* Syntax directed translation. (SDT)

Grammar + Semantic rules = SDT

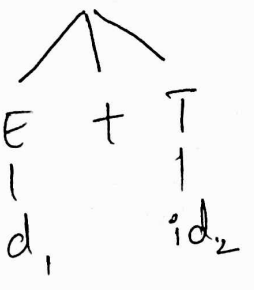
Syntax Directed Definition. (SDD)

SDD -> CFG + Semantic Rule

-> With every symbol present in grammar syntax Directed Definition associates a set of attributes

-> & with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production.

Production	Semantic Rule
$E \rightarrow E+T \mid id_1$	$E.val = E.val + T.val \mid E.val = id_1.lexval$
$T \rightarrow id_2$	$T.val = num.lexval$



E.val denotes that node x has an attribute called val

value of x.a is found using semantic rule for the x production used at the node

Annotated parse tree:- A parse tree with attribute values at each node, is annoted parse tree

* Different types of Attributes
  - Synthesized attribute
  - Inherited attribute

Synthesized Attributes :-
parent takes the value from its children

Ex:- A → BCD
- A → parent node
- BCD → children node

A.s = B.s → attribute associated with B
A.S = C.S
A.S = D.S
→ Parent node A can have value from its children B, C, D

* Inherited Attribute :-
If a node takes value from its parent/sibling node

A → BCD
C.S → A.S — parent node
C.S → B.S — sibling node
C.S → D.S — sibling node

* Types of SDD:-

↳ S-Attributed SDD | S-Attribute definition | S-attribute grammar

↳ L-Attributed SDD | L-Attribute definition | L-attribute grammar

S. Attribute SDD

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

$A \rightarrow BCD$

$A.S = B.S$

$A.S = C.S$

$A.S = D.S$

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

3) Attribute are evaluated with Bottom-up parsing.

L-Attribute SDD

A SDD that uses both synthesized & Inherited attribute is called as L-attributed SDD, But each Inherited attribute is restricted to inherits from parent | left sibling only.

$A \rightarrow XYZ$

$Y.S = A.S$ ✓

$Y.S = Z.S$ ✓

$Y.S = X.Y$ —X

semantic Action are placed anywhere on RHS

Attribute are evaluated by traversing parse tree depth first, left to right order

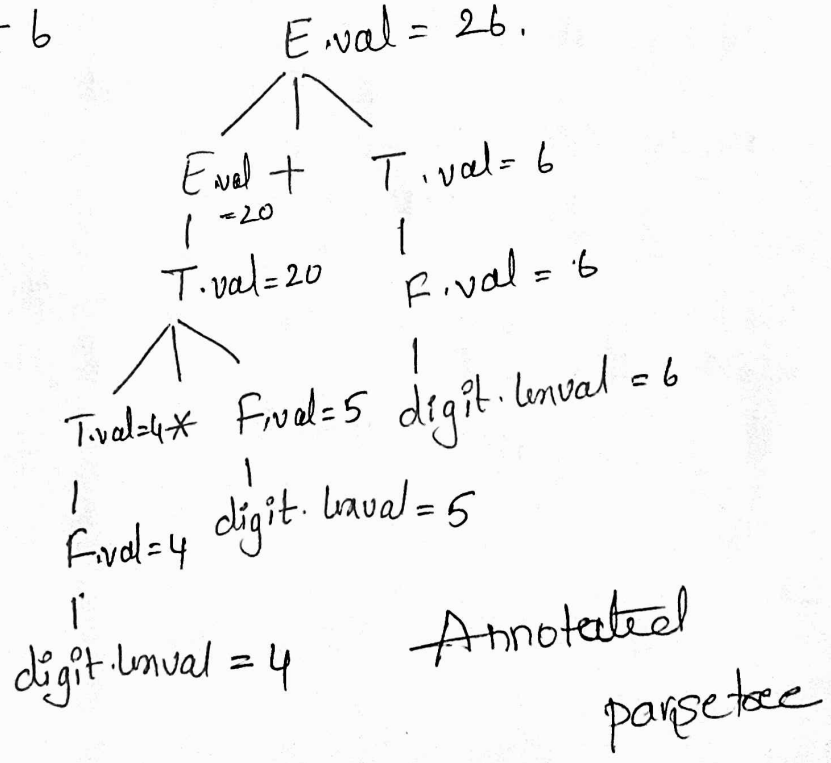
Evaluation orders for SDD's

Dependency graph represent the flow of inform" among the attributes in a parse tree

