Code No.:

INTERMEDIATE CODE GENERATION

- 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 called intermediate language.
- The generation of an intermediate language leads to efficient code generation.

* Benefits of Intermediate Code (IC) Generation

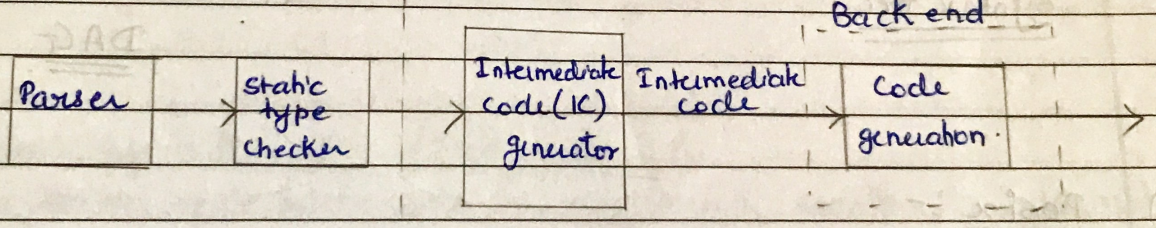
There are certain benefits of generating machine independent IC.

i) A compiler for different m/cs can be created by attaching different backend to the existing front ends of each m/c.

2) A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.

3) A m/c independent code optimizer can be applied to IC in order to optimize the code generation.

The role of IC generator in compiler is depicted in fig. below.



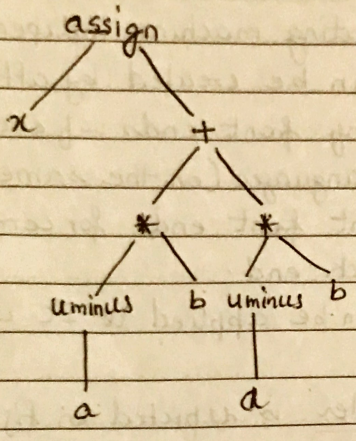
*) Intermediate Languages.

There are mainly three types of IC representations.

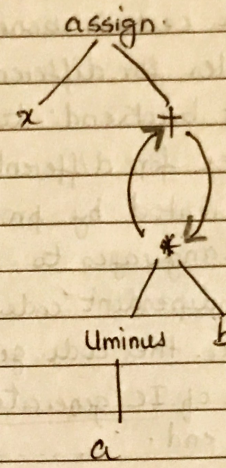
- Syntax tree
- Posix
- Three addren code.

1) Syntax tree :-

- The natural hierarchical structure is represented by syntax trees.
- Directed Acyclic Graph or DAG is very much similar to syntax trees but they are in more compact form.
- The code being generated as ~~Ex~~ intermediate should be such that the remaining processing of the subsequent phases should be easy.
- Consider the i/p string x := - a * b + - a * b for the syntax tree and DAG representation.



Syntax tree



DAG

2) Posix :-

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

  x a - b * a - b * + :=

- Basically, the linearization of syntax trees is posix notation. In this representation, the operator can be easily associated with the corresponding operands.
- This is the most natural way of representation in expression evaluation.

3) **Three address code :-** i/p is in the form of an annotated syntax tree or syntax tree then can be converted into a linear representation, postfix

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

$$a := b \text{ op } c$$

where a, b or c are the operands that can be names, constants, compiler generated temporaries and op represents the operator.

- The operators can be fixed or floating point arithmetic operator or logical operators on Boolean valued data.
- Only single operation at right side of the expression is allowed at a time.
- For the expression like a = b+c+d the three addren code will be

$$t_1 := b + c$$
$$t_2 := t_1 + d$$
$$a := t_2.$$

- Here t_1 and t_2 are the temporary names generated by the compiler.
- There are at the most three addresses are allowed (two for operands and one for result) hence the name of this representation is three-addren code.

(*) **Types of 3 Addren Statements :-**

- The form of 3 addr code is very much similar to assembly language. Here are some commonly used 3-addr codes for typical language constructs.

