

Department of Computer Science

Gujarat University



Certificate

Roll No: 10

Seat No: _____

This is to certify that Mr./Ms. Pradip S Karmakar _____
student of MCA Semester – III has duly completed his/her term work for
the semester ending in December 2020, in the subject of
System Software (SS) _____ towards partial fulfillment of
his/her Degree of Masters in Computer Applications.

Date of Submission
10 - December - 2020

Internal Faculty

Head of Department

Department Of Computer Science
Rollwala Computer Centre
Gujarat University

MCA - III

Subject: - System Software (SS)

Name: - Pradip S Karmakar

Roll No.: - 10

Exam Seat No.: - _____

Sr. No.	Contents	Pg. No	Date	Signature
1	Assembler	3	10-12-20	
2	Macro	28	10-12-20	
3	Scanner & Parser	39	10-12-20	
4	Linker & Loader	57	10-12-20	
5	Compiler	69	10-12-20	
6	Programs 1. Assembler 2. Macro Preprocessor 3. Top Down without Backtracking 4. Recursive Decent Parser 5. Operator Precedence Parser 6. LL1 Parser 7. Scanner	84 91 101 105 108 112 115	10-12-20	

Assignment - 1

Define the following instruction

- ① STOP :- It is an imperative statement. It stop the execution of program. OPCODE is 00.
- ② SUB :- It is an imperative statement. Its OPCODE is 02. It subtract the value with the register value.
- ③ ADD :- It is an imperative statement. Its OPCODE is 01. It add the value with the register value.
- ④ MULT :- It is an imperative statement. Its OPCODE is 03. It multiply the value with the register value.
- ⑤ MOVER :- It is an imperative statement. Its OPCODE is 04. It is used to move the value from memory to register.
- ⑥ MOVEM :- It is an imperative statement. Its OPCODE is 05. It is used to move the value from register to memory.
- ⑦ COMP :- It is an imperative statement. Its OPCODE is 06. It is used to compare and set the condition code.

- ⑧ BC :- It is an imperative statement. Its opcode is 07. It is used for the branch condition.
- ⑨ READ :- It is an imperative Statement. Its opcode is 09 . It is used to read a value
- ⑩ PRINT:- It is an imperative statement. Its opcode is 10. It is used to print the contents of register.
- ⑪ ORIGIN:- It is an Assembler directive statement. This directive instruct the Assembler to put the address given by <address specification> in the location counter counter.
- ⑫ EQU :- It is an Assembler directive statement. In EQU left side is label / symbol and right side is address.
- ⑬ PURGE :- It is an Assembler directive statement. It is used undefined the symbol names which are defined by the ORIGIN statement.
- ⑭ ASSUME :- It is an Assembler directive statement. It tells the assembler that it can assume the address of the indicated

segment to be present in (register)

- (15) SEGMENT :- It is the memory used to store three components of a program i.e. Program code, data & stack.
- (16) PROC :- It indicate that it is a procedure
Example CALCULATE PROC
Here CALCULATE is procedure.
- (17) NEAR :- It indicate that whether the call to the procedure is to be assembled as far call ~~be~~ i.e. the procedure is called from the same segment.
- (18) FAR :- It indicate that whether the call to the procedure is to be assembled as far call. i.e. the procedure is called from the another segment.
- (19) PUBLIC :- When a symbol is define as a public in a program that means it can be referred by the another program units also.
- (20) EXTERN :- When an assembly module wishing to use a symbol declare in another assembly module. The symbol

Should be declare as extern.

Q1) OFFSET :- It is a displacement from the base address of a segment.

* Explain the meanings.

Q1) Base Register :-

→ The Base register is used for program relocation.

→ It holds the base address of data in RAM to be executed recently.

→ Example :- Let's say we have memory and in that we have stored a program P1 from 100 to 199

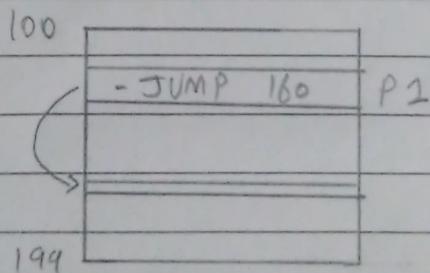


Diagram 1

→ So according to diagram 1 there is an instruction JUMP 160 (using absolute address) so now control will be at address 160. Here all works good.