- Dependingy graphs are uyful for determing Evaluation order for attributes in a parse tree
- While an Annotated parse tree shows the values of attibutes / dependency graph determines how those values can be computed

production	semantic Actions
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lexval.$

i/p string 4 * 5 + 6



Annotated parsetree

Example (Synthesized Attributes)

Let's Consider an Example of arithematic Expression and then you will see how SDT will be Constructed.

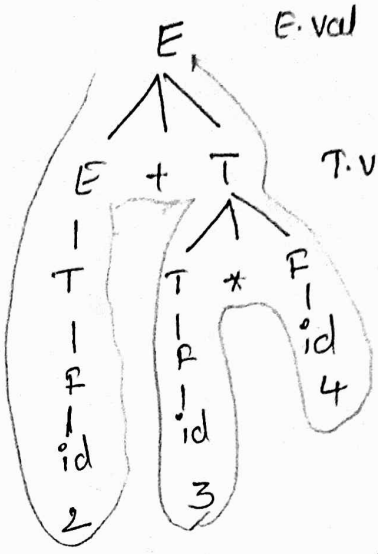
Input : 2+3*4

output : 14

E ⇒ E+T | T

T → T*F | F

F → id.



E.val = E.val + T.val =
2 + 12 = 14.

T.val = T.val * F.val
3 * 4
12

E → E+T { E.val = E.val + T.val then print(E.val) }
/T { E.val = T.val

T → T*F { T.val = T.val * F.val }
/F { T.val = F.val }

F → id { F.val = id }

Example (Inherited Attributes)

The annotated parse tree is generated and attributes values are Computed in top down manner

The Grammar

S → TL

T → int

T → float

T → double

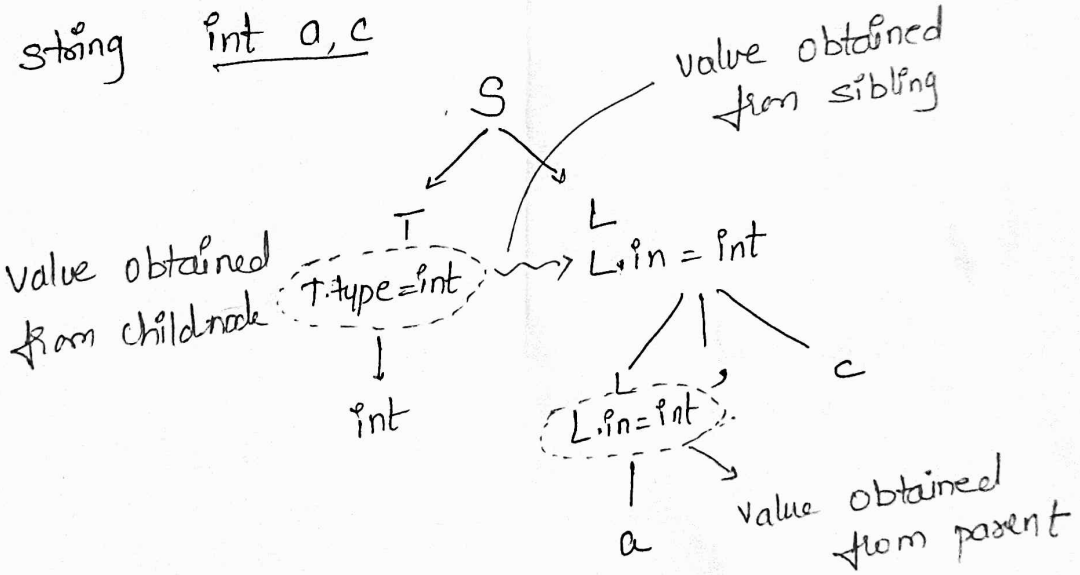
L → L1, id

L → id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
$S \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$T \rightarrow double$	$T.type = double$
$L \rightarrow L_1, id$	$L_1.in = L.in$ Enter_type (id.entry, L.in)
$L \rightarrow id$	Entry_type (id.entry, L.in)