Language construct	IC form	Meaning.
Assignment statement	$x := y \text{ op } z$	Here binary operation is performed using 'op'.
Assignment statement	$x := \text{op } y$	Here the unary operation is performe. The operator 'op' is an unary operator
Copy statement	$x := y$	Here the value of y is assigned to x

P.T.O.

unconditional jump	goto L	The control flow goes to the statement labelled by L.
conditional jump	if x relop y goto L	The relop indicates the relational operators such as $<, =, >, \geq, \leq$. If x relop y is true then it executes goto L statement.
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 as parameters to the procedure p.
	return y	The return stmts indicates the return value y.
Array statements	x := y[i]	The value at i^{th} index of array y is assigned to x.
	x[i] := y	The value of identifier y is assigned at the index i of the array x.
Address & pointer assignments	x := &y	The value of x will be the addr or location of y.
	x := *y	The y is a ptr whose value is assigned to x.
	*x := y	The r-value of object pointed by x is set by the l-value of y.

(*) Implementation of Three Address Code

- Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields.
- There are 3 representations used for three address code such as quadruples, triples and indirect triples.

(1) Quadruple representation

- The quadruple is a structure with at the most four fields such as op, arg1, arg2, result.
- The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.
- Eg:- Consider the input statement **x := -a * b + -a * b**

The three address code is

$t_1 := \text{uminus } a$

$t_2 := t_1 * b$

$t_3 := -a$

$t_4 := t_3 * b$

$t_5 := t_2 + t_4$

$x := t_5$

Quadruple

	Op	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

2) Triples :- Temporaries are not used & instead references to instr. are used.

- In the triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.
- For the expression x := -a * b + -a * b the triple representation is as given below.

Number	op	Arg1	Arg2	
(0)	uminus	a		> references to instr
(1)	*	(0)	b	
(2)	uminus	a		
(3)	*	(2)	b	
(4)	+	(1)	(3)	
(5)	:=	x	(4)	

3) Indirect triples :- in addn to triples we use a list of ptrs to triples

- In the indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements. x := -a*b + -a*b

Numbers	op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	x	(15)

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

- In the quadruple representation using temporary names the entries in the symbol table against those temporaries can be obtained. The advantage with quadruple representation is that one can quickly access the value of temporary

variables using symbol table.

- Use of temporaries introduces the level of indirection for the use of symbol table in quadruple representation. Whereas, in triple representation the pointers are used. By using pointers one can access directly the symbol table entry.
- The quadruple representation is beneficial for code optimization. In indirect triple list of all references to computations is made separately and stored.
- Thus indirect triple and quadruple representations are similar as far as their utility is concerned.
- But indirect triple saves some amount of space as compared with quadruple representation.

(*) Generation of Three Address Code

- The translation scheme for various language constructs can be generated by using the appropriate semantic actions.

(1) Declarations :-

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

eg:

$S \rightarrow D$	{offset := 0}
$D \rightarrow id : T$	{enter_tab(id.name, T.type, 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 T_1	{T.type := array(num.val, T_1.type) T.width := num.val X T_1.width}
$T \rightarrow *T_1$	{T.type := pointer(T.type) T.width := 4}

- Initially the value of offset is set to zero. The computation of offset can be done by using the formula offset = offset + width.
- In the above translation scheme T.type, T.width are the synthesized attributes. The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type.
  For instance integer has width 4 and real has 8.
- The rule $D \rightarrow id : T$ is a declarative statement for id

declaration. The enter-tab is a function used for creating the symbol table entry for identifier along with its type and offset.

- The width of array is obtained by multiplying the width of each element by number of elements in the array.
- The width of pointer type is supposed to be 4.

(2) Assignment statements :-

- The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record.

Eg: Obtain the translation scheme for obtaining the three address code for the grammar

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$.

Soln Here we will use the translation scheme 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 ':=' E_1.place '+' E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place := newtemp(); append(E.place ':=' E_1.place '*' E_2.place) }
$E \rightarrow -E_1$	{ E.place := newtemp(); append(E.place ':=' 'uminus' E_1.place) }

$E \rightarrow (E_1)$	{ 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 va
- E.place is used to hold the value of E.

Consider the assignment statement $x := (a+b) * (c+d)$ we will assume all these identifiers are of the same type. Let us have bottom-up parsing method.

