Symbol Table Management

Symbol Table

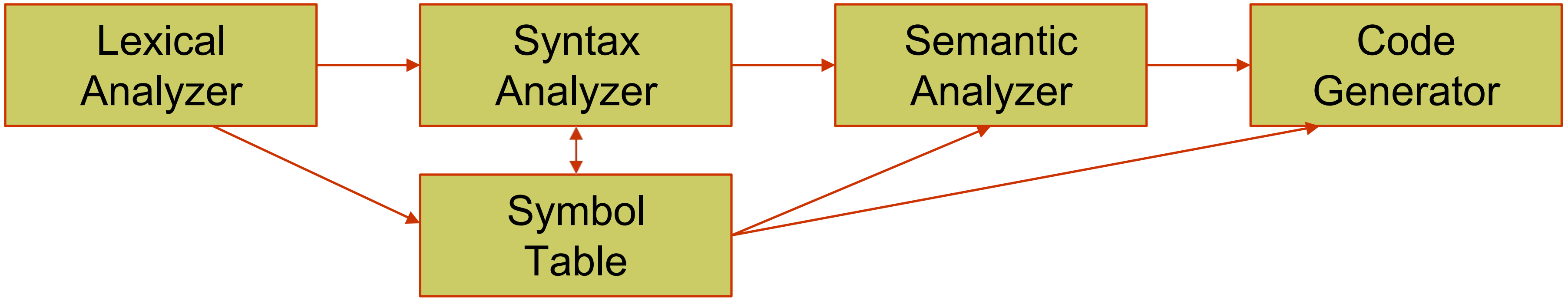
- Symbol table is structure created and maintained by data compilers to store information about the occurrence of various entities such as **variable names, function names, objects, classes, interfaces, etc.**

- Symbol table is used by both the analysis and the synthesis parts of a compiler.

Symbol Table

- When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers and other symbols, variables, constants, procedures statements e.t.c,
- This information about the name:-
- Type
- Its Form , Its Location

It will be used later by the semantic analyzer and the code generator.



- Efficient to add new entries to the S.T
- Dynamic in nature

Issues in Symbol Table:

- Format of entries
- Method of Access
- Place where they are stored

Contents of Symbol Table

Name	Information

<u>Capabilities of S.T :</u>

1)Checking (Determine whether the given information is in the table)

2) Adding New Information

3)Access the information of Name

4)Deletion

Symbol Table Entries

- following information about Name and each entry in the symbol table is associated with attributes that support the compiler in different phases:
  - The name (as a string).
  - Size and Dimension
  - The data type.
  - Its scope (global, local, or parameter).
  - Its offset from the base pointer (for local variables and parameters only).

Implementation

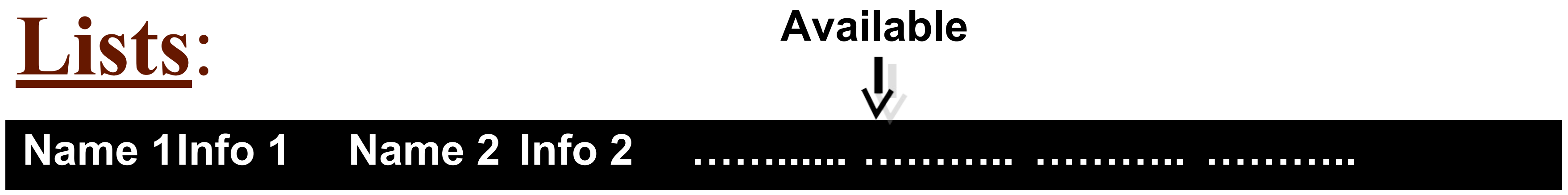
- Use linear Array of records ,one record per name.
- Entries of S.T are not uniform so to make it uniform some information is kept outside the table and pointer to this information stored in S.T.
- **Record** (consist known no. of consecutive words of memory,, so names stored in record)

It is appropriate if upper bound on the length of identifier is given…..

