

Chapter 6

Compilers & Interpreters

By:

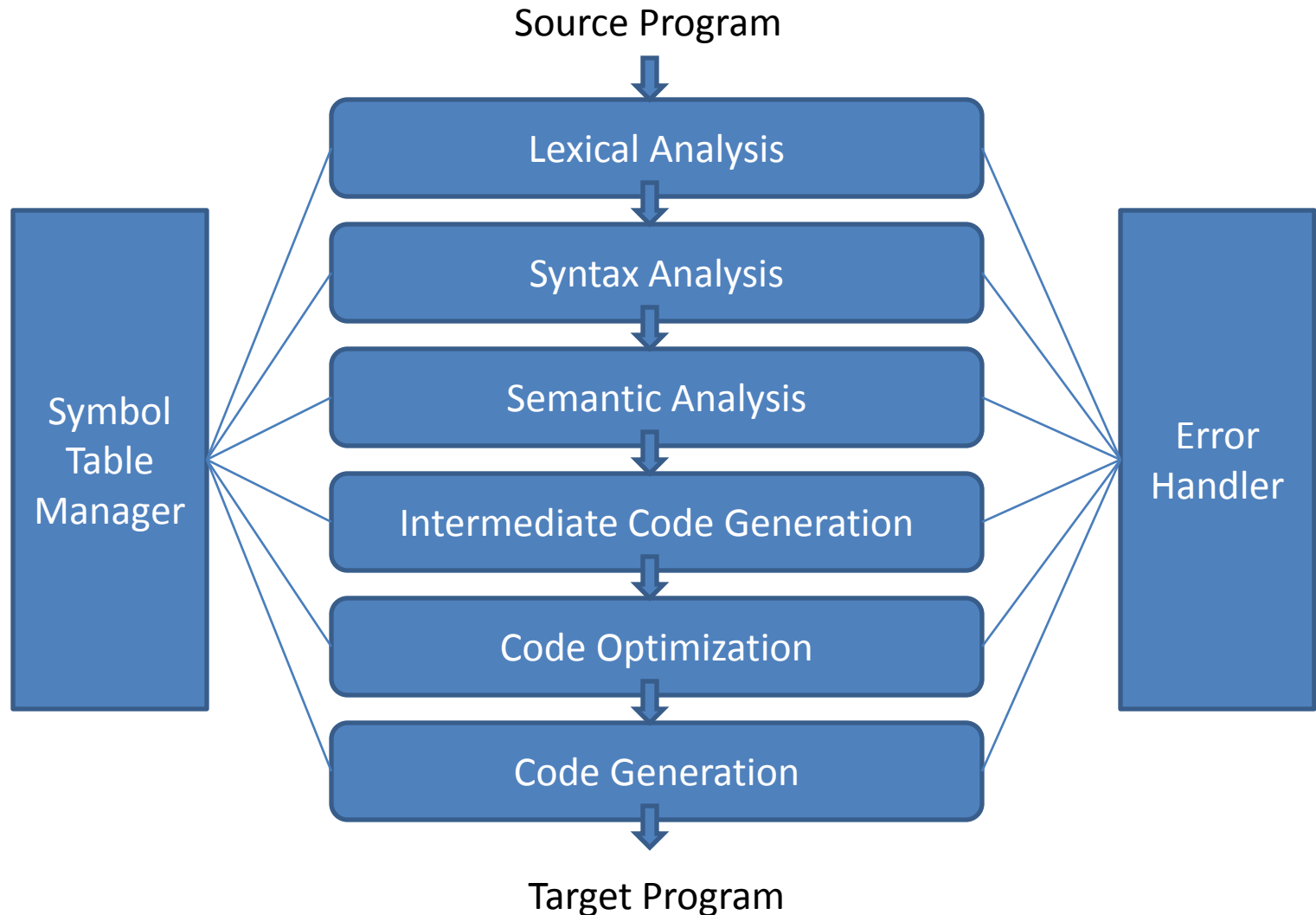
Prof. Bhargavi H Goswami,
Sunshine Group of Institutes,
Rajkot, Gujarat, India.

Email: bhargavigoswami@gmail.com

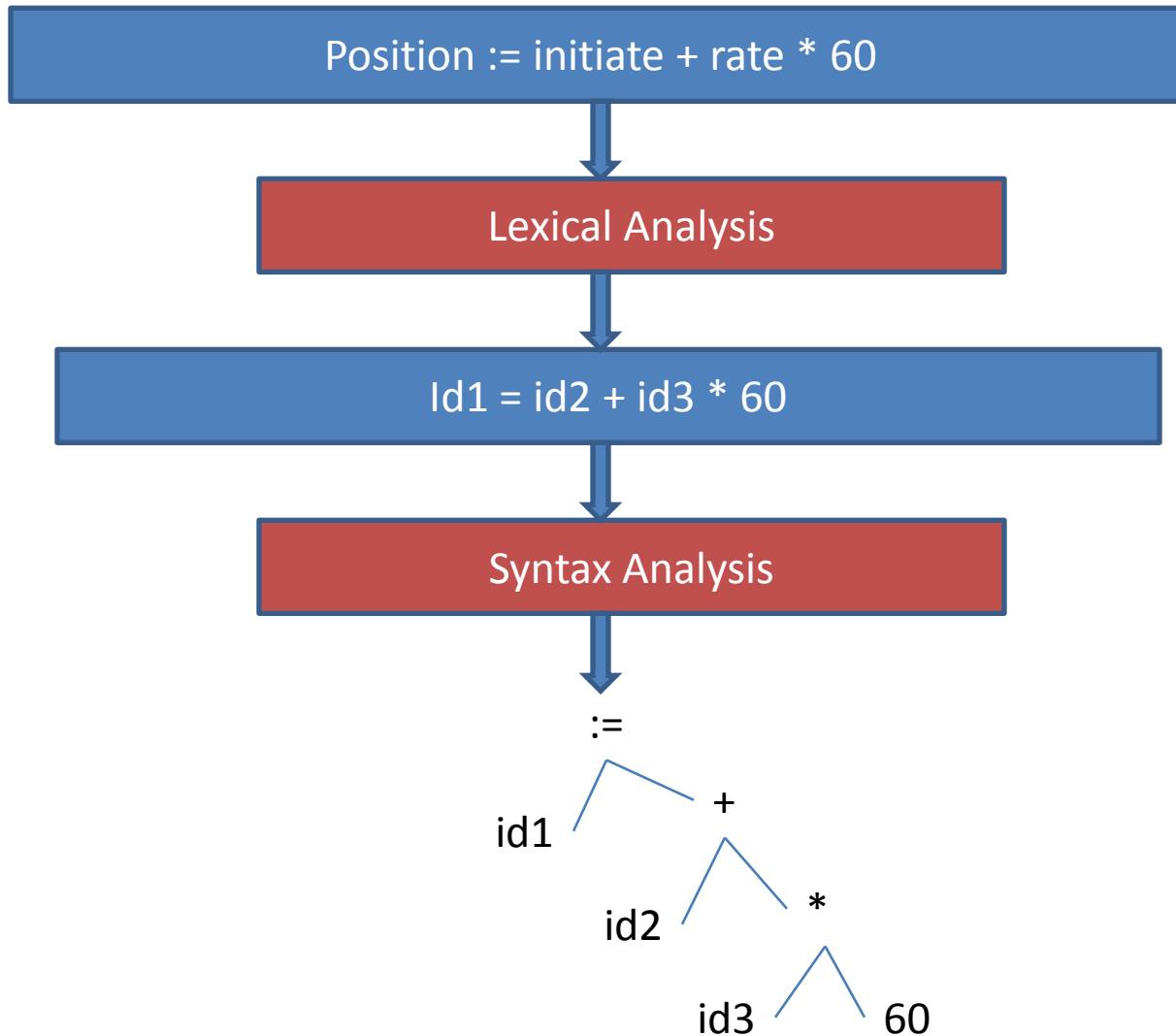
Mob: +91 8140099018

1. Phases of Compilation

Phases of Compiler



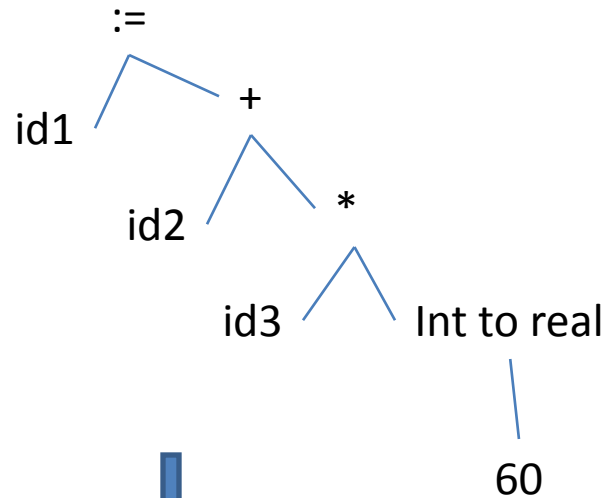
Example:



Example Conti...



