UNIT – III

INTERMEDIATE CODE GENERATION

Types of intermediate code – Representation of three address code – syntax Directed translation scheme – Intermediate code generation for : Assignment statements – Boolean statements – switch -case statement – Procedure call – Symbol Table Generation.

Types of Intermediate code:

* The translation of the source code into the object code for the target m/c, a compiler can produce middle level language code, which is referred to intermediate code.

Types : 8 types.

1) Postfix Notation
2) Syntax Tree
3) Three Address code
   - Quadruples
   - Triples
   - Indirect Triples.
4) DAG
   (Directed Acyclic Graph)

Postfix Notation:

→ operator comes after an operand.

→ ie, the operator follows an operand

Eg:

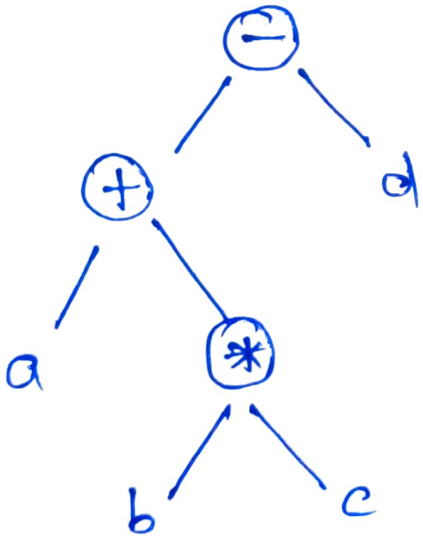
$-(a+b)*(c+d) \longrightarrow ab+cd+*$

$-(a*b)-(c+d) \longrightarrow ab*+cd+-$

Syntax Tree:

→ leaf node describes an operand & each interior node is an operator.

Eg: $a+b*c-d$.



Three Address code:

→ sequence of stmts of the form $A = B\ op\ C$ where A, B, C are either programmer-defined names, constants or compiler generated temp-names,

The op represents an operator.

→ reason for the name 3-addr code is that each stmt generally includes 3 addr, 2 for the operands and one for result.

Types of 3 address :

Quadruples :

→ it is a structure which consists of 4 fields namely op, arg1, arg2 and result.

→ op denotes the operator

→ arg1 & arg2 denotes two operands

→ result is used to store the result of the expn.

Triples :

→ doesn't make use of extra temporary variables to represent a single operation, instead when a reference to another triple's value is needed, a ptr to a triple is used.

→ it consist of only three fields namely op, arg1 and arg2.

Indirect Triples:

→ it makes use of pointer to the listing of all references to computation which is made separately and stored.

→ similar in utility as compared to quadruple representation but requires less space than it.

→ Temporaries are implicit and easier to rearrange code.

Representation of 3 addr code: (Implementation)

* 3-address code is a type of intermediate code, which is easy to generate and can be easily converted to m/c code.

* It makes use of atmost three addresses.

General representation

$a = b \ op \ c$

where

- a, b or c represents operands like names, constants or compiler generated temp names.
- op represents operator.

Example: convert a * - (b+c) into 3-address code.

$t_1 = b + c$

$t_2 = \text{uminus } t_1$

$t_3 = a * t_2$

Implamantation:

* There are 3 -representations of 3-address code.

1) Quadruples
2) Triples
3) Indirect Triples.

Quadruple:

* It is a structure which consists of 4 fields.

* op → arg1, arg2 and result.
  - op → operator
  - arg1, arg2 → two operands
  - result → store the result.

Advantage:

→ Easy to re-arrange code for global optimization

→ one can quickly access value of temporary variable using symbol table.

Disadvantage:

→ Contain lot of temporaries.

→ temporary variable creation increases time and space complexity.

Example $a = b * -c + b * -c$

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

$t_3 = \text{uminus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	Arg 1	Arg 2	Result
(0)	uminus	c		t_1
(1)	*	t_1	b	t_2
(2)	uminus	c		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Triples:

* It doesn't make use of extra temporary variable to represent single operation instead when a reference to another triple's value is needed, a ptr to that triple is used.

* It consist of only three fields namely op, arg1 and arg2.

