given by 'call id' where id denotes the name of procedure. E.place gives the value of the parameter which is inserted in the queue.

For L → E the queue gets empty & a single pointer to the symbol table is obtained. This pointer denotes the value of E.

Q.1) Write three address code statements for the following source language statement.

$X = FUN(2, Y-1) + 2$, where FUN is a function.

→ $X = FUN(2, Y-1) + 2$ where FUN is function

$t1 = Y - 1$

Param 2

Param t1

call FUN, 2/*2 denotes total number of params

Return t2

$t3 = t2 + 2$

$X = t3$

2) Write the three address code that will be generated by yor scheme.

$x = fun(y + 2 * 3, 0) - 1$

→ Consider, three address code for

$x = fun(y + 2 * 3, 0) - 1$

$t1 = 2 * 3$

$t2 = y + t1$

param $t2$

param 0

call fun, 2

Return $t3$

$t4 = t3 - 1$

$x = t4$