Semantic Analysis

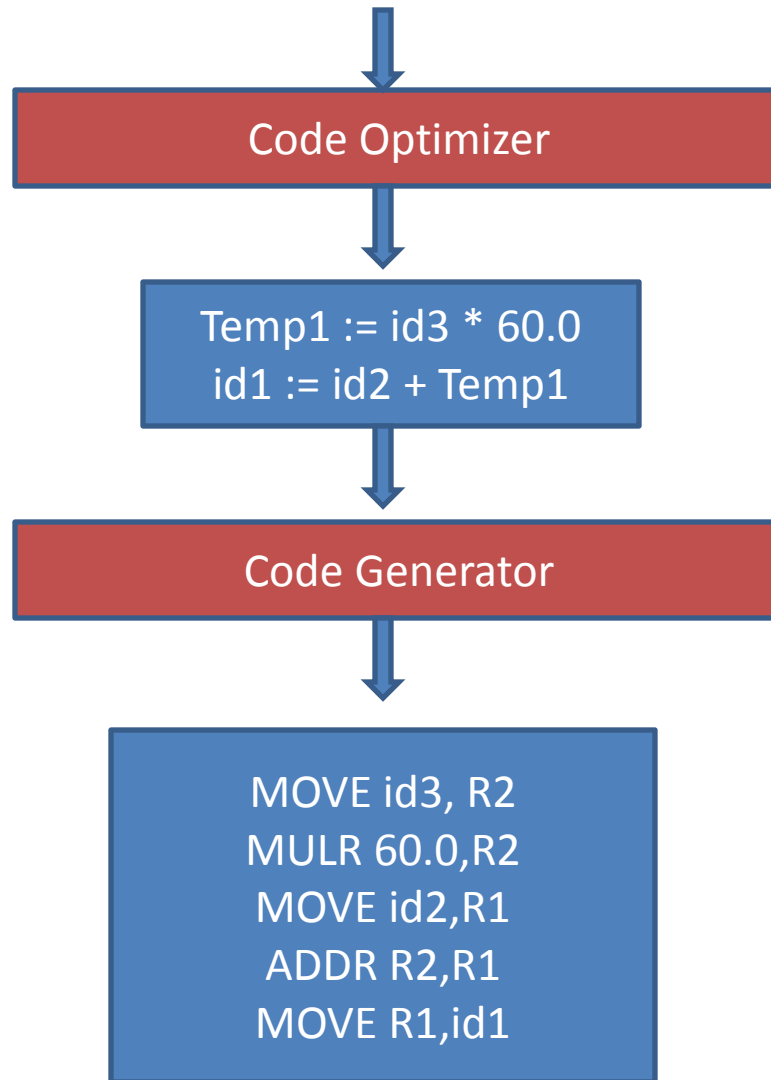


Intermediate Code Generator



```
Temp1 := int to real (60)
Temp2 := id3 * Temp1
Temp3 := id2 + Temp2
id1 := Temp3
```

Example conti...



- Each phase transfers source program from one representation to another.
- Typical decomposition of compiler into phases results to conversion of source program into target program.

1. Symbol Table Manager:

- Important Functions:
 - Record Identifiers used in program
 - Collect info about various attributes of each identifier.
- Stores:
 - Storage Location
 - Type
 - Scope
 - Procedure name
 - No. of type of arguments
 - Return Type
- Symbol table is data structure containing record for each identifiers with fields for attributes of identifiers.
- It is prepared at lexical analysis and later phases add information.

2. Error Detection & Reporting:

- Each phase can encounter errors.
- Compilation can proceed only after solving errors generated by lexical, syntax & semantic analysis.
- If code doesn't form tokens, structure is violated which is detected by lexical analysis.
- If syntax is violated, error is detected by syntax phase.
- During semantic analysis, compiler tries to detect constructs that have right syntactic structure.

3. Analysis Phase:

- Performs lexical, syntax and semantic analysis.

4. Intermediate Code Generation:

- After semantic analysis, some compiler generate explicit Intermediate Representation (IR) of source program.
- This IR has two properties:
 - 1. Easy to produce
 - 2. Easy to translate into target program.

5. Code Optimization:

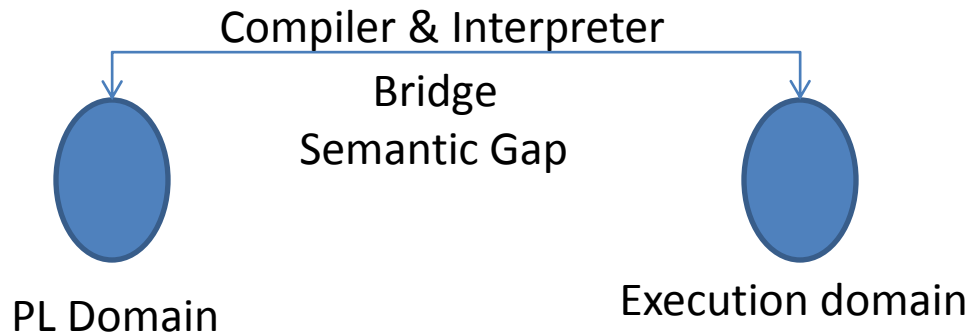
- Attempts to improve IC.
- Results to faster running machine code.
- Optimization improve running time of target program.
- But it doesn't slow down compilation.

6. Code Generation:

- Final phase of compiler that generates target code.
- Consist of re-locatable machine code or assembly code.
- Here memory locations are selected for each variable used in the program.

Aspects of Compilation

Aspects of Compilation



- Two aspects:
 - Generate code.
 - Provide Diagnostics.
- To understand the implementation issue, we should know PL features contributing to semantic gap between PL and Execution domain.
- PL Features:
 1. **Data Types**
 2. **Data Structure**
 3. **Scope Rules**
 4. **Control Structures**

1. Data type

- **Definition:**

A datatype is the specification of

(i) legal values for variables of type

(ii) legal operations on the legal values of the type.

- **Task:**

1. Check legality of operations for type of its operands.
2. Use type conversion operations wherever necessary & permissible.
3. Use appropriate instruction sequence.

```
var
  x,y : real;
  i,j : integer;
```

```
Begin
```

```
  y := 10;
  x := y + i;
```

Type conversion of i is needed.

```
  i : integer;
  a,b : real;
  a := b + i;
```

```
CONV_R    AREG, I
ADD_R     AREG, B
MOVEM     AREG, A
```

2. Data Structure

- PL permits declaration of DS and use it.
- To compile the reference of element of DS compiler must develop memory mapping to access allocated area.
- A record, heterogeneous DS leads to complex memory mapping.
- User defined DS requires mapping of different kind.
- Proper combination of DS is required to manage such complexity of structure.
- Two kind of mapping is involved
 - Mapping array reference
 - Access field of record

Example:

Program example (input,output);

type

 employee = record

 name : array [1...10] of character;

 sex : character;

 id : integer

 end;

 weekday = (mon,tue,wed,thur,fri,sat,sun);

var

 info : array [1..500] of employee;

 today : weekday;

 i,j : integer;

begin {main program}

 today := mon;

 info[i].id := j;

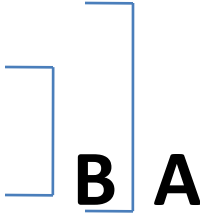
 if today = tue then....

end

3. Scope Rules

- Determine the accessibility of variable declared in different blocks of a program.
- Eg:

```
x, y : real;  
      y, z : integer;  
      x := y;
```



The diagram illustrates nested blocks. A large rectangle labeled **A** contains a smaller rectangle labeled **B**. The code is positioned to the left of the rectangles. The first two lines of code, `x, y : real;` and `y, z : integer;`, are aligned with the top of block **A**. The third line, `x := y;`, is aligned with the top of block **B**. This indicates that the assignment statement is executed within the scope of block **B**, and thus it uses the values of `x` and `y` declared in block **B**.
- Stat `x:=y` uses value of block 'B'.
- To determine accessibility of variable compiler performs operation:
 - Scope Analysis
 - Name Resolution

4. Control Structure

- Def: It is collection of language features for altering flow of control during execution.
- This includes:
 - Conditional transfer of control
 - Conditional execution
 - Iterative control
 - Procedure call
- Compiler must ensure non-violation of program semantics.
- Eg: for i = 1 to 100 do
 begin
 lab1 : if i = 10 then

 End
- Forbidden: control is transferred to label1 from outside the loop.
- Assignment statements are also not allowed.