Disadvantage:

→ temporaries are implicit and difficult to rearrange code.

→ difficult to optimize because optimization involves moving intermediate code.

↳ when a triple is moved, any other triple referring to it must be updated also.

↳ with a help of ptr one can directly access symbol table entry.

Example: $a = b * -c + b * -c$

#	op	Arg 1	Arg 2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Indirect Triples:

* It makes use of ptr to the listing of all reference to computations which is made separately stored.
* Its similar to utility as compared to quadruples but requires less space.
* Temporaries are implicit and easier to rearrange code

Example : a = b * -c + b * -c

#	op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

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

Eg: 2

(x+y) * (y+z) + (x+y+z)

3 Addr code

$t_1 = x + y$

$t_2 = y + z$

$t_3 = t_1 * t_2$

$t_4 = t_1 + z$

$t_5 = t_3 + t_4$

#	op	Arg1	Arg2	result
(1)	+	x	y	t_1
(2)	+	y	z	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	z	t_4
(5)	+	t_3	t_4	t_5

Quadruple.

#	op	Arg 1	Arg 2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triple

#	op	Arg 1	Arg 2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Stmt
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triple.

Types of 3-addr stmts:

$x := y$ op z ⟶ Assignment

$x :=$ op y ⟶ unary assignment

$x := y$ ⟶ copy

goto L ⟶ unconditional jump

if x relop goto L ⟶ conditional jump

Param x
call p n
return y
} ⟶ Procedure call

$x := y[i]$ $x[i] := y$ ⟶ Indexed Assignment.

Syntax Directed Translation ~~Schema~~

SDT
↳ Grammar + semantic rules

* helps in producing intermediate code.

* Intermediate code

  - postfix notation
  - 3-address code
  - DAG.

~~DAG~~!

informal notation

* Every NT can get 0 or more attributes.
* in semantic rule,

att ← VAL, str, no, num, loc, ...

Page 5

Syntax Directed Definition (SDD)

* Specifies the value of attributes by associating semantic rules with the grammar productions.

* It is a CFG with attributes and rules together which are associated with grammar symbols and productions respectively.

Process of SDT

* Construction of syntax tree
* Computing values of attributes at each node by visiting the node of syntax tree.

Eg: SDT (Grammar + Semantic Rules)

Production	Semantic Rule - (Action)
$E \rightarrow E+T$	E.val := E.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T*F$	T.val := T.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow Num$	F.val := num.lexval

⇓

Attribute return by LA.

SDT Scheme:

* It is a CFG.
* It is used to evaluate the order of semantic rules.
* In translation scheme, the semantic rules are embedded with the right side of the production.
* The position at which an action is to be executed in shown by enclosed b/w braces. It is written with the right side of the production.

Eg:

Production	Semantic Rules
$S \rightarrow E\ \$$	{Print E.val}
$E \rightarrow E + E$	{E.val := E.val + E.val}
$E \rightarrow E * E$	{E.val := E.val * E.val}
$E \rightarrow (E)$	{E.val := E.val}
$E \rightarrow I$	{E.val := I.val}
$I \rightarrow I$ digit	{I.val := 10 * I.val + lexval}
$I \rightarrow$ digit	{I.val := lexval}

Implementation of SDT:

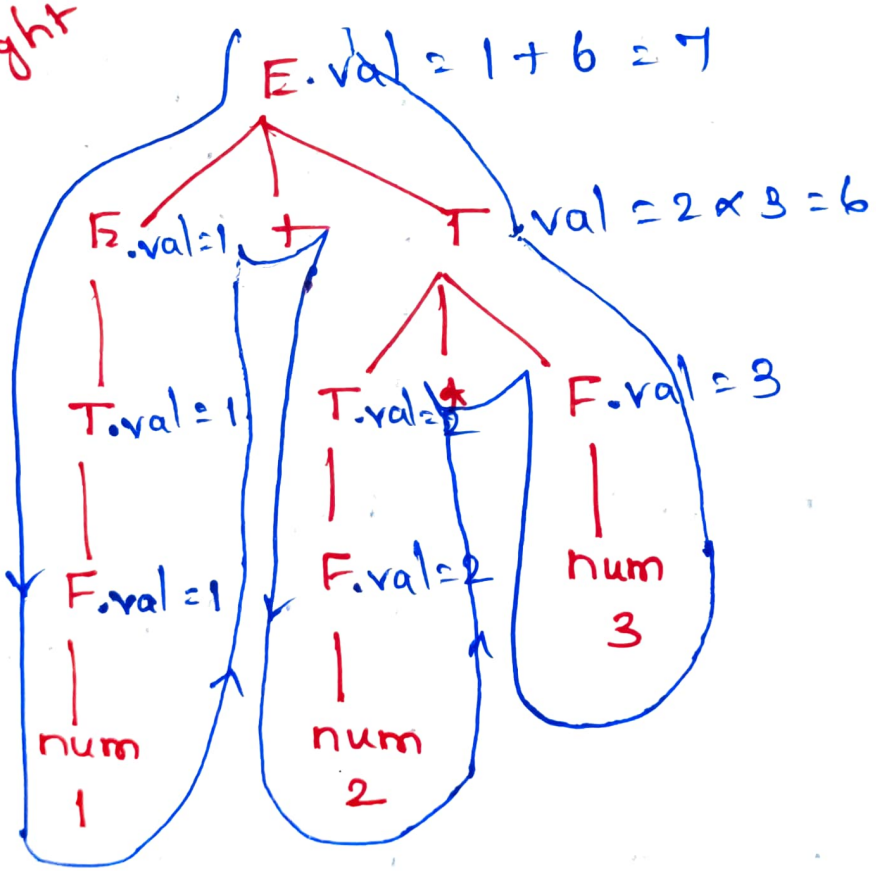
* SDT is implemented by constructing a parse tree and performing the action in a left to right depth first order. $a+b*c \rightarrow$

* SDT is implementing by parse the input and a parse tree as a result.

Grammar	Action (Semantic Rule)
$E \rightarrow E + T$	$\{E.val = E.val + T.val\}$
$E \rightarrow T$	$\{E.val := T.val\}$
$T \rightarrow T * F$	$\{T.val := T.val * F.val\}$
$T \rightarrow F$	$\{T.val := F.val\}$
$F \rightarrow num$	$\{F.val := num.lexval\}$

Eg 1 + 2 * 3 Construct Syntax Tree

- Top down
- left to right



Intermediate code Generation for Assignment stmt

* In SDT, Assignment stmt is mainly deals with expressions.

* Expression can be of the type real, integer, array & records.

Eg:

S → id := E

E → E1 + E2

E → E1 * E2

E → (E)

E → id

The translation-scheme of this grammar is

Production Rule	semantic Actions.
S → id := E	{ p = look_up (id.name); (symbol table) if p ≠ null then Emit (p = E.place) /* addr code Else Error; }
E → E1 + E2	{ E.place = new temp(); (t1) Emit (E.place = E1.place '+' E2.place) /* 3-addr code */ }
E.place = t1 t1 = t1 + t2	
E → E1 * E2	{ E.place = new temp(); Emit (E.place = E1.place '*' E2.place) }
E.place = t2 t2 = t1 * t2	

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

$E \rightarrow id$ { P = loop-up (id.name);
val
assign val if P ≠ null then
P ≠ null. Emit (P = E.place) 3 addr code
3-addr code
Else
Error;
append 3 addr code }

→ The P returns the entry fn id.name in the symbol table.

