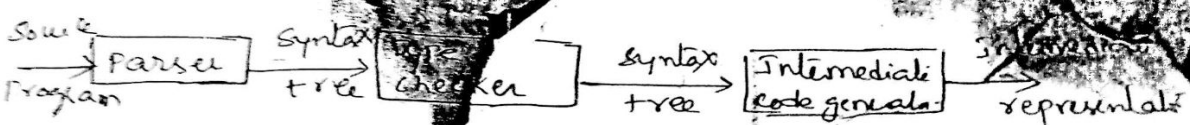
V

Type Checking

- Type checking is an important activity done in the semantic analysis phase. The need for type checking is
  - To detect errors arising in the expression due to incompatible operand
  - To generate intermediate code for expression and statements. Typically language supports two types
    1. Basic : int, char, float
    2. constructed : Array, structure and pointer types



- Type checker verifies that the type of a construct matches that expected by its context.

Type expression :- The type of a language construct will be denoted by type expression.

- The systematic way of expressing type of language construct is called type expression. Thus the type expression can be defined as

1. The basic type is called type expression. Hence int, char, float, enum are type expression.
2. While performing typechecking two special basic types are needed such as type_error and void

(a) type-error is for reporting error occured during type check

(b) void indicates that there is no type (NULL) associated with statements.

(iv) Pointer :- type expression for pointer is given as Pointer(T) where T is data type.

Ex:- float *A.

The type expression for identifier A is given as Pointer(float)

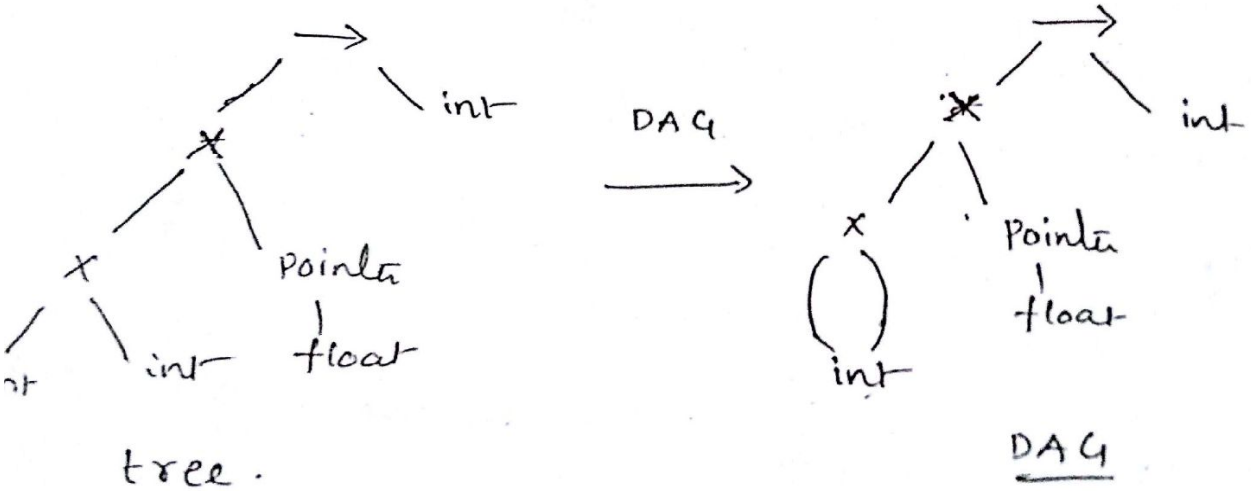
(v) Function :- A function maps elements of one set, the domain to another set range. So type expression for function is given by domain-range. So type expression for function is $D \rightarrow R$.

Ex:- int Sum(int a, int b)

type expression for Sum is int x int → int

· type expression can be represented using tree (or) DAG

Ex:- int max(int a, int b, float *c)



tree.

DAG

Type System :- is a collection of rules for assigning type expressions to the language constructs. The type system is implemented by type checker.

* Type conversion :- Process of converting one type to another.

two types of conversions 1. Implicit conversion
2. explicit conversion.

(i) Implicit Conversion (or) Coercions :- Process of converting one type to another automatically by the compiler then it is called Coercions.

In this type of conversion there is no loss of data when integer can be converted to float. but there is a loss from real to integer.

(ii) explicit conversions: The conversion is said to be be explicit if Proglammer specially writes some thing for converting one type to another.

Ex: int xyz, p.
p = float (xyz);

identifier xyz is type casted and this is how explicit Conversion from int to float takesplace.

- typechecking rules for coercion from integer to float are as given bellow.

duce e
that
zati
hould
t

Production	Semantic rule
$E \rightarrow num$	E.type := integer
$E \rightarrow num.num$	E.type := real
$E \rightarrow id$	E.type := lookup(id.entry)
$E \rightarrow E_1$ op E_2	E.type := if E_1.type = int and E_2.type = int then integer elseif E_1.type = int and E_2.type = real then real elseif E_1.type = real and E_2.type = int then real elseif E_1.type = real and E_2.type = real then real else type-error.