Memory Allocation

Memory Allocation

- 3 important task:
 - Determine memory **requirement**
To represent value of data items.
 - Determine memory **allocation**
To implement lifetime & scope of data item.
 - Determine memory **mapping**
To access value in non-scalar data items.
- Binding: A memory binding is an association b/w memory address attributes of the data item & address of memory area.
- Topic list:
 - A. Static and dynamic memory allocation
 - B. Memory allocation in block structured language
 - C. Array allocation & access

[A] Static & Dynamic Allocation

Static Memory Allocation

1. Memory is allocated to variable before execution of program begins.
2. Performed during compilation.
3. At execution no allocation & de-allocation is performed.
4. Allocation to variable exist even if program unit is not active.

Dynamic Memory Allocation

1. Memory allocation of variable is done all the time of execution of program.
2. Performed during execution.
3. Allocation & de-allocation actions occurs only at execution.
4. Allocation to variable exist only if program unit is active.

Static Memory Allocation

5. Variable remains allocated permanently till execution does not end.
6. Eg: Fortran
7. No. of flavors or types
8. Adv: Simplicity and faster access.
9. Memory wastage compared to dynamic.

Memory
Code(A)
Data (A)
Code (B)
Data (B)
Code (C)
Data (C)

Dynamic Memory Allocation

5. Variables swap from and to allocation state & free state.
6. Eg: PL/I, ADA, Pascal.
7. Two types/ flavors
 - 1. automatic allocation
 - 2. Program controlled allo
8. Adv: Recursion support DS size dynamically.
9. Less memory wastage compared to formal.

Memory: Only A is active
Code(A)
Code (B)
Code (C)
Data (A)

Memory: A calls B. A & B is active
Code(A)
Code (B)
Code (C)
Data (A)
Data (B)

Automatic v/s Program Controlled Dynamic Allocation

Automatic Dynamic Allocation

1. Implies memory binding performed at execution initiation time of program unit.
2. Memory is allocated to declared variables when execution starts.
3. De-allocated when program unit is exited.
4. Different memory areas may be allocated to same variable in different activation of program unit.
5. Implemented using stack since entry, exit is LIFO by nature.
6. Implemented variables of program are accessed using displacement from this pointer.

Program Controlled Dynamic Allocation

1. Implies memory binding performed during the execution of program unit.
2. Memory is allocated not at execution but when used for very first time during execution.
3. De-allocated when arbitrary points are left while execution.
4. Here allocation is done when program modules start purely based on scope of variables.
5. Implemented using heap DS, as not always LIFO by nature.
6. Pointer is needed now to point to each memory area allocated.

[B] Memory allocation in Block Structured Language

- Block contain data declaration
- There may be nested structure of block
- Block structure uses dynamic memory allocation
- Eg: PL/I, Pascal, ADA
- Sub Topic List
 1. Scope Rules
 2. Memory Allocation & Access
 3. Accessing Non-local variables
 4. Symbol table requirement
 5. Recursion
 6. Limitation of stack based memory allocation

1. Scope Rules:

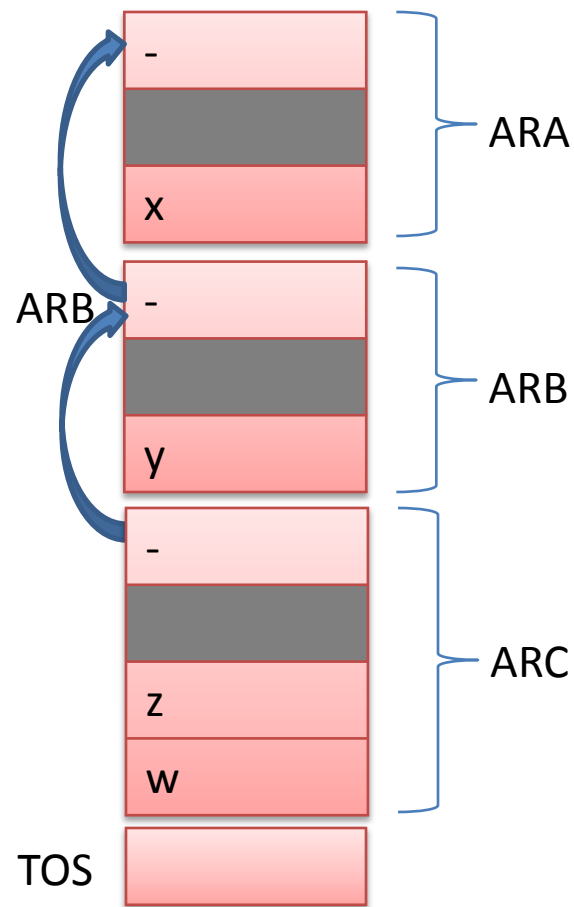
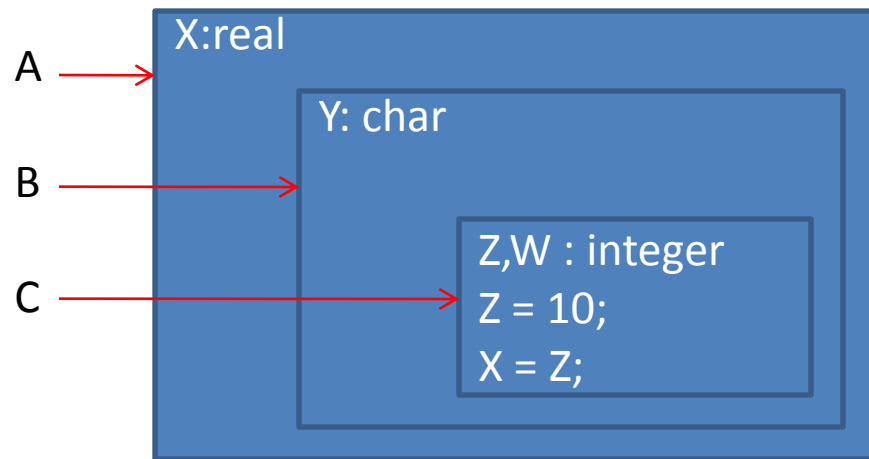
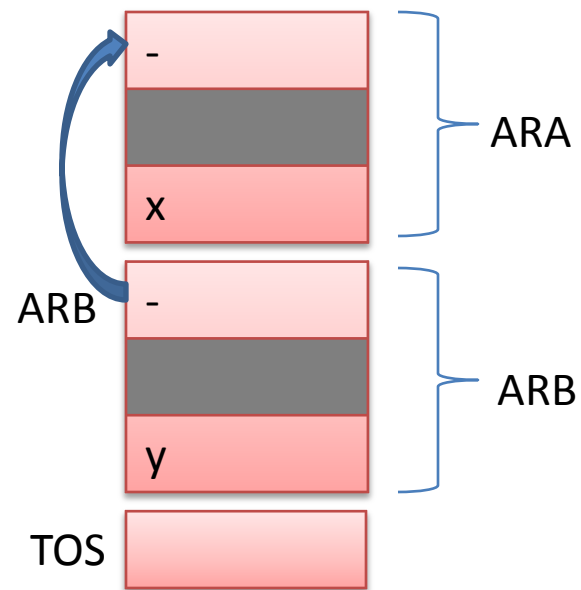
- If variables var_i is created with name $name_i$ in block B.
- Rule 1: var_i can be accessed in any statement situated in block B.
- Rule 2: Rule 1 + 'B' is enclosed in 'B' unless 'B' contains declaration using same name $name_i$.
- Eg: variables B = local
B' = non-local

```
A{  
    x,y,z : integer;  
    B{  
        g : real;  
        C{  
            h,z : real;  
        }C  
    }B  
    D{  
        i,j : integer;  
    }D  
}A
```

Block	Accessibility of Variables	
	Local	Non-Local
A	xA,yA,zA	
B	gB	xA,yA,zA
C	hC, zC	xA,yA,gB
D	iD, jD	xA, yA, zA

2. Memory allocation & Access

- Implemented using extended stack model.
- Automatic dynamic allocation is implemented using extended stack model.
- Minor variation: each record has two reserved pointers, determining its scope.
- AR: Activation Record
- ARB: Activation Record Block
- See the following figure:
 - When execution starts, state changes from (a) to state (b).
 - When execution exit block C, state change (a) to (b).