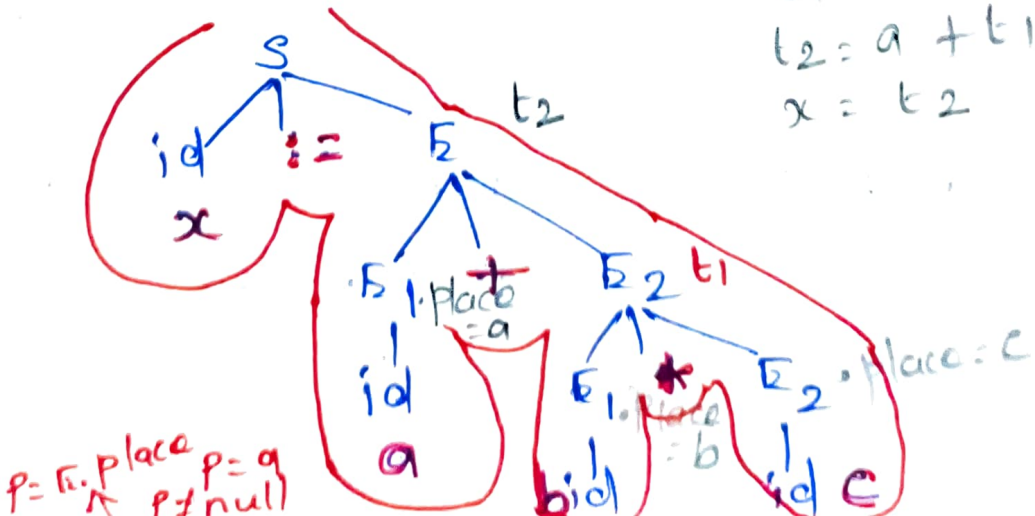
→ The Emit fn is used for appending the 3-addr code to the o/p file. Otherwise it will report an error.

→ The new temp() is a fn used to generate new temporary variables.

→ E.place holds the value of E.

Example $x = a + b * c$

$t_1 : b * c$
$t_2 : a + t_1$
$x : t_2$



Boolean stmts:

* 2 primary purposes.
  1. used for computing logical values
  2. used as conditional Expressions using if-then-else (or) while-do.

⇒ consider a grammar

relational operators <, >

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 AND & OR are left associated.

⇒ NOT has higher precedence than AND & left OR.

Production Rule	Semantic Action
1. $E \rightarrow E_1$ OR E_2	{ E.place = newtemp(); Emit(E.place ':=' E_1.place 'OR' E_2.place) }
2. $E \rightarrow E_1$ AND E_2	{ E.place = newtemp(); Emit(E.place ':=' E_1.place 'AND' E_2.place) }

3. E → NOT E1 { E.place = newtemp(); Emit(E.place ':=' 'NOT' E1.place }

4. E → (E1) { E.place = E1.place }

5. E → id1 relop id2 { E.place = newtemp();
Emit(if id1.place relop id2.place 'goto' nextstate + 3);
Emit (E.place ':=' '0')
Emit ('goto' next state + 2)
Emit (E.place ':=' '1')
}

6. E → True { E.place := newtemp();
Emit (E.place := '1')
}

7. E → False { E.place := newtemp();
Emit (E.place ':=' '0')
}

⇒ Emit generates 3-addr code & newtemp() generates temporary var.

⇒ E → id1 relop id2 contains next_state & it gives the index of next 3-addr stmt in the o/p sequence.

Example: a or b and not c

$E \rightarrow E$ OR E

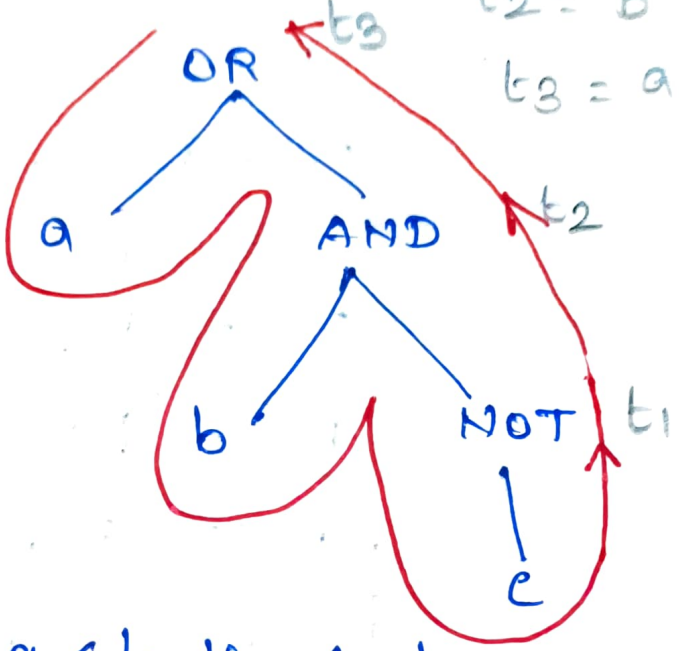
$E \rightarrow E$ AND E

$E \rightarrow$ NOT E

t_1 = not c

t_2 = b and t_1

t_3 = a or t_2



Example: if a < b then 1 else 0.