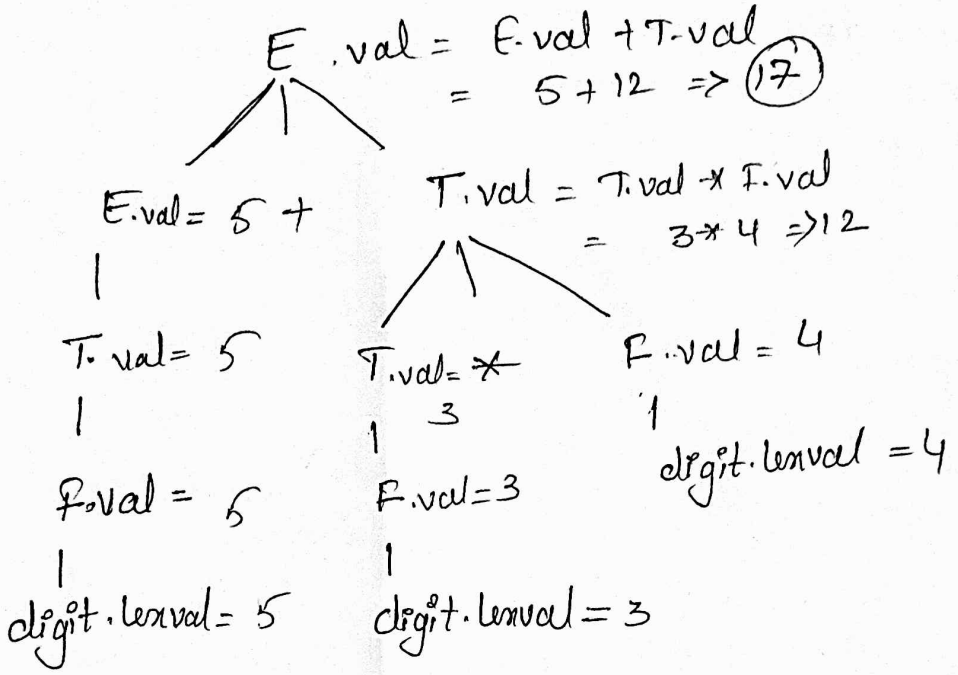
i/p string int a, c



Example Grammar production

	Semantic Action / Rule
$E \rightarrow E+T \mid T$	$E.val = E.val + T.val \mid E.val = T.val$
$T \rightarrow T*F \mid F$	$T.val = T.val * F.val \mid T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lexval$

input string 5+3*4



* Application of Syntax Directed Translation

Syntax Directed Translation

It is used for semantic analysis and SDT is basically used to construct the parse tree with Grammar and semantic action. In Grammar, need to decide who has the highest priority will be done first and In semantic action, will decide what type of action done by grammar.

Example

SDT = Grammar + semantic Action

Grammar = $E \rightarrow E_1 + E_2$

Semantic action = if (E_1.type != E_2.type) then print "type mismatching"

* Application of syntax Directed Translation:

→ SDT is used for Executing Arithmetic Expression

→ In the Conversion from infix to postfix Expression

→ In the Conversion from infix to prefix

→ It is also used for Binary to decimal Conversion

→ In Counting number of Reduction

→ In creating a syntax tree

→ SDT is used to generate intermediate code

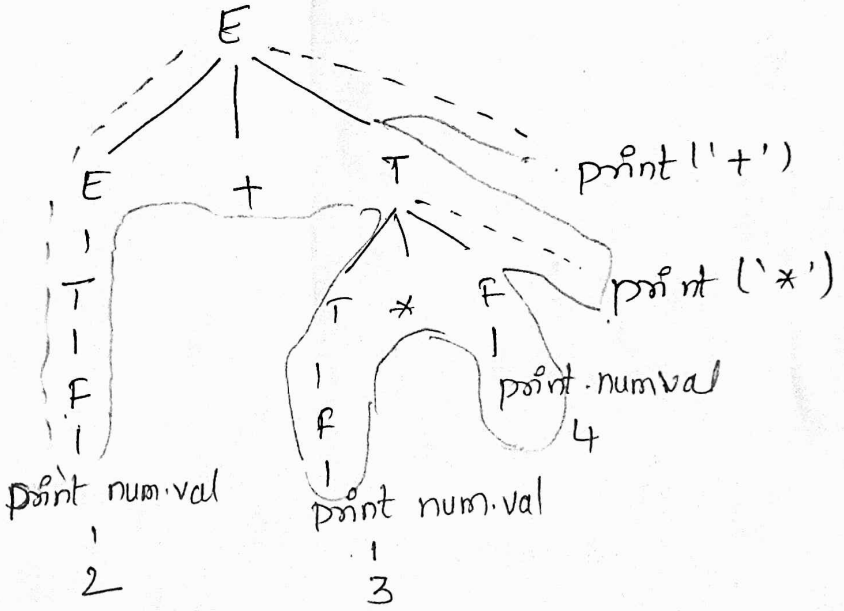
→ In storing information into symbol table

→ SDT is Commonly used for type checking also

* Construct Infix to postfix Expression (2+3*4)

production	semantic rules
E -> E+T	{ printf ('+'); }
E -> T	-
T -> T* F	{ printf ('*'); }
T -> F	-
F -> num	{ printf (%'num.val%); }