Specification of Simple type checker.

- We specify a type checker for simple language in which the type of each identifier must be declared before the identifier is used.
- Type checker is a translation scheme in which type of each expression from the types of subexpressions is obtained. Type checker can handle arrays, pointers, statements and functions.

Type checker ensure following things:

(i) Each identifier must be declared before the use.
(ii) The use of identifier must be within the scope
(iii) An identifier must not have multiple dimensions at a time with in the same scope.

Write a translation scheme for grammar.

$S \rightarrow D; E$	means all declarations before expression
$D \xrightarrow{T} LIST$	{ LIST.type := T.type }
$LIST \rightarrow id$	{ addtype (id.entry, LIST.type) }
$T \rightarrow char$	{ T.type := char }
$T \rightarrow float$	{ T.type := float }
$T \rightarrow int$	{ T.type := int }
$LIST \rightarrow * L$	{ L_1.type := Pointer (LIST.type) }
$LIST \rightarrow L_1[num]$	{ L_1.type := array (0..num.val-1, LIST.type) }

Type checking of Expression

- The Synthesized attribute type for E gives the type expression assigned by the type system to the expression generated by E.
- Following Semantic rules says that Constant represented by tokens literal and num have type char and integer respectively.

$E \rightarrow$ literal { E.type := char }

$E \rightarrow$ num { E.type := integer }

- we use function lookup (id.entry) to fetch the id type in the symbol table and assigned to the attribute type

$E \rightarrow id$ { E.type := lookup (id.entry) }

$E \rightarrow E_1$ mod E_2 { E.type := if E_1.type = int and E_2.type = int then int else type_error }

- for array reference

$E \rightarrow E_1 [E_2]$ { E.type := if E_2.type := int and E_1.type := array(s,t)
then t else
type-error }

Here s is set of index and t is the datatype of array

Hence type returned for expression E will be t.

$E \rightarrow *E_1$ { E.type := if E_1.type = pointer(t) then t else type-error }

Type checking of statements

- Generally language constructs like statements do not have values. and therefore the special type void is used.
- Type-error is the basic type used to raise error signal

translation scheme for checking the type of statement

Production	semantic rule
$S \rightarrow id := E$	{ S.type := if id.type = E.type then void else type-error }
$S \rightarrow if (E) S_1$	{ S.type := E.type := boolean(true) then S_1.type else type-error }
$S \rightarrow while (E) S_1$	{ S.type := if E.type := boolean(true) then S_1.type else type-error }
$S \rightarrow S_1 ; S_2$	{ S.type := if S_1.type := void and S_2.type = void then void else type-error }

* EQUIVALENCE OF TYPE EXPRESSIONS

- In type checking when two expressions have the identical type expressions then return certain type else return type-error.
- It is necessary to check whether two expressions posses the same type.
- Hence checking type equivalence in the type checking expression is an important task.

This type equivalence is of two categories:

1. Structural equivalence 2. Name equivalence

Structural equivalence of type expressions

- Two expressions are either the same basic type, (or) are formed by applying the same Constructor to structurally equivalent types.
- i.e two type expressions are structurally equivalent if and only if they are identical.

Ex:- type expression integer is equivalent to integer only.

ALGORITHM:- checking structural equivalence of two type expression

```
(1) function Sequiv(s,t) : boolean
    begin
(2) if s and t are same basic type then
(3)   return true
(4) else if s = array(s1, s2) and t = array(t1, t2) then
(5)   return Sequiv (s1, t1) and Sequiv (s2, t2)
(6) else if s = s1 x s2 and t = t1 x t2 then
(7)   return Sequiv (s1, t1) and Sequiv (s2, t2)
(8) else if s = Pointer (s1) and t = Pointer (t1) then
(9)     return Sequiv (s1, t1)
(10) else if s = s1 → s2 and t = t1 → t2 then
(11)   return Sequiv (s1, t1) and Sequiv (s2, t2)
       else
(12)   return false
    end.
```

Name equivalence for type expressions

- Two type expressions are said to be name equivalent if and only if they are identical.
- In the name equivalence the type expressions are given the names.

EX:-

```
type def Struct node
{ int x ;
} node ;
node * first , * second
Struct node * last1 , * last2 ;
```

- In the above first and second are name equivalent and last1 and last2 are name equivalent.
- But first and last1 are not name equivalent. Because names for these two type expressions are different.
- Although first, second, last1, last2 are similar under structural equivalence since they are representing the same type pointer ('Struct Node').

Chomsky hierarchy of languages and Recognizers

- The chomsky hierarchy represents the class of language that are accepted by different machine.