100 : if a < b goto 103

101 : t_1 = 0 (false)

102 : goto 104

103 : t_1 = 1 (true)

109 : t_3 = 0

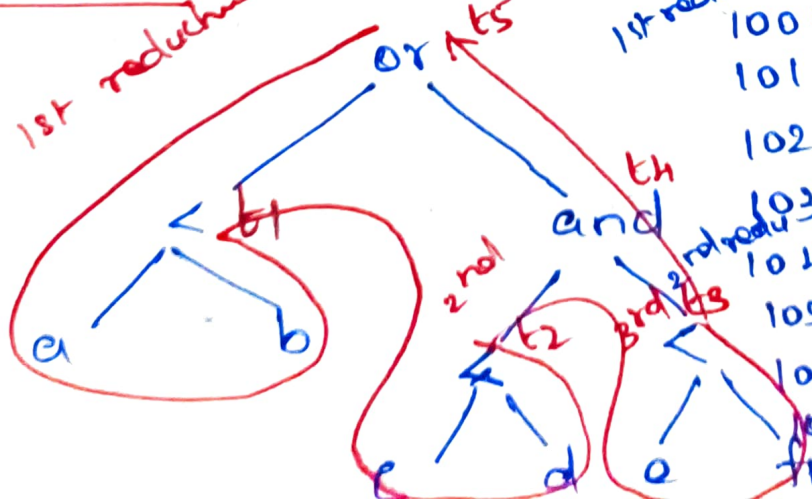
110 : goto 112

111 : t_3 = 1

112 : t_4 := t_2 and t_3

113 : t_5 := t_1 OR t_4

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



1st reduction

100 : if a < b goto 103

101 : t_1 = 0

102 : goto 104

103 : t_1 = 1

2nd reduction

104 : if c < d goto 107

105 : t_2 = 0

106 : goto 108

107 : t_2 = 1

3rd reduction

108 : if e < f goto 111

Switch case statement :

Syntax

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

→ Code to evaluate E (expr) into t (result)

goto 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 then goto L1
if t = V2 then goto L2
if t = Vn-1 then goto Ln-1
goto Ln

next : (break)

Types of translation.

★ L-attributed Translation.
- performs translation during parsing itself.
- no need of Explicit tree construction
- L represents Left to Right.

★ S-attributed translation.
- performs in connection with bottom up parsing.
- S represents Synthesized.