Production rule	Semantic action for attribute evaluation	Output
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a+b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	$t_2 := c+d$
$E \rightarrow E_1 * E_2$	E.place := t_3	$t_3 := (a+b) *$
$S \rightarrow id := E$		$x := t_3$

The annotated parse tree can be drawn as follows: -

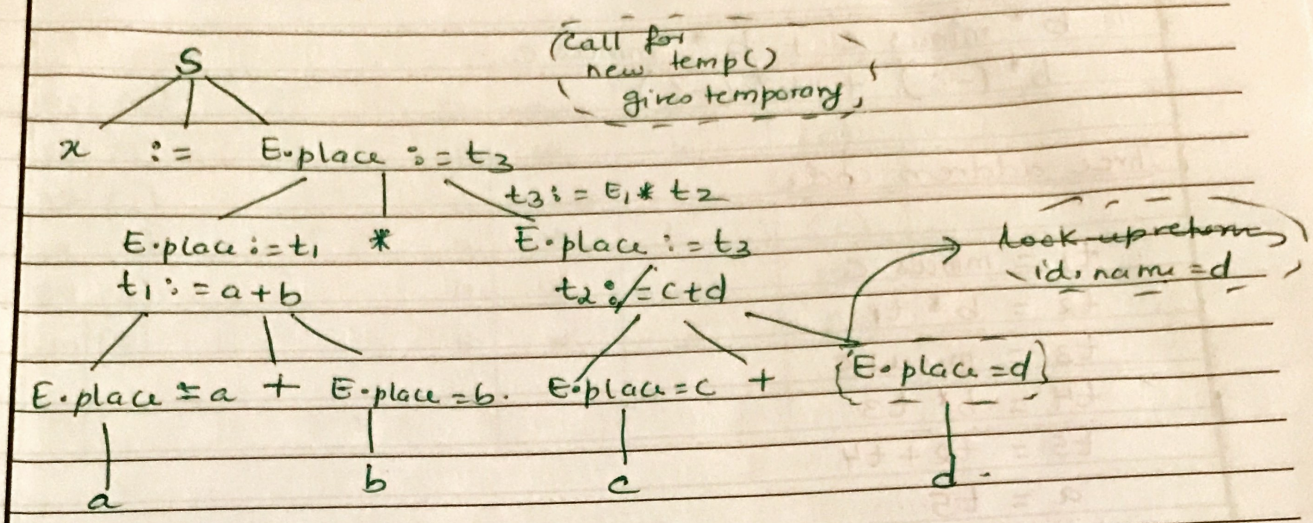


fig :- Annoted parse tree for generation of three address code

Example of three address code

b * minus c + b * minus c.
b * (-c) + b * (-c)

Three address code

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

Quadruples.

op	arg1	arg2	result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

Triples.

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect triples

	op
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)

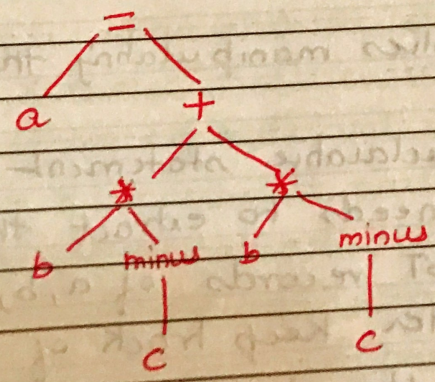
eg

Triples Examples :

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

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Syntax tree



Syntax-Directed Definition (SDD)

SDD = CFG + Semantic Rules

→ A SDD is a context Free grammar together with Semantic Rules.

→ Attributes are associated with grammar symbols and semantic rules are associated with productions.

→ If 'X' is a symbol and 'a' is one of its attribute then X.a denotes value at node 'X'.

→ attributes may be numbers, strings, references, datatypes etc.

attribute b
y
y.b → value at node y.

production	Semantic Rule.
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val.

E & T are grammar symbols, val is attribute
Semantic means it provides meaning to the corresponding production.

→ Every symbol must contain an attribute

Types of Attributes:-

1) **Synthesized Attribute :-**

If a node takes value from its children then it is synthesized attribute.

