INDEX

Intermediate code generator: If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language independent (three-address code).
1.Postfix Notation: Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an 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 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 is : ab + c ***
2.Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.
eg:expression a * – (b + c) into three address code.t1= b+c,t2=uminus t1,t3=a*t2
A)**Quadruple** is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression. **B)Triples:**This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2. **C) Indirect Triples:** This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.
3. 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 the single link in the syntax tree the internal nodes are operators and child nodes are operands. To form a syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Issues of Code generator:1. Memory Management – Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for the name. Then from the symbol table entry, a relative address can be determined for the name.**2. Instruction selection** – Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straightforward. **three-address statements P:=Q+R; S:=P+T; would be translated into the latter code sequence MOV Q, R0; ADD R, R0; MOV R0, P; MOV P, R0; ADD T, R0; MOV R0, S.** Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. **3. Register allocation issues:** Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers is subdivided into two subproblems: **During Register allocation –** we select only those sets of variables that will reside in the registers at each point in the program. **During a subsequent Register assignment phase**, the specific register is picked to access the variable. To understand the concept consider the following three address code sequence t:=a+b; t:=t*c; t:=t/d. Their efficient machine code sequence is:
MOV a,R0; ADD b,R0; MUL c,R0; DIV d,R0; MOV R0,t. **4.Evaluation order** – The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem. **5.Approaches to code generation issues:** Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are: Correct, Easily maintainable, Testable & Efficient.

The **code optimization** in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. So optimization helps to:
1.Reduce the space consumed and increases the speed of compilation.
2.Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer**. 3.**An optimized code often promotes re-usability.
Types of Code Optimization: A.Machine Dependent Optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy**. *Peephole Optimization Techniques 1. Redundant load and store elimination:** In this technique, redundancy is eliminated. y = x + 5; i = y; z = i; w = z * 3; →y = x + 5; w = i * 3; **2.Constant folding:** The code that can be simplified by the user itself, is simplified.
3. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time. **4. Null sequences/ Simplify Algebraic Expressions :** Useless operations are deleted. a := a + 0; a := a * 1; a := a/1; a := a - 0**; 5. Combine operations:** Several operations are replaced by a single equivalent operation. **6. Deadcode Elimination:** A part of the code which can never be executed, eliminating it will improve processing time and reduces set of instruction.
Objectives of Peephole Optimization: To improve performance, To reduce memory footprint, To reduce code size. **Techniques:1.Redundant load and store elimination:** In this technique, redundancy is eliminated.**2.Constant folding:** The code that can be simplified by the user itself, is simplified.**3.Strength Reduction:** The operators that consume higher execution time are replaced by the operators consuming less execution time.**4.Null sequences**:Useless operations are deleted.**5.Combine operations:**Many operations are replaced by one equivalent operation.

B. Machine Independent Optimization phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations **1. Common subexpression** is the one which was computed and doesn't change after it last computation, but is often repeated in the program t1=x*z; t2=a+b; t1=x*z;. ->t1=x*z; t2=a+b;. **2. Constant Folding** is a technique where the expression which is computed at compile time is replaced by its value. Eg. int x= 5+7+c;-> int x=12+c**; 3. Constant Propagation:** If any variable is assigned a constant value and used in further computations, constant propagation suggests using the constant value directly for further computations. int c = 5 * 2; int z = a;-> int z = a[10]; **4. Dead code** is a program snippet that is never executed or never reached in a program. It is a code that can be efficiently removed from the program without affecting any other part of the program. intx=a+3; z=a+b;printf("%d",z) ->z=a+b;printf('%d',z) **5.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. Loop Optimization is a machine independent optimization.
Loop Optimization Techniques:1.Frequency Reduction (Code Motion):
In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop**. Initial code:**while(i<100)/n{ a = Sin(x)/Cos(x) + i;/n i++;}**Optimized code**:t = Sin(x)/Cos(x);/n while(i<100)/n{ a = t + i;/n i++;} **2.Loop Unrolling:L**oop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions. **Initial code:**for (int i=0; i<5; i++)/nprintf("Pankaj\n");**Optimized code:**printf("Pankaj\n") write 5 times. **3. Loop Jamming**: Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops. **Initial Code:**for(int i=0; i<5; i++)/n a = i + 5;/n for(int i=0; i<5; i++)/n b = i + 10;**Optimized code:**for(int i=0; i<5; i++)/n{a = i + 5; b = i + 10;}

Phases of a complier:
Lexical Analyzer: It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.
Syntax Analyzer: It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree. The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not. The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree. Any identifier is an expression. Any number can be called a expression. Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression. The parse tree can be compressed to form a syntax tree. **Semantic Analyzer:** It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking. **Intermediate Code Generator –** It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.
Code Optimizer – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent. **Target Code Generator** – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

LL Parser(top down); LR Parser(bottom up)
1)First L of LL is for left to right and second L is for leftmost derivation. L of LR is for left to right and R is for rightmost derivation. 2)**It follows the left most derivation.** It follows reverse of right most derivation. 3)**Using LL parser parser tree is constructed in top down manner.** Parser tree is constructed in bottom up manner**. 4)In LL parser, non-terminals are expanded.** In LR parser, terminals are compressed. 5)**Starts with the start symbol(S).** Ends with start symbol(S). 6)**Ends when stack used becomes empty.**Starts with an empty stack. 7)**Pre-order traversal of the parse tree.** Post-order traversal of the parser tree. 8)**Terminal is read after popping out of stack.** Terminal is read before pushing into the stack. 9) **It may use backtracking or dynamic programming.** It uses dynamic programming. 10) **Example: LL(0), LL(1)** Example: LR(0), SLR(1), LALR(1), CLR(1)

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation. DAG are a type of data structure and they are used to apply transformations to basic blocks. (DAG) facilitates the transformation of basic blocks. DAG is an efficient method for identifying common sub-expressions. It demonstrates how the statement's computed value is used in subsequent statements. The graph's leaves each have a unique identifier, which can be variable names or constants. The interior nodes of the graph are labelled with an operator symbol. In addition, nodes are given a string of identifiers to use as labels for storing the computed value.DAG have topological orderings defined.

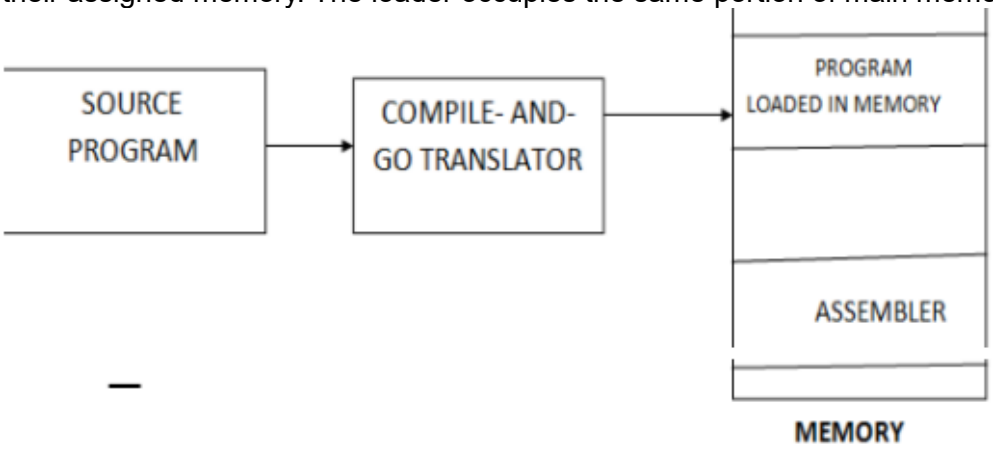
(SDT)Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down to the parse tree in form of attributes attached to the nodes. **Syntax-directed translation rules use** 1) lexical values of nodes, 2) constants & 3) attributes associated with the non-terminals in their definitions. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree. **Evaluation of Semantic rules may**: 1)Generate code 2) Insert information into the symbol table 3) Perform Semantic checks 4) Issues error messages.
Types of attributes: Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production). For e.g. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
Inherited attributes: An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production). For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute. **S-attributed SDT :1)** If an SDT uses only synthesized attributes, it is called as S-attributed SDT.**2)** S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.**3)** Semantic actions are placed in rightmost place of RHS**. L-attributed SDT: 1)**If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.**2)** Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.**3)** Semantic actions are placed anywhere in RHS.