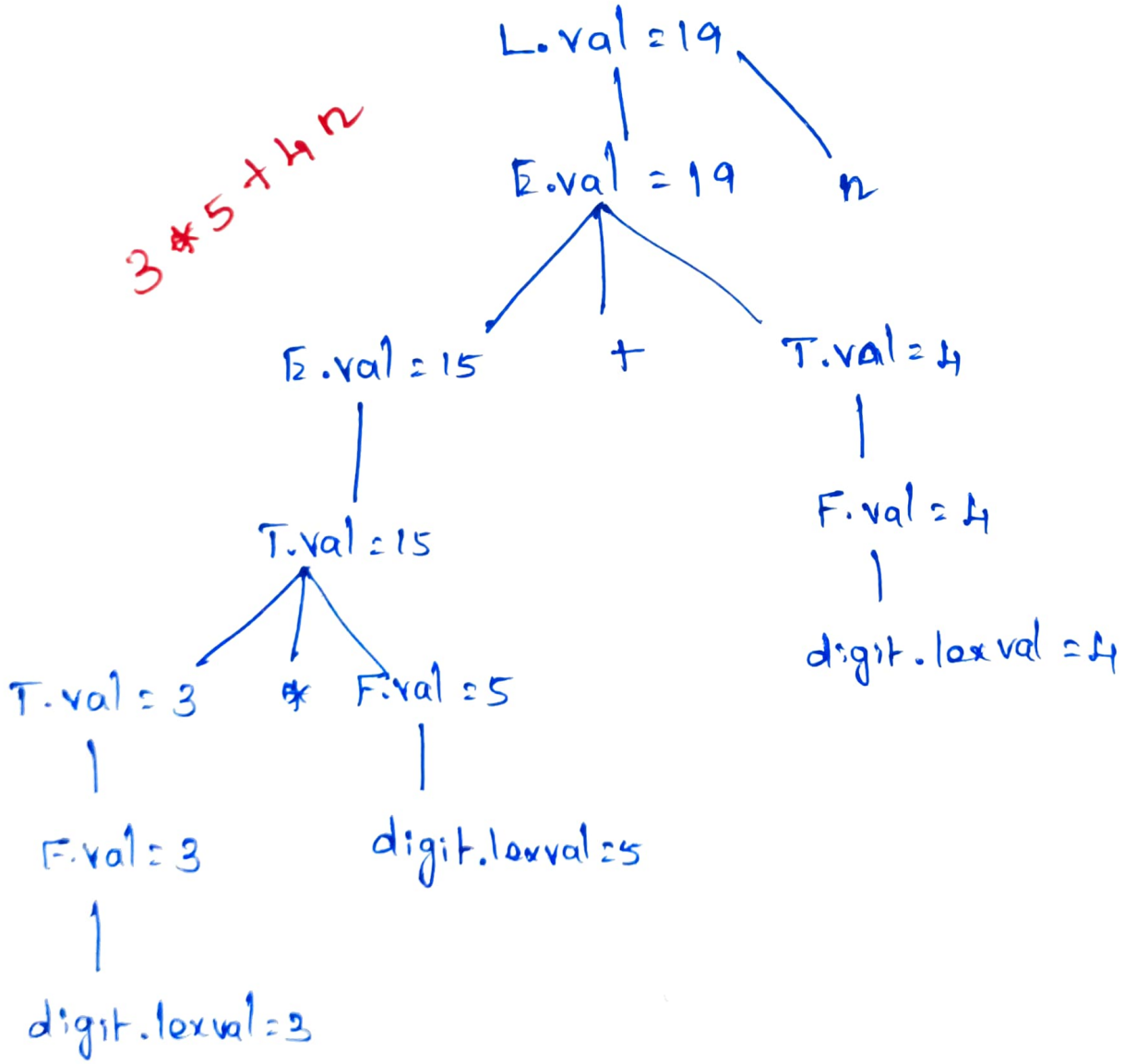
Types of Attributes:

★ Inherited attributes
- defined by semantic rule associated with production at the parent of node.
- attribute values are confined ~~by~~ to the parent of node, its siblings and by itself.
- NT concern must be in the body of the production

★ Synthesized attributes.
- denoted by semantic rules associated with the production at the node.
- attribute values are confined ~~with~~ to the children of node and by itself.
- NT concern must be in the head of production.
- T have synthesized attributes (lexical value).

Annotated Parse Tree.

- Translation of SDD can be viewed by constructing parse tree.
- A parse tree which displays the values of attributes at each node is called annotated parse Tree.
- The process of computing attribute value at the node is called annotated parse tree. (or) parse tree annotation.



Procedure call

Procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

Calling Sequence

The translation for a call includes a sequence of actions taken on entry and exit from each procedure.

$p(x_1, x_2, \ldots x_n)$ (procedure call statement)

p - Name of the procedure

procedure p must be called with 'n' parameters

n - No. of Actual parameters.

The 3 Address statement for the procedure call is

```
param x1
param x2
  :
param xn
call p, n
```

(param x_1 evaluated first, x_2 upto x_n)

This is the sequence of 3 Address statement generated by using the syntax directed translation

Whenever procedure call occurs some sequence of Actions must be done

eg. Consider C program statements

```
main ()  ——————> calling procedure
{
   :
   add(x,y); ——————> called procedure
   :
}
void add (int a, int b)
{
   :
}
```

1. The space must be allocated for the Activation record of the called procedure.
   - Activation record - It will have information about the procedures
   - eg: local variables in the procedure
     parameters " " "
     return address
   - All the Actions stored in the Activation record.
2. parameters must be evaluated
3. Return Address must be stored
4. Whenever the program is executed, the main function calling procedure is executed.
5. Whenever the procedure called occurs the main procedure is temporarily suspended

so, jump is performed to called procedure.