Data Structure for S.T

- Required to make n-entries and m-inquiries

1) **Lists**:



- **It is easy to implement**
- **Addition is easy**
- **Retrieve Information**
- **ADVANTAGES:** **Minimum space is required**

Addition in table is easy

- **DISADVANTAGE:** **Higher Access time**

2)Binary Search Tree:

- Efficient approach to organize S.T with two field :::
- Left and Right
- **Algorithm for searching name in B.S.T**

```
P= initially a pointer to root
1)  If Name = Name (P) then   Return   /* success */
2)  Else if Name < Name (P) then
        P:= left(P) /* visit left child */
3)  Else Name (P) < Name  then
        P:= Right (P)    /* visit right child */
```

Addition

Firstly search, if doesn’t exist then create new node at proper position.

3)Hash Table

- Consists K words [0….k-1]
- Pointers into the storage table (linked list)

- **Searching Name in S.T**
- Apply hash function to name

  h(Name) -> {0…..k-1 (integer) }

Addition new Name

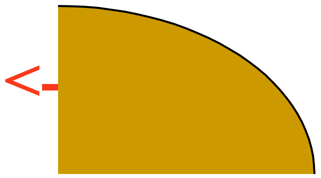
Create a record at available space in storage table and link that record to h(Name)th list.

Hashing > BST > Linear List

Representing Scope information in S.T

- **Scope:** The region of a program where a binding is active
- The same name in a different scope can have a different binding
- **<u>Rules for governing scope :</u>**
- 1) If name declared within a block B then valid only within B
- 2) If B1()

```
        {.............<-
              B2()
               {.......}
        }
```



- Require complicated S.T organization
- So use Multiple symbol tables, one for each block
- **Each Table has : Name and Information**
- If **New block entered**

  Then Push an empty table into the stack for storing names and information.

Ex:-

```
Program     main
  Var x,y : integer ;
  Procedure P:
     Var x,a : boolean ;
   Procedure Q:
     Var x,y,z : real;
    Begin …… end
    Begin ……end
    Begin ……end
```

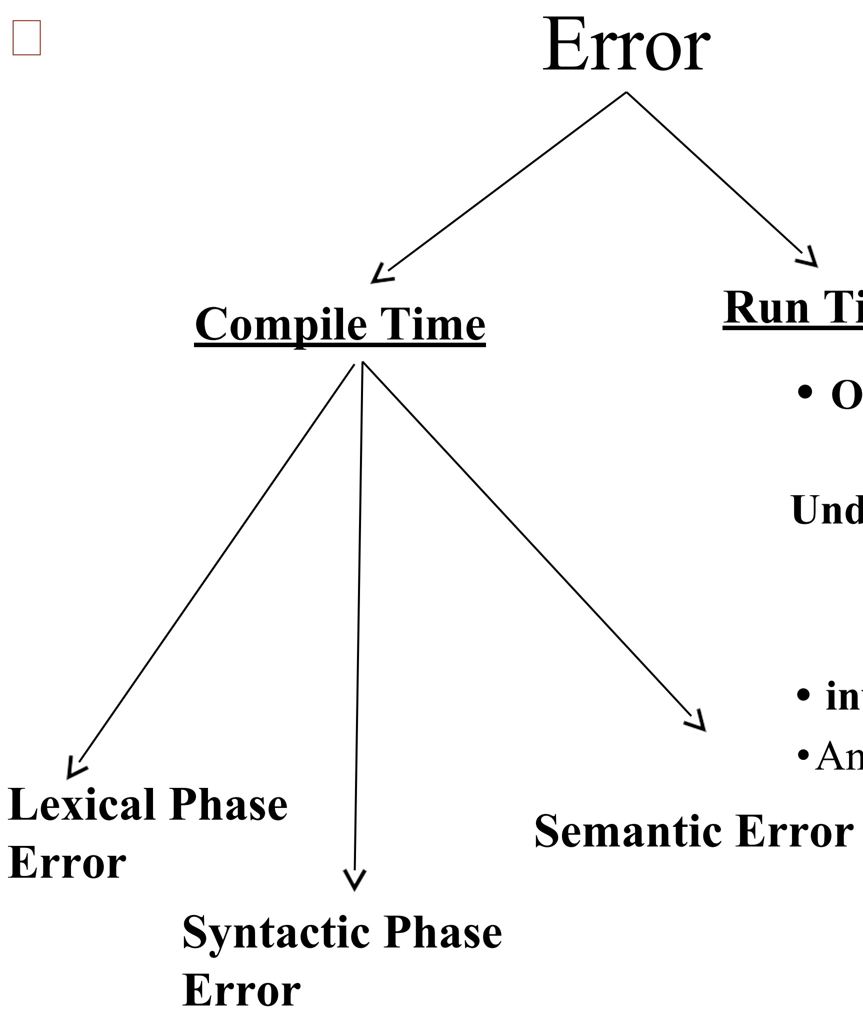
Symbol Table organization that compiles with static scope information rules