Diff b/w Top-Down Parsing Bottom-Up Parsing:
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.
Top-down parsing attempts to find the left most derivations for an input string. Bottom-up parsing can be defined as an attempt to reduce the input string to the start symbol of a grammar. **we start parsing from the top (start symbol of parse tree) to down (the leaf node of parse tree) in a top-down manner.** we start parsing from the bottom (leaf node of the parse tree) to up (the start symbol of the parse tree) in a bottom-up manner. **Left Most Derivation.** Right Most Derivation. **Example: Recursive Descent parser.** Example: ItsShift Reduce parser.

Difference between System & Application Software: 1. System Software maintains the system resources and gives the path for application software to run. **2. Low-level** languages are used to write the system software. **3. It is** general-purpose software**. 4. Without** system software, the system stops and can't run**. 5. System** software runs when the system is turned on and stops when the system is turned off. **6. Eg**: System software is an operating system, etc. **7. Sys** Software programming is more complex than application software. **8. The Software** that is designed to control, integrate and manage the individual hardware components and application software is known as system software. **9. A sys** software operates the system in the background until the shutdown of the computer. **10. The system** software has no interaction with users. It serves as an interface between hardware and the end user. **11. Sys** software runs independently.
1. Application software is built for specific tasks. **2.** While high-level languages are used to write the application software. 3. While it's a specific purpose software. 4. While Without application software system always runs. 5. While application software runs as per the user's request. 6. Eg: Application software is Photoshop, VLC player, etc. 7. Application software programming is simpler in comparison to system software. 8. A set of computer programs installed in the user's system and designed to perform a specific task is known as application software. 9. Application software runs in the front end according to the user's request. 10. Application software connects an intermediary between the user and the computer. 11. Application software is dependent on system software because they need a set platform for its functioning.

Q.) Loader is a system program that loads machine codes of a program into the system memory. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program. **Functions of Loaders** The main function of a loader is to load executable code into memory and prepare it for execution. several other functions:**1 Allocation/memory managemnet:** Allocates space for program in memory, by calculating size of program**. 2 Error Handling:** The loader must handle any errors that occur during the loading process. This includes detecting and reporting errors such as missing libraries or incorrect file formats. **3 Linking** combines two or more separate objects programmes and supplies the information needed to allow references between them, done by Linker. Resolves symbolic references between object decks. **4 Relocation** modifies the object programme so that it can be loaded at an address different from the location originally specified by the linking loader**. 5 LOADING** allocates memory locations and brings the object program into memory for execution, done by Loader.

Q) schemes of loaders: 1 Compile & go loader: In this scheme, the source code goes into the translator line by line, and then that single line of code loads into memory. In another language, chunks of source code go into execution. Line-by-line code goes to the translator so there is no proper object code. Because of that, if the user runs the same source program, every line of code will again be translated by a translator. So here re-translation happens. **Disadvantages:** In this loader scheme, the source program is converted to object program by some translator. The loader accepts these object modules and puts the machine instruction and data in an executable form at their assigned memory. The loader occupies the same portion of main memory.



2 General loader scheme: In this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts the machine instruction and data in an executable form at their assigned memory. The loader occupies the same portion of main memory. **Advantages:** In this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts the machine instruction and data in an executable form at their assigned memory. The loader occupies the same portion of main memory.

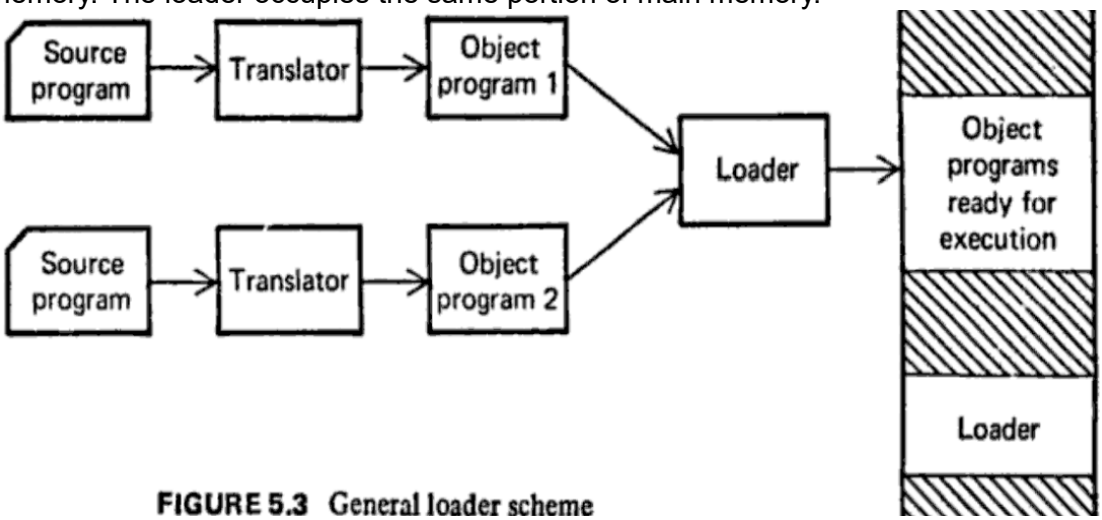
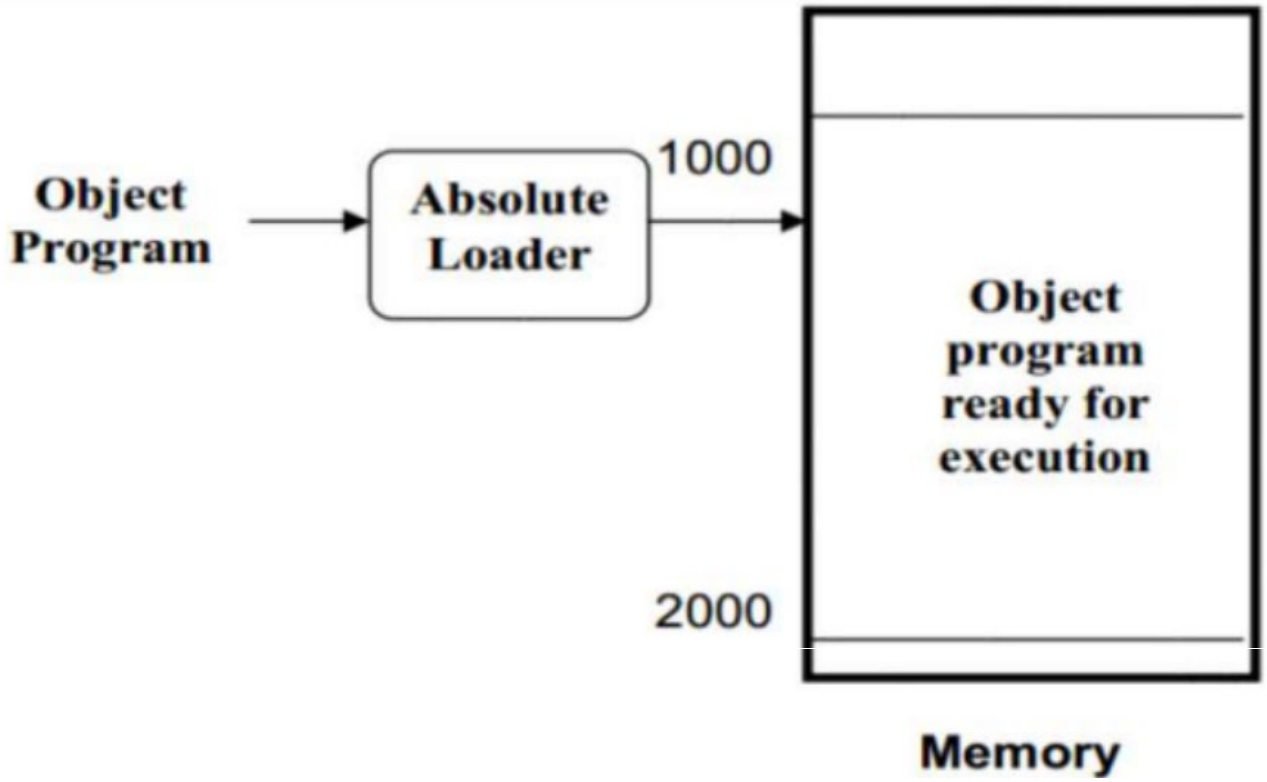