Language class	Language	Grammar	Machine	example
Type 3	Regular Language	Regular Grammar	FSM i.e NFA (or) DFA	$a^* b^*$
Type 2	Context free Language	Context free Grammar	PDA	$a^n b^n$
Type 1	Decidable language	Context Sensitive Grammar	Linear bounded automata	$a^n b^n c^n$
Type 0	Computable languages	Unrestricted grammar	Turing m/c	$n!$

- This is hierarchy therefore every language of type 3 is also type 2, type 1 and 0. Similarly every language of type 2 is also of type 1 and type 0 etc.

Type 3 - Regular languages

Regular languages are those languages which can be described using regular expressions. These languages can be modeled by NFA & DFA.

Type 2 - Context free language: which can be represented by context free grammar (CFG). The production rule is of the form $A \rightarrow \alpha$

$A \rightarrow$ any single Non-terminal

$\alpha \rightarrow$ any combination of terminals and non terminals

Type 1 :- Context Sensitive languages

The context sensitive grammars are used to represents context sensitive languages. C.S.G is follows the the following rules

1). Context Sensitive grammars may have more than one symbol the left hand side of their Production rule

2) The number of symbols on the left hand side must not exceed the number of symbols on the right hand side.

3) Rule of the form $A \rightarrow \epsilon$ is not allowed unless A is a start symbol. It does not occure on the right hand side of any rule.

Linear bounded automata

4) Type 0 - unrestricted languages:

There is no restriction on the grammar rules of the type of languges. These languges can be effectively modeled by Turing machine

UNIT - V

STORAGE ORGANISATION

Runtime Storage:-

The compiler demands the block of memory from the operating system. So, this block of memory utilizes by the compiler. So this block or amount of memory is called as Runtime storage.

(or)

The compiler utilizes block of memory running the compiled program is called as 'Runtime storage'.

Language Issues:-

1) Does the programming language supports recursion or does the source language allows recursion.

While handling the recursive calls there may several instances of recursive procedures that are acti simultaneously. So, memory allocation must be needed each and every instance, with its copy of local variables and parameters, passed through that recur procedure.

2) How the parameters are passed to the procedure

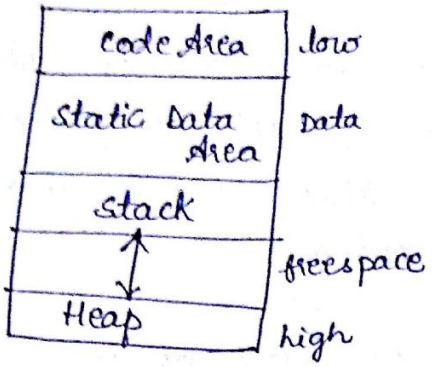
Parameter passing methods call by value and call by Reference

- The allocation strategy for each method is differe Some languages supports passing parameters as procedure and return also procedure.

3) Does the procedure refer non local names? How?

4) Does language supports memory allocation stati or dynamically?

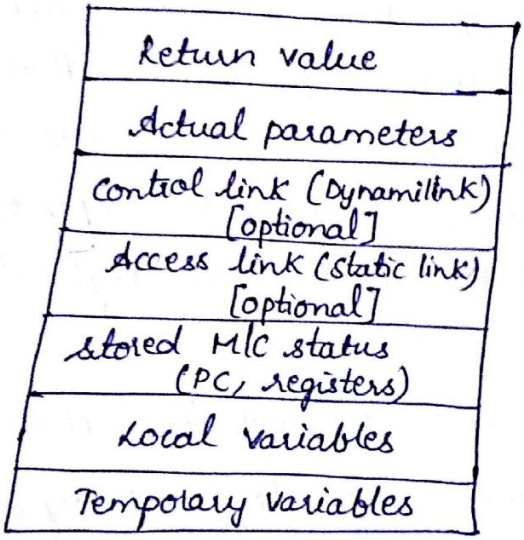
d) Does language supports memory allocation statically or dynamically?



(Run time storage organisation)

Activation Record:-

The block of memory is used to manage the information needed by single execution of procedure. The activation record contains some fields as follows:-



- Fortran uses the static data area to store the activation record, whereas in PASCAL and C, the activation record is situated in stack area.
- The size of each field of activation record is determined at the time when a procedure is called.

) Temporary values:-

The temporary variables are needed

during the evaluation of expressions. Such variables are stored in the temporary field of activation record.

2) Local variables:-

The local data is data i.e., local to execution, of procedure is stored in this field of activation record.

3) Saved machine registers:-

This field holds information regarding the status of machine just before procedure is called. This field contains machine registers and program counter.

~~4) Program counter:-~~

4) Control link:-

This field is optional. It points to activation record of calling procedure. This link is also called dynamic link.

5) Access link:-

This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.