- **Allocation:**

1. $TOS := TOS + 1;$
2. $TOS^* := ARB;$
3. $ARB := TOS;$
4. $TOS := TOS + 1;$
5. $TOS^* := \dots\dots\dots(\text{sec reserved pointer 2})$
6. $TOS := TOS + n;$

- So address of 'z' is $\langle ARB \rangle + z;$

- **De-allocation:**

1. $TOS := ARB - 1;$
2. $ARB := ARB^* ;$

3. Accessing non-local variables

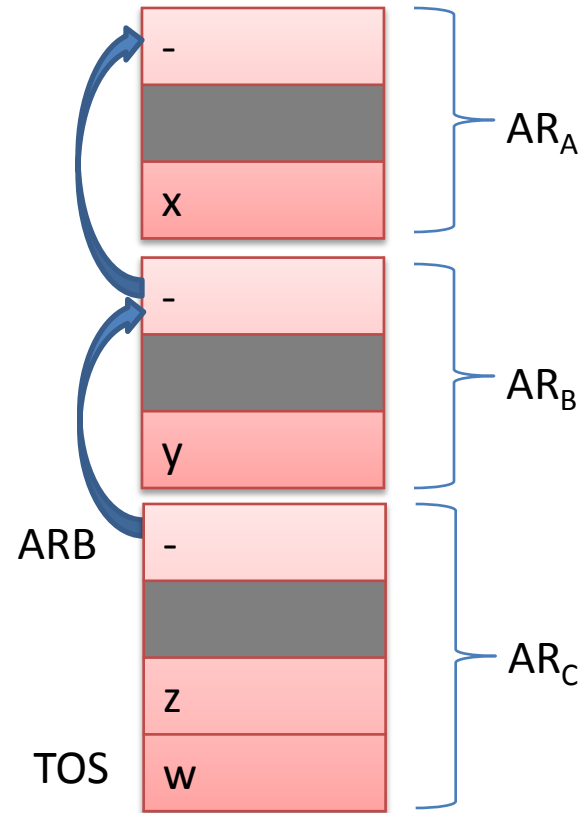
- $n1_var$: is a non local variable
- b_def : block defined into
- b_use : block in use
- Then textual ancestor of block b_use is a block which encloses block b_use that is b_def .
- Level m ancestor is a block which immediately encloses level $m-1$ ancestor.
- $S_nest_b_use$: static nesting level of block b_use .
- Rule: when b_use is in execution, b_def must be active.
- i.e $ARBb_def$ exist in stack while execution $ARBb_use$.
- $n1_var$ are accessed by start address of
 $ARB_def + dn1_var$
where, $dn1_var$ is displacement of $n1_var$ in ARB_def .
- Lets learn few more terms
 - Static Pointers
 - Display

(i) Static Pointer:

- Access of non-local variables is implemented using second reserved pointer in AR.
- It has
 - 1 (ARB)
 - 0 (ARB)
- 1 (ARB) is static pointer.
- At the time of creation of AR for Block B its static pointer is set to point AR of static ancestor of b.
- Access non-local variables:
 1. $r := \text{ARB};$
 2. Repeat step-3 m times
 3. $r := 1(r)$
 4. Access n1_var using address $\langle r \rangle + \text{dn1_var}$.
- Example: Status after execution:
 - $\text{TOS} := \text{TOS} + 1$
 - $\text{TOS}^* := \text{address of AR at level 1 ancestor.}$
 - $\langle r \rangle$ to access x at statement $x := z$
 - $r := \text{ARB};$
 - $r := 1(r);$
 - $r := 1(r);$
 - Access x using address $\langle r \rangle + dx$.