Method-I



o/p : 2 3 4 * +

Method-II Convert infix to postfix 1+2+3

E -> E+T { print('+'); } | T

T -> num { print num.val; }

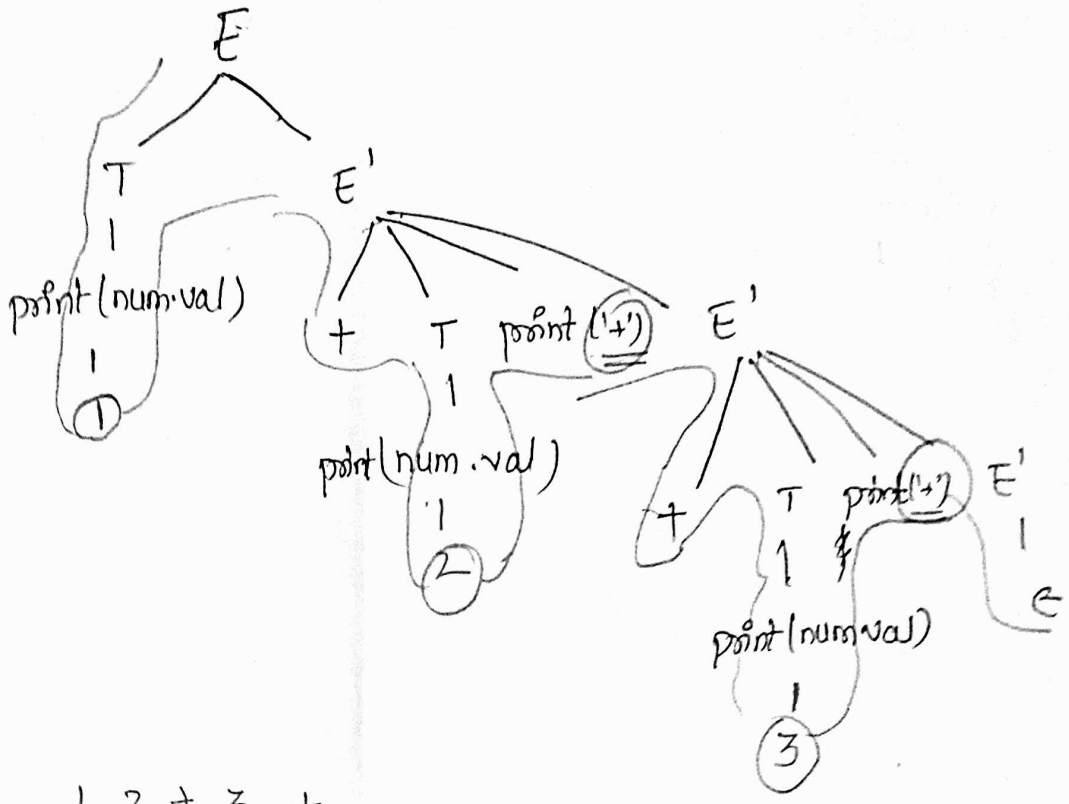
-> Grammar having left Recursion

E -> TE'

E' -> +T { print ('+'); } E'

E' -> ∈

T -> num { print (num.value; }



o/p:- 1 2 + 3 +

** Symbol Table :-

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variables, names, function names, objects, classes, interfaces, etc. symbol table is used by both the analysis and the synthesis parts of a compiler.

→ It is built in lexical and syntax analysis phases

→ The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code

→ It is used by compiler to achieve compile time efficiency

-> It is used by various phases of Compiler as follows :-

1. lexical Analysis :- Creates new table entries in the table, Example like entries about token.
2. Syntax Analysis :- Adds information regarding attribute type, scope, dimension, line of reference, use etc in the table.
3. Semantic Analysis :- Use refers symbol table for type Conflict issue.
4. Intermediate Code Generation :- refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information
5. Code optimization :- uses information present in symbol table for machine depend optimization
6. Target Code generation :- Generates code by using address information of identifier present in the table.

* Symbol table entries :- Each entry in symbol table is associated with attributes that support Compiler in different phases.

⇒ Items stored in symbol table

* Variable names and constants
* Procedure and function names
* Literal constants and strings
* Compiler generates temporaries
* Labels in source languages

⇒ Information used by Compiler from symbol table

* Data type and name
* Declaring procedures
* Offset in storage
* If structure or record then pointer to structure table
* For parameters, whether parameter passing by value / by reference
* Number and type of arguments passed to fn
* Base Address

** operations in symbol table

Example :- Format of symbol table :-

Compiler uses following type of information in Symbol. 1. Data type , 2 Name 3 scope 4. Address 5 : other attributes

Ex :- static int a;
float int b;

S.N	Name	type	Attribute
1	a	int	static
2	b	float	—

* Symbol table Representation:-

↳ Fixed - length Name

↳ Variable - length Name

Ex:- int Calculate ;
int sum ;
int a, b;

1) Fixed length Name:- A fixed space for each name is allocated in Symtable. In this space of storage if name is too small then there is wastage of space

Name										type
c	a	l	c	u	l	a	t	e		int
s	u	m								int
a										int
b										int

fixed length Symbol table

2) Variable - length Name:- The amount of space required by string is used to store the names The name can be stored with the help of starting index and length of each name

Name		type
Starting Index	length	
0	10	int
10	4	int
14	2	int
16	2	int

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c	a	l	c	u	l	a	t	e	$	s	u	m	$	a	$	b	$

Operation on symbol table

1. Insert : insert (a , int)
2. look up lookup (symbol) : lookup (a)
3. Delete delete (symbol) : delete (a)
4. Scope mgnt local & Global variable

1. Insert :- This operation is more frequently used by analysis phase. i.e, the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about the unique names occurring in the source code.

→ An attribute for a symbol in the source code is the information associated with that symbol.

→ This information contains the value, state, scope and type about the symbol.

for int a;
should be processed by the compiler as
insert (a, int)

2 lookup()

Lookup() operation is used to search a name in symbol to determine

* if the symbol exists in the table.
* if it is declared before it is being used.
* if the name is used in the scope
* if the symbol is initialized.
* if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following.

lookup (symbol)

→ This method returns 0 (zero) if the symbol does not exist in the symbol table.