6. After jump is performed to the called procedure again the jump is performed to the calling procedure. Because the execution must be resumed at this point the return address stored.

7. Before jumping the back to the calling procedure some sequence of actions takes place.
- result of the called procedure must be stored in a known place
- the return address
- Activation record of the calling procedure restored because, After Activating the Activation record of calling procedure. Now jump is performed to the calling procedure.

These are the actions done when performing procedure call statement

SDT for producing 3 Address code for procedure calls

Grammar	Semantic rules
$S \rightarrow call\ id\ (\ Elist\ )$	{ for each item p on queue do emit ('param' p); emit ('call' id.place) }
$Elist \rightarrow Elist, E$	{append E.place to the end of queue
$Elist \rightarrow E$	{initialize queue to contain only E.place}

↓
Name of procedure.

elist → elist , E

- This is a recursive procedure for any no. of parameters we can use this rule.

- for evaluating the parameters, we are using the data structure 'queue'

→ elist → E

The parameter E must be evaluated and result stored into the queue

E.place represents value of 'E' that is stored in the queue

→ elist → elist, E

Ahead queue have some value

E.place append to the end of queue

→ S → call id (elist)

Here, we need to produce list of parameter statements and then call

The parameters stored in the queue

for each item p on queue do emit ('param' p

'param' x

'param' y

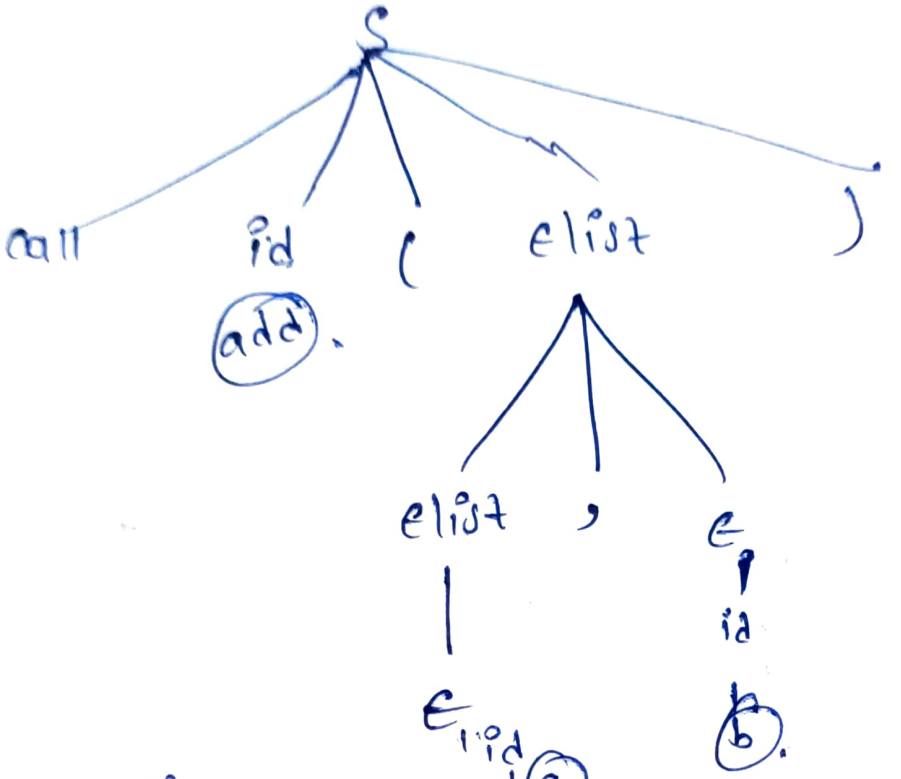
'param' z

:

call id.place (Name of procedure should be done)

PARSE TREE

Eg: add (a,b)



In Bottom up Approach, ~~reduct~~

queue

3 Address code

param a
param b
call add

SYMBOL TABLE GENERATION:

Symbol Table (ST) Important data structure

* Created and maintained by compiler to store the info about various entries such as variable names, function names, objects, classes, interfaces etc.
* It is used in both the analysis & synthesis parts.
* Analysis phase collects the info for symbol table.
* Synthesis phase uses that info to generate code.
* It is built in both lexical & syntax analysis phases.
* It is used by compiler to acheive compile time efficiency.