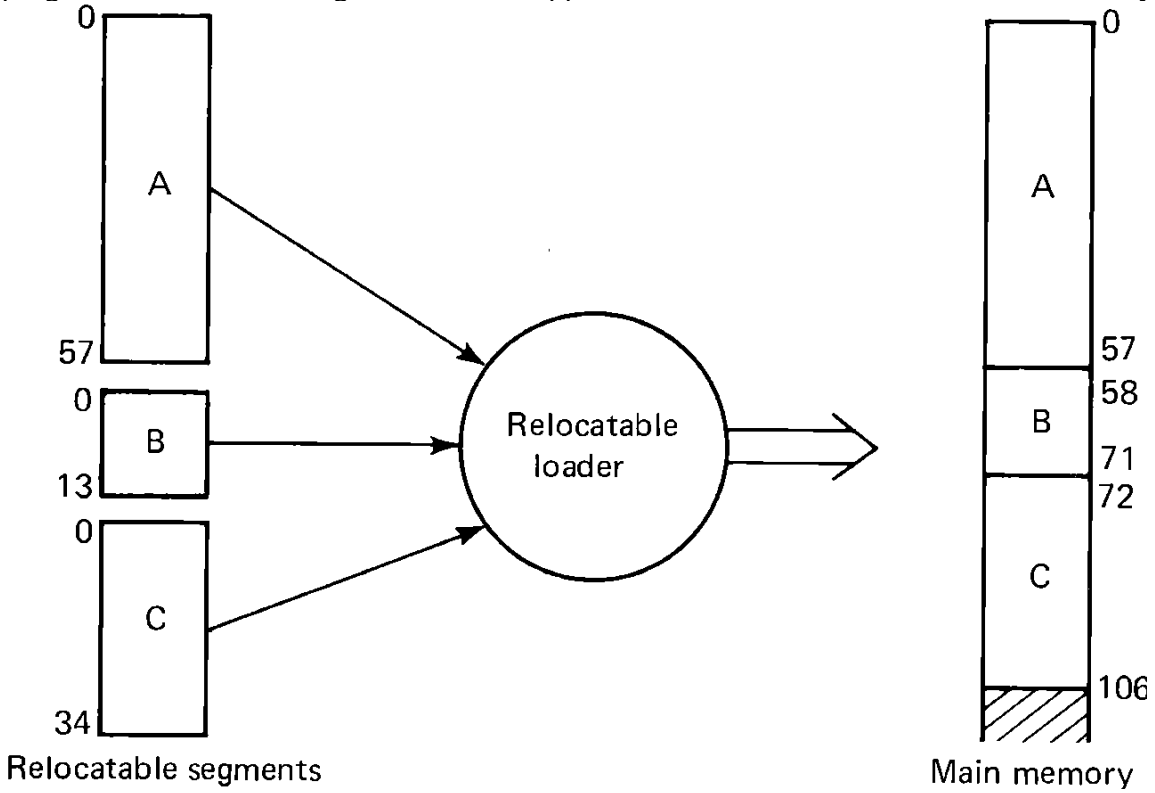
FIGURE 5.3 General loader scheme

Q)Types of loaders 1. Absolute Loader: Absolute loader loads the object code to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. **DISADVANTAGES :** The need for programmer to specify the actual address. Difficulty in using subroutine libraries.



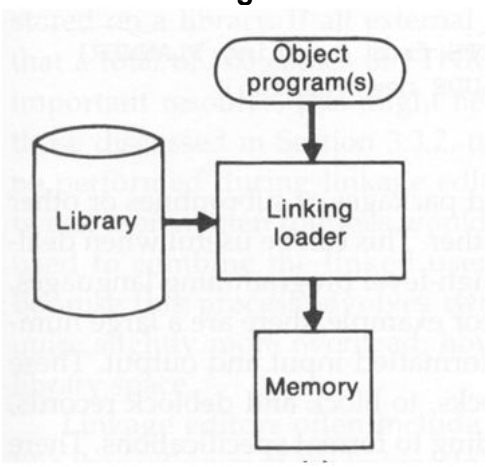
2. Bootstrap Loader: When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer - usually an operating system. The bootstrap itself begins at address 0.

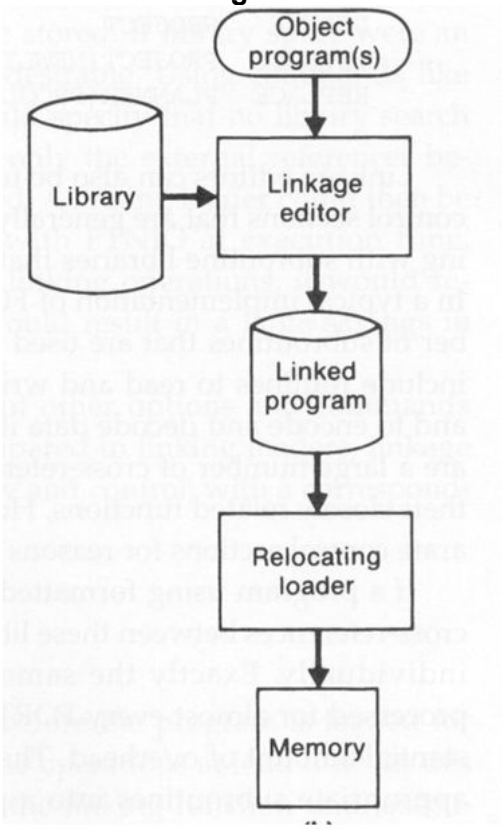
3. Relocating Loader: It is a type of loader that can load the program code into any available memory location. It modifies the program code to adjust for the new memory location. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently.



4**. Linking Loader**: Type of loader that combines multiple object files into a single executable file. It resolves external references between object files. Perform all linking and relocation at load time.