- <u>Next technique to represent scope information in S.T:</u>

1) Write nesting depth of each procedure block
2) Use pair (Procedure name, nesting depth) to access the information from the table

Error Detection & Recovery

- programmers make mistakes



- **Overflow** {Indicates that the magnitude of a computational result is too large to represent.}

Underflow {ndicates that the magnitude of a computational result is too close to zero to represent.}

- **invalid subscript** {}
- An integer division by zero

Sources of Error

- Algorithmic Error
- Coding Error
- A program may exceed a compiler or machine limit

Ex:- Array declaration with too many dimensions to fit into S.T

- Error in the phases of compiler (during translating program into object code)

Some Transcription Errors

- The insertion of an extra character
- Deletion of required character
- Replacement of correct character by an incorrect character

1) Lexical Phase Error

- If after some processing lexical analyzer discover that **no prefix of remaining input fits to any token class then invoke error recovery routine.**

<u>Simplest way to recover it</u>

- **skip the erroneous character until L.A finds next token**
- **Disadvantage:**

set the problems for next phases of compiler

•Too long numeric literals

•Long identifiers

•Misspelling of identifiers or keywords

•Missing operators

Error Recovery

- **Panic Mode Recovery:**

1) The parser discovers an error.

2)If any unwanted character occurs then delete that character to recover error.

3) Rejects input symbols until a “synchronizing” token usually **a statement delimiter as: semicolon ; , end } is encountered.**

3) Parser delete stack entry until finds an entry with which it can continue parsing.

2) Syntactic Errors

- Examples of Syntactic Errors

1) **Missing right Parenthesis:**

   max(A, 2* (3+B) **{ Deletion error }**

2) **Extra Comma: for(i=0;,1<100;i++)** **{ insertion error }**

3) **Colon in place of semicolon :**

   I = 1: **{Replacement Error}**

4) **Misspelled keyword :**

   Void mian () **{Transposition Error}**

5) **Extra Blank:**

   /* comment * / **{Insertion Error}**

Minimum Distance correction of syntactic error

- Theoretical way of defining errors and their location
- It is called “**Minimum Hamming distance**” Method.
- **Let a program P has errors = k**
- **Find shortest sequence of error transformations that will map to valid program**

Ex:

```
        IFA =B THEN
            SUM =SUM + A;
        ELSE
            SUM =SUM - A;
```

$\Longrightarrow$

```
IF A =B THEN
        SUM =SUM + A;
    ELSE
        SUM =SUM - A;
```

“**Minimum Hamming distance**” **= 1** (Transformation may be the insertion or deletion of a single character)

Recovery from syntactic Error

I) Panic Mode Recovery:

The parser discovers an error. It then discards input symbols till a designated set of synchronizing token is found.

- **Synchronizing tokens** selected are such that their role in the program is unambiguous, **like Delimiters ; } etc.**
- **Advantage:** Simple and never goes into an infinite loop.

Algorithm of panic mode recovery in LL(1)Parsing

1) Parser looking for entry in parsing table

```
2)  if M[A, a] = 'Blank'
      then input symbol a skipped
    else if M[A, a]= "Synch"
      then pop off the nonterminal from the top of the stack
    else top[token] ≠ Input symbol
      then pop off the token from the stack
```

Panic Mode Recovery in LL Parser

E→ TE′   E′→+TE′ | ε   T→ FT′   T′→ *FT′ | ε

F→(E) | id

	First	Follow
E	{ (, id }	{), $ }
E′	{ +, ε }	{) , $ }
T	{ (, id }	{ +,), $ }
T′	{ * , ε }	{ + ,) , $ }
F	{ (, id }	{ * , + ,) , $ }