Eg:- A → BCD , A be a parent node
B, C, D are children nodes.

A.S = B.S
A.S = C.S
A.S = D.S

} parent node A taking value from its children B, C, D.

Parent node is taking value from children node.

2) Inherited Attribute :-

If a node takes value from its parent or siblings.

Eg: A → BCD A produces BCD.

C.i = A.i ---> parent node i - attribute associated with symbol BCD.
C.i = B.i ---> sibling node
C.i = D.i ---> sibling node.
C is taking value from parent A.

Types of SDD :-

(1) S - Attributed SDD or S - Attributed Definitions or S - Attributed grammar

(2) L - Attributed SDD or L - Attributed Definitions or L - Attributed grammar

(1) S - Attributed SDD

1) A SDD that uses only synthesized Attributes is called as S - Attributed SDD.

Eg:- A → BCD
A.S = B.S
A.S = C.S
A.S = D.S.

2) Semantic Actions are always placed at right end of the production. It is also called as "postfix SDD".

3) Attributes are Evaluated with Bottom-up parser.

(2) L - Attributed SDD

1) A SDD that uses both synthesized & Inherited Attributes is called as L - Attributed SDD but each inherited Attribute is restricted to inherits from parent or left sibling only.

Eg:- A → XYZ { Y.S = A.S, Y.S = X.S, Z.S = X.S }.

2) Semantic Actions are placed anyway at on R.H.S.

3. Attributes are Evaluated by traversing parse tree depth first, left to right order.

Translation of Assignment statements

- In SDT, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array & records.

Eg: The translation scheme of this grammar is

Production Rule	Semantic Actions
$S \to id := E$	$\{ P = \text{look-up}(id.name);$ (symbol table)
$E \to E_1 + E_2$	if $P \neq nil$ then
$E \to E_1 * E_2$	Emit ($P = E.place$) (generate TAC)
$E \to (E_1)$	else
$E \to id$	error; $\}$

② $E \to E_1 + E_2$

$\{ E.place = newtemp();$

Emit ($E.place = E_1.place + E_2.place$) (generate TAC)

$\}$

$E.place = t_1$

$t_1 = E_1 + E_2$

③ $E \to E_1 * E_2$

$\{ E.place = newtemp();$

Emit ($E.place = E_1.place$ '*' $E_2.place$) (TAC)

$\}$

$E.place = t_2$

$t_2 = E_1 * E_2$

④ $E \to (E_1)$

$\{ E.place = E_1.place \}$

⑤ $E \to id$

$\{ P = look\text{-}up(id.name);$

If $P \neq nil$ then

Emit ($p = E.place$)

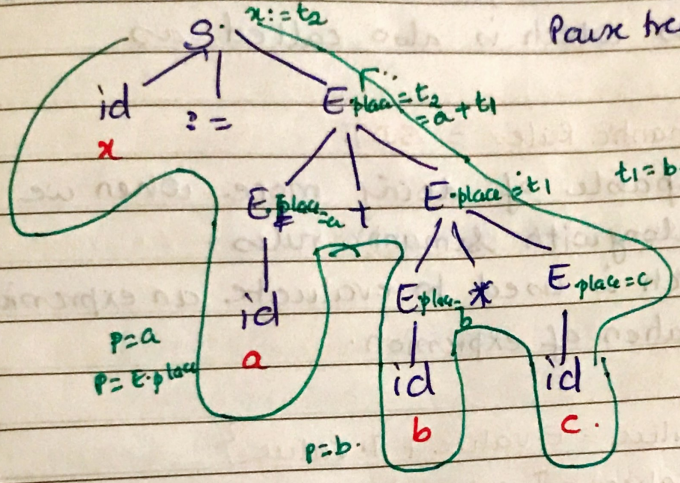
Else

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.

Eg. $x = a + b * c$. ⇒ expression.



Parse tree

p = a.

→ Traverse parse tree top down left to Right

Syntax - Directed Translation SDT

- Whenever for a given grammar we have a parse tree for that grammar.
- Till now we have seen some grammars and for grammars we have seen parsers. Now in practice whenever we go for any compiler design which we implement along with the grammar we have some informal notations which is also called as Semantic Rules.