6) Actual parameters:-

This fields holds information about actual param These actual parameters are passed to called procedu

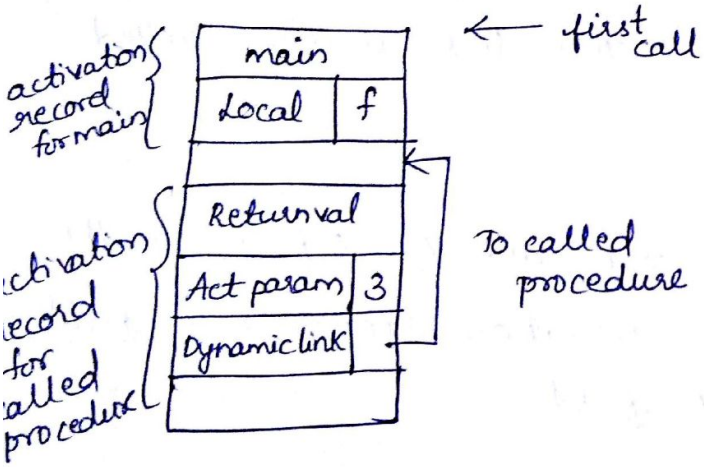
Return values:-

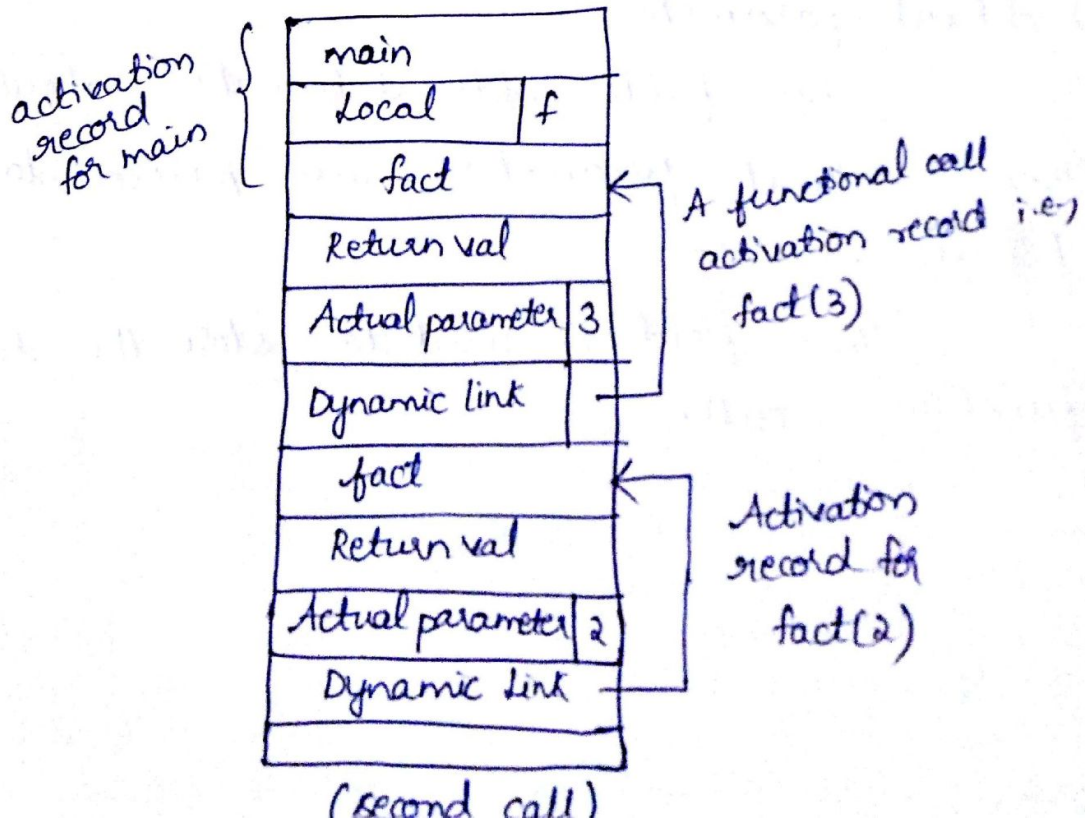
This field is used to store the result of a function call.