Panic Mode Recovery in LL Parser

LL Parsing Table for a given Grammar

	+	*	(	Id	)	$
E			E TE’	E TE’		
E’	E +TE’				E’ ε	E’ ε
T			T FT’	T FT’		
T’	T’ ε	T *FT’			T’ ε	T’ ε
F			F(E)	F id		

Panic Mode Recovery in LL Parser

Modified LL parsing table

	+	*	(	Id	)	$
E			E→ TE’	E→ TE’	Synch	Synch
E’	E→ +TE’				E’→ ε	E’→ ε
T	Synch		T→ FT’	T→ FT’	Synch	Synch
T’	T’→ ε	T→ *FT’			T’→ ε	T’→ ε
F			F→(E)	F→ id		

Error recovery in LL Parsing Table using panic mode recovery

Input	Stack	Action
$id+*id	E$	M[E, id] = E→ TE’
$id+*id	TE’ $	M[T, id] = T→FT’
$id+*id	FT’E’ $	M[F, id] = F→id
$id+*id	idT’E’ $	Pop ‘id’ from stack and move the input pointer further
$id+*	T’E’ $	M[T, *] = T→ *FT’
$id+*	*FT’E’ $	Pop ‘*’ from stack and move the input pointer further
$id+	FT’E’ $	M[F, +] = Synch Pop F
$id+	T’E’ $	M[T’, +] = T’→ ε
$id+	E’ $	M[E’, +] = E’→+TE’
$id+	+TE’ $	Pop ‘+’ from stack and move the input pointer further
$id	TE’ $	M[T, id] = T→ FT’
$id	FT’E’ $	M[F, id] = F→ id
$id	idT’E’ $	Pop ‘id’ from stack and move the input pointer further
$	T’E’ $	M[T’, $] = T’→ ε
$	E’ $	M[E’, $] = E’→ ε
$	$	Accept

Panic Mode Recovery in LR Parser

E E + E | E * E | (E) | id

LR Parsing table

	Id	+	*	$	E
I0	S2				1
I1		S3	S4	Accept	
I2		R3	R3	R3	
I3	S2				5
I4	S2				6
I5		R1	S4	R1	
I6		R2	R2	R2	

Panic Mode Recovery in LR Parser

E E + E | E * E | (E) | id

LR Parsing table with error routines

	Id	+	*	$	E
I0	S2	e1	e1	e1	1
I1	e2	S3	S4	Accept	
I2	R3	R3	R3	R3	
I3	S2	e1	e1	e1	5
I4	S2	e1	e1	e1	6
I5	R1	R1	S4	R1	
I6	R2	R2	R2	R2	

Input	Stack	Action
$0	Id+*id$	M[0, id] =S2
$0id2	+*id$	M[2, +] =S3
$0id2+3	*id$	M[3, *] =e1
$0id2+3id2	*id$	M[2, *] =R3
$0E1	*id$	M[1, *] =S4
$0E1*4	Id$	M[4, id] =S2
$0E1*4id2	$	M[2, $] =R3
$0E1	$	Accept

Processing:

Fill the synch entries under the follow of nonterminals

	id	+	*	(	)	$
E	E -> TE'			E -> TE'	synch	synch
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> F T'	synch		T -> F T'	synch	synch
T'		T' -> ε	T' -> * F T'		T' -> ε	T' -> ε
F	F -> id	synch	synch	F -> (E)	synch	synch

Fill "synch" under the follow of nonterminals ………..
Then perform the operation for the

Input string (w) = * id *+ id $ According to Algorithm

STACK	INPUT	REMARK
$\$E$	***id** $*$ + **id** \$	error, skip $*$
$\$E$	**id** $*$ + **id** \$	**id** is in FIRST(E)
$\$E'T$	**id** $*$ + **id** \$	
$\$E'T'F$	**id** $*$ + **id** \$	
$\$E'T'$**id**	**id** $*$ + **id** \$	
$\$E'T'$	$*$ + **id** \$	
$\$E'T'F*$	$*$ + **id** \$	
$\$E'T'F$	+ **id** \$	error, $M[F, +]$ = synch
$\$E'T'$	+ **id** \$	F has been popped
$\$E'$	+ **id** \$	
$\$E'T+$	+ **id** \$	
$\$E'T$	**id** \$	
$\$E'T'F$	**id** \$	
$\$E'T'$**id**	**id** \$	
$\$E'T'$	\$	
$\$E'$	\$	
\$	\$	

Parsing and error recovery moves made by predictive parser.