Grammar + Semantic Rules = SDT

- parsers are capable of doing more when we give grammar along with semantic rules.
- Let's take eg which is used to evaluate an expression SDT for evaluation of expression.

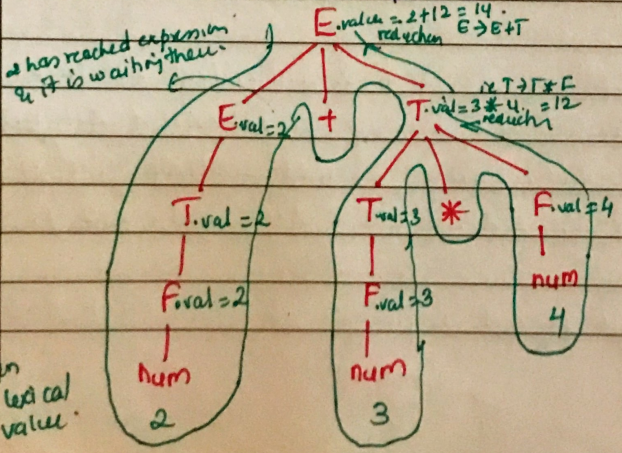
$E \Rightarrow E_1 + T$ { E.value = E_1.value + T.value }
$\quad\quad / T$ { E.value = T.value }.

$T \Rightarrow T_1 * F$ { T.value = T_1.value * F.value }
$\quad\quad / F$ { T.value = F.value }

$F \Rightarrow num$ { F.val = num.lvalue }

Let us take a string 2 + 3 * 4. What will be the output of this ie 14.

- I have to get 14 as the output if I carry out this SDT on this expression.
- We have to drive the parse tree.



num is token
2, 3, 4 are lexical value.

After generating parse tree we have to traverse it top to bottom down left to right.
And whenever there is a reduction, we go to the production and carry out the actions.
What is the reduction num is reduced to F. What is the semantic action. { F.val = num.lvalue }
F is associated with a value ie val. it is also called as attribute.
variable is NT
Every NT is going to get an attribute in this SDT. it is not necessary that in all SDT we should get it but in this SDT every variable is going to get an attribute.
What is attribute associated with F it is F.value.
So F.val = num.lvalue, what is the lexical value ie 2.
Which means this information of 2 is passed to F.
So again there is a reduction F is reduced to T.
So what is the action T.value = F.value. ie 2.
again there is reduction.

NOTE :- In SDT every variable is going to get one or more attribute, sometimes it may get even zero attribute also that depends on the type of SDT we are talking about.

Syntax Directed Translation

[Boolean Expression - short circuit]

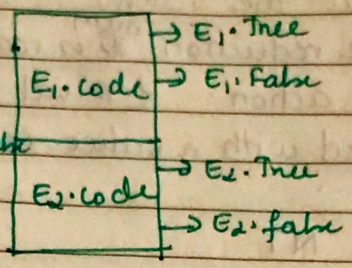
$E \rightarrow E_1$ or E_2

$E_1 \cdot true = E \cdot true$

$E_1 \cdot false$ = new lable $E_1.false$

$E_2 \cdot true = E \cdot true$

$E_2 \cdot false = E \cdot false$



code

$E \cdot code = E_1 \cdot code \,||\, gen(E_1 \cdot false) \,||\, E_2 \cdot code$

Boolean Expressions

Boolean Expressions have two primary purposes
1. used for computing logical values.
2. used as conditional expressions using if-then-else or while-do.

consider a grammar.

1) E → E OR E
2) E → E AND E
3) E → NOT E
4) E → (E)
5) E → id relop id
6) E → True
7) E → False

The relop is denoted by <, >.

So for this grammar we have to construct SDT.

The AND & OR are left associated.

NOT has higher precedence then AND & lastly OR.