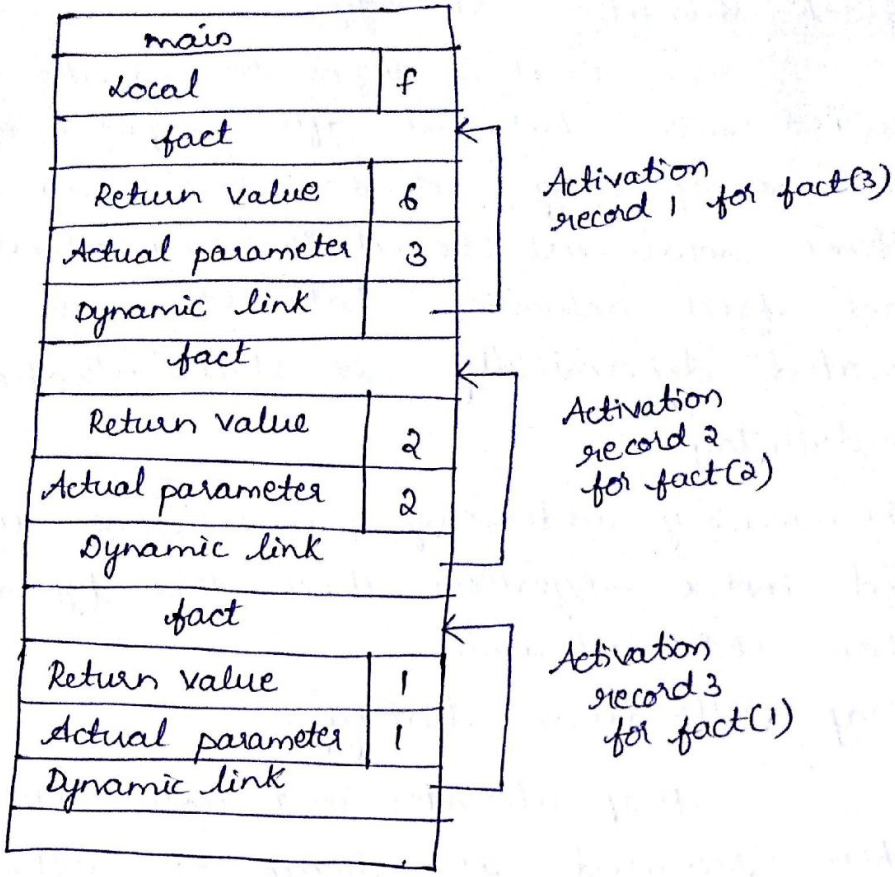
Ex:- Calculate factorial program, how the activation record will look like every recursive call of factorial 3.

Sol:-

```
main()
{
int f;
 f= fact(3);
}
int fact(int n)
{ if (n==1)
  return 1;
  else
  return (n*fact(n-1));
}
```







(Third call)

Storage Allocation Strategies:-

1) Static Allocation strategy:-

Some of the data objects in program may know at compile time, so these data objects allocated memory with help of static allocation strategy Now the compiler allocates a memory to data objects of at compile time, by using static allocation strategy. FORTRAN uses static allocation strategies.

Disadvantages:-

Recursive procedures are not supported by this type of allocation.

2) Stack Allocation strategy:-

As activation begins the activation records are pushed into stack and after completion of this activation, the corresponding activation records can be pop.

- Hence locals are bound to each activation record, on each fresh activation. Data Structures can be created dynamically for stack allocation.

Disadvantage:-

- The memory addressing can be done using pointers and index registers. Hence this type of allocation is than static allocation.

3) Heap Allocation Strategy:-

Heap allocates continuous block of memory when required for storage of activation record or other data object. This allocated memory can be deallocated when activation is end. This deallocated space can be reuse by heap manager. The efficient heap management can be done by

- Creating a linked list for free blocks and when any memory is deallocated that block of memory is append to the linked list.
- Allocate the most suitable block of memory from the linked list i.e. use best-fit technique for allocation of block.

Block Structured and Non Block Structure Storage Allocation

The storage allocation can be done for 2 types of data variables. 1) Local data 2) Non-local data.

- The local data can be handled using activation record whereas non local data can be handled using scope, information.
- The 'block structured storage' allocation can be done using 'static scope' (or) 'lexical scope' and the non-block structured storage allocation can be done using 'dynamic scope'.

Local Data :-

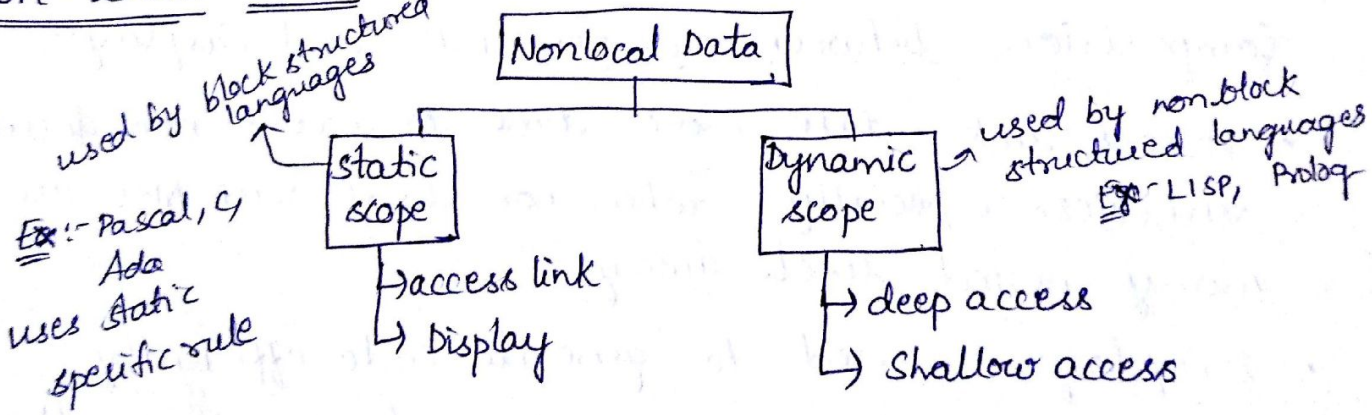
The local data can be accessed with help of activation record. The offset relative to base pointer of an activation record points to local data variables within an activation record. Hence,