→ If the symbol exists in the symbol table, it returns its attributes stored in the table.

3. delelete :- It is used when we want to delete any symbol.

delete (symbol))

4. Scope management :-

A Compiler maintains two types of symbol tables:- a global symbol table which can be accessed by all the procedures and scope symbol tables, that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown

```
int value = 10;   -> Global scope.

void fn1()
  {
    int one_1;
    int one_2;
      {
        int one_3;   -> inner scope
        int one_4;
      }
  }
```

** Implementation of symbol table

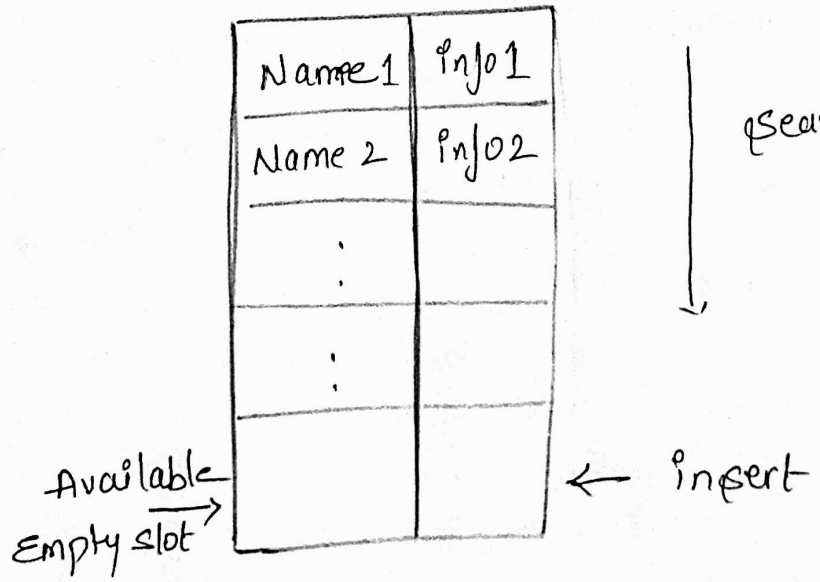
A block of structured language is a kind of language in which section of source code is within some matching pair of delimiters such as "{" and "}" / begin and end. and such section executed as one unit one or one procedure or a function or it may be controlled by some conditional statement (if, while, do while) Normally block structured languages support structured programming approach.

Following are data structures that are used organiza-
- tion of block structured languages.

* linear list
* self organizing list
* trees
* Hashing.

* Linear list

- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information
- New names can be added in the order as they arrive
- The pointer 'avaible' is maintained at the end of all stored records.



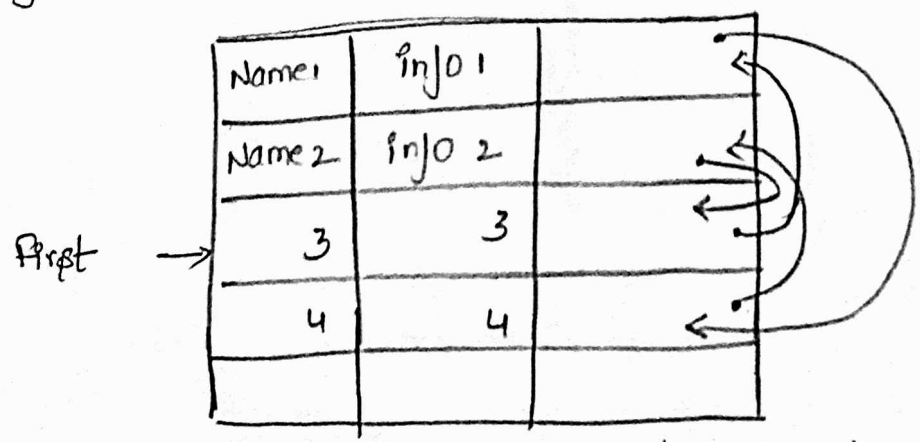
- To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach search at pointer available without finding a name we get an error "use of undeclared name"

- while inserting a new name we should ensure that it should not be already there. If it is already present there then another error occurs i.e. multiple defined name.