Production Rule	Semantic Action
1) $E_0 \rightarrow E_1$ OR E_2	{ E.place = newtemp(); TAC ← Emit (E.place ':=' E_1.place 'OR' E_2.place) }
2) E → E_1 AND E_2	{ E.place = newtemp(); Emit (E.place ':=' E_1.place 'AND' E_2.place) }
3) E → True = 1	{ E.place := newtemp(); Emit (E.place ':=' 1) }
4) E → False = 0	{ E.place := newtemp(); Emit (E.place ':=' '0') }

Emit generates TAC & newtemp() generates temporary variable

3) $E \rightarrow NOT\ E_1$ { E1.place = newtemp();
emit (E.place ':=' 'NOT' E1.place)
}

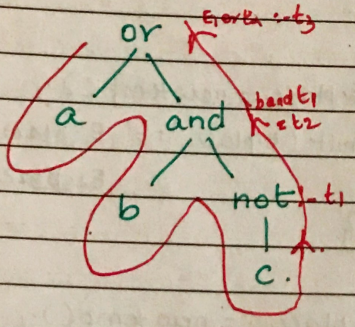
4) $E \rightarrow (E_1)$ { E.place = E1.place }

* (5) $E \rightarrow id_1\ relop\ id_2$ { E.place = newtemp();
emit ('if' id1.place relop.op id2.place
jump to 3 steps ahead 'goto' nextstate+3);
emit (E.place ':=' '0')
emit ('goto' next-state+2)
emit (E.place ':=' '1')

$E \rightarrow id\ relop\ id_2$ contains next state & it gives the index of next three address statement in the output sequence.

eg. for 1, 2, 3.

a or b and not c



t_1 = not c

t_2 = b and t_1

t_3 = a or t_2

Eg: if a<b then 1 else 0 — statement.

lets take location 100 ; series of steps.

100 : if a<b goto 103 → 100+3 jump to 103 — do first take the condition. if a<b goto next stm +3

101 : $t_1 = 0$ — E.place = 0 , E.place = temp var ie t_1

102 : goto 104 → 102+2

103 : $t_1 = 1$ — E.place = 1 → temp variable.

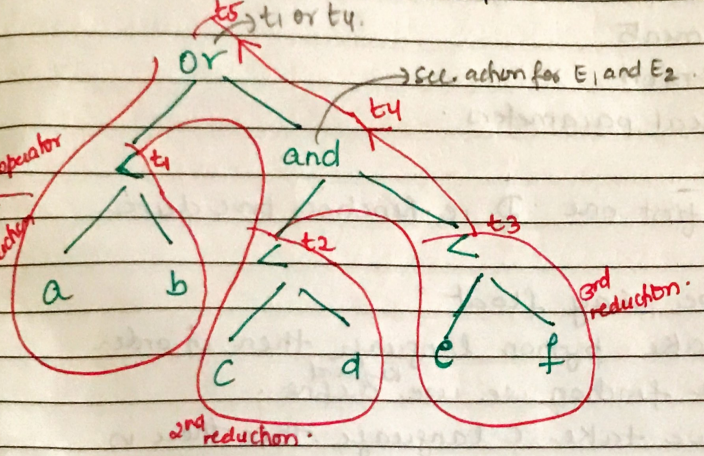
104 :

if a<b goto 103 , 103 if $t_1 = 1$, otherwise 0 . if stm is false it jumps to 104 ie goes to next stm.

lets take example which covers all the operations.

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

first let us draw the parse tree.



if a<b

100 : if a<b goto 103

101 : $t_1 = 0$

102 : goto 104

103 : $t_1 = 1$

104 : if c<d goto 107 → 104+3 → if c<d goto 107 if wrong continue to 105 → E.place = 0

105 : $t_2 = 0$

106 : goto 108

107 : $t_2 = 1$

108 : if e<f goto 111

109 : $t_3 = 0$

110 : goto 112

111 : $t_3 = 1$

112 : $t_4 = t_2$ and t_3

113 : $t_5 = t_1$ or t_4

Intermediate code for Procedures

- procedures are functions

data type → function name

D → define T id (F) { S }

id → variable name, F → formal parameters.

F → ∈ | T id , F

S → return E ;

E → id (A) ;

A → ∈ | E , A

return add( ).

- We use several NTs. first one is D, D stands for function procedure definition.
T - stands for return type or data type
id - identifier.
F - formal parameters.
S - Statements
E - expression.
A - Actual parameters.