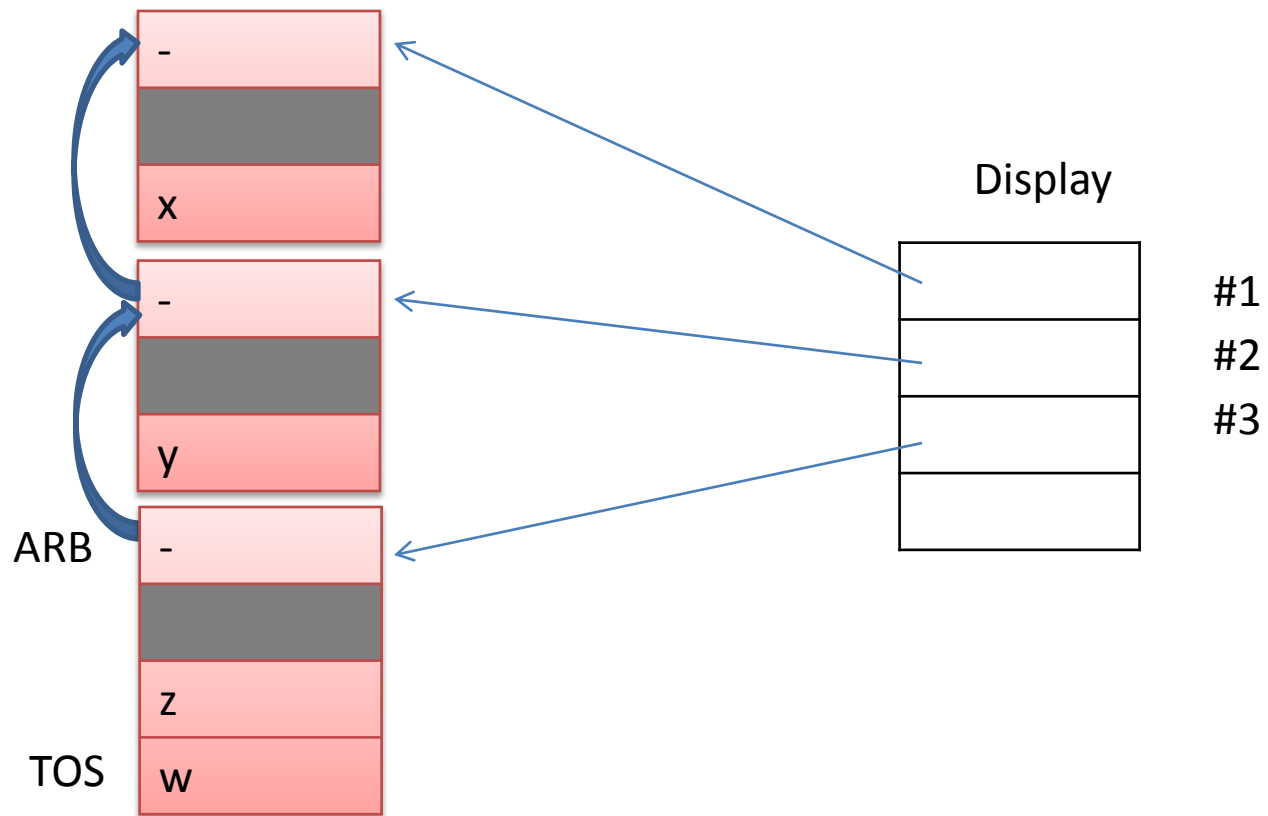
Dynamic
Pointers

Static
Poiners



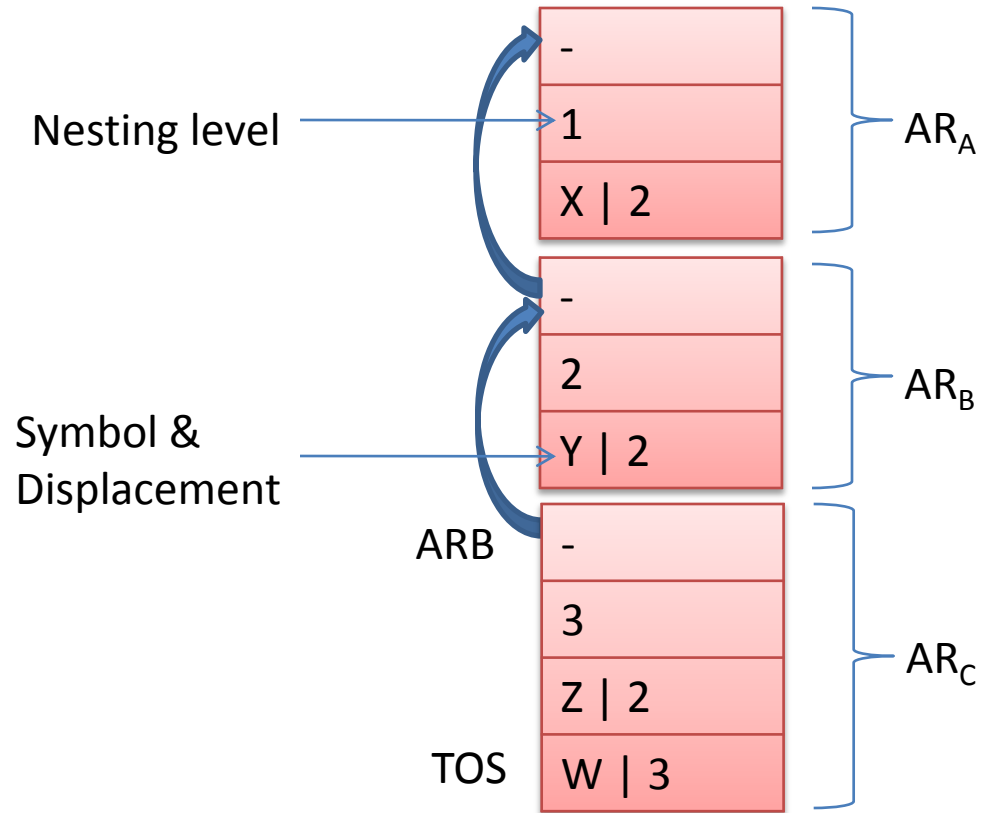
(ii) Displays:

- Display (1) = address of level (S_nest_b-1) ancestor of B.
- Display (2) = address of level (S_nest_b-2) ancestor of B.
- Display [S_nest_b-1] = address of level 1 ancestor of B.
- Display [S_nest_b-1] = address of AR_B .
- For large value of level difference, it is expensive to access non-local variables using static pointers.
- Display is an array used to improve the efficiency of non-local variables accessibility.
- Eg: $r := display[1]$
access x using address $\langle r \rangle + dx$.



4. Symbol Table Requirement:

- To improve dynamic allocation & access, compiler should perform following task.
 - Determine static nesting level of `b_current`
 - Determine variable designated with scope rules.
 - Determine static nesting level of block & `dv`.
 - Generate code.
- Extended stack model is used bcz it has
 - Nesting level of `b_current`
 - Symbol table for `b_current`.
- Symbol table has
 - Symbol
 - Displacement.



5. Recursion:

- Extended stack model best for recursion.
- See program and figure in book pg. no. 175 and 176.

6. Limitations of stack based memory allocation:

- Not good for program controlled memory allocation.
- Not adequate for multi-active programs.

[C] Array allocation & Access

- $A[5,10]$. 2D array arranged column wise.
- Lower bound is 1.
- Address of element $A[s1,s2]$ is determined by formula:
- $Ad.A[s1,s2] = Ad.A[1,1] + \{ (s2-1) \times n + (s1-1) \} \times k$.
- Where,
 - n is number of rows.
 - k is size, number of words required by each element.
- General 2D array can be represented by $a[l1:u1,l2:u2]$
- The formula becomes
 $Ad.A[S1,S2] = Ad.A[l1,l2] + \{ (s2-l2) \times (u1-l1+1) + (s1-l1) \} \times k$.
- Defining the range:
 - Range1: $u1-l1+1$
 - Range2: $u2-l2+1$
- $Ad.A[s1,s2]$
 - $= Ad.A[l1,l2] + \{ (s2-l2) \times \text{range1} + (s1-l1) \} \times k$.
 - $= Ad.A[l1,l2] - (l2 \times \text{range1} + l1) \times k + (s2 \times \text{range1} + s1) \times k$.
 - $= Ad.A[0,0] + (s1 \times \text{range1} + s1) \times k$.
- We can compute following values at compilation time or allocation time (given that subscripts are constants).

A[1,1]
-
-
A[5,1]
A[1,2]
-
A[5,2]
-
-
A[1,10]
-
A[5,10]

Dope Vector:

- Definition: Dope vector is an descriptor vector
 - Accommodated in symbol table
 - Used by generation of code phase.
- No. of dimension of array determine format & size of DV.
- Code generation use d.Dv to check validity of subscripts.
- See figure 6.10 pg. no. 178.
- The code for array reference on pg. no. 179.
- See Example 6.15 on pg. no. 179 and memory allocation made for this example on pg. no. 180 fig. 6.12.

Compiler of Expression

Topic List

A. A Toy generator for expression

- a) Operand Descriptor
- b) Register Descriptor
- c) Generating an Instruction
- d) Saving Partial Results

B. Intermediate code for expression

- a) Postfix string
- b) Triples & Quadruples
- c) Expression Trees

[A] A Toy Generator for Expressions

- Major issues in code generation for expression:
 - Determination of execution order for operators.
 - Selection of instruction used in target code.
 - Use of registers and handling partial results.

a) Operand Descriptor:

- Attributes:
- Addressability:
- Specifies:
 - Where operand is located
 - How it can be accessed.
- Addressability Code:
 - M : operand is in memory
 - R : operand is in register
 - AR : address is in register
 - AM : address is in memory
- Address:
 - Address of CPU register or memory

Type	Length	Miscellaneous
------	--------	---------------

Addressability Code	Address
---------------------	---------

- Operand descriptor is build for every operand that is id's, constant, partial results.
 - PRi :- Partial Results
 - Opj :- Some operator
- Operand descriptor is an array in which operand descriptions are stored.
- Descriptor # is a descriptor in operand descriptor array.
- Eg: code generation for $a * b$

```

      MOVER      AREG, A
      MULT      AREG, B
    
```
- See the skeleton code from book 6.13, pg. 182.

(int, 1)	M, addr (a)	Descriptor for a
(int, 1)	M, addr (b)	Descriptor for b
(int, 1)	R, addr (AREG)	Descriptor for a * b

b) Register Descriptor

- It has 2 fields:
 - Status: free / occupied
 - Operand Descriptor #
- Stored in register descriptor array.
- One register descriptor exist for each CPU register.
- Register Descriptor for AREG, after generating code for $a*b$.

Occupied	#3
----------	----

c) Generating on Instruction

- Rule: any one operand need to be in register to perform any operation over it.
- If not, one of the operand is brought to register.
- Function codegen is called with OP; and descriptors of its operands as parameters.

d) Saving Partial Results

- If all the registers are occupied, register are freed by transferring content of temporary location in memory.
- r is available to evaluate operator $O P_i$.
- Thus, 'temp' array is declared in target program to hold partial results.
- Descriptor of partial result must change/modify when partial result are moved to temporary location.
- After partial result $a*b$ is moved to temp location.
- Eg: $a*b$
 1. int,1 m,addr(a)
 2. int,1 m,addr(b)
 3. int,1 m,addr(temp[1])
- Code generation routine on pg. no. 184 to 187. just have a look.