Reference to any variable 'x' in procedure

= Base pointer pointing to start of procedure + offset of variable 'x' from base pointer.

access to local data x := base address + offset address refers to local variable

Non-local Data :-



Static scope rule :- (Lexical scope)

Static scope is applied on block and neste procedures. Lexical scope can be implemented using ac link and displays.

1) Access link :-

Implementation of lexical scope can be

obtained by using pointer to each activation record. These pointers are called access links. If procedure 'p' is nested within a procedure 'q' then access link of 'p' points to access link of most recent activation record of procedure 'q'.

2) Displays:-

It is expensive to traverse down access link every time when a particular non local variable is accessed. To speed up access to non locals can be achieved by maintaining an array of pointers called display. In display,

a.) An array of pointers to activation record is maintained.

b.) Array is indexed by nesting level

c.) The pointers point to only accessible activation record

d.) The display changes when a new activation occurs and it must be reset when control returns from the new activation.

Comparision between access link and display:-

- Access link take more time to access non-local variables, especially when non local variables are at many nested levels away.
- Display is used to generate code efficiently.
- Display requires more space at run time than access links.

Dynamic scope rule:-

It determines the scope of declaration of names at run time by considering the current activa

- In dynamic scoping a use of non-local variable refers to non-local data declared in most recently called and still alive procedure.
- Therefore each time new bindings are setup for local names called procedure. In dynamic scoping symbol, tables can be required at run time.
- There are 2 ways to implement non local accessing under dynamic scoping.

i) Deep access:-

The idea to keep a stack of active variables, use control links instead of access links and when we want to find variable then search stack from top to bottom, looking for most recent activation record, that contains space for desired variables. This method of accessing non local variables is called deep access. In this method a symbol table needs to be used at run time.

ii) Shallow access:-

The idea is to keep a central storage with one slot for every variable name. If names are not created at run time then that storage layout can be fixed at compile time otherwise when new allocation of procedure occurs, then that procedure changes the storage entries, for its locals of entry and exit (i.e, while entering in the procedure and while leaving the procedure).

Comparision of Deep and Shallow access:-

- Deep access takes longer time to access non locals while shallow access allows fast access.
- Shallow access has overhead of handling procedure entry & ex
- Deep access needs a symbol table at run time.

Parameter Passing Methods:-

There are 2 types of parameters

i.) Formal parameters ii.) Actual parameters.

Based on these parameters there are various parameter passing methods, the most common methods are

1.) Call by Value:-

The actual parameters are evaluated and their r-values are passed to called procedure. The operations on formal parameters do not change values of actual parameter.

Ex:- Languages like C, C++ use actual parameter passing method.

2) Call by reference :-

This method is also called as 'call by address', or 'call by location'. The L-value, the address of actual parameter is passed to called routines activation record. The values of actual parameters can be changed.

Ex:- Reference parameters in C++, PASCAL's var parameter.

3) Copy - restore:-

This method is hybrid between call by value and call by reference. This method is also known as copy-in - copy-out or values result. The calling procedure calculates value of actual parameter and it is then copied to activation record for called procedure. During execution of called procedure, the actual parameter value is not affected. If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

Ex:- In Ada, this parameter passing method is used.

4) Call by name:-

This is less popular method of parameter passing. Procedure is treated like macro, the procedure body is substituted for call in caller with actual parameters substituted for formals. The actual parameters can be surrounded by paranthesis to preserve their integrity. The local names of called procedure and names of calling procedure are distinct.

Ex:- ALGOL uses call by name method.

Symbol Table:-

Symbol table is a data structure used by compiler to keep track of semantics of variable, means symbol table stores information about scope and binding information about names.

- Symbol table is ~~used by~~ built on lexical and syntax analysis phases.
- The symbol table is used by various phases as follows:

1) Lexical analyzer:-

stores information of symbols in symbol table.

2) Parser:- (syntax analyser)

while checking the syntax of source, makes use of symbol table to verify the information about tokens being generated.

3) Semantic Analysis:- This phase refers symbol table for type conflict issue.

) Code generation:- refers symbol table knowing how much run time space is allocated? What type of run time scope is allocated.

l-Value and r-value:-

The 'l' and 'r' prefixes from left and right side assignment.