Linking Loader **Linkage Editor**





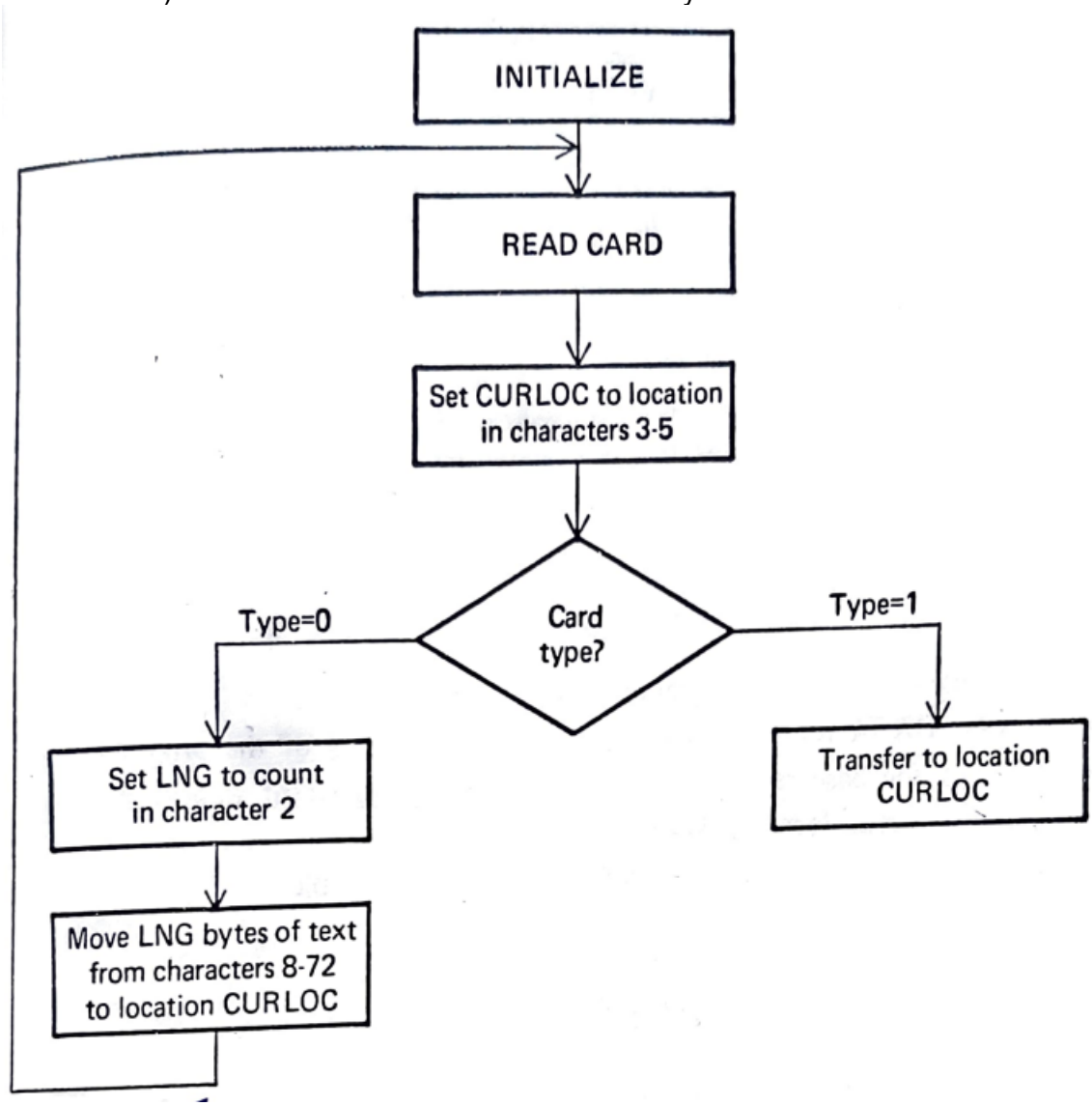
Linkage Editor: Produces a linked version of the program called a load module or an executable image. This load module is written to a file or library for later execution. The linked program produced is suitable for processing by a relocating loader. Using Linkage editor , an absolute object program can be created, if starting address is already known.

4. Dynamic Linker: It is a type of loader that loads and links shared libraries at run time. It is used to reduce the memory usage of large programs by sharing common code between processes. This scheme postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called. It is usually called dynamic linking, dynamic loading or load on call. Allows several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when they are needed. Run-time linker uses dynamic linking approach which binds dynamic executables and shared objects at execution time.

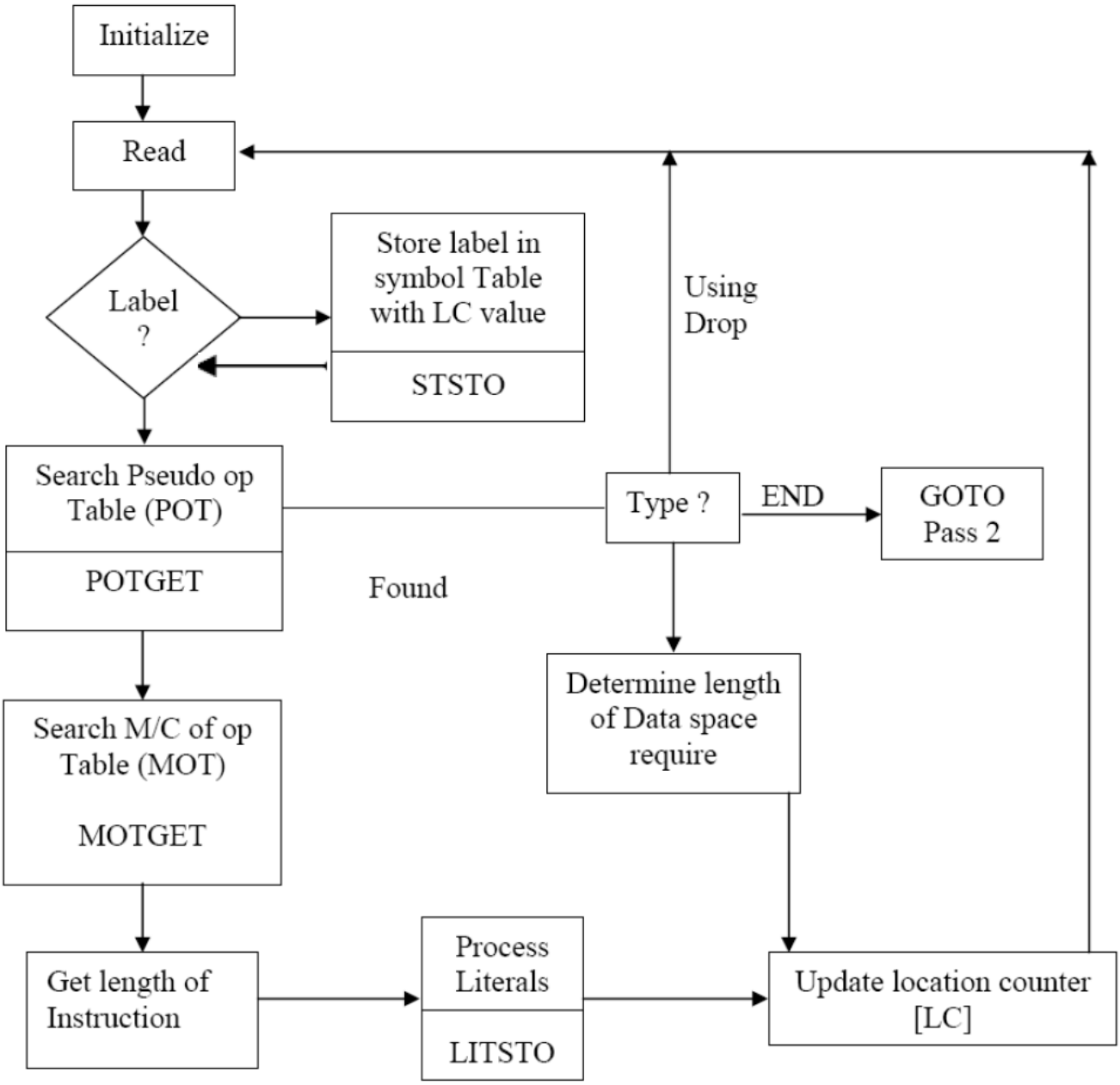
Q.4) Linking loader vs dynamic direct linking loader

Linking loader and dynamic direct linking loader are both programs that load executable code into memory and handle the linking phase of program execution. Here are the main differences between the two: **Linking:** A linking loader performs the linking phase at load time, whereas a dynamic direct linking loader performs linking at run time. In other words, a linking loader combines multiple object files into a single executable program at load time, while a dynamic direct linking loader does this at run time. **Dependency resolution:** A linking loader resolves external references to functions and data during load time by linking multiple object files. A dynamic direct linking loader resolves external references during run time by dynamically linking the necessary libraries. **Load time:** A linking loader may take longer to load a program into memory since it has to perform the linking phase at load time. A dynamic direct linking loader loads a program faster since it can resolve external references at run time. **File size:** A linking loader can reduce the size of the executable file by eliminating redundant information, while a dynamic direct linking loader may result in larger executable files since all the necessary information is included in the object file. **Flexibility:** A linking loader provides more flexibility since it allows for dynamic linking of libraries at run time.