[B] Intermediate Codes for Expressions

- a) Postfix string
- b) Triples & Quadruples
- c) Expression Trees

a) Postfix Strings

- Here each operands appear immediately after its last operand.
- Thus, operators can be evaluated in order in which they appear in string.
- Eg: $a+b*c+d*e^f$
 $abc*+def^*+$
- We perform code generation from postfix string using stack of operand descriptors.
- Operand appears and then operand descriptors are pushed to the stack. i.e stack would contain descriptor for a, b & c when first * is encountered.
- Little modification in extended stack model can manage postfix string efficiently.

b) Triples & Quadruples

1) Triples:

Operator	Operand 1	Operand 2
----------	-----------	-----------

- Triple is a representation of elementary operations in the form of a pseudo machine instruction.
- Slight change in algorithm of operator precedence help us convert infix string to triples.
- See figure 6.19 on pg.no. 189 for the expression $a+b*c+d*e^f$.

2) Indirect Triples:

- Are useful in optimizing compiler.
- This arrangement is useful to detect the identical expression occurrence in the program.
- For efficiency Hash Organization can be used for the table of triples.
- Indirect triple representation provide memory economy.
- Aid: Common sub expression elimination.
- See figure 6.20 and example 6.22 on pg.no.189.

3) Quadruple:

- Result name: designates result of evaluation that can be used as operand for other quadruple.
- More convenient than using triples & indirect triples.
- For example : $a + b * c + d * e^f$
- See figure 6.21 on pg. no. 190.
- Remember, they are not temporary locations (t) but result name.
- For elimination, result name can become temporary location.

Operator	Operand 1	Operand 2	Result name
----------	-----------	-----------	-------------

c) Expression Tree

- Operator are evaluated in order determined by bottom up parsing which is not most efficient.
- Hence, compiler's back-end analyze expression to find best evaluation order.
- What will help us here?
- Expression Tree: is a AST (Abstract Syntax Tree) which depicts the structure of an expression.
- Thus, simplifying analysis of an expression to determine best evaluation order.
- Eg: 6.24 pg.no. 190.
- How to determine best evaluation order for the expression?
 - Step 1: Register Requirement Label (RR Label) indicates no. of register required by CPU to evaluate sub-tree.
 - Step 2: Top Down parsing and RR Label information is used and order of evaluation is determined.
 - See the algorithm on pg.no. 191.
 - Example 6.25 and expression tree on pg.no. 191 and 192.

COMPILATION OF CONTROL STRUCTURES

Compilation of Control Structure

Topic List:

- A. Control Transfer, Conditional Execution & Iterative Constructs
- B. Function & Procedure Calls
- C. Calling Conventions
- D. Parameter Passing Mechanism
 - a) Call by Value
 - b) Call by Value Result
 - c) Call by Reference
 - d) Call by Name

Definition: Control Structure

- Control structure of a programming language is the collection of language features which govern the sequencing of control through a program.
- Control structure of PL construct for
 - Control Transfer
 - Conditional Execution
 - Iterative Construct
 - Procedure Call

[A] Control Transfer, Conditional Execution & Iterative Construct

- Control transfer is implemented through conditional & un-conditional 'goto' statements are the most primitive control structure.
- Control structure like if, for or while cause semantic gap b/w PL domain & execution domain.
- Why? Control transfer are implicit rather than explicit.
- This semantic gap is bridge in two steps:
 - Control structure is mapped into goto.
 - Then this programs are translated to assembly program.
- See Fig. 6.25 and Example 6.26 on pg.no. 193 and 194.

[B] Function & Procedure Call

$x := \text{fn-1}(y, z) + b * c;$

- In the given statement function call on fn-1 is made after execution returns the value to calling function.
- This may result to some side effects.
- Def: Side Effect:
Side effect of a function (procedure) call is a change in the value of a variable which is not local to the called function (procedure).
- A procedure call only achieves side effect, it doesn't return a value.
- A function call achieves side effect also returns same value.
- Compiler must ensure following for function call:
 - 1. Actual (y,z) parameters are accessible in called function.
 - 2. Called function is able to produce side effect according to rules of PL.
 - 3. Control is transferred to, and is returned from the called function.
 - 4. Function value is returned to calling program.
 - 5. All other aspects of execution of calling program are unaffected by function call.

- Compiler uses set of features to implement functions:
 - Parameter List: contains descriptor for each parameter.
 - Save Area: Register $\leftarrow \rightarrow$ Save Area
 - Calling Conventions: there are few execution time assumptions:
 - i. How parameter list is accessed?
 - ii. How save area is accessed?
 - iii. How transfer of control at call & return are implemented?
 - iv. How function value is returned to calling program.
 - (iii) & (iv) are performed by machine instruction or CPU?
 - Others are assumptions they are not achieved by CPU or machine instruction.

[C] Calling Conventions

Static Calling Convention

- Static memory allocation & environment.
- Parameter list & save area are allocated in calling program.
- Calling conventions required.
Address of function, parameter list and save area to be contained in specific CPU registers at call.
- During execution Dp has the address: $\langle r_par_list \rangle + (dDp)par_list$.
Where, r_par_list is register containing address of parameter, dDp is displacement of Dp in parameter list.
- Eg: pg no. 195 example 6.29

Dynamic Calling Convention

- Dynamic memory allocation & Environment.
- Calling program construct parameter list and saved area using stack.
- This becomes part of called function's AR when execution is initiated.
- During execution Dp has address: $\langle ARB \rangle + (dDp)AR$.
Where, ARB is start of parameter list and $(eDp)AR$ is computed and stored in symbol table.
- Eg: pg.no. 196 example same.

[D] Parameter Passing Mechanism

- Define semantics of parameter usage inside a function.
- Thus, defines the kind of side effects, a function can produce on its actual parameter.
- Types:
 - Call by value
 - Call by value result
 - Call by reference
 - Call by name

1. Call by Value

- Actual parameters are passed to called function.
- These values are assigned to corresponding formal parameters.
- Values are passed in 'one direction'. i.e from calling program to called program.
- If function changes value of formal parameter, changes are not reflected on actual parameter.
- Thus, can't produce any side effect on parameters.
- Generally used in built in function.
- Advantage:
 - Simplicity
 - Efficient if parameters are scalar variables.

2. Call by Value Result

- Extends capability of call by value.
- Copies values of formal parameter back to corresponding actual parameter at return.
- So, side effects are reflected at return.
- Advantage:
 - Simplicity
- Disadvantage:
 - Incurs higher overhead.

3. Call by Reference

- Address of actual parameter is passed to called function.
- Parameter list is actually the list of addresses.
- At every access, corresponding actual parameter is obtained from parameter list.
- Code:
 - 1. $r \leftarrow \langle \text{ARB} \rangle + (\text{dDp})\text{AR}$ or
 $r \leftarrow \langle \text{r_par_list} \rangle + (\text{dDp})\text{par_list}$
 - 2. Access value using address contained in register.
- Code analysis:
 - Step 1: incurs overhead
 - Step 2: produces instantaneous side effects.
- Mechanism is popular because has clear semantics.
- Plays important role at the time of nesting of structures.
- How? It provides updated value.
- See eg. 6.29 on pg. no. 197.
- Here, z, i are non local variables of α .
- α be called as $(d[i], x)$.
- Value of 'x' changes as 'b' also changes.

4. Call by Name

- Same effect as call by reference.
- Every occurrence of formal parameters in the called function is replaced by the name of the corresponding actual parameter.
- Eg:

```
a = d[i];  
z = d[i];  
i = i + 1;  
-b = d[i] + 5;  
x = d[i] + 5;
```
- Achieves instantaneous side effects.
- Has implication of changes in parameters during execution.
- Code:
 - 1. `r <- <ARB> + (dDp)AR`
 - `r <- <r_par_list> + (dDp)par_list`
 - 2. Call the function whose address is contained in r.
 - 3. Use the address returned by the function to access p.
- Advantage:
 - Changes are made dynamically which makes call by name mechanism immensely powerful.
- Dis-advantage:
 - High overhead at step 2.
 - Not much practically practiced.