- The advantage of this is that it takes minimum amount of space.

~~&~~ Self organizing list

- This symbol table implementation is using linked list.
  A link field is added to each record
- we search the records in the order pointed by the link of the link field.
- A pointer "First" is maintained to point to first record of the symbol table.



The reference to these names can be Name 3, 4, 2, 1

- When the name is referenced / created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

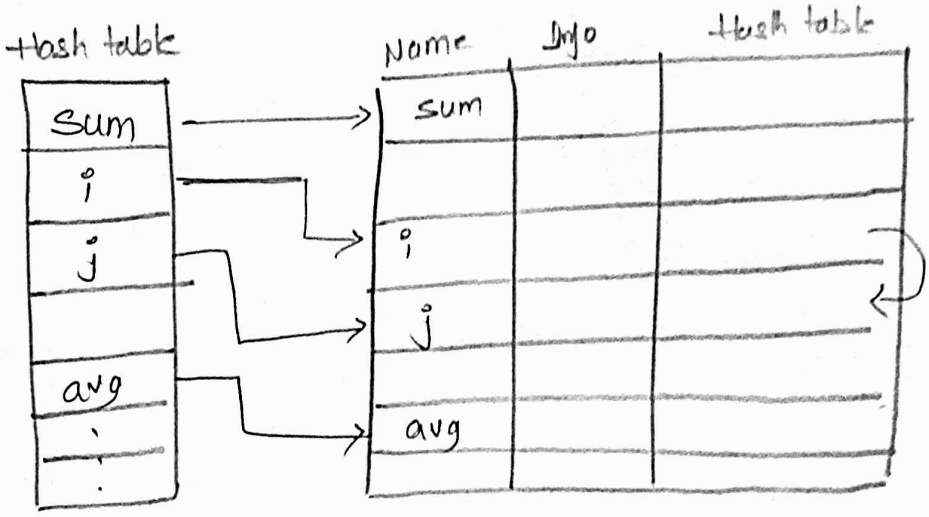
** Hashing

- Hashing is an important technique used to search the records of symbol table. This method is superior to list organization
- In hasing scheme two tables are maintained a hash table and symbol table.
- The hash table consists of k entries from 0, 1, to k-1 These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'name' is in symbol table, we use a hash function 'h' such that h(name) will result any integer between 0 to k-1. we can search any name by.

  position = h(name). - using this position we can obtain the exact locations of name in S.T

-> The hash

- The hash table and symbol table can be as shown



Hashing for symbol table

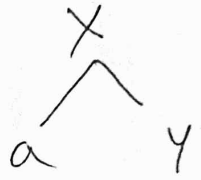
- The hash function should result in uniform distribution of names in symbol table.
- The hash function should be such that there will be minimum number of collision. Collision is such a situation where hash function results in same local for storing the names.
- Various collision resolution techniques are open addressing, chaining, rehashing.
- The advantage of hasing is quick search is possible and this disadvantage is that hashing is complicated to implement

* Tree (Binary tree)

When the scope information is presented in hierarchial manner then it forms a tree structure representation. To overcome the drawback of sequential representation being slow in searching the desired identifies, this representation is been designed.

A binary tree is build using lexicographic ordering of tokens

Ex int a, X, Y



left child	Name	more Infor	Right child

* ~~Management~~ of ST

Scope management

A compiler maintains two types of symbol tables: a global symbol table which can be accessed by all the procedures and scope symbol tables that are created for each scope in the program.