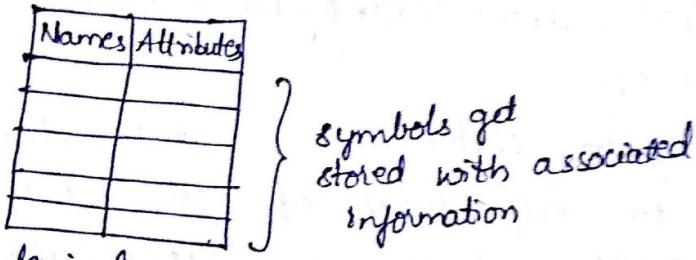
Ex:-

a := i + 1

↑ l-value ↑ r-value

Entries in Symbol table:-

- To achieve compile time efficiency compiler makes use of symbol table.



- It associates lexical names with their attributes.
- Items to be stored in symbol table are:-
  1.) Variable names, identifiers also.
  2.) Constants
  3.) Procedure names
  4.) Function names
  5.) Literal constants and strings
  6.) Compiler generated temporaries
  7.) Labels in source languages.

Compiler uses following types of information from symbol tabl

1.) Data type 2.) Name 3.) Declaring procedures
4.) Offset in storage.
5.) If structure or record then pointer to structure table
6.) For parameters, whether parameters passing is by value or reference?
7.) Number and type of arguments passed to function.
8.) Base address.

Methods used how to store in Symbol table :-

There are 2 types of name representation.

1) Fixed - length name :-

A fixed space for each name is allocated in symtab. In this type of storage if name is too small then there is wastage of space. The name can be referred by pointer to symbol table entry.

Name										Attribute
c	a	l	c	u	l	a	t	e		
s	u	m								
a										
b										

2) Variable - length name :-

The amount of space required by string is used : store the names. The name can be stored with help of starting index and length of each name.

Ex :-

Name		Attribute
Starting Index	length	
0	10	
10	4	
14	2	
16	2	

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	$

Need for Symbol Table :-

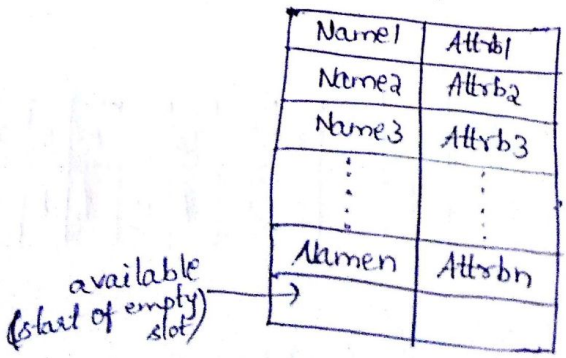
The symbol tables are required for

i.) For quick insertion of identifies and related information.
ii.) For quick searching of identifies.

Symbol table can be organised in 3 ways:-
- List Data structure
- Self organizing list
- Hash tables.

1) Symbol table organization using linear list:-
- Linear list is simplest kind of mechanism to implement symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in order as they arrive.
- The pointer available is maintained at end of all stored records. To retrieve information about some we start from beginning of array and go on searching up to available pointer.
- If we reach 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 than another error occurs i.e.) "multiple defined name"

The advantage of list organization is that it takes minimum amount of space



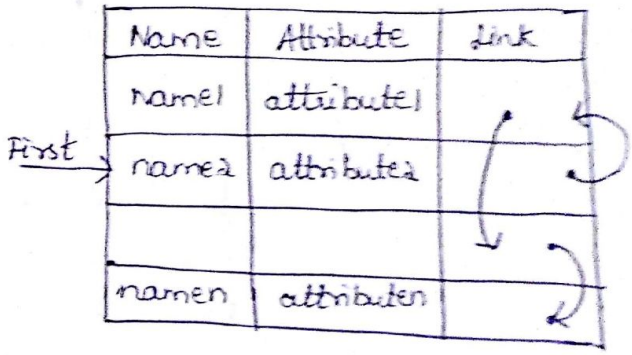
(For list data structure using arrays)

2). Symbol Table organization using self organizing list:-

- This symbol table implementation is using linked list. A link field is added to each record.
- We search the records in order pointed by link of link field. A pointer "First" is maintained to point to first record of the symbol table.
- The references to these names can be Name3, Name1, Name2, when name is referenced or created it is moved to front of list.
- Most frequently referenced names will tend to be front of list. Hence access time to most frequently referred names will be the least.

Advantages:-

- Fast to access
- Memory wastage is reduced.