II) Phrase –level Recovery

- Local Correction by parser on remaining input, by some string which allows parser to continue.
- Replacing comma by semicolon, inserting extra semicolon etc.
- Perform local correction on the input to repair the error
- **Drawbacks:** Improperreplacement might lead to infinite loops.

  Hard to find where is actual error.
- **Advantage:** It can correct any input string.

III) Global Correction

- Compiler perform some changes to process the input string.
- It uses simple way , **where choose minimal sequence of changes to obtain least cost correction**.
- Input:: **Incorrect I/P string = X**

  Grammar= G
- **Then algorithm will find the parse tree for related string = Y**
- Transform X toY by performing some insertion, deletion and changes in to the token stream.

Disadvantages

- Too costly to implement in terms of space and time.
- Basically includes theoretical interest.

IV) Error Production

- A method of predict common errors that might be encountered.
- Augmenting the grammar for the language at hand, with productions as : **A-> Error.**
- Such a parser will detect expected errors when an error production is used.
- **Ex:- Automatic Error recovery in YACC**

  Use error production with semantic actions

  A : Error ε {semantic action to recover error}.

- **Advantage: Error diagnostics is very fast.**

3) Recovery from Semantic error

- **Sources of Error**

i) Undeclared names and type incompatibilities.

ii) **Recovery**

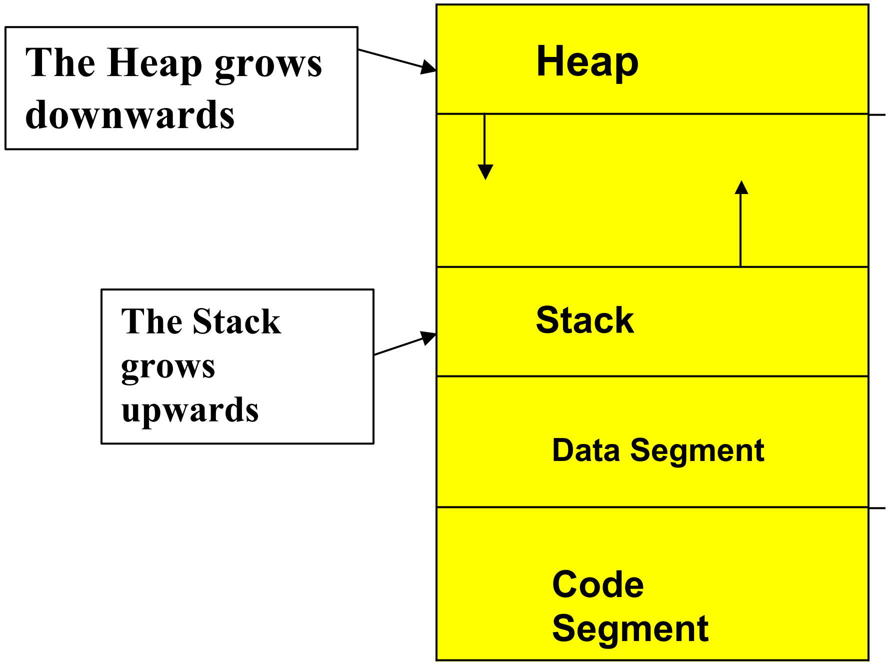
a) Type Checking, where compiler report the nature and location of error.

b) Declare the undeclared names and stored into the symbol table

Stack and Heap Allocation

Program Address Space

- Any program you run has, associated with it, some memory which is divided into:
  - Code Segment
  - Data Segment (Holds Global Data)
  - Stack (where the local variables and other temporary information is stored)
  - Heap