→ But now this program is send to secondary memory for sometimes, then another program i.e. P2 will occupy address 100 to 199.

→ Then again program P1 comes to back to main memory than new address is allocated.

200

P2

199

100

JUMP 160

P1

99

Diagram 2

→ So now the problem will occur as we have change the memory address of program P2 but it is having an instruction JUMP 160 so it create an illegal instruction so the program P2 memory is from 100 to 199.

→ So to solve this problem we have to add the displacement instead of giving absolute value

→ So the effective address

EA : Base register value + displacement

→ So there will be no problem if address is changed.

② Index register :-

→ The index registers have the source index of array and distinction index. i.e., offset. They are provided with auto increment and auto-decrement facility.

Example :-

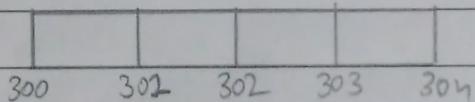


Diagram 1

Here in Diagram 1 the source index is 300. If we want to move at 303

Then,

$$\begin{aligned}\text{Effective address} &= \text{source index} + \text{offset} \\ &= 300 + 3 \\ &= 303\end{aligned}$$

This way we can move to 303 address.

③ Symbol table :-

→ An identifier used in the source program is called symbol. Thus names of variables, functions and procedures are symbols.

→ Language Processor uses the symbol table to maintain the information about the symbols used in source program.

Symbol table performs following things:-

- ① Add the symbol
- ② Add the location of symbol
- ③ Add the length of symbol
- ④ Delete the symbol entry
- ⑤ Access the symbol entry

SYMTAB

Symbol	Address	length
LOOP	202	1
NEXT	214	1

⑥ Literal Table :-

→ The use of literal table is to collect all literals used in a program. At any stage the current literal pool is the last pool in Littab. On encountering an LTORG statement literals in the current pool are allocated addresses starting with the current value in LC.

Littab

literal	address
= '5'	211
= '2'	212

⑤ Pool table:-

→ This table contains the literal number of the starting literal of each literal pool

Pool table

<u>literal</u>	<u>no</u>
#1	
#2	

⑥ Optab :-

→ optab contains the fields mnemonic code, opcode and size of the operator. It also contain the class field. which tells us that the statement is imperative, Assembler directive or declaration statement.

→ If an imperative statement the mnemonic into field contains the pair else it contain the id of a routine to handle the declaration or directive statement.

⑦ Declaration Statement:-

→ The syntax of declaration statement is as follows :-

[label] DS <constant>
[label] DC <values>

→ The DS statement reserves area of memory and associate names with them.

→ The DC statement constructs a memory word containing constants.

Eg. A DS 200
ONE DC '1'

⑧ Imperative Statements:-

→ As imperative statement indicates an action to be performed during the execution of the assembled program each imperative statement typically translate into one machine instruction

Eg. MOVER AREG, X

① Assembler Directives:-

Assembler Directives instruct the assembler to perform certain action during the assembly program.

Eg. START.

This directive indicate that first word of machine should be placed in the memory word with address <constant>

START <constant>

START 200

* Difference between the following.

① Application Domain

→ In this domain designer express their ideas.

Execution Domain

→ In this domain the ideas express by designer are implemented.

→ The code of any program is written

→ On basis of code written output is generated.

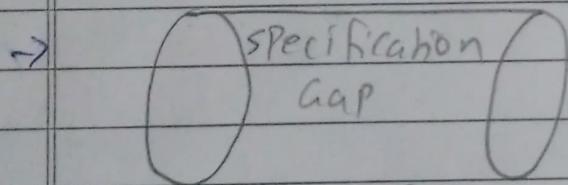
→ In application domain algorithm and syntax is checked

→ In execution domain the run time problem are checked.

- | | |
|---|--|
| → In application domain work of analysis phase is done. | → In execution domain work of synthesis phase is done. |
| → Eg. <code>Pointf("Hello world");</code>
Here we write the command and its symbol is checked. | → Eg. Hello world
By processing the command output is generated by execution domain |

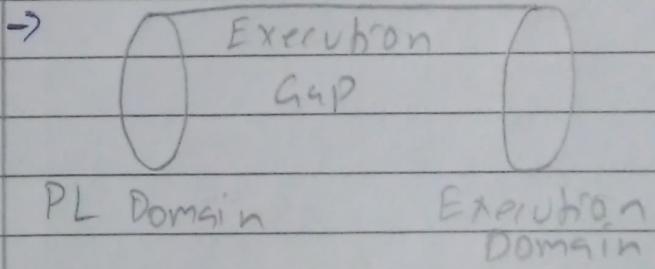
② Specification Gap

→ The gap between the application domain and the programming domain is called specification gap.



Execution Gap

→ The gap between the programming domain and execution domain is called execution gap.



→ Specification gap is the semantic gap between two specification and same task.

→ Execution gap is the gap between the semantic of program written in different programming language.

→ The specification gap is bridged by the software development team.

→ Execution gap is bridged by designers of PL processor, translator or interpreter.

→ Eg. suppose a person who don't know programming language can use the application. but no able to do programming

→ Eg. Programmer write program in high level language and computer does not understand the high level language directly.

③

Source program

→ A program written in a PL is called source program.

target program.

→ Target program is an o/p generated after compiling the source program.

→ Source program is written in PL by programmer → Target program is compiled and generated by the compiler.

→ Source program contains English words according to the syntax of ~~PL~~ PL.

→ Target program only contains Binary code.

→ Programmer can read the SOURCE program.

→ Machine can read the target program.

→ Source program is an input to compiler

→ Target program is the output to compiler.

(4)

Compiler

→ Compiler converts the entire program

→ It produce optimized code.

→ Analysis time is more.

→ Machine code is generated

→ There is a execution gap

Interpreter

→ Interpreter translate code one line at a time.

→ Code is not too much optimized

→ Program analysis time is less

→ No machine code is generated

→ There is no execution gap

(5)

Language translator

→ Language translator is bridges an execution gap to the machine language of the computer system.

→ Assembler is an language translator whose source code is assembly language and convert it to machine language.

Preprocessor

→ It is a language processor which bridges an execution gap but is not a language translator.

→ Preprocessor process high level language before language translator takes over.

⑥ Problem oriented

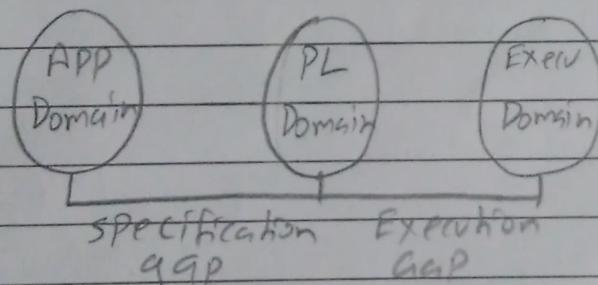
→ Problem oriented language are specific oriented language.

→ Problem oriented language have large execution gap

→ Execution gap is bridged by translator or interpreter

→ Eg. COBOL, FORTRAN

→



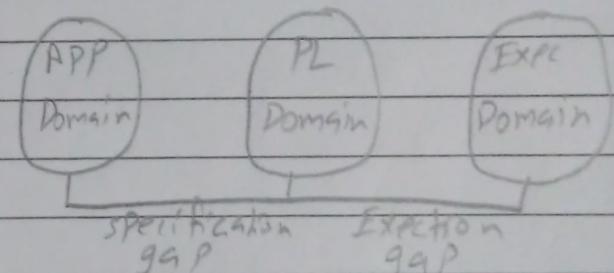
Procedure oriented.

→ procedure oriented language are common oriented language.

→ procedure oriented language have large specification gap

→ specification gap is bridged by an application designer

→ Eg. C, C++.