Design of Absolute Loader: Assembler takes as input assembly language program and generates the object program ina format i.e. req by the loader. **2 cards: A) TXT card:** this card contains the binary information. **1) card type: always 0** for TXT card **2) count:** which indicates in bytes the amount of binary information available **3) address:** specifies address at which binary information should be transferred **4) contents:** which contains the binary information
B) Transfer card: this card specifies the address at which the control should be transferred. 1) card type: is always 1 for transfer card. 2) count: this field is always 0 as this card contains no binary information 3) address: this field specifies the address at which the control should be transferred 4) contents: is blank as it contains no binary information.



Design of Direct linking loader: works in 2 passes. Purpose of pass 1 is to define the segments and their local definitions and purpose of pass 2 is to transfer the text cards and perform relocation and linking. **A) Pass 1 Databases: 1) Object cards** (ESD, TXT, RLD, END, EOF) produced by assembler in a format that is req by the loader. **2) IPLA:** obtained by the loader from OS. **3) PLA (program load address)** which is used to assign address to the segments. **4) GEST (global external symbol table)** keeps a track of segments and their local definitions. **5) Copyfile:** which is prepared to be used by Pass 2. **B) Pass 2 Databases: 1)Copyfile** prepared by pass 1. **2) IPLA 3)PLA 4)GEST 5) EXADDR:** execution address which specifies the address at which the control should be transferred. **6) LESA (Local external symbol array):** which is prepared with the help of ESD and GEST. if an address is specified on the END card, that address is used as the execution start address otherwise, execution will commence at the beginning of the first segment.



Machine Opcode Table (MOT): 000-RR- 2 bytes-Absolute; **001-RX-4 bytes- Relocatable**;

Mnemonic op-code (4 byte) character	Binary op-code (1 byte) Hex	Instruction length (2 bits) (binary)	Instruction format (3 bits) (binary)	Not used in this design 3
LOAD	5A	10	001	-
ST	4A	11	001	-

Pseudo-Opcode Table:

Pseudo-op (5 bytes) (character)	Address of routine to process pseudo-op 3bytes = 24 bit
START	
USING	
DC/DS	

Symbol Table and Literal Table:

Symbol/Literal (8 byte) Character	LC Value (4 byte)	Length (1 byte) Hex	Relative/Absolute
RAM	0	1	R
FOUR	12	4	R
FIVE	16	4	R
TEMP	20	4	R
F'5' (literal)	24	4	R