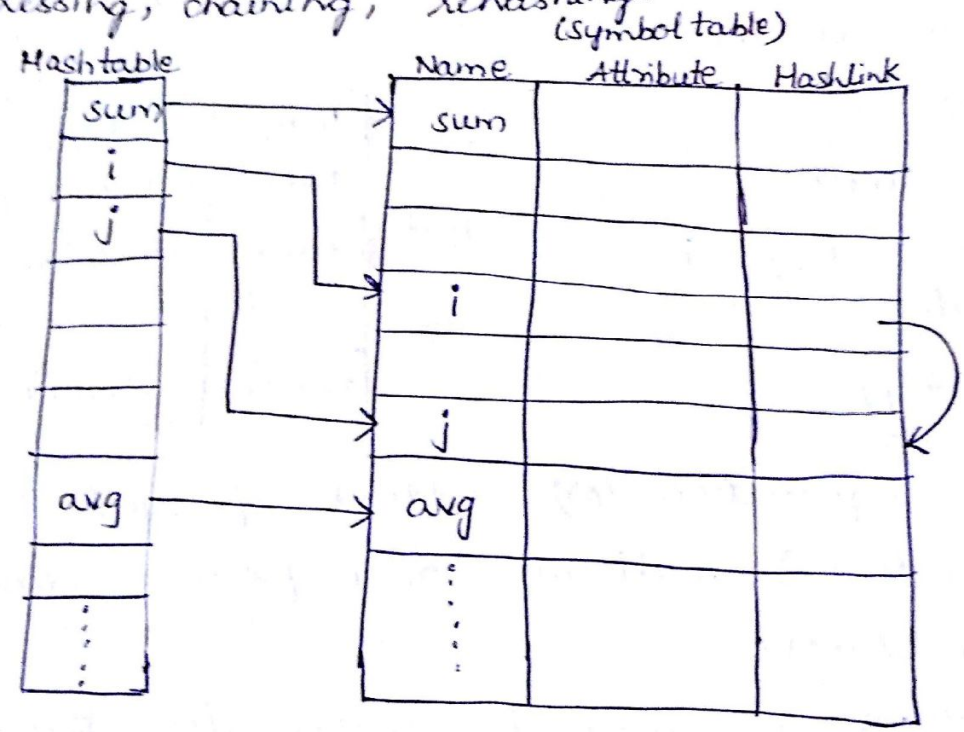
3) Hash Table:-

- Contains information (or) address of symbol table. we can allocate (or) deallocate data from a memory at arbitrary times.
- In hashing is used to search the records of symbol table. In hashing scheme 2 tables are maintained a hash table and symbol table.
- Hash table consists of K entries from 0, 1 to K-1. These entries are basically pointers to symbol table pointing to names of symbol table.
- To determine whether '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,

$$\text{position} = h(\text{name})$$

- Using this position we can obtain exact locations of name in 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 location for storing names. Various collision resolution techniques are open addressing, chaining, rehashing.



Advantage:-

- Quick search is possible.

Disadvantages:-

Some extra space is required, because of storing hash and symbol tables.

Obtaining scope of variables is very difficult.

Dynamic Storage Allocation Techniques:-

- Allocating memory using some functions.
- 'new' is a function used to create memory.
- 'dispose' is a function used to destroy memory created.

Dynamic storage allocation techniques is of 2 types.

1.) Explicit Allocation <— fixed sized block, variable sized block

2.) Implicit Allocation

- In PASCAL, uses the functions 'new' and 'dispose'.
- Using 'new' and 'dispose' function user explicitly creates or destroys memory (record).

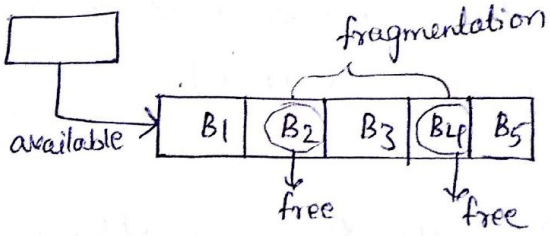
- **Fixed sized block:-**

Amount of memory required to complete the program is fixed in the function written by user.

- malloc function is used for

Ex:- Linked list creation of a node in linked list needs malloc function. Each node is fixed with some memory

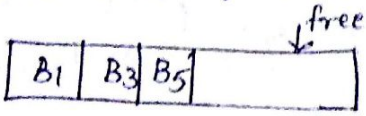
- **Variable sized block:-**



In linked list, each node is fixed with some memory, but we can vary size of linked list by either adding or deleting nodes.

If we delete some nodes in the list then there will be free space created such that the block of memory allocated for linked list gets fragmented.

- Memory can be allocated for this technique is by heap allocation storage, so that memory allocated can get fragmented.
- Using some algorithms free space created by deallocating nodes is moved to a side so that free space is available at one end of list.



Implicit Allocation:-

- Compiler automatically allocates and deallocates memory.
- Some languages which support this type of allocation are LISP, SNOBAL.
- Block of memory is created with help of packages where block contains some set of operations like.

Block size	→ amount of memory required to construct basic block.
Reference count	→ address of another node is stored, means, if reference count is 1, then it refers to next block. If it is zero no reference is made.
Marking options	→ We can mark user block using this technique. It indicates whether reference count is used or not.
User data	→ where user data is stored.

- By using Implicit allocation, problems might occur they are 'dangling reference' and 'garbage collection'.