Parameter Passing Mechanism supported by Language

Language

1. C
2. Pascal
3. Fortran
4. PL/I
5. ADA
6. Algol-60

Uses

1. Call by value
2. Call by value
Call by reference
3. Call by reference
4. Call by reference
5. Call by value result
Call by reference
6. Call by value
Call by name

CODE OPTIMIZATION

Code Optimization: Topic List

A. Optimizing Transformation

- a) Compile Time Evaluation
- b) Combination of Common Sub-expression
- c) Dead Code Elimination
- d) Frequency Reduction
- e) Strength Reduction
- f) Local & Global Optimization

B. Local Optimization

- a) Value Numbers

C. Global Optimization

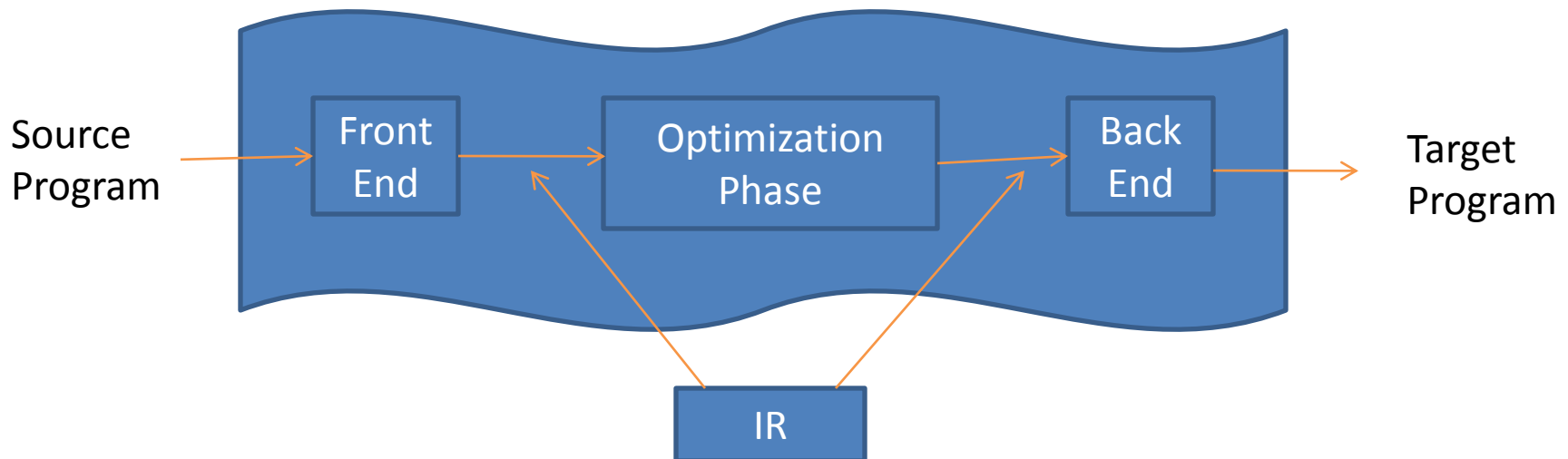
- a) Program Representation
- b) Control & Data Flow Analysis

Code Optimization: Introduction

- Aim: Improving the execution efficiency of program.
- How to fulfill this aim?
 - Eliminating Redundancy
 - Re-arrangement and re-writing program computation to execute it efficiently.
- Axiomatic: Code optimization must not change the meaning of the program.
- Scope of Optimization:
 - Improves program rather than improving algorithm.
 - Not possible to generate efficient code for specific target machine.
- Thus, optimization techniques are independent of both PL and target machine.

Schematic of Optimizing Compiler:

- Eg: IBM 1360 system, optimizing compiler for fortran 'H':
- Consumes 40% extra compilation time due to optimization.
- Occupy 25% less storage.
- Executes three times faster.



[A] Optimizing Transformation:

- Optimization Transformation is a rule for 're-writing' a segment of a program.
- Improve execution efficiency without affecting its meaning.
- Types:
 - Local (Small segments)
 - Global (Entire Segments)
- What is the need of this classification?
 - Reason is difference in cost.
 - Benefits of optimization transformation
 - What is needed is provided, not less, not more.
- Lets see few Optimizing transformations:

a) Compile Time Evaluation

- Certain actions are performed at compile time, certain at execution time.
- This distribution improves execution efficiency as certain actions are eliminated at execution.
- This is called 'constant folding'.
- 'Constant Folding':
 - When all operands are constant.
 - Perform operation at compilation
 - Result is also constant
 - Thus, can be replaced by original evaluation.
- Thus, we eliminate division operation at time of execution.
- Eg: $a = 3.14157/2$ is replaced by $a = 1.570785$

b) Elimination of Common Sub-Expression

- See ex. 6.31
- Common sub expression are occurrences of expressions yielding same value, called 'equivalent expression'.
- Second occurrence $b*c$ can be eliminated.
- They are identified by using triples & quadruples.
- Some compilers use rule of 'algebraic equivalence' in common sub-expression elimination.
- Advantages: Improves effectiveness of optimization.
- Dis-advantages: Also increase cost of optimization.

c) Dead Code

- The code which can be omitted from a program without affecting its results is called dead code.
- Now question is how to detect or check whether it is a dead code or not.
- By checking whether the value assigned is an assignment statement which is used anywhere in the program or not.
- If not, it's a dead code.
- Eg: `x:=<exp>`
- Has dead code if value, assigned to x is not used anywhere in the program.
- i.e Expression constitutes dead code only if it is not producing any side effects.

d) Frequency Reduction

- Eg: 6.33
- Those who are independent of 'for loop' is called loop invariant.
- Here, x is a loop invariant, which has been moved out of loop to perform frequency reduction.
- Y is indirectly dependent of loop. i.e z,i ;
- So, frequency reduction is not possible.
- Thus, transformation of loop optimization moves loop invariant code out of loop and places it prior to loop entry.

e) Strength Reduction

- Strength reduction optimization replaces the occurrence of a time consuming operation (also called 'high strength' operation) by n occurrence of a faster operation (also called 'low strength' operation).
- Example 6.34
- Here, we are replacing multiplication operation with addition.
- Beneficial in array reference.
- This results in strength reduction.
- Dis-advantage: Not recommended for floating point operands. Reason, it doesn't guarantee equivalence of result.

f) Local & Global Optimization

- Two phases:
 - Local
 - Global
- Local Optimization: applied over small segments consisting of few statements.
- Global Optimization: applied over a program unit over function or procedure.
- Local optimization is preparatory phase for global optimization.
- Local optimization simplifies certain aspects of global optimization.
- Global optimization eliminates only first occurrence of $a+b$, all other occurrences will eliminate automatically with local optimization.

[B] Local Optimization

- Provides limited benefits at low cost.
- Scope? Basic block which is essentially sequential segment in a program.
- Cost is low. Why?
 - Sequential Nature
 - Simplified Analysis
 - Applied to basic block.
- Limitations? Loop optimization is beyond the scope of local optimization.
- See def of basic block in book, pg no. 203.
- Is a single entry point.
- Essentially sequential.

Value Number:

- Provides simple means to determine whether two occurrence of an expression in a basic block are equivalent or not.
- This technique is applied while identifying the basic block.
- Steps / Conditions for value numbers:
 - Two expression e_i and e_j are equivalent if they are congruent and their operands have same value number.
- See eg. 6.35 and 6.36 pg. 204
- Starting of variable is 0.
- Value no is considered only when operation need to be performed over variables.
- Flag checks to see if value needs to be stored in temporary location.
- This semantic can be extended to implement “constant propagation”.
- See eg. 6.37 on pg. no. 206
- Last value is treated as constant and quadruple for $b=81.9$ is generated.
- This leads to possibility of constant propagation and folding of last statement.

[C] Global Optimization

- Require more analytic efforts to establish the feasibility of an optimization.
- Global common sub expression elimination is done here.
- Occurrence can be eliminated if it **satisfy two condition**:
 - 1. Basic Block b_j is executed only after some block $b_k \in SB$ has been executed one or more times (Ensure $x*y$ is evaluated before b_j)
 - 2. No assignment to x or y have been executed after the last (or only) evaluation of $x*y$ block b_j .
- $x*y$ is saved to temporary location in all block b_{12} satisfying condition 1.
- Requirement? Ensure that every possible execution of program satisfy both the condition.
- How we would do this?
- By analysing program using two techniques:
 - Control Flow Analysis
 - Data Flow Analysis
- Before that, let us see program representation which is done in the form of PFG.