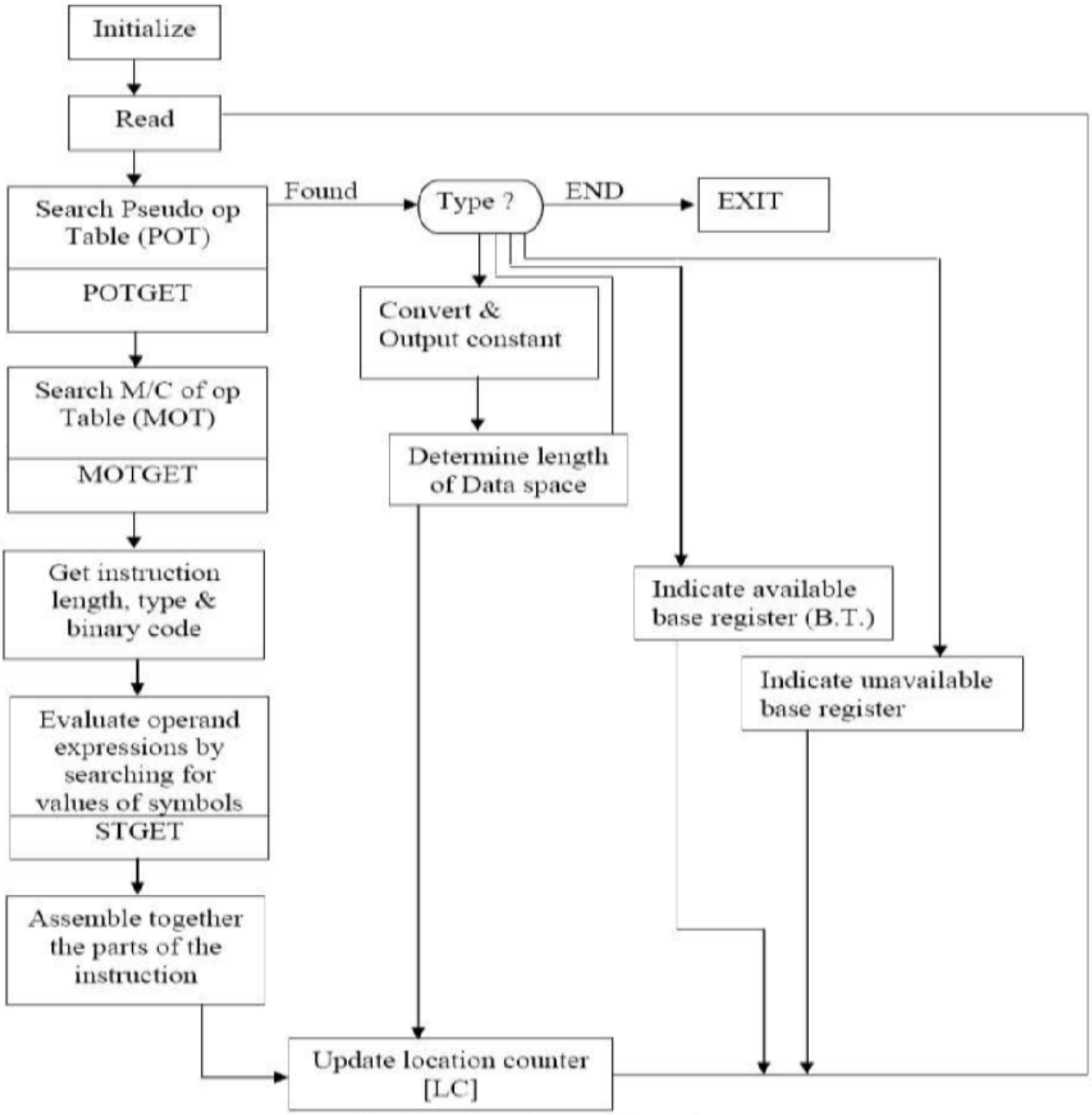


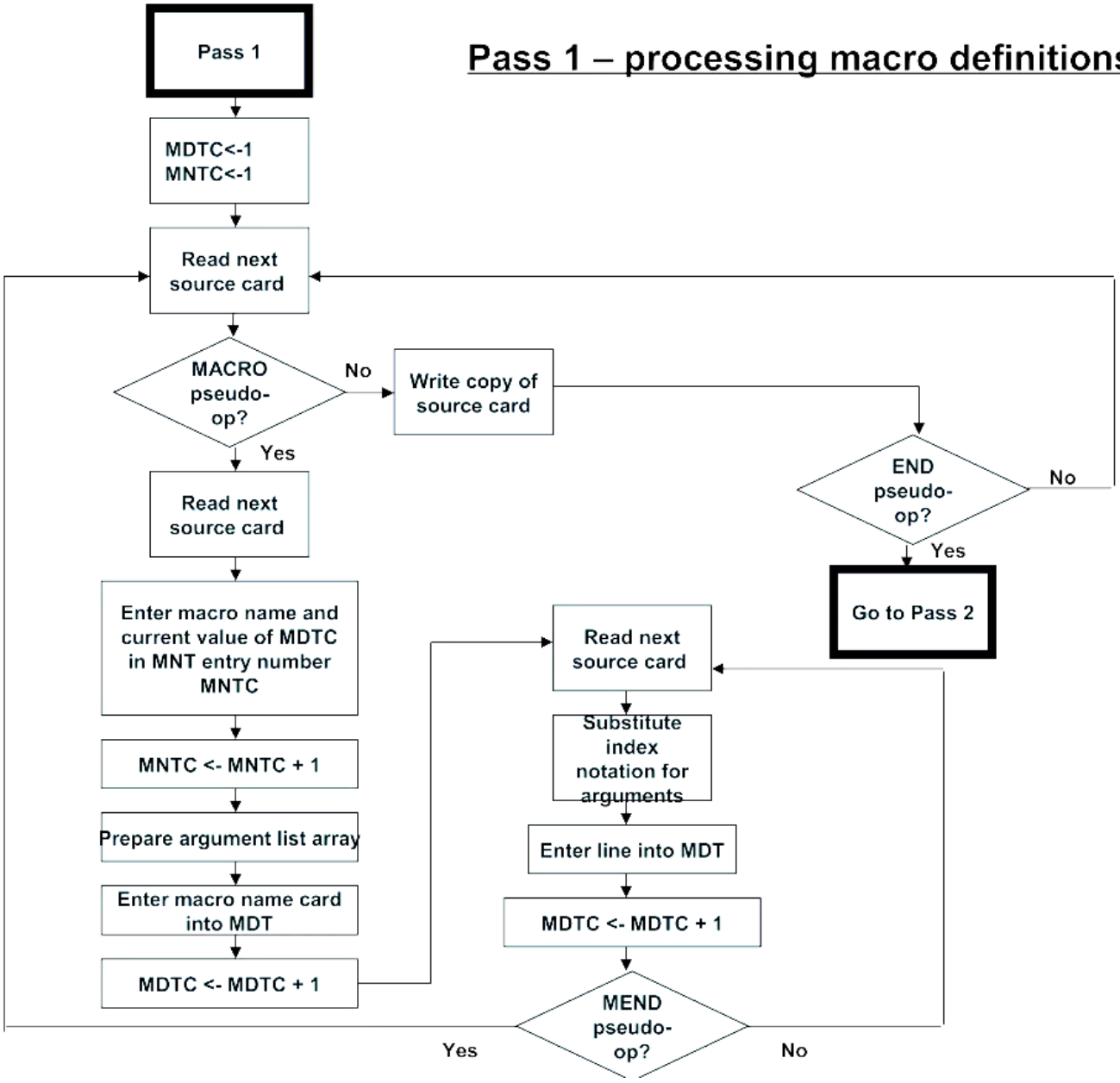
Fig. 1.5 Flow chart of Pass -2

BT indicates which register are currently specified as base registers by USING pseudo-ops and what specified contents of these registers are

Base Table (when USING*, 15)

←---------- 4 bytes / entry ----------→

	Availability Indicator 1 byte Character	**Designated relative–address contents of base register (3 bytes = 24 bit address) Hex**	
1	**"N"**	**-**	**^**
2	**"N"**	**-**	**|**
.	**.**	**.**	**15 entries**
.	**.**	**.**	**|**
15	**"Y"**	**00 00 00**	**^**



Macro Name Table (MNT): used to store name of macro along with pointer to MDT (MDT index) where the corresponding definistion is stored

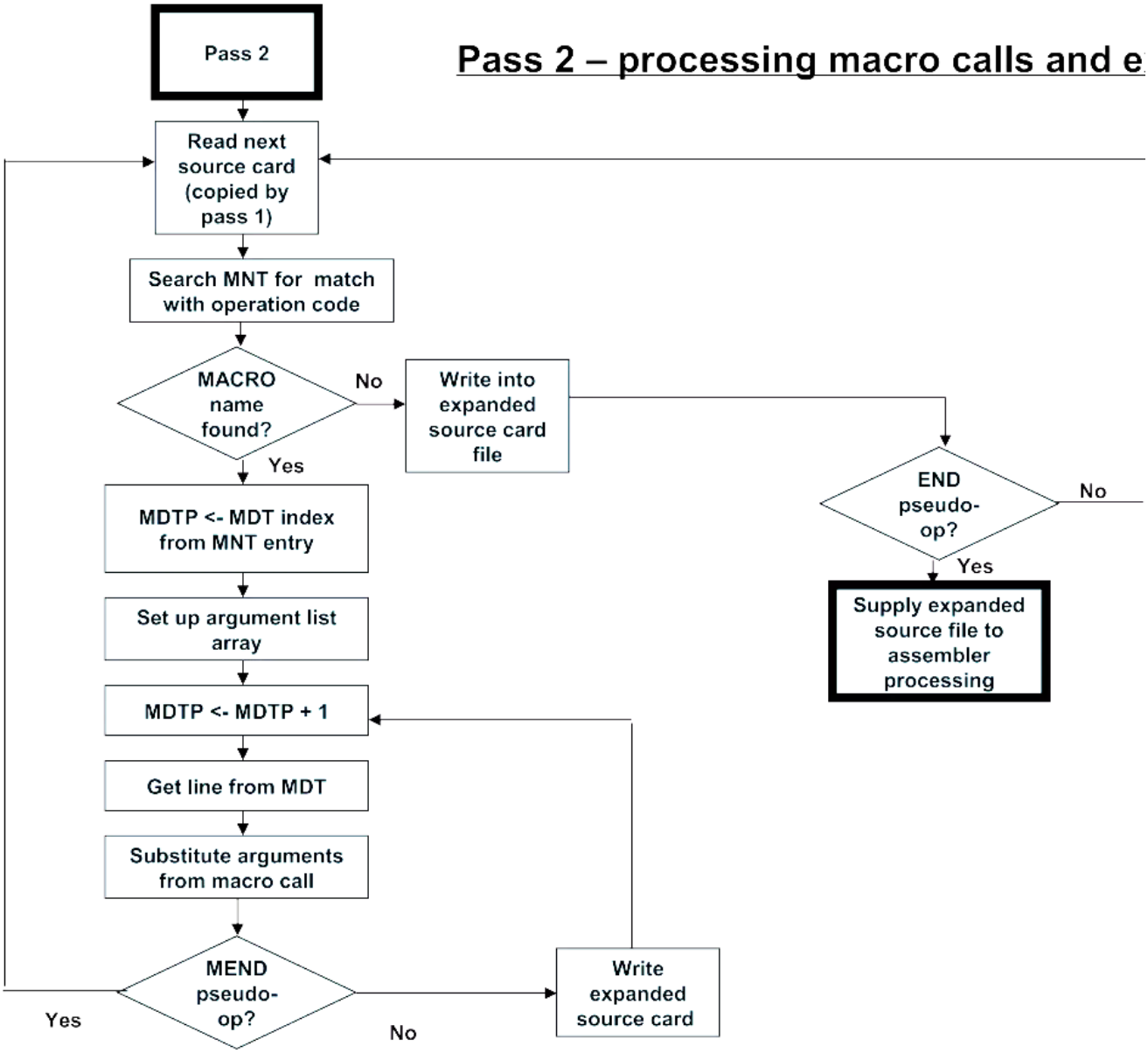
MNTC	Macroname	MDT index
1	INCR	1
2	INCR2	6