Local Variables:Stack Allocation

- When we have a declaration of the form “int a;”:
  - a variable with identifier “a” and some memory allocated to it is created in the stack. The attributes of “a” are:
    - **Name: a**
    - **Data type: int**
    - **Scope: visible only inside the function it is defined, disappears once we exit the function**
    - **Address: address of the memory location reserved for it. Note: Memory is allocated in the stack for a even before it is initialized.**
    - **Size: typically 2 bytes**
    - **Value: Will be set once the variable is initialized**
- **Since the memory allocated for the variable is set in the beginning itself, we cannot use the stack in cases where the amount of memory required is not known in advance. This motivates the need for HEAP**

Pointers

- We know what a pointer is. Let us say we have declared a pointer "int *p;" The attributes of "a" are:
  - Name: p
  - Data type: Integer address
  - Scope: Local or Global
  - Address: Address in the data segment or stack segment
  - Size: 32 bits in a 32-bit architecture
- We saw how a fixed memory allocation is done in the stack, **now we want to allocate dynamically. Consider the declaration:**
  - "int *p;". **So the compiler knows that we have a pointer p that may store the starting address of a variable of type int.**
  - **To point "p" to a dynamic variable we need to use a declaration of the type " p = new int;"**

Pointers : Heap Allocation

- Dynamic variables are never initialized by the compiler, so it is a good practice to initialize it.

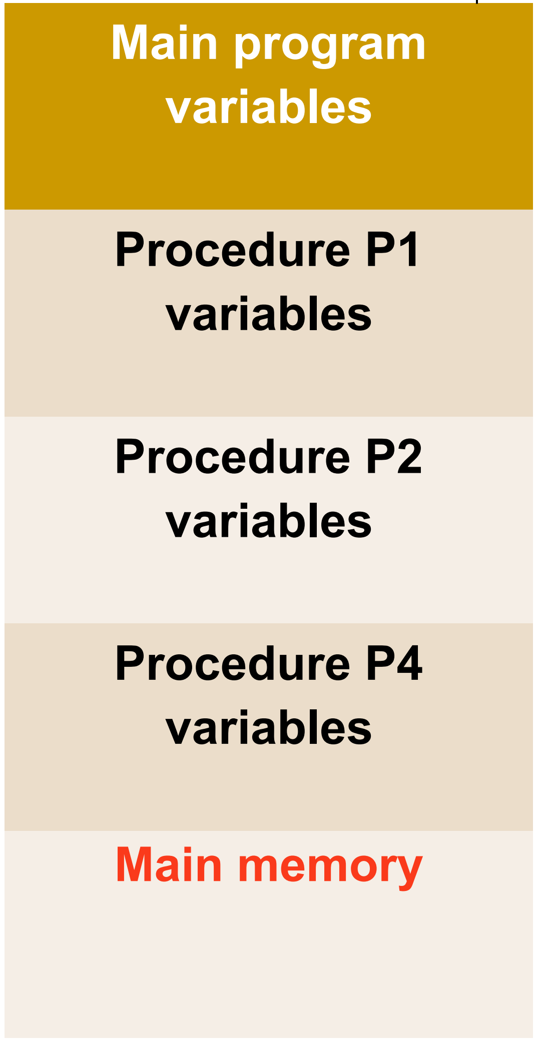
```
int *p;
p   =   new int;
*p  =   0;
```

- In more compact notation:

```
int *p =
new  int(0);
```

Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile timeNo stack/heap allocation; no overheads
- Ex: Fortran IV and Fortran 77
- Variable access is fast since addresses are known at compile time
- No recursion



Dynamic Data Storage Allocation

- Compiler allocates space only for golbal variables at compile time
- **Space for variables of procedures will be allocated at run-time Stack/heap allocation**
- Ex: **C, C++, Java, Fortran 8/9**
- **Variable access is slow** (compared to static allocation) since addresses are accessed through the stack/heap pointer
- Recursion can be implemented

Variable Storage Offset Computation

- The compiler should compute the offsets at which variables and constants will be stored in the activation record (AR)

- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

Static Scope and Dynamic Scope

- **Static Scope A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text**
- Uses the static(unchanging) relationship between blocks in the program text
- **Dynamic Scope A global identifier refers to the identifier associated with the most recent activation record**
- Uses the actual sequence of calls that are executed in the dynamic(changing) execution of the program
- Both are identical as far as local variables are concerned