PFG: Program Flow Graph

- Def: A PFG for a program P is directed graph $G_p = (N, E, n_0)$
- Where,
 - N : set of blocks
 - E : set of directed edges (b_i, b_j) indicating the possibility of control flow from the last statement of b_i (source node) to first statement of b_j (destination node).
 - n_0 : start node of program.

Control & Data Flow Analysis

- **Control & Data Flow Analysis:** Used to determine whether the condition governing and optimizing transformation are satisfied or not.
- **1. Control Flow Analysis:** Collects information concerning its structure i.e nesting of loops.
- **Few concepts:**
 - **Predecessors & Successor:-** If $(b_i, b_j) \in E$, b_i is a predecessor of b_j & b_j is a successor of b_i .
 - **Paths:-** A path is a sequence of edges such that destination node of one edge is the source node of the following edge.
 - **Ancestors & Descendants :-** If path exist from b_i to b_j , b_i is an ancestor of b_j and b_j is a descendant of b_i .
 - **Dominators & Post Dominators:-** Block b_i is a dominator of block b_j if every path from n_0 to b_j is passed through b_i . And b_i is the post dominator of b_j if every path from b_j to an exit node passes through b_i .
- **Advantages:** Control flow concepts answer the question of condition 1.
- **Drawbacks:** Incurs overhead at every expression evaluation.
- **Solution?** Restrict the scope of optimization.

2. Data Flow Analysis:

- Analyse the use of data in the program.
- Data flow information is computed for the purpose of optimization at entry & exit of each basic block.
- Determines whether optimization transformation can be applied or not.
- Concepts:
 - Available Expression
 - Live Variables
 - Reaching Definition
- Use:
 - Common sub expression elimination.
 - Dead code elimination
 - Constant variable propagation

Available Expression:

- Occurrence of global common sub expression can be eliminated only if
 - 1. Condition 1 & 2 are satisfied at entry of b_i .
 - 2. No assignment to x or y precedes the occurrence of $x*y$ in b_i .
- How to determine availability of an expression at entry or exit of basic block b_i ?
- Rules:
 - 1. Expression e is available at the exit of b_i if
 - (i) b_i contains evaluation of e not followed by assignment to any operand of e , or
 - (ii) value of e is available at the entry to b_i & b_i doesn't contain assignment to any operand of e .
 - 2. Expression e is available at entry to b_i if it is available at exit of each predecessor of b_i in G_p .

- It is forward data flow concept
- Availability at exit of node determines availability at entry of successor.
- We associate two Boolean properties with block b_i to determine the effect of computation called 'local properties' of block b_i .
- Eval i:- 'True' if e is evaluated in B_i and operands of e are not modified.
- Modify :- 'True' if operand of e is modified in b_i .
- See page no. 209 equations and pg. No. 210 PFG Example 6.38.

Live Variables:

- Variable Var is said to be live, at a program point P_i basic block b_i if the value contained in it at P_i is likely to be used during subsequent execution of program.
- Otherwise, its a dead code which can be eliminated.
- How to determine liveness? By 2 property specified on pg. No. 211.
- See data flow information again on pg. No. 211.
- Availability at entry of block determines availability at exit of its predecessor.
- Hence called “Backward Data Flow” concept.
- Also called “any path concept”.
- Why?
- Liveness of an entry at successor is sufficient to ensure, liveness at exit of block.
- Eg:
 - a is live at entry of each block except fourth block.
 - B is live at all block .
 - X & Y are live at 1,5,6,7,8,9

INTERPRETERS

Interpreters

- Topic List
 - Interpreters – use
 - Overview of Interpreters
 - Toy Interpreter
 - Pure & Impure Interpreters

Interpreters : Introduction

- Avoid overhead of compilation
- Modification at every execution is managed by interpreters.
- Dis-advantage: Expensive in terms of CPU time.
- Why? Each statement is subject to follow interpreters cycle.
- Cycle:
 - Fetch the Instruction
 - Analyse statement & Determine meaning
 - Execute the meaning of statement
- What is the difference between compiler and interpreter.

Compiler v/s Interpreter

Compiler

- **Next Phase:** During compilation analysis of statement is followed by code generation.
- **.exe** : Compiler convert into exe only once.
- **Development:** One time infrastructure development.
- **Rate of Development:** Slow
- **Access Rate:** Faster access at later stage.
- Error: Not easy to solve & debug errors.

Interpreter

- During interpretation analysis is followed by actions for implementation.
- We can never run a program without interpretation.
- Repetitive development.
- Rate of development: faster.
- Slower access rate.
- Error can be easily solved.

Compiler

- **Best for**: static languages
- **Required at**: only one time compiler is required.
- **Cost**: proved cheaper at longer run.
- **Loading**: Compiler is loaded only first time.

Interpreter

- Best for dynamic languages.
- Each time program is executed interpreter is required.
- Proved costly at language run.
- Interpreter is needed at each load.

Interpreter : Introduction

- Notation:
 - T_c : Average Compilation time per statement
 - T_e : Average Execution time per statement
 - T_i : Average Interpretation time per statement
- Here we assure $T_c = T_i.T_e$
- $SizeP$ = number of statements in program P.
- $Stmt_ExecutedP$ = number of statements executed in Program P.

Example:

- Let,
 - Size = 200
 - 20 statements are executed
 - Loop has 10 iterations
 - Loop has 8 instructions in it
 - 20 stmts are followed by loop for printing result.
- Then, $\text{stmt_executedP} = 20 + (10 * 8) + 20$
 $= 120.$
- Total Execution Time:
 - For compilation model:
 - $200 \cdot T_c + 120 \cdot T_e$
 - $= 206 \cdot T_c$
 - For interpretation model:
 - $120 \cdot T_c$
 - $120 \cdot T_c$
- Conclusion: Clearly interpretation is better than compilation as far as execution time is concerned.

Why Use Interpreter?

- Simplicity
- Efficiency & Certain environmental benefits.
- If required modification, recommended when stmt execution is less than size P.
- Preferred during program development.
- Also when programs are not executed frequently / repeatedly.
- Best for writing programs for an editor, user interface, OS development.

Components:

- 1. Symbol Table: Holds information concerning entities present in program.
- 2. Data Store: Contains values of data items declared.
- 3. Data Manipulation Routines: A set containing a routine for every legal data manipulation actions in the source language.
- Advantages:
 - Meaning of source statement is implemented using interpretation routine which results to simplified implementation.
 - Avoid generation of machine language instruction.
 - Helps make interpretation portable.
 - Interpreter itself is coded in high level language.

A Toy Compiler:

- Steps:
 - Ivar[5] = 7;
 - Rvar [13] = 1.2;
 - Add is called (for a=b+c)
 - Addrealint is called.
 - Rvar[r_tos] = rvar[13] + ivar[5];
 - Type conversion is made by interpreters
 - Rvar[addr1] + ivar[addr2]
 - End.
- See program of interpreter on pg. No. 125.
- See example for given above steps on pg. No. 126

Pure & Impure Interpreters:

- **Pure Interpreters:**

- Here, source program is retained in source form all through interpretation.
- Dis-advantage: This arrangement incurs substantial analysis overheads while interpreting the statement.
- Eliminates most of the analysis during interpretation except type analysis.
- For type analysis pre processor is needed.
- See fig. 6.34 (a). Pg. No. 217
- See Ex. 6.42. IC intermediate code for postfix notation.

- **Impure Interpreters:**

- See fig. 6.34 (b). Pg. no. 217.
- See Ex. 6.43. IC intermediate code for postfix notation.
- Perform some preliminary processing of the source program to reduce the analysis overhead during interpretation.
- Pre-processor converts program to an IR which is used during interpretation.
- IC can be analysed more efficiently than source program.
- Thus, speed up interpretation.
- Dis-advantage: Use of IR implies that entire program has to be pre-processed after any modifications.
- Thus, incurs fixed overhead at the start of interpretation.
- Conclusion Statement: Thus, eliminates most of the analysis during interpretation.

6th Chapter Ends Here.

- Please start preparing.
- U will not get 1 month study leave.
- Start preparation Now.
- Every day complete 1 chapter.
- Chapter 1, 3, 4 from class work.
- Chapter 5, 6, 7, 8 and Unit 6 from slides.
- From tomorrow we would start chapter 7.
- THANK YOU..... 😊 😊 😊 😊