A quality product by

Brainheaters™ LLC

Brainheaters Notes

SS | Computer Semester-6

SERIES 313 - 2018 (A.Y 2020 - 21)

BH.Index

(Learn as per the Priority to prepare smartly)

MODULE-1

Q1. What is the difference between System Software and Application software? (P4 - Appeared 1 Time) (5-10M)

Ans: **System software and application software:**

Sr No	System Software	Application Software
1.	System software is used for operating computer hardware.	Application software is used by users to perform specific tasks.
2.	System softwares are installed on the computer when the operating system is installed.	Application softwares are installed according to the user's requirements.
3.	In general, the user does not interact with system software because it works in the background.	In general, the user interacts with application softwares.

4.	System software can run independently. It provides a platform for running application softwares.	Application software can't run independently. They can't run without the presence of system software.
5.	Some examples of system softwares are compiler, assembler, debugger, driver, etc.	Some examples of application softwares are word processors, web browsers, media players, etc.

Q2. Describe following data structures: OPTAB, SYMTAB, LITTAB and POOLTAB. (P4 - Appeared 1 Time) (5-10M)

Ans : **OPTAB**

- Each entry in the operation table contains fields like mnemonic , class and mnemonic info.
- The class field many contain . IS (Imperative Statements), AD (Assembler Directives) or DL (Declarative Statements)

Symbol Table (SYMTAB)

- Each entry in the symbol table has two primary fields: name and address.

- The symbol table uses the concept of forward reference to achieve the address of symbols.

Literal Table (LITTAB)

- Literal table contains all literals and address of all literals. For literals
- LTORG directives place the all constants at consecutive memory locations. If we are not using LTORG then all literals are placed after END in memory

Pool Table (POOLTAB)

- Awareness of different literal pools is maintained using the auxiliary table POOLTAB. At any Stage , the current literal pool is the last pool in LITTAB.

Q3. Explain the following terms with suitable examples. 1.

Expansion time variable 2. Positional parameter. (P4 - Appeared 1 Time) (5-10M)

Ans: **Expansion time variable:**

- Expansion time variables are variables which can only be used for the expansion of macro calls.
- Local EV is created for use only during a particular macro call.
- Global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.
- <EV specification > has syntax & <EV name>, where EV name is an ordinary string.
- Example:

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

Positional parameters:

- A positional formal parameter is written as &<parameter name>.The <actual parameter spec> in call on a macro using positional parameter is simply an<ordinary string>.
    - Step-1: find the ordinary position of XYZ in the list of formal parameters in the macron prototype statement.
    - Step-2 : finds the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement.
    - Examples- INCR A, B, areg
- The rule of positional association values of the formal parameters are

Formal parameter	**value**
MEM_VAL	A
INCR_VAL	B
REG	AREG

Q4. Explain with examples - expansion time variables, expansion time Statements - AIF and AGO for macro programming. Show their usage for the expansion time loop by giving examples. (P4 - Appeared 1 Time) (5-10M)

Ans : **Expansion time variable:**

- Expansion time variables are variables which can only be used for the expansion of macro calls.
- Local EV is created for use only during a particular macro call.
- Global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.
- LVL<EV specification>[,<EV specification>...]
- GBL<EV specification>[,<EV specification>...]
- <EV specification > has syntax & <EV name>, where EV name is an ordinary string.
- Initialize EV by preprocessor statement SET.
  <EV specification >SET <SET-expression>
- Example:

```
        MACRO
        CONSTANTS
        LCL     &A
&A      SET     1
        DB      &A
&A      SET     &A+1
        DB      &A
        MEND
```

  - A call on macro CONSTANTS is expanded as follows: The local EV A is created. The first SET statement assigns the value '1' to

it. The first DB statement thus declares a byte constant '1'. The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

AIF Statement

- It is used to specify the branching condition.
- This statement provides a conditional branching facility.
- Syntax : AIF (condition) Label
- If the condition is satisfied the label is executed else it continues with the next execution.
- It performs an arithmetic test and branches only if the tested condition is true.

AGO Statement

- This statement provides unconditional branching facilities.
- We do not specify the condition
- Syntax: AGO. Label
- It specifies the label appearing on some other statement in the macroinstruction definition.
- The macro processor continues the sequential processing of instructions with the indicated statement.
- These statements are directives to the macro processor and do not appear in a macro expansion.

it. The first DB statement thus declares a byte constant '1'. The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

AIF Statement

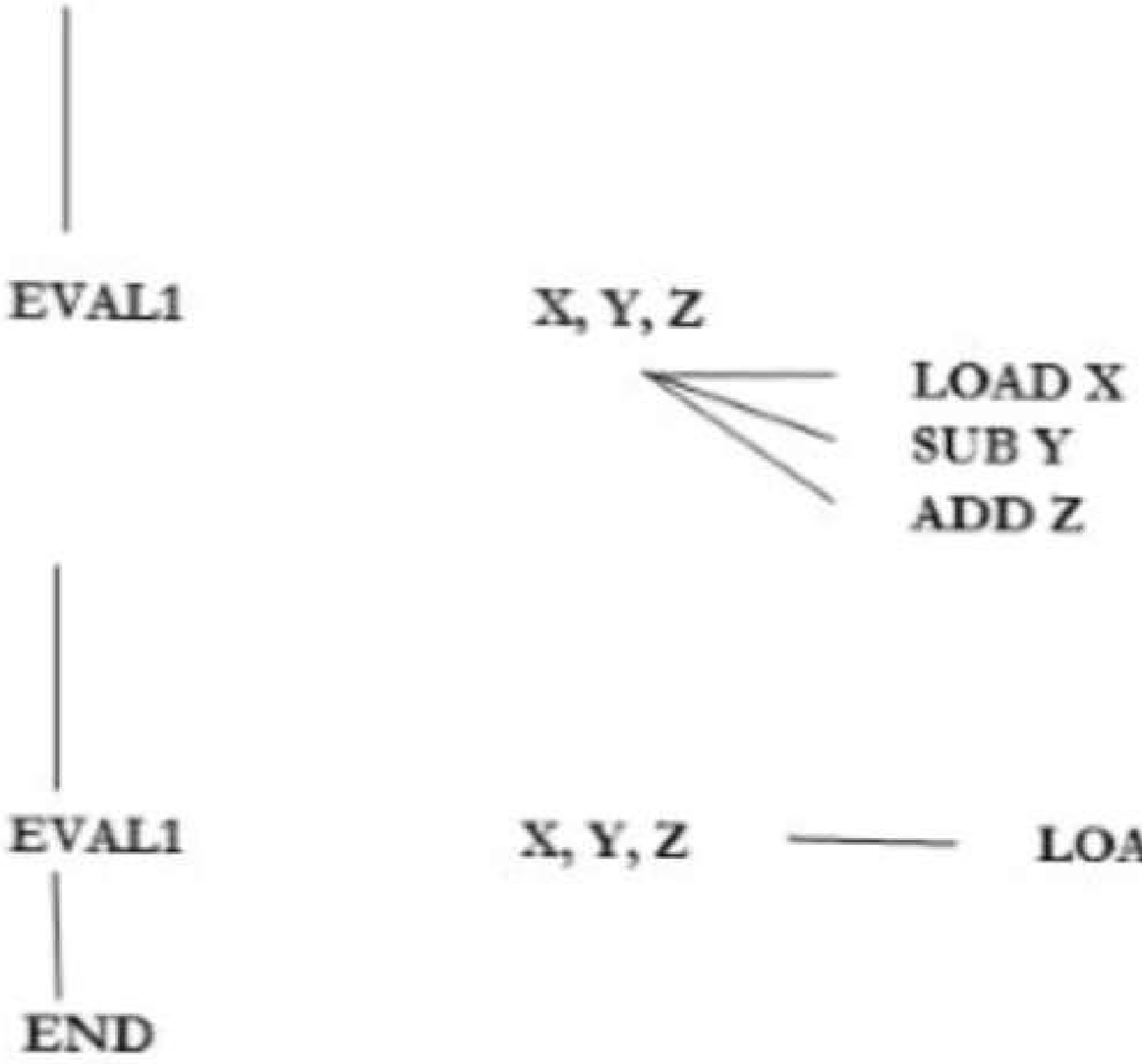
- It is used to specify the branching condition.
- This statement provides a conditional branching facility.
- Syntax : AIF (condition) Label
- If the condition is satisfied the label is executed else it continues with the next execution.
- It performs an arithmetic test and branches only if the tested condition is true.

AGO Statement

- This statement provides unconditional branching facilities.
- We do not specify the condition
- Syntax: AGO. Label
- It specifies the label appearing on some other statement in the macroinstruction definition.
- The macro processor continues the sequential processing of instructions with the indicated statement.
- These statements are directives to the macro processor and do not appear in a macro expansion.

Example:

```
MACRO
EVAL1       &A, &B
AIF (&A EQ &B).NEXT
LOAD        &A
SUB         &B
ADD         &C
AGO         .FIN
.NEXT       LOAD &C
.FIN        MEND
```



Q5. Define the following terms: 1. System Software 2. Semantic gap 3. Specification gap 4. Execution gap (P4- Appeared 1 time) (3-7 marks)

Ans: **System software:**

- System Software can be designed as the software in such a way so that it can control and work with computer hardware. It acts as an interface between the device and the end-user. It also provides the platform for the running of other software.

- Example: operating systems, antivirus software etc.

Semantic gap:

- The semantic gap is defined as the difference between high-level programming sets in various computer languages, and the simple computing instructions that microprocessors work within machine language.

Specification gap:

- It is the semantic gap between the application domain and the PL domain. It can also be defined as the semantic gap between the two specifications of the same task. The specification gap is bridged by the software development team.