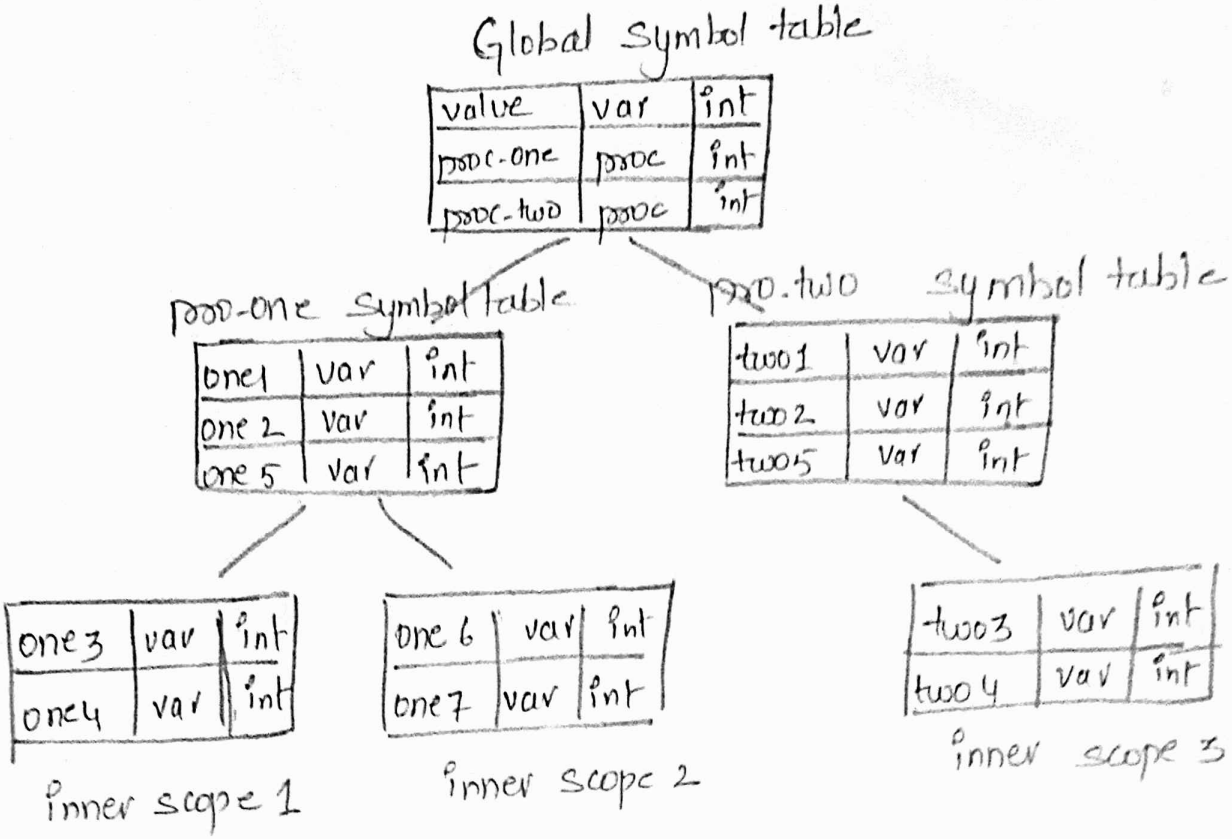
To determine the scope of a name, symbol tables are arranged in hierarchial structure as shown

```
int value = 10;

void pro_one()
  {
   int one1;
   int one2;
    {
     int one3;   } inner scope 1
     int one4;
    }
   int one5;
    {
     int one6;   } inner scope 2
     int one7;
    }
  }

void pro_two()
  {
    int two1;
    int two2;
     {
      int two3;   } - inner scope 3
      int two4;
     }
    int two5;
  }
```

The above program can be represented in a hierarchial structure of symbol table.



→ The global symbol table contains names for one global variable int value. and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the proo-one symbol table and all its children are not available for proo-two symbols and its child tables.

→ This symbol tables data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a ST, it is searched using the following algorithm.

- First a symbol will be searched in the current scope, i.e current symbol table
- If a name is found, then search is completed, else it will be searched in the parent symbol table
- either the name found / global ST has been searched for name