Macro Definition Table (MDT)

MDTP	Definition	MDTP	Definition
1	INCR1 & ARG	6	INCR2 & ARG
2	A 1, # 0	7	A 1, # 0
3	A 2, # 0	8	A 2, # 1
4	A 3, # 0	9	A 3, # 0
5	MEND	10	MEND

Argument List Array (ALA): IN pass 1 ALA is used to substitute index notations for the arguments before saving it in MDT. IN pass 2 ALA is used to substitute arguments from macro call for resp. index notation. Eg: formal arg indices and actual arguments. Label arg is first->0 index

Index	Argument
0	data 1
1	data 2



Output: expanded source code

First:set of terminals that begin strings serived from alpa, when alpha is in sentential form.
Rule 1: if X is a terminal, then first(X)={X} **Rule2:** if X ->aaplha, then first of X={a}
Rule 3: IF X->Y1,Y2…Yn, then j=0, repeat 1) j=j+1 2) First(X) = First (Yj)-{epsilon} 3) if First (Yj) dpes not contain epsilon, then break
Follow: set of terminals which can appear to the RHS of A in some sentential form. Repeat the following until you can add nothing more **1)** Follow(A)={$} if A is a start variable
2) if A->alpha B beta, then Follow of (B)= first(beta)-{epsilon}, if first (beta) contains epsilon, then add Follow(A) to Follow of (B) 3) If A->alpha B, then follow (B) = Follow (A)
Left Recursion: if A->A alpha|Beta, then 2 new rules: A-> beta A' and A' -> alphaA'
Predictive Parser: if epsilon occurs in first, then v use follow in the parsing table
Left Factoring: if A->a aplha1| a alpha2| a alpha3, then A->aA' and A'->alpha1,alpha2,aplha3
LL1: if parsing table has 2 entries in same cell, then its not LL1 or if first=follow of the same A
for . after terminal, we reduce and write r with no of original production and for LR0 write in all 3 columns and in SLR, v write on in that particular cell, not all 3 columns
operator precedence: ^ raised to symbol has highest precedence and) too. (has lowest precedence. Id has higher precedence than *
Operator precedence: a<b **shift**; a>b **reduce**, if non terminal, ignore and use $

FORWARD REFERENCE PROBLEM: If the symbol is used before its declaration, tehn the synthesis phase cannot continue its task
Types of Assembly Language statements: 1) Imperative statements: An imperative statement in assembly language indicates the action to be performed during execution of assembly statement assembly statement. Ex:- A 1,FOUR **2) Declarative Statement:** These statements declares the storage area or declares the constant in program. EX A DS 1; ONE DC '1'; **3) Assembler Directives:** These are the statements used to indicate certain thing regarding how assembly of input program is to be performed. Ex START 100; USING *, 15

Error Recovery Techniques: 1) Panic mode: When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.**2) Statement mode:** When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.
3)Error productions: Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.**4) Global correction:** The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet. **5) Abstract Syntax Trees:** Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

Macro in system programming is a feature of a programming language that allows programmers to define an abbreviation for a piece of code. The programmer uses this abbreviation wherever he/she has to repeat the corresponding piece of code in the program.
Features of Macro Facility:
a)Macro with no argurment: MACRO;INCRI;A 1,5; MEND **B) Macro with 1 Argument:** MACRO;INCRI & ARG; A 1,& ARG; MEND….INCRI DATA1..>A1, DATA 1 **C)Macro with multiple arguments:** MACRO; INCRI &ARG1, &ARG2, &ARG3; A 1, &ARG1; A2,&ARG2; A3,&ARG3; MEND…INCRI DATA1, DATA2,DATA3…>A1, DATA1; A2,DATA2; A3,DATA3 **D) Macro argument with label:** MACRO; INCRI &ARG1, &ARG2, &ARG3, &LAB; &LAB A 1, &ARG1; A2,&ARG2; A3,&ARG3; MEND…INCRI DATA1, DATA2,DATA3,Loop1…>Loop1 A1, DATA1; A2,DATA2; A3,DATA3 **e) Conditional assembly:** MACRO; EVAL &A,&B,&C; AIF (&A EQ &B).NEXT; LOAD &A; SUB &B; ADD &C; AGO FINISH; NEXT LOAD &C FIISH MEND….. EVAL X,Y,Z ..> LOAD X; SUB Y; ADD Z **f)Nested Macro call** is when a macro is within a macro **g)Expansion time control flow h) Lexical substitution**

Macro call leads to macro expansion. Macro call statement is replaced by a sequence of assembly statement in macro expansion. **2 actions performed during macro expansion. A)Lexical substitution/ Parameterized Macro:** Like function definitions, we write macro definitions using formal parameters. Lexical substitution replaces the formal parameters with the actual parameters present in the macro call. **B)Semantic Expansion:** This process generates a sequence of statements. These statements are specifically tailored to meet the requirements of the corresponding macro call. **Macro Definition:**
To define a macro you have to just choose an abbreviation that suits your macro definition. The macro definition starts with the MACRO keyword followed by the macro abbreviation. The MACRO abbreviation is followed by formal parameters. **Algorithm for macro expansion:**
1. Initialise the Macro Expansion Counter (MEC) 2. Check the statement which is pointed by MEC is not a MEND statement. a) If the statement is model statement then expand the statement and increment the MEC by 1 else MEC = new value specified in the statement;
3. Exit from the macro expansion.

Difference between Token, Lexeme, and Pattern:
Token:1) Token is basically a sequence of characters that are treated as a unit as it cannot be further broken down**. 2)** all the reserved keywords of that language(main, printf, etc.) **3)** name of a variable, function, etc **4) all** the operators are considered tokens**. 5)** each kind of punctuation is considered a token. e.g. semicolon, bracket, comma, etc. **6)** Interpretation of type Literal: a grammar rule or boolean literal. **7)** int a = 10; //Input Source code. **Tokens:** int (keyword), a(identifier), =(operator), 10(constant) and ;(punctuation-semicolon). Total number of tokens = 5
Lexeme:1) It is a sequence of characters in the source code that are matched by given predefined language rules for every lexeme to be specified as a valid token. **2)** int, goto **3)** main, a **4)** +, = **5)** (,), {, } 6) "Welcome to SPCC!" **7)** main is lexeme of type identifier(token)
(,),{,} are lexemes of type punctuation(token).
Pattern 1) It specifies a set of rules that a scanner follows to create a token. 2) The sequence of characters that make the keyword.3) it must start with the alphabet, followed by the alphabet or a digit. 4) +, = 5) (,), {, } 6) any string of characters (except ' ') between " and 7) For a keyword to be identified as a valid token, the pattern is the sequence of characters that make the keyword. For identifier to be identified as a valid token, the pattern is the predefined rules that it must start with alphabet, followed by alphabet or a digit.

Garbage Collection 1)It is a process for automatically freeing up pooled storage that a program no longer needs.2) Garbage collection is an automatic memory management feature in many modern programming languages.3) It is responsible for freeing up memory for use by other programs and ensuring that a program using increasing amounts of pooled storage does not reach its quota.4) In older programming languages like C and C++, memory allocation and freeing is done manually by the programmer. 5) Manual memory control can introduce bugs in the code like memory leaks or dangling pointers. 6) Automatic garbage collection tries to eliminate these bugs by automatically detecting when a piece of data is no longer needed. 7) A GC has two goals: freeing up any unused memory and not freeing up any memory that the program will still use.
Compaction:1) Compacting is a process used by GC to free up space in the heap by moving cells between arenas. **2)** To free up space, the GC moves cells between arenas and consolidates the live cells in fewer arenas. **3)** This operation is potentially expensive as every pointer to a moved cell must be updated. **4)** Compacting the heap is usually done only when memory is low or the user is inactive. **5)** The algorithm for compaction works in **three phases:** selecting the cells to move, moving the cells, and updating the pointers to those cells.