- lets take first one. D ie function procedure definition.
T - data type say float.
- so if you take python language then in order to define the function we use keyword define.
- whereas if we take C language then there is no need of define keyword
Here T means data type so float, id means identifier ie ~~not~~ here it is name of the function.

```
float add (F)
```

↳ formal parameters.

we can write like this F → ∈ | T ⇒ data type which is int

```
float add ( )  or
          (int a)  or
          (int a, float b)
{.  return add( )
    statements
}
```

```
main ()
{
    add ();
}
```

→ actual parameter.

Intermediate code for switch statement
or
Translation of switch statement

switch statement syntax :-

switch (E) → expression (it may be integer expression or character expression)
{
case V_1 : S_1 → case values are integer or character value
case V_2 : S_2 → statement blocks
.....
case V_{n-1} : S_{n-1}
default : S_n.
}

- switch is useful when we have no of choices and if we want to select one choice among the list of choices then we use a switch
- E → expression will be evaluated
- if Expression matches case value V_1 then corresponding stmt S_1 will be executed.
- after stmt blk we may need to use break stmt.
- if no break then remaining case will also be executed.
- if expr doesn't match with any then default blk will be executed.

After evaluation of this expression it may produce integer or character value

optional

If

Translation of switch statement

t - stores result of expression

```
Code to evaluate E into t
    goto test          → control goes to label test
L1 : code for S1
    goto next
L2 : code for S2
    goto next
.....
Ln-1 : code for Sn-1
    goto next
Ln : code for Sn
    goto next
```

```
test : if t = V1 goto L1      (1)  → expression result
       if t = V2 goto L2      (2)
       .....
       if t = Vn-1 goto Ln-1  (3)
       goto Ln
next :
```

no stmt so execution will be over.

Q. Generate three address code for:

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

Soln :-

```
100.      if ch = 1 goto L1
101.      if ch = 2 goto L2
102. L1 : t1 := a+b.
103.      c := t1
104.      goto Last
105. L2 : t1 := a-b.
106.      c := t1
107.      goto last
108.      Last
```

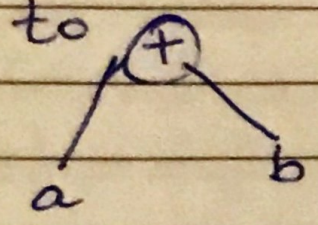
Directed Acyclic Graph

DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too.

- leaf nodes represent identifiers, names or constants
- interior nodes represent operators
- interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

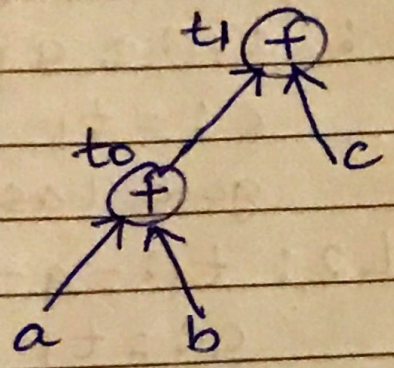
eg.

$t_0 = a + b$



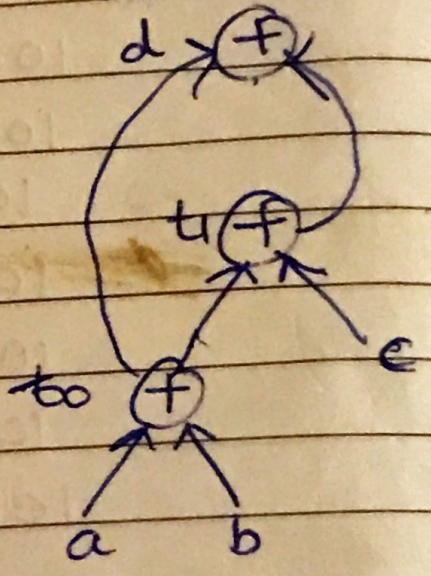
[$t_0 = a + b$]

$t_1 = t_0 + c$



[$t_1 = t_0 + c$]

$d = t_0 + t_1$



[$d = t_0 + t_1$]