Use of Symbol Table in various phases of Compiler:

1. Lexical Analysis: Creates new table entries (token) in the table.
2. Syntax Analysis: Adds info about attribute type, scope, dimension, line of reference, use etc in the table.
3. Semantic Analysis: Checks for semantic errors (type checking) by using available info.
4. Intermediate Code Generation: To know how much and what type of run-time is allocated we use the symbol table in this phase.

Code Optimization: Uses info in symbol table for machine dependent optimization.

Target Code Generation: Generates code by using address info of identifier present in the table.

* A symbol table is simply a table that is either linear or a hash table.

* Maintains an entry for each name in the format "<Symbol Name, Type, Attribute>"

Example: Static int interest; → Variable declaration

Stores above in following format in symbol table

<interest, int, static>

Operations of Symbol Table:

1. Allocate: To allocate a new empty symbol table.

2. Free: To remove all entries and free the storage of ST.

3. Insert: Insert() function is used to insert name in a ST and return a pointer to its entry.

Example: int x;

the compiler process the above in the format of {insert(x,int)}

4. Lookup: Used to search for a name and return a pointer to its entry.

5. Set-Attribute: To associate an attribute with a given entry

6. Get-Attribute: To get the associated attribute with a given entry.

Implementation of ST:

* We use some of the data structures commonly to implement the ST. They are:

1. List
2. Linked List
3. Hash Table
4. Binary Search Tree (BST)
5. Scope Management.

1. List: Arrays are used to store names and its associated information.

* New names are added in the order as they arrive.
* For inserting new names, it must not already present or else it occurns an error as "Multiple defined name".
* For searching a name, we start from ~~searching~~ of beginning of list till Available pointer and if not found we get an error as- "use of undeclared name".
* "Available" pointer is used in List.

* Time complexity for insertion - O(1) - fast
" " Lookup - O(n) - slow for large table

* Advantage: Takes minimum amount of space.

2. Linked List:

* Informations are linked together in the form of list.
* A link field is added to each record.
* Time complexity for insertion - O(1) - fast
" " Lookup - O(n) - slow for large table.

3. Hash Table:

* It is an array with index range of table size 0 to -1.
* To search for a name we use hash function which results in integer b/w 0 to -1.
* ~~Inserti~~
* Time complexity for insertion & Lookup - O(1) - very fast
* Advantage: Search is possible
* Disadvantage: Hashing is complicated to implement.

4. Binary Search Tree (BST):

* We add 2 link fields i.e; left & right childs.
* All names are created as child of root node.
* Time complexity for insertion & Lookup - $O(\log_2 n)$.

5. Scope Management:

* We are having 2 types of symbol Tables.

1. Global Symbol Table: It can be accessed by all procedures.
2. Scope Symbol Table: Created for each scope in the program.

* To determine the scope of a name, symbol tables are arranged in hierarchical structure.

Example:

```
int value=10;
int sum_num()
 {
 int num_1;
 int num_2;
  {
  int num_3;
  int num_4;
  }
int num_5;
  {
  int num_6;
  int num_7;
}
```

```
int sum_id;
 {
 int id_1;
 int id_2;
  {
  int id_3;
  int id_4;
  }
int id_5;
}
```

We represents above example code in Hierarchical structure of ST.

Value	Var	int
Sum_num	Sum	int
Sum_id	Sum	int

→ Global Symbol Table.

Sum_num Symbol table

num_1	Var	int
num_2	Var	int
num_5	Var	int

Sum_id symbol table

id_1	Var	int
id_2	Var	int
id_5	var	int

Local Symbol tables.

num_3	Var	int
num_4	Var	int

inner Scope 1

num_6	Var	int
num_7	Var	int

inner scope 2

id_3	Var	int
id_4	Var	int

inner scope 3