Execution gap:

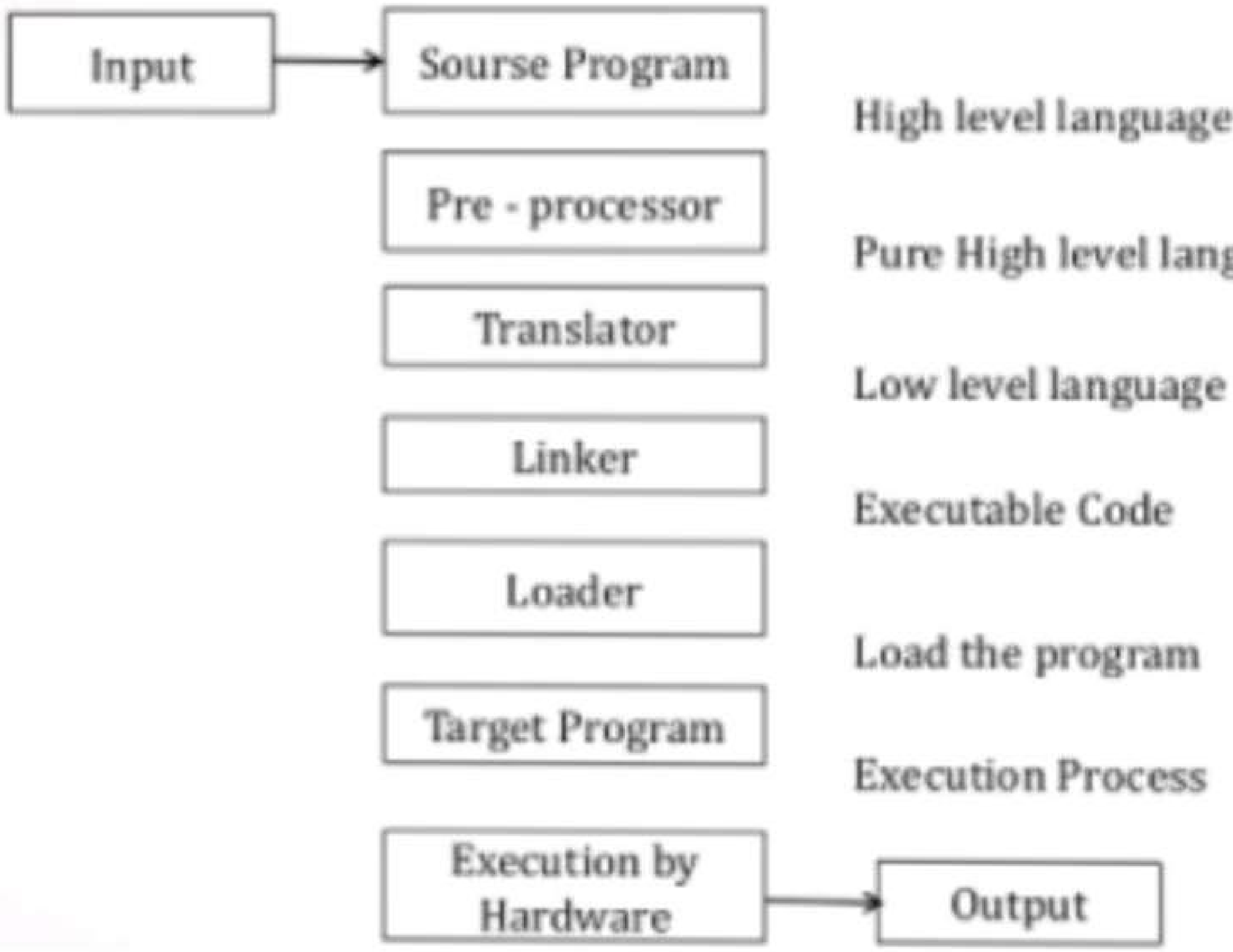
- It is the gap between the semantics of programs written in different programming languages. The execution gap is bridged by the translator or interpreter.

Q6. Explain the Life cycle of the source program with a neat sketch. (P4- Appeared 1 time) (3-7 marks)

Ans : In the life cycle of a source program the exact procedure behind the compilation task and step by step evaluation of source code are High level languages, Low level languages, Pre-processors, Translators, Compilers, Assembler, Interpreters, Linkers and Loaders.

- A computer program goes through many phases from its development to execution. From the human readable format to binary encoded computer instructions.

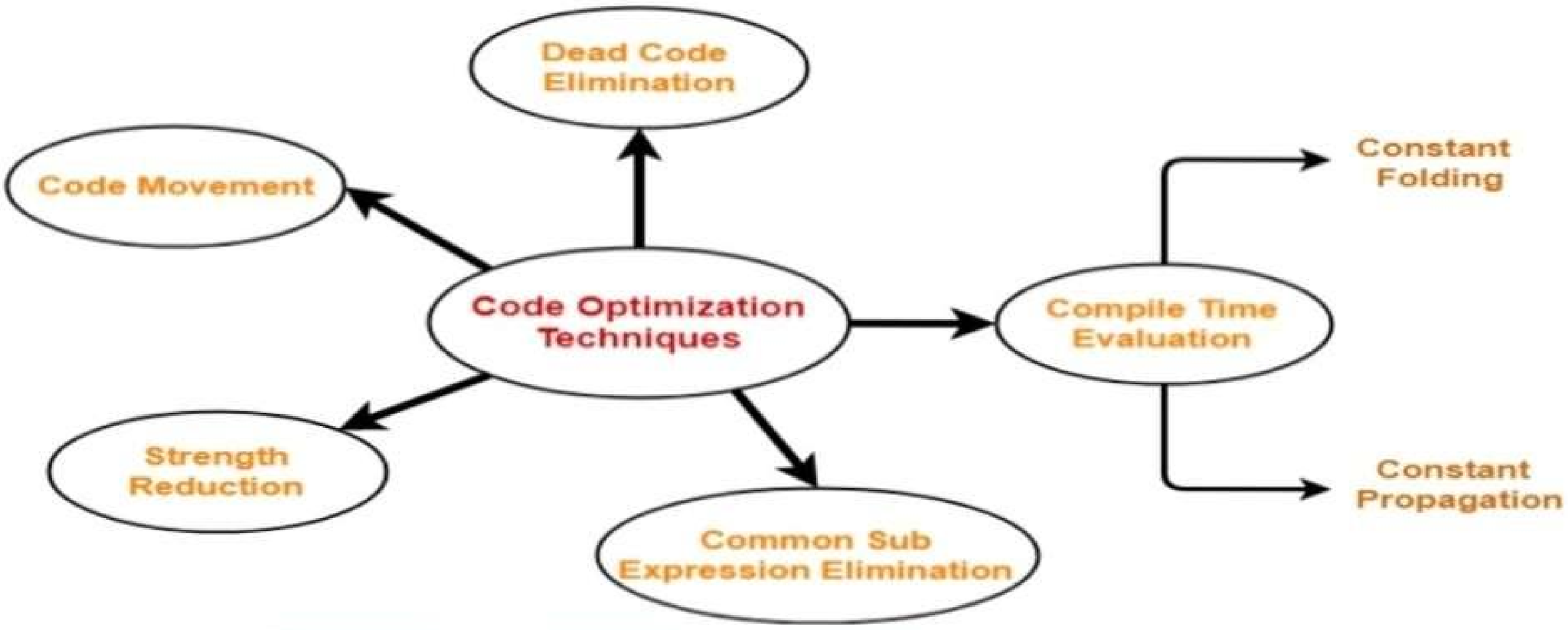
Life cycle of a souce program



MODULE-2

Q1. Explain any three Code Optimization Techniques. (P4 - Appeared 1 Time) (5-10M)

Ans : The code optimization techniques are as follows:



- Compile Time Evaluation
- Common subexpression elimination
- Dead Code Elimination
- Code Movement
- Strength Reduction

Compile Time Evaluation-

- Two techniques that falls under compile time evaluation are-

Constant Folding-

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.

- Those expressions are then replaced with their respective results.
- Example-

  Circumference of Circle = (22/7) x Diameter

  Here,
- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves time at run time.

Constant Propagation-

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of the variable must not get altered in between.
- Example-pi = 3.14
- radius = 10
- Area of circle = pi x radius x radius
- Here,
- and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves time at run time.

Common Subexpression Elimination-

- The expression that has been already computed before and appears again in the code for computation is called Common Sub-Expression.
- As the name suggests, it involves eliminating the common subexpressions.

- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

- Example-

Code Before Optimization	Code After Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i // Redundant Expression
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

Code Movement-

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets executed again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

- Example-

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

Dead Code Elimination-

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.
- Example-

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

Strength Reduction-

- As the name suggests, it involves reducing the strength of expressions.

- This technique replaces the expensive and costly operators with the simple and cheaper ones.
- Example-

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression "A x 2" is replaced with the expression "A + A".
- This is because the cost of the multiplication operator is higher than that of the addition operator.

Q2. Define following: Language Migrator, Execution gap, Token, Handle (P4 - Appeared 1 Time) (5-10M)

Ans : **Execution gap:**

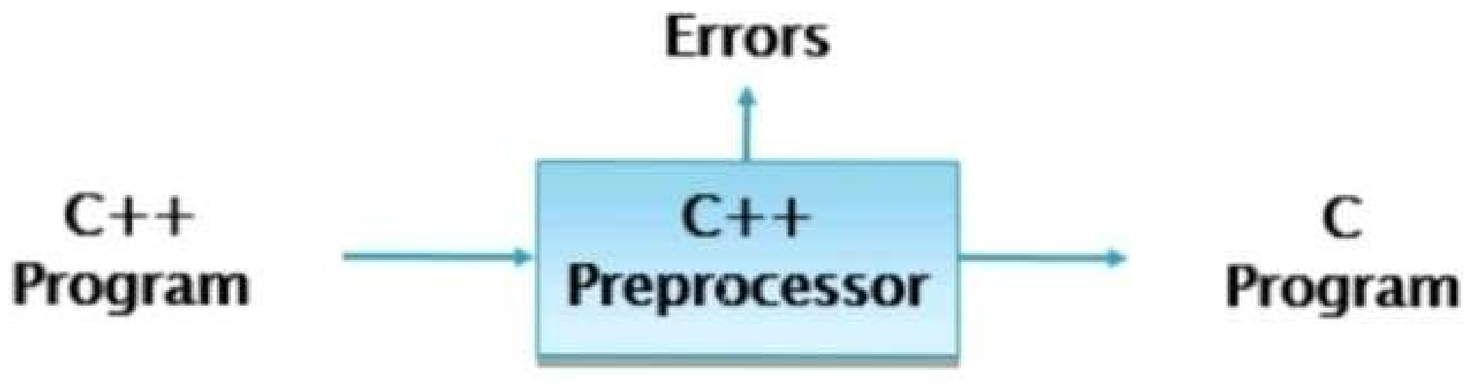
- It is the gap between the semantics of programs written in different programming languages.
- The execution gap is bridged by the translator or interpreter.

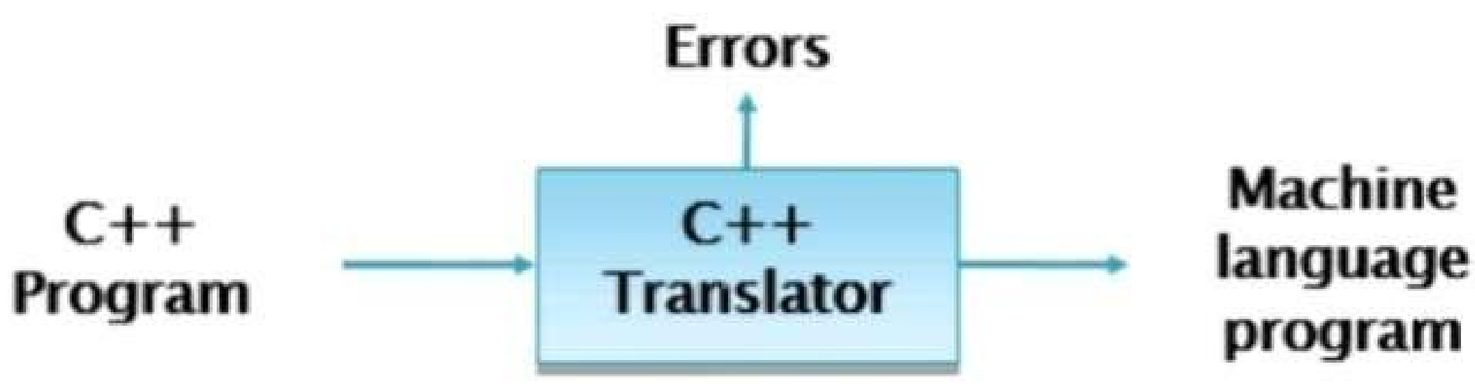
Language Migrator

- Bridges specification gap between two programming languages.
- Example: Preprocessor and Translator

Token

- A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:
    - Keywords
    - Identifiers
    - Constants
    - Strings
    - Special Symbols
    - Operators





Handle

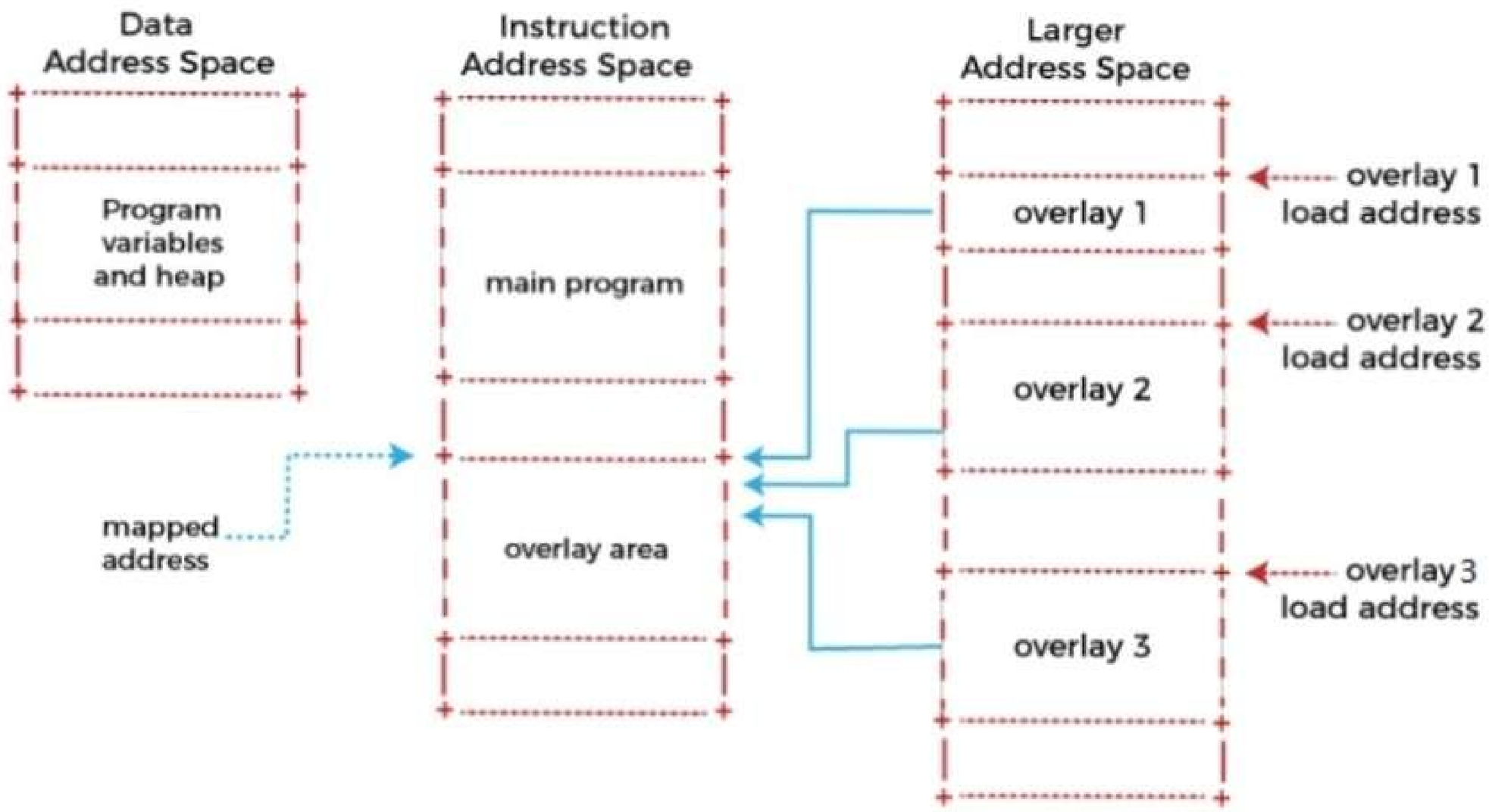
- A handle is a value used to identify an object to the operating system. For example, at any given time, each active window has a unique number, called a window handle, that identifies it to the operating system.

Q3. Define overlay. Explain the execution of an overlay structured program. (P4 - Appeared 1 Time) (5-10M)

Ans: **Overlay:**

- The process of transferring a block of program code or other data into main memory, replacing what is already stored.

How does Overlays Work



A code overlay

- Suppose you have a computer whose instruction address space is only 64 kilobytes long but has much more memory than others can access, such as special instructions, segment registers, or memory management hardware. Suppose that you want to adopt a program larger than 64 kilobytes to run on this system.
- One solution is to identify modules (overlays) of your program which are relatively independent and need not call each other directly. Separate the overlays from the main program, and

place their machine code in the larger memory. Place your main program in instruction memory, but leave at least enough space there to hold the largest overlay as well.

- To call a function located in an overlay, you must first copy that overlay's machine code from the large memory into the space set aside for it in the instruction memory and then jump to its entry point there.

Q4. What are the advantages and disadvantages of procedure oriented language? (P4 - Appeared 1 Time) (5-10M)

Ans: There are various advantages of procedure oriented programming languages :-

- It is good for general purpose programming
- We can reuse pieces of code in a program without writing them again, by the virtue of method calling.
- We can structure a program based on these methods and have a cleaner and more maintainable code when compared to the languages it succeeded (e.g Assembly language).

Some common disadvantages of procedure oriented languages are

- Data is exposed to the whole program at once, so there is no security of data available.
- Since the focus is on the instructions, it is rather difficult to relate to real world objects and in transition some real world problems

- There is no hierarchy in code. (It is a very big problem, trust me)

Q5. Give examples of language processors. (P4 - Appeared 1 Time) (5-10M)

Ans : A language processor is a software program designed or used to perform tasks such as processing program code to machine code. Language processors are found in languages such as Fortran and COBOL.

Examples of language processors

There are two main types of language processors:

- Interpreter - allows a computer to interpret, or understand, what a software program needs the computer to do, what tasks to perform.
- Translator - takes a program's code and translates it into machine code, allowing the computer to read and understand what tasks the program needs to be done in its native code. An assembler and a compiler are examples of translators.

Q6. Why are translators needed? (P4 - Appeared 1 Time) (5-10M)

Ans : A program written in high-level language is called source code. To convert the source code into machine code, translators are needed.

- A translator takes a program written in source language as input and converts it into a program in target language as output.
- It also detects and reports the error during translation.

MODULE-3

Q1. Compare various intermediate code forms for an assembler.

(P4 - Appeared 1 Time) (5-10M)

Ans: The following are commonly used intermediate code representation:

1. Postfix notation
2. Syntax tree
3. Three-address code

Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +.
- No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.
- In postfix notation the operator follows the operand.
- Eg : the postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + ab -+*.

Three-Address Code:

- A statement involving no more than three references(two for operands and one for result) is known as three address

statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z , here x, y, z will have an address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.

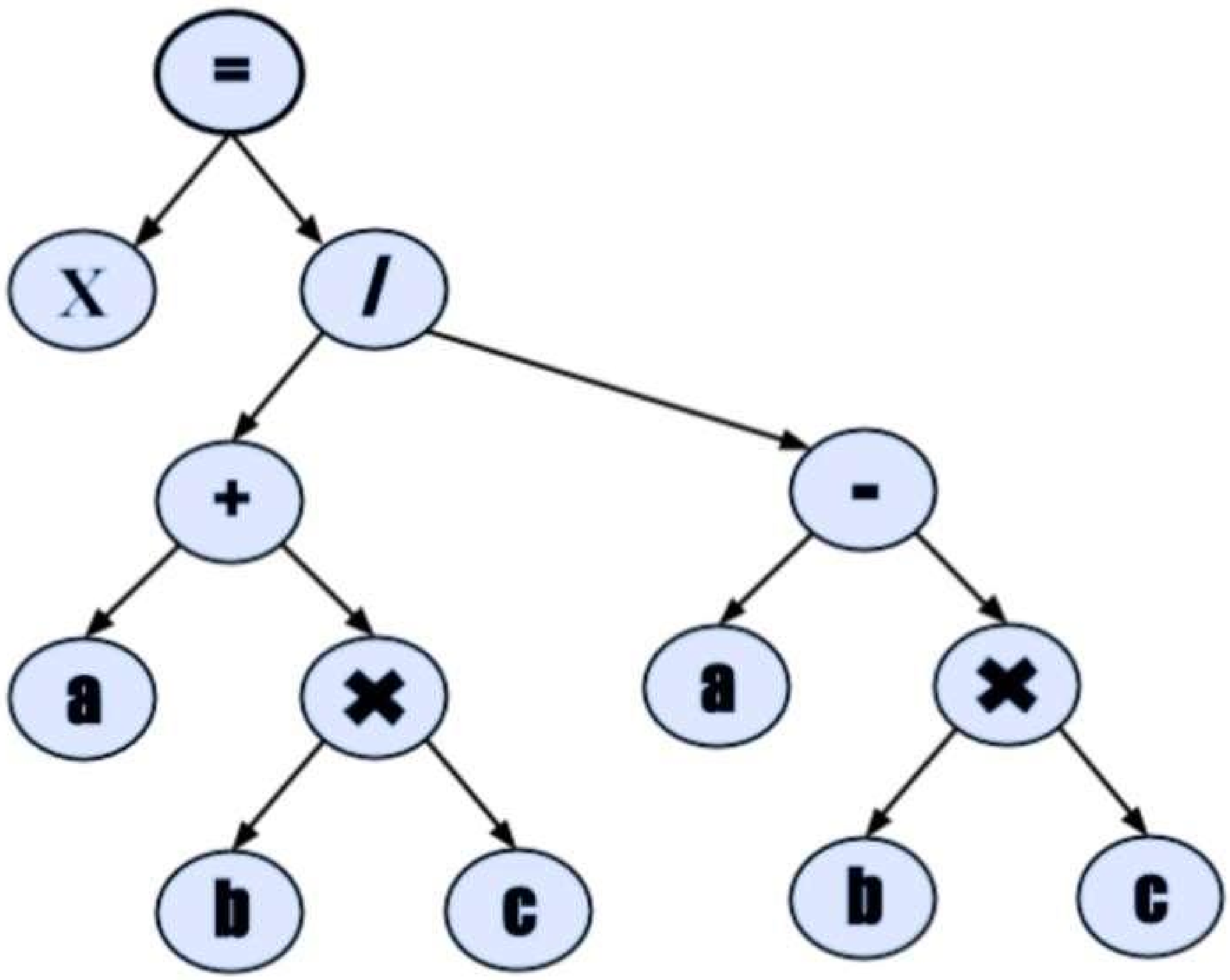
- Eg: the three address code for the expression a + b * c + d :T 1 = b * c T 2 = a + T 1 T 3 = T 2 + dT 1 , T 2 , T 3 are temporary variables.

Syntax Tree

- Syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by a single link in the syntax tree; the internal nodes are operators and child nodes are operands. To form syntax trees put parentheses in the expression, this way it's easy to recognize which operand should come first.
- Example : x = (a + b * c) / (a – b * c)

$$X = (a + (b* c)) / (a - (b * c))$$

Operator Root

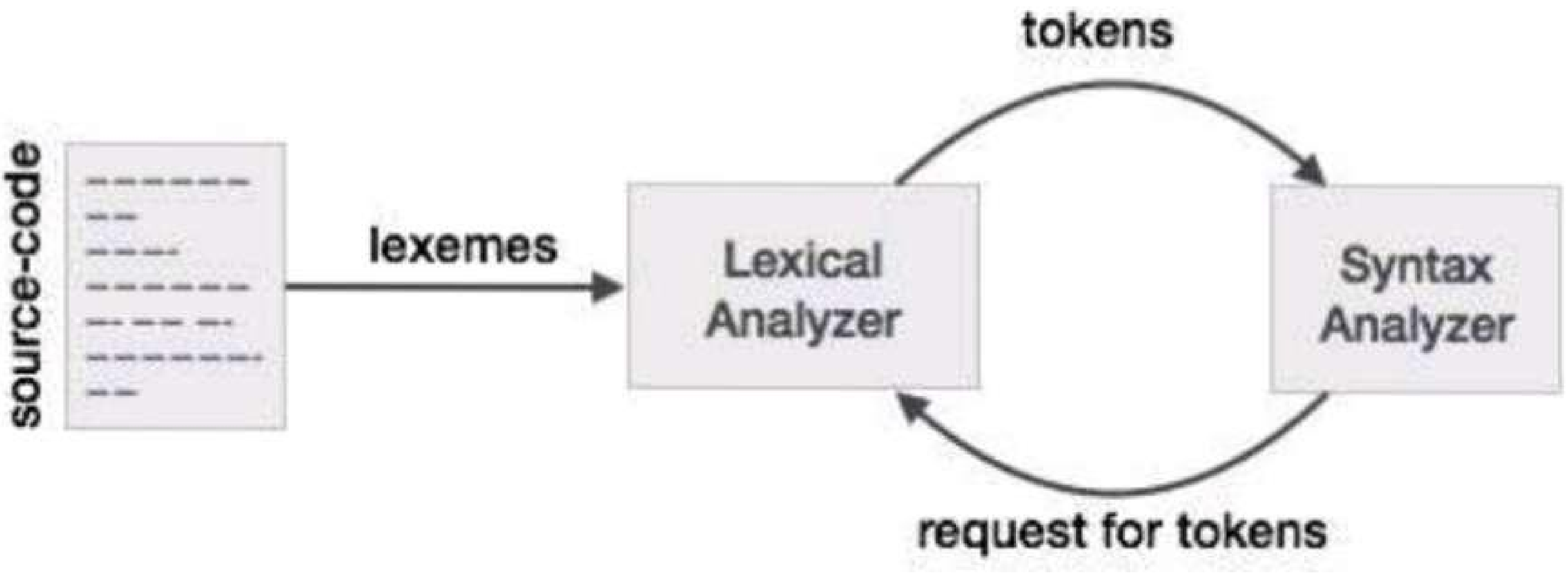


Q2. Explain lexical analysis of language processors. (P4 - Appeared 1 Time) (5-10M)

Ans : Lexical analysis is the first phase of a compiler.

- It takes the modified source code from language preprocessors that are written in the form of sentences.

- The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- If the lexical analyzer finds a token invalid, it generates an error.
- The lexical analyzer works closely with the syntax analyzer.
- It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Longest Match Rule-

- When the lexical analyzer reads the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.
- While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword int or the initials of identifier int value.
- The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

- The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input.
- That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Q3. Difference between one pass and two pass assembler. (P4 - Appeared 1 Time) (5-10M)

Ans:

One pass assembler	**Two pass assembler**
1) Scans entire source file only once	1) Require two passes to scan the source file. - First pass: responsible for label definition and introducing them in the symbol table. - Second pass: translates the instructions into assembly language or generates machine code.
2) Generally • Deals with syntax. • Constructs symbol table • Creates label list	2) Along with pass1 pass 2 also required which • Generates actual opcode. • Compute actual address of every label.

• Identifies the code segment, data segment, stack segment, etc.	• Assign code address for debugging the information. • Translates operand name to appropriate register or memory code
3) Cannot resolve forward references of data symbols.	3) can resolve forward references of data symbols.
4) No object program is written, hence no leader is required	4) A leader is required as object code is generated.
5) Tends to be faster compared to two pass	5) two pass assemblers require rescanning. Hence slow compared to one pass assembler.
6) Only created tables with all symbols are calculated.	6) address of symbols can be calculated.

MODULE-4

Q1. Define following terms: 1) Assembler 2) Macro 3) Parsing 4)Interpreter. (P4 - Appeared 1 Time) (5-10M)

Ans: **Assembler:**

- Assembler is the language translator that accepts inputs as assembly language (ALP) and obtains its machine equivalent code (Object code) along with the information required by the loader.

Macro:

- A macro (which stands for " macroinstruction ") is a programmable pattern which translates a certain sequence of input into a preset sequence of output.

Parsing:

- Parsing or syntactic analysis is the process of analyzing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech).

Interpreter:

- An interpreter is a common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Q2. Write Macro definition with the following and explain. (i) Macro using expansion time loop (ii) Macro with REPT statement. (P4 - Appeared 1 Time) (5-10M)

Ans: **MACRO**

- In an assembly language program there can be a group of instructions that may be repeated again and again.
- So we combine these instructions to a single group and assign a name to it which is called MACRO..
- A macro is an extension to the basic ASSEMBLER language. They provide a means for generating a commonly used sequence of assembler instructions/statements.
- The sequence of instructions/statements will be coded ONE time within the macro definition. Whenever the sequence is needed within a program, the macro will be "called"
- MACRO is defined as a single line abbreviation for a group of instructions.
- Syntax : Macro

  Macro name

  } macro body

  MEND
- Execution is faster in macro .There is no transfer of control in Macro.
- Features of Macro processor:

  1) Recognize the macro definition.

  2) Save macro definition.

  3) Recognize the macro call.

  4) Perform macro expansion.

Expansion time loop To generate many similar statements during the expansion of a macro. This can be achieved by similar model statements in the macro.

Example:

```
MACRO
 CLEAR &A
MOVER AREG, =„0“
MOVEM AREG, &A
MOVEM AREG, &A+1
MOVEM AREG, &A+2
MEND
```

Expansion time loops can be written using expansion time variables and expansion time control transfer statements AIF and AGO.

Example:

```
                    MACRO
                    CLEAR   &X, &N
                    LCL     &M
&M                  SET     0
                    MOVER   AREG,=„0“
.MOVE MOVEM                 AREG, &X+&M
&M                  SET     &M+1
            AIF             (&M NE N) .MORE
                    MEND
```

Comparison with execution time loops:

- Most expansion time loops can be replaced by execution time loops.
- An execution time loop leads to more compact assembly programs.

- In execution time loop programs would execute slower than programs containing expansion time loops.

Other facilities for expansion time loops:

- REPT statement
- Syntax: REPT <expression>
- <expression> should evaluate to a numerical value during macro expansion.
- The statements between REPT and an ENDM statement would be processed for expansion <expression> number of times.
- Example :

```
        MACRO
        CONST10
        LCL     &M
&M      SET     1
        REPT    10
        DC      '&M'
&M      SETA    &M+1
        ENDM
        MEND
```

Q3. Explain Nested macro calls with suitable examples. (P4 - Appeared 1 Time) (5-10M)

Ans: A macro body may also contain further macro definitions. However, these nested macro definitions aren't valid until the enclosing macro has been expanded! That means, the enclosing macro must have been called, before the nested macros can be called.

- Example:A macro, which can be used to define macros with arbitrary names, may look as follows:

```
DEFINE MACRO MACNAME
MACNAME MACRO
DB 'I am the macro &MACNAME.'
ENDM
ENDM
```

In order not to overload the example with "knowhow", the nested macro only introduces itself kindly with a suitable character string in ROM. The call

```
DEFINE Obiwan
```

would define the macro

```
Obi Wan MACRO
DB 'I am the macro Obiwan.'
ENDM
```

and the cal

```
DEFINE Skywalker
```

would define the following macro:

```
Skywalker MACRO
DB 'I am the macro Skywalker.'
ENDM
```

Q4. Attributes of formal parameter and expansion time variable in macro. (P4 - Appeared 1 Time) (5-10M)

Ans : Attributes of formal parameters:

- Represents information about the value of the formal parameter about corresponding actual parameters.

- The type, length and size attributes have the name T,L and S.

Example:

```
MACRO
DCL_CONST     &A
AIF           (L'&A EQ 1) .NEXT
- -
.NEXT         - -
              - -
              MEND
```

- Here expansion time control is transferred to the statement having.
- NEXT in its label field only if the actual parameter corresponds to the formal parameter. A has the length of 'l'.

Q5. Explain positional parameters, keyword parameters and default value parameters for macro. (P4 - Appeared 1 Time) (5-10M)

Ans: **Positional parameters:**

- Positional parameters are symbolic parameters that must be specified in a specific order every time the macro is called.
- The parameter will be replaced within the macro body by the value specified when the macro is called. These parameters can be given a default value.
- A parameter can be either a simple string or a quoted string. It can be passed by using the standard method of putting

variables into shared and profile pools (use VPUT in dialogs and VGET in initial macros).

- This method is best suited to parameters passed from one dialog to another, as in an edit macro.
- Additionally, what is the difference between keyword parameters and positional parameters? Positional parameters are mentioned by their positions and they came before keyword parameters. Positional parameters have only locations.. keyword parameters have keyword , equal sign(=) and information..

Keyword parameter:

- For keyword parameter,
    - <parameter name> is an ordinary string and
    - <parameter kind> is the string '='
      in syntax rules.
- The <actual parameter spec> is written as <formal parameter name>=<ordinary string>.
- Note that the ordinal position of the specification XYZ=ABC in the list of actual parameters is immaterial.
- This is very useful in situations where long lists of parameters have to be used.

Example:Following are macro call statement:

INCR_M MEM_VAL=A, INCR_VAL=B, REG=AREG

INCR_M INCR_VAL=B, REG=AREG, MEM_VAL=A

- Both are equivalent.
- Following is macro definition using keyword parameter:
- MACRO
- INCR_M &MEM_VAL=, &INCR_VAL=,&REG=

- MOVER &REG, &MEM_VAL
- ADD &REG, &INCR_VAL
- MOVEM &REG,&MEM_VAL
- MEND

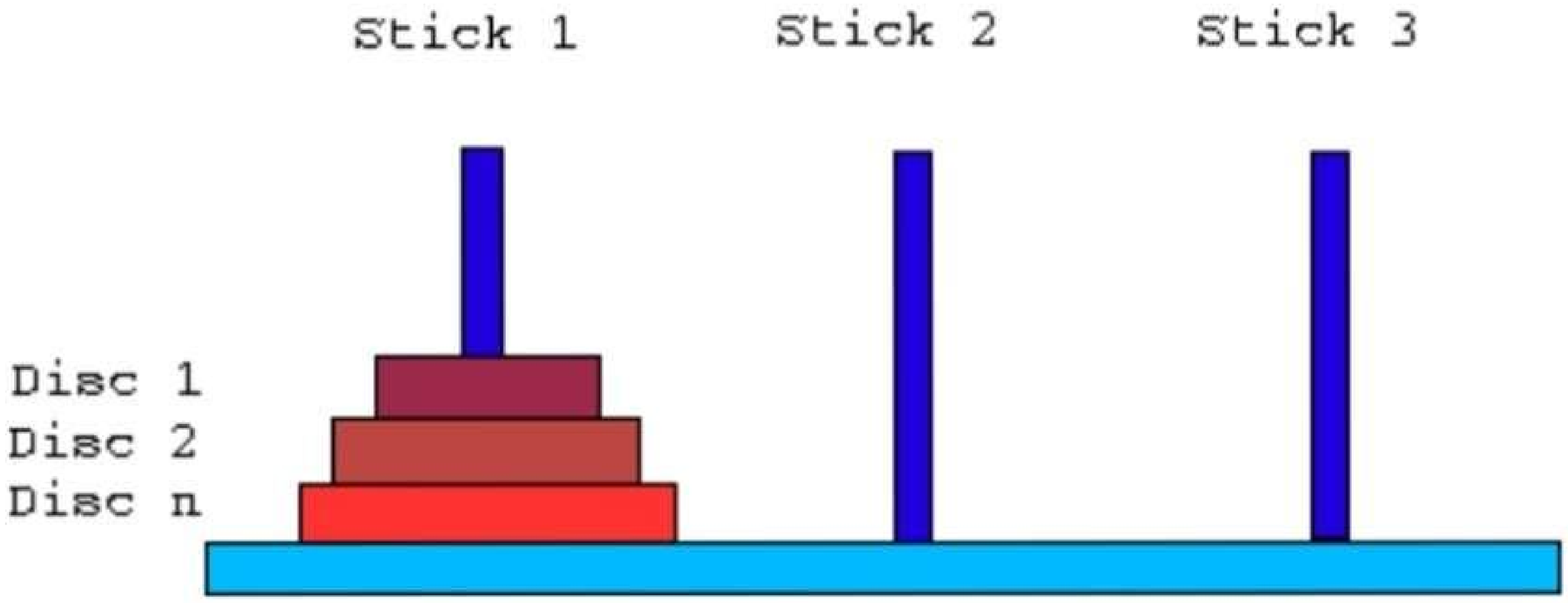
Default value parameter:

- If the myParameter macro variable has a value, it will be used, otherwise defaultValue will be assigned as the value.
- This technique can also be used in macros instead of using a %IF statement.
- The coalescec function is used regardless of whether the expected value for the parameter is numeric since macro is a text manipulation facility, (i.e., everything in macro is interpreted as a character string)

Q6. Write an example of recursive macros. (P4 - Appeared 1 Time) (5-10M)

Ans : **Example: The Towers of Hanoi**

- There are three vertical sticks on the table. On stick 1 There are n discs with different diameters and a hole in the middle, the smallest disc on top, the biggest on the bottom.



- The Problem is to transfer the tower of discs from stick 1 to stick 2 with a minimum of moves. But only the topmost disc on a tower may be moved at one time, and no disc may be laid on a smaller disc. Stick 3 may be used for scratch purposes. This is a Solution with ASEM-51 macros:
    - ;The Towers of Hanoi
    - $GENONLY CONDONLY
    - DISCS EQU 3 ;number of discs
    - HANOI MACRO n, SOURCE, DESTINATION, SCRATCH
      IF n >
- The recursive macro HANOI generates an instruction manual for the Problem, where the instructions appear as comment lines in the list file. The symbol DISCS must be set to the desired number of discs. If HANOI is called like this,

the following "instruction manual" is generated

```
27+ 3    ;  move topmost disc from stick 1 to stick 2
35+ 2    ;  move topmost disc from stick 1 to stick 3
44+ 3    ;  move topmost disc from stick 2 to stick 3
```

```
53+ 1     ;  move topmost disc from stick 1 to stick 2
64+ 3     ;  move topmost disc from stick 3 to stick 1
72+ 2     ;  move topmost disc from stick 3 to stick 2
81+ 3     ;  move topmost disc from stick 1 to stick 2
```

- The GENONLY and CONDONLY controls ensure that the table doesn't contain all the macro calls and IF constructions.

MODULE-5

Q1. What is program relocation? How is relocation performed by linker? Explain with examples. (P4 - Appeared 1 Time) (5-10M)

Ans : **Relocation:**

- It modifies the object program by changing certain instructions so that it can be loaded at a different address from the location originally specified.
- There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by the loader is called relocation.
- In absolute Loader relocation is done by the assembler as the assembler is aware of the starting address of the program.
- In relocatable loaders, relocation is done by the loader and hence the assembler must supply to the loader the location at which relocation is to be done.
- In the complete process translation origin and linking are always different. but linking origin and loading origin may be the same or different.
- If loading origin and linking origin are different this means relocation must be performed by the loader.
- Linker always performs relocation. But some leaders do not do this.

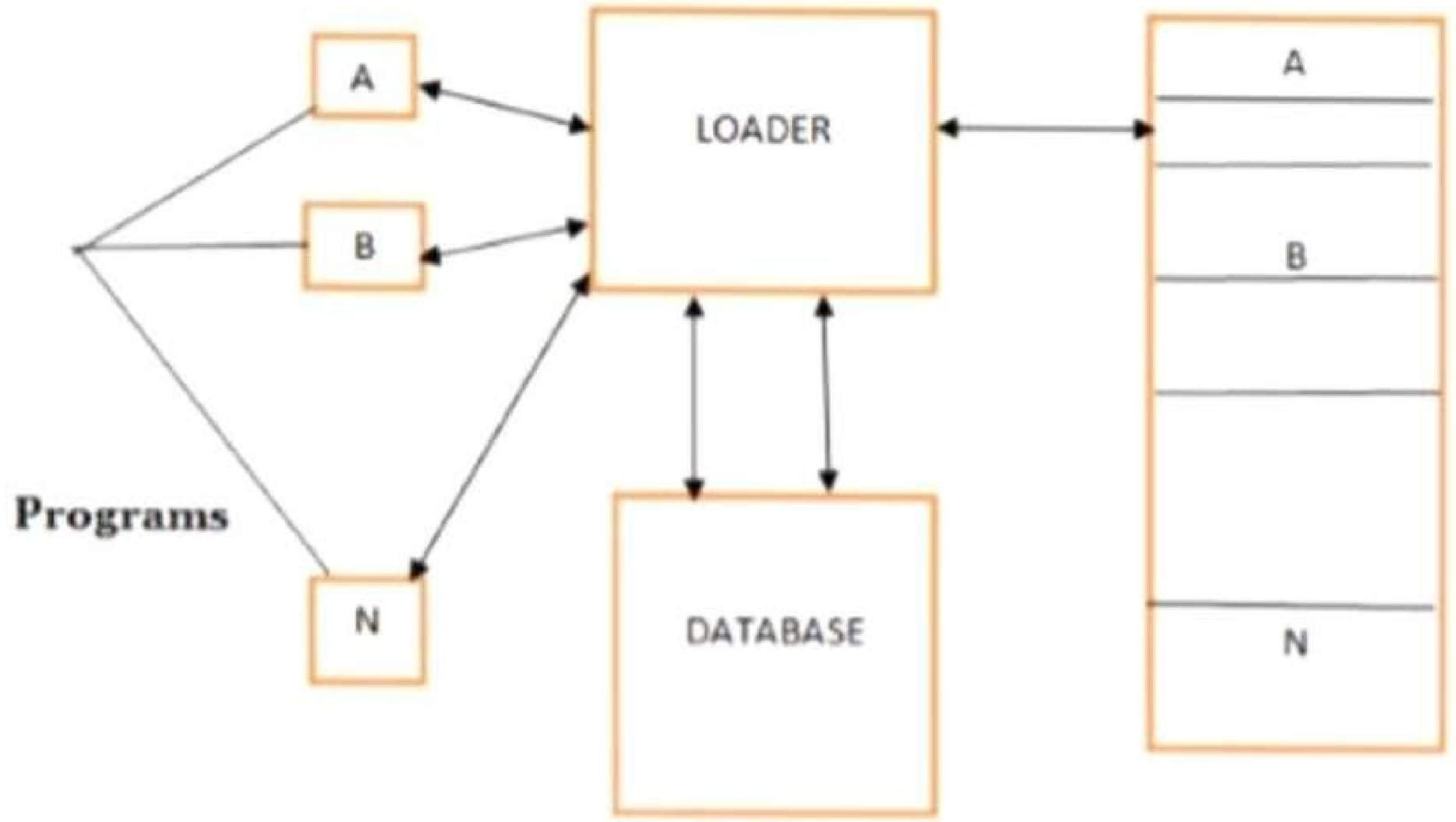
- If loaders do not perform relocation this means load origin and link origin are equal, and this loader is called absolute loader.
- If loaders perform relocation then loaders are called relocation loaders.
- Performing relocation: we can calculate the relocation factor by linking origin and translate origin. Suppose translated origin and linked origin of program are t_origin and l_origin respectively. Consider a symbol symbol in program P. its translation time address be tsymb and link time address be symb. The relocation factor of program is defined as relocation_factor = l_orgin – t_origin
- Relocation factors can be positive, negative or 0.
- Here symb is working as an operand. The translator puts the address tsymbal in the instruction for it. Tymb = t_origin + dsymb
- where dsymb is the offset of symbols in the program
- lamb = l_origin + dsymb
- lsymb = t_origin +relocation_faction +dsymb
- = t_origin + dsymb + relocation_factior
- = tsymb + relocation_factor

Q2. Explain the term loader with its basic function. (P4 - Appeared 1 Time) (5-10M)

Ans: **Loader:**

- A loader is a system program, which takes the object code of a program as input and prepares it for execution.

Diagram



- Programmers usually define the Program to be loaded at some predefined location in the memory.
- But this loading address given by the programmer is not coordinated with the OS.
- The loader does the job of coordinating with the OS to get the initial loading address for the .EXE file and load it into the memory.

Loader Function: The loader performs the following functions:

1) Allocation
2) Linking
3) Relocation
4) Loading

Allocation:

- Allocates the space in the memory where the object program would be loaded for Execution.

- It allocates the space for the program in the memory, by calculating the size of the program. This activity is called allocation.
- In absolute loaders, allocation is done by the programmer and hence it is the duty of the programmer to ensure that the programs do not overlap.
- In reloadable loaders allocation is done by the loader hence the assembler must supply the loader the size of the program.

Linking:

- It links two or more object codes and provides the information needed to allow references between them.
- It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.
- In absolute loader, linking is done by the programmer as the programmer is aware of the runtime address of the symbols.
- In relocatable loaders, linking is done by the loader and hence the assembler must supply to the loader the locations at which the loading is to be done.

Relocation:

- It modifies the object program by changing certain instructions so that it can be loaded at a different address from the location originally specified.
- There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by the loader is called relocation.

- In absolute Loader, relocation is done by the assembler as the assembler is aware of the starting address of the program.
- In relocatable loaders, relocation is done by the loader and hence the assembler must supply to the loader the location at which relocation is to be done.

Loading:

- It brings the object program into the memory for execution.
- Finally, it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus the program now becomes ready for execution, this activity is called loading.
- In both the loaders (absolute, relocatable) Loading is done by the loader and hence the assembler must supply to the loader the object program.

Q3. Describe the facilities for dynamic debugging. (P4 - Appeared 1 Time) (5-10M)

Ans : **Dynamic Debugging:**

It involves observing the contents of register or output after execution of each instruction (in single step technique) or a group of instructions (in breakpoint technique).

In a single board microprocessor, techniques and tools commonly used in dynamic debugging are:

1. Single Step: This technique allows one to execute one instruction at a time and observe the results of each instruction. Generally, this is built using a hard-wired logic

circuit. As we press the single step run key we will be able to observe the contents of the register and memory location. This helps to spot:

- incorrect addresses
- incorrect jump location in loops
- incorrect data or missing codes

2. However, if there is a large loop then single step debugging can be very tiring and time-consuming. So instead of running the loop n times, we can reduce the number of iterations to check the effectiveness of the loop. The single step technique is very useful for short programs.

Q4. Define linking. How external reference is resolved in linking?

(P4 - Appeared 1 Time) (5-10M)

Ans : **Linking**

- Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed.

How external reference is resolved in linking?

- The linker uses public symbols to resolve the addresses of external references. For each external reference, a public symbol with the same name and attributes must exist.
- The linker builds a table of all public and external symbols it encounters. External references are resolved with public references (as long as the names match and the symbol attributes correspond). The linker reports an error if the symbol types of an external and public symbol do not match.

The linker also reports an error if no public symbol is found for an external reference.

- The absolute addresses of the public symbols are resolved after the location of the segments is determined.

Q5. Differentiate Linker and Loader. (P4 - Appeared 1 Time) (5-10M)

Ans :

BASIS FOR COMPARISON	**LINKER**	**LOADER**
Basic	It generates the executable module of a source program.	It loads the executable module to the main memory.
Input	It takes as input, the object code generated by an assembler.	It takes an executable module generated by a linker.
Function	It combines all the object modules of a source code to generate an executable module.	It allocates the addresses to an executable module in main memory for execution.

Type/Approac h	Linkage Editor, Dynamic linker.	Absolute loading, Relocatable loading and Dynamic Run-time loading.

Q6. Write a brief note on MS-DOS Linker. (P4 - Appeared 1 Time) (5-10M)

Ans : MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program. This executable program has the file name extension. EXE. LINK can also combine the translated programs with other modules from object code libraries.

1. MS-DOS compilers and assemblers produce object modules(.OBJ).
2. Each .OBJ contains a binary image of the translated instructions and data of the program.
3. MS-DOS LINK is a linkage editor that combines one-or more modules to produce a complete executable program (.exe).
4. MS-DOS object module.

Record Types	**Description**
THEADR	Translator header similar to header record in SIC/XE
TYPEDEF	Type definitions of external symbols

PUBDEF	Public Definitions of external symbols defined in the object module.
EXTDEF	Information about the data and external name.
NAMES	List of all the segments and class names used in the program.
SEGDEF	Describes the segments in the object module, including name, length, alignment.
GDPDEF	Grouping of Segments.
LEDATA	Similar to Text record (SIC/XE) contain translated instruction and data from the source program
LIDATA	Translated Instruction and data that occur In repeated pattern
FIXUP	Resolve external references, contains relocation and linking information and must immediately follow the LEDATA or LIDATA record to which it applies
MODEND	End of object module.

Link performs its processing in two passes.

Pass 1: Computes a starting address for each segment in the program.

- Segments are placed in the same order as given in SEGDEF records.
- Segments from different object modules that have the same segment name and class are combined.
- Segments with the same class, but different names are concatenated.
- Segments starting address is updated as these combinations and concatenation are performed.

Pass 2:

Extracts the translated instructions from the object modules.

- Process each DATA and DATA record along with FIXUP
- Relocation operations that involve the starting address of the segment are added to a table of Segment Fixups.
- Build an image of the executable program in memory.
- Write it to the executable (.EXE.) file.
- This file includes a header that contains a table of fixups,information about memory requirements .

Q7. Compare Absolute Loader with Relocating Loader (BSS Loader). (P4 - Appeared 1 Time) (5-10M)

Ans : An absolute loader is the most basic type of loading technique which loads the file into memory at the location determined by the header of the file, after that it passes control to the user program.

- A relocatable loader, on the contrary, loads the user's program anywhere in memory as defined by the user or my memory logic.

- This leads to alteration of addresses which are required for correct referencing.

MODULE-6

Q1. Explain Left recursion, Left factoring in top down parsing. (P4 - Appeared 1 Time) (5-10M)

Ans : **Left Recursion**

- Left Recursion. The production is left-recursive if the leftmost symbol on the right side is the same as the non-terminal on the left side.
- For example, expr → expr + term. If one were to code this production in a recursive-descent parser, the parser would go in an infinite loop.

Elimination of left Recursion

We eliminate left-recursion in three steps.

- eliminate ε -productions (impossible to generate ε!)
- eliminate cycles (A ⇒+ A)
- eliminate left-recursion

Algorithm:

Arrange the non terminals in some order, A_1 , A_2 , A_3,..... A_n.

For 1 :- 1 TO n DO

BEGIN (FOR i)

FOR j :- 1 TO N DO

BEGIN (FOR j)

(1) Replace each product of the form $A_i \rightarrow A_i\gamma$ by the productions:

$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots\ldots \mid \delta_k\gamma$

Where:

$A_j \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots\ldots \mid \delta_k\gamma$

are all the current A_j - productions.

(2) ELiminate the direct left recursion from the A_i productions

END (FOR j)

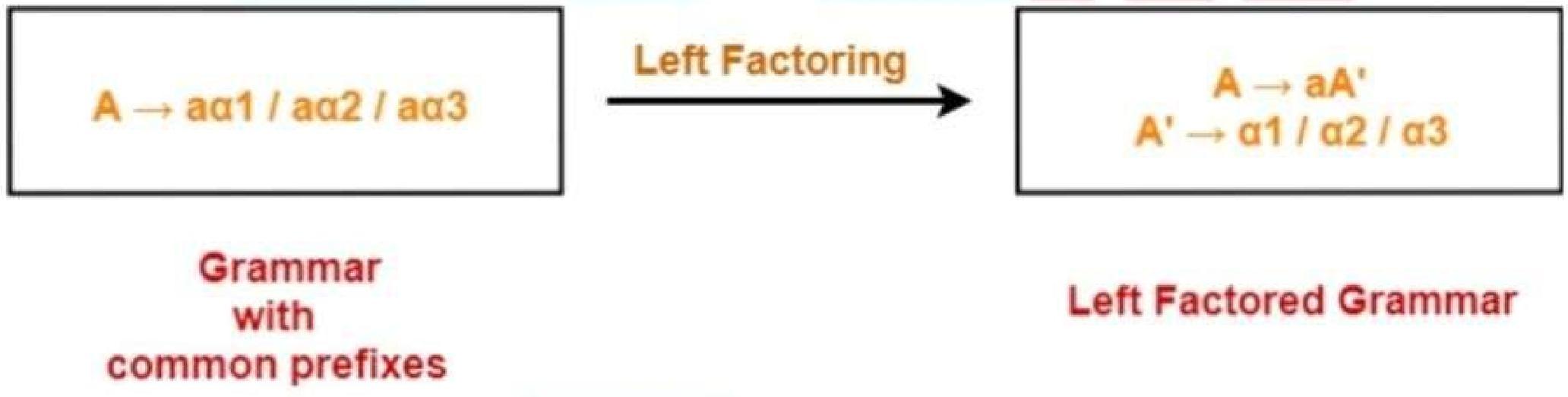
END (FOR i)

Left factoring

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

Example-

- This kind of grammar creates a problematic situation for top down parsers.
- Top down parsers can not decide which production must be chosen to parse the string in hand. To remove this confusion, use left factoring.



Q2. What is operator precedence parsing? (P4 - Appeared 1 Time) (5-10M)

Ans : **Operator precedence parsing**

Operator precedence grammar is a kind of shift-reducing parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only be established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

- Both ends of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first leftmost ⋖ is encountered.
- Everything between leftmost ⋖ and rightmost ⋗ is a handle.
- $ on $ means parsing is successful.

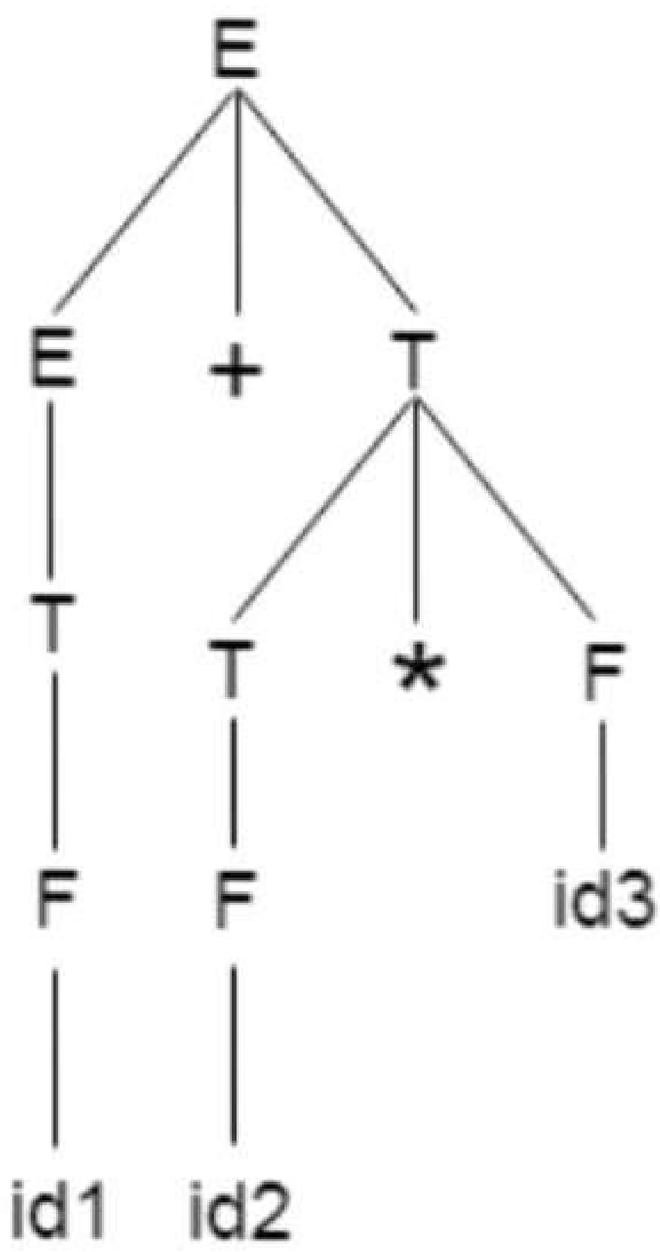
ExampleGrammar:

1. E → E+T/T
2. T → T*F/F
3. F → id

Given string:

1. w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Q3. Explain types of grammar. (P4 - Appeared 1 Time) (5-10M)

Ans : **Types of grammar:**

1. **Comparative Grammar-**
    - The analysis and comparison of the grammatical structures of related languages is known as comparative grammar. Contemporary work in comparative grammar is concerned with "a faculty of language that provides an explanatory basis for how a

human being can acquire a first language . . .. In this way, the theory of grammar is a theory of human language and hence establishes the relationship among all languages" (R. Freidin, Principles and Parameters in Comparative Grammar. MIT Press, 1991).

2. **Generative Grammar-**
    - Generative grammar includes the rules determining the structure and interpretation of sentences that speakers accept as belonging to the language. "Simply put, a generative grammar is a theory of competence: a model of the psychological system of unconscious knowledge that underlies a speaker's ability to produce and interpret utterances in a language" (F. Parker and K. Riley, Linguistics for Non-Linguists. Allyn and Bacon, 1994).
3. **Mental Grammar-**
    - The generative grammar stored in the brain that allows a speaker to produce language that other speakers can understand is mental grammar. "All humans are born with the capacity for constructing a Mental Grammar, given linguistic experience; this capacity for language is called the Language Faculty (Chomsky, 1965). A grammar formulated by a linguist is an idealized description of this Mental Grammar" (P. W. Culicover and A. Nowak, Dynamical Grammar: Foundations of Syntax II. Oxford University Press, 2003).
4. **Pedagogical Grammar-**
    - Grammatical analysis and instruction designed for second-language students. "Pedagogical grammar is

a slippery concept. The term is commonly used to denote

- (1) pedagogical process--the explicit treatment of elements of the target language systems as (part of) language teaching methodology;
- (2) pedagogical content--reference sources of one kind or another that present information about the target language system; and
- (3) combinations of process and content" (D. Little, "Words and Their Properties: Arguments for a Lexical Approach to Pedagogical Grammar." Perspectives on Pedagogical Grammar, ed. by T. Odlin. Cambridge University Press, 1994).

5. **Performance Grammar-** A description of the syntax of English as it is actually used by speakers in dialogues.

"[P]performance grammar . . . centers attention on language production; it is my belief that the problem of production must be dealt with before problems of reception and comprehension can properly be investigated" (John Carroll, "Promoting Language Skills." Perspectives on School Learning: Selected Writings of John B. Carroll, ed. by L. W. Anderson. Erlbaum, 1985).

6. **Reference Grammar-**

- A description of the grammar of a language, with explanations of the principles governing the construction of words, phrases, clauses, and sentences. Examples of contemporary reference grammars in English include A Comprehensive Grammar of the English Language, by Randolph Quirk

et al. (1985), the Longman Grammar of Spoken and Written English (1999), and The Cambridge Grammar of the English Language (2002).

7. **Theoretical Grammar-**
    - The study of the essential components of any human language. "Theoretical grammar or syntax is concerned with making completely explicit the formalisms of grammar, and in providing scientific arguments or explanations in favour of one account of grammar rather than another, in terms of a general theory of human language".
8. **Traditional Grammar-**
    - The collection of prescriptive rules and concepts about the structure of the language. "We say that traditional grammar is prescriptive because it focuses on the distinction between what some people do with language and what they ought to do with it, according to a pre-established standard. . . . The chief goal of traditional grammar, therefore, is perpetuating a historical model of what supposedly constitutes proper language".
9. **Transformational Grammar-**
    - A theory of grammar that accounts for the constructions of a language by linguistic transformations and phrase structures. "In transformational grammar, the term 'rule' is used not for a precept set down by an external authority but for a principle that is unconsciously yet regularly followed in the production and interpretation of sentences. A

rule is a direction for forming a sentence or a part of a sentence, which has been internalized by the native speaker" .

10. **Universal Grammar-**
    - The system of categories, operations, and principles shared by all human languages and considered to be innate. "Taken together, the linguistic principles of Universal Grammar constitute a theory of the organization of the initial state of the mind/brain of the language learner--that is, a theory of the human faculty for language" .

Q4. What is the structure of the LEX program? (P4 - Appeared 1 Time) (5-10M)

Ans : LEX programs are used in "Lexical analysis" of the input stream, which is to break the input stream into usable and meaningful elements.

- The job of a lexical analyzer is to identify and match the patterns from the input stream with respect to the rule part mentioned in the LEX program.

Structure of LEX program:

```
%{
/* C includes */
}%
/* Definitions */
%%
/* Rules */
%%
/* user subroutines */
```

The LEX structure consists of 4 different sections, they are as follows –

- DECLARATION PART – This section starts with '%{' , ends with '}%' and includes the libraries like stdio.h which are to be included before executing the LEX program. We can also declare variables in this section that can be used in the rule part.
- DEFINITION PART – This section follows the declaration part and is written in between the definition part and the rule part. It is used to define patterns which can be assessed by the rule part.
- RULE PART – This section starts with '%%' and ends with '%%'. Rule part is necessary and the most important part of a LEX program, it gives the rule according to which the patterns are matched and the associated action is undertaken by the LEX program.
- USER DEFINED PART – This section is the basic sequence of flow for a LEX program, this section can be used to take input from the user or to print some data on to the terminal/console and is written in C.

Q5. What is ambiguity in the grammatic specification? (P4 - Appeared 1 Time) (5-10M)

Ans : **Ambiguity:**

- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string.
- If the grammar is not ambiguous then it is called unambiguous.

Q6. Difference between Top Down parser and Bottom Up parser.

(P4 - Appeared 1 Time) (5-10M)

Ans :

S.No	Top Down Parsing	Bottom Up Parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the	Bottom-up parsing can be defined as an attempt to

	leftmost derivations for an input string.	reduce the input string to the start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

MODULE 7

Q1. Explain the terms Binding and Binding Times. (P4 - Appeared 1 Time) (5-10M)

Ans : A binding is an association between a name and the thing that is named

- Binding time is the time at which an implementation decision is made to create a binding
    - Language design time: the design of specific program constructs (syntax), primitive types, and meaning (semantics)
    - Language implementation time: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc.
    - Program writing time: the programmer's choice of algorithms and data structures
    - Compile time: the time of translation of high-level constructs to machine code and choice of memory layout for objects
    - Link time: the time at which multiple object codes (machine code files) and libraries are combined into one executable
    - Load time: the time at which the operating system loads the executable in memory

- Run time: the time during which a program executes (runs)

Binding Time Examples

Language feature	Binding time
Syntax, e.g. if (a>0) b:=a; in C or if a>0 then b:=a end if in Ada	Language design
Keywords, e.g. class in C++ and Java	Language design
Reserved words, e.g. main in C and writeln in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. float and struct in C	Language design
Internal representation of literals, e.g. 3.1 and "foo bar"	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time

Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. printf in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Non Static allocation of space for variable	Run time

Q3. **Which are the methods used for identifying free memory?**

Ans : Free Space Management

- A file system is responsible to allocate the free blocks to the file therefore it has to keep track of all the free blocks present in the disk. There are mainly two approaches by which the free blocks in the disk are managed.

1. Bit Vector
    - In this approach, the free space list is implemented as a bitmap vector. It contains the number of bits where each bit represents each block.
    - If the block is empty then the bit is 1 otherwise it is 0. Initially all the blocks are empty therefore each bit in the bitmap vector contains 1.
    - LAs the space allocation proceeds, the file system starts allocating blocks to the files and setting the respective bit to 0.
2. Linked List
    - It is another approach for free space management. This approach suggests linking together all the free blocks and keeping a pointer in the cache which points to the first free block.
    - Therefore, all the free blocks on the disks will be linked together with a pointer. Whenever a block gets allocated, its previous free block will be linked to its next free block.

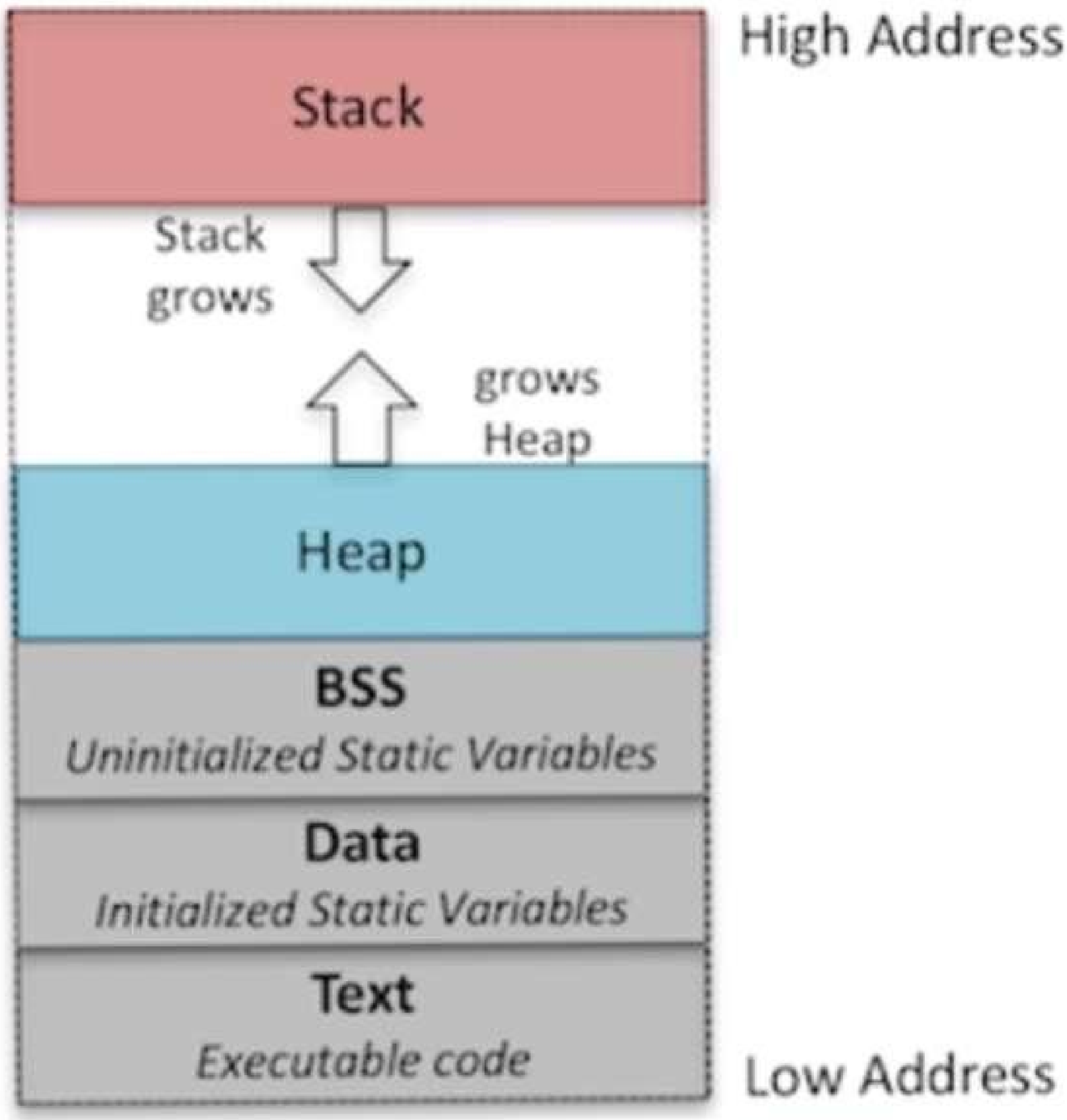
Q4. What is memory binding? Explain dynamic memory allocation using an extended stack model. (P4 - Appeared 1 Time) (5-10M)

Ans : Dynamic Memory Allocation:

- The mechanism by which storage/memory/cells can be allocated to variables during the run time is called Dynamic Memory Allocation (not to be confused with DMA).
- So, as we have been going through it all, we can tell that it allocates the memory during the run time which enables us to

use as much storage as we want, without worrying about any wastage.

- Dynamic memory allocation refers to managing system memory at runtime.



- Dynamic memory management in the C programming language is performed via a group of four functions named malloc(), calloc(), realloc(), and free().
- These four dynamic memory allocation functions of the C programming language are defined in the C standard library header file <stdlib.h>.
- Dynamic memory allocation uses the heap space of the system memory.
- Stack region is used for storing program's local variables when they're declared.

- Also, variables and arrays declared at the start of a function, including main, are allocated stack space.
- Stacks grow from high address to low address.
- Heap region is exclusively for dynamic memory allocation. Unlike stacks, heaps grow from low address to high address.

Code region can be further divided as follows:

- BSS segment: stores uninitialized static variables
- Data segment: stores static variables that are initialized
- Text segment: stores the program's executable instructions

Q5. Differentiate Compiler and Interpreter. (P4 - Appeared 1 Time) (5-10M)

Ans :

Sr. no	Compiler	Interpreter
1	Compiler scans the whole program in one go.	Translates program one statement at a time.
2	As it scans the code in one go, the errors (if any) are shown at the end together.	Considering it scans code one line at a time, errors are shown line by line.
3	Main advantage of compilers is their execution time.	Due to interpreters being slow in executing the object code, it is preferred less.
4	It converts the source code	It does not convert source

	into object code.	code into object code instead it scans it line by line.
5	It does not require source code for later execution.	It requires source code for later execution.
Eg.	C,C++,C# etc.	Python, Ruby, Perl, SNOBOL, MATLAB, etc.

Q6. What is the analysis and synthesis part of compilation? (P4 - Appeared 1 Time) (5-10M)

Ans: There are two parts of complication

Analysis: (means breaks in parts) Analysis parts break the source program into constituent pieces and give the source program an intermediate representation.

Synthesis: synthesis output (target program) From the intermediate representation.

MODULE 8

Q1. What is backtracking? (P4 - Appeared 1 Time) (5-10M)

Ans : Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions.

- This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems.
- Backtracking is used when we have multiple solutions, and we require all those solutions.
- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again.
- It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

Q2. Explain the drawbacks and benefits of Interpretation (P4 - Appeared 1 Time) (5-10M)

Ans :**Advantages Of Interpreter**

- **Cross-Platform :** In interpreted language we directly share the source code which can run on any system without any system incompatibility issue.
- **Easier To Debug :** Code debugging is easier in interpreters since it reads the code line by line, and returns the error message on the spot. Also, the client with the source code can debug or modify the code easily if they need to.
- **Less Memory and Step :** Unlike the compiler, interpreters don't generate new separate files. So it doesn't take extra Memory and we don't need to perform one extra step to execute the source code, it is executed on the fly.
- **Execution Control :** Interpreter reads code line by line so you can stop the execution and edit the code at any point, which is not possible in a compiled language. However, after being stopped it will start from the beginning if you execute the code again.

Disadvantages Of Interpreter

- **Slower :** Interpreter is often slower than compiler as it reads, analyzes and converts the code line by line.
- **Dependencies file required :** A client or anyone with the shared source code needs to have an interpreter installed in their system, in order to execute the code.
- **Less Secure :** Unlike compiled languages, an interpreter doesn't generate any executable file so to share the program

with others we need to share our source code which is not secure and private. So it is not good for any company or corporations who are concerned about their privacy.

Q3. A program computes i*5 for 10 times. What type of optimization can it be applied?

Ans : **Loop Optimization**

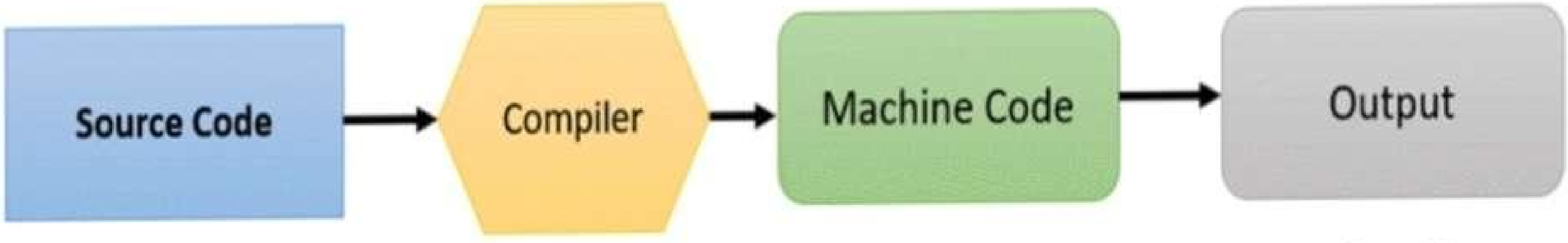
- Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

Q4. Describe three components of the interpreter.

Ans : An interpreter is a computer program, which converts each high-level program statement into the machine code. This includes source code, pre-compiled code, and scripts.

- Both compiler and interpreters do the same job which is converting higher level programming language to machine code. However, a compiler will convert the code into machine code (create an exe) before the program runs.
- Interpreters convert code into machine code when the program is run.

How Compiler Works



How Interpreter Works

3. Above (40 Marks)

1. What is the difference between System Software and Application software?
2. Explain Life cycle of the source program with a neat sketch.
3. Describe following data structures: OPTAB, SYMTAB, LITTAB and POOLTAB.
4. Which type of gap makes the software buggy or unreliable? Which methods can be used to overcome this situation?
5. Define following: Language Migrator, Execution gap, Token, Handle
6. Define overlay. Explain the execution of an overlay structured program.
7. What are the advantages and disadvantages of procedure oriented language?
8. Compare various intermediate code forms for an assembler.
9. Explain lexical analysis of language processor
10. Define following terms: 1) Assembler 2) Macro 3) Parsing 4)Interpreter
11. Write Macro definition with following and explain. (i) Macro using expansion time loop (ii)

12. Explain Nested macro calls with suitable examples.
13. What is program relocation? How relocation is performed by linker? Explain with examples.
14. Explain the term loader with its basic function.
15. Describe the facilities for dynamic debugging.
16. Define linking. How external reference is resolved in linking?
17. Differentiate Linker and Loader.
18. Write a brief note on MS-DOS Linker.
19. What is operator precedence parsing?
20. Explain types of grammar.
21. What is ambiguity in grammatic specification?
22. Difference between Top Down parser and Bottom Up parser.
23. What is memory binding? Explain dynamic memory allocation using an extended stack model.
24. Differentiate Compiler and Interpreter.
25. Explain the drawbacks and benefits of Interpretation