Lex and YACC
1) Lex, short for "Lexical Analyzer", breaks up an input stream into more usable elements. **2)** It identifies the "interesting bits" in a text file by recognizing the significance of symbols and keywords in a given language**. 3)** The purpose of Lex is to provide a more structured view of the input stream by separating the tokens from irrelevant characters such as spaces and newlines. **4)** In the context of a C-compiler, Lex would identify symbols like { } ( ) ; and recognize keywords like int, char, etc. **5)** Lex also determines whether a particular occurrence of a word is a keyword or a variable by looking at the context in which it appears. **6)** The lexical analyzer ensures that the syntax of the input stream is preserved while providing a more structured view of the input stream.

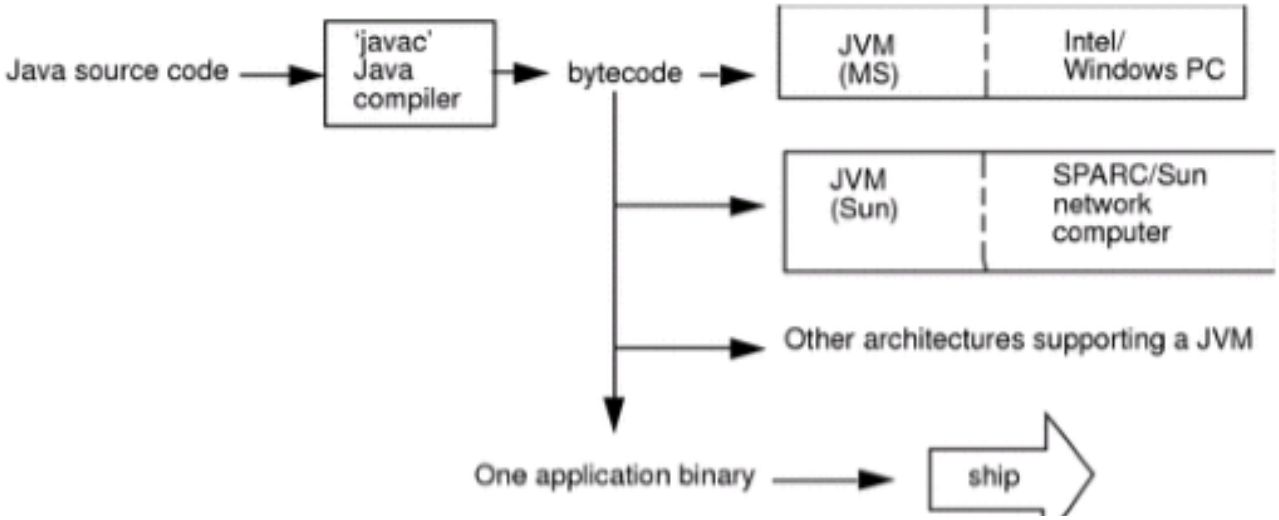
1) Yacc, short for "Yet Another Compiler Compiler", is a parser that analyzes the structure of the input stream. **2)** Its main job is to operate on the "big picture" and ensure that the input is syntactically sound.**3)** In the context of a C-compiler, Yacc verifies that a word is a function name or a variable based on whether it is followed by a (or a =. **4)** It also ensures that there is exactly one } for each { in the program, among other things.**5)** Yacc provides a high-level view of the input stream by grouping the tokens identified by Lex into larger structures**. 6)** The parser decides if the token represents a valid structure based on the grammar of the language. **7)** The grammar defines the rules of the language, such as the syntax for function declarations, loops, and conditional statements. **8)** Yacc checks that the input stream adheres to the grammar of the language and produces meaningful output based on the input.

Java Compiler Environment: Javac compiles Java programs.

SYNOPSIS: Javac [options] filename.java ...
javac_g [options] filename.java ...

Java virtual machine (JVM) is an abstract computing machine that enables a computer to run a Java program. **Java Runtime Environment (JRE)** is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library. **Java Development Kit (JDK)** is a superset of a JRE and contains tools for Java programmers, e.g. a java compiler.

New Portable Java Compile-Time Environment:



Description: 1) The javac command compiles Java source code into Java bytecodes. You then use the Java interpreter - the java command - to interprete the Java bytecodes.
2) Java source code must be contained in files whose filenames end with the .java extension. The file name must be constructed from the class name, as classname.java, if the class is public or is referenced from another source file.
3) For every class defined in each source file compiled by javac, the compiler stores the resulting byte codes in a class file with a name of the form classname.class. Unless you specify the -doption, the compiler places each class file in the same directory as the corresponding source file.
4) When the compiler must refer to your own classes you need to specify their location. Use the -classpath option or CLASSPATH environment variable to do this. The class path is a sequence of directories (or zip files) which javac searches for classes not already defined in any of the files specified directly as command arguments. The compiler looks in the class path for both a source file and a class file, recompiling the source (and regenerating the class file) if it is newer.
5) Set the property javac.pipe.output to true to send output messages to System.out. Set javac.pipe.output to false that is, do not set it, to send output messages to System.err.
javac_g is a non-optimized version of javac suitable for use with debuggers like jdb.

Difference b/w Compiler & Interpreter: 1)Compiler is a sys program which compiles complete source program at a time. Interpreter compiles one line at a time. **2) Compiler generates intermediate code.** Doesn't generate IC **3) as compiler takes complete code, it needs more memory to store it during compilation.** Consumes less memory as it takes single lines during interpretation. **4) Source code is compiled once and run anytime.** Needs to be interpreted everytime to run. **5) Errors are displayed after entire source code is compiled.** Errors are displayed as soon as encountered**. 6) eg: gcc compiler.** Eg: java byte code interpreter.

Difference b/w macro and function: 1) Macros are Preprocessed. Functions are Compiled. **2) No Type Checking is done in Macro.** Type Checking is Done in Function. **3) Speed of Execution using Macro is Faster.** Speed of Execution using Function is Slower **4) Before Compilation, macro name is replaced by macro value.** During function call, transfer of control takes place. **5) Macros are useful when small code is repeated many times.** Functions are useful when large code is to be written. **6) Macro does not check any Compile-Time Errors.** Function checks Compile-Time Errors

Algorithm for Operator Precedence: repeat forever {
if the input is scanned completely and stack contains only the start variable then accept and halt
else let a be the topmost terminal symbol on the stack and let b be the input terminal symbol
{if a<. b or a=.b then
shift
else if a.> b then
reduce
Else ERROR() } }

Algorithm for Predictive Parser LL(1): { Let x be the stack top and a be the input symbol.
if x=a=$ tehn accept and break
else if x=a=/(not equal to)$ then
pop x from stack
remove a from input
else if M[X,a]= X->Y_1, Y_2….Y_k
pop X from stack
push Y_k Y_{k-1} ….Y_1
else Error() }

Recursive Descent Parser Algorithm for given BNF: S->AB; A-> aA | b; B->bB | a

```
procedure S();
  begin
        A()
        B()
  end
procedure A();
  begin
        if (input_symbol = 'a') then
        begin
              ADVANCE();
              A()
        end
        else
        begin
              if (input-symbol ='b') then
              begin
                    ADVANCE()
              end
              else
              begin
                    ERROR()
              end
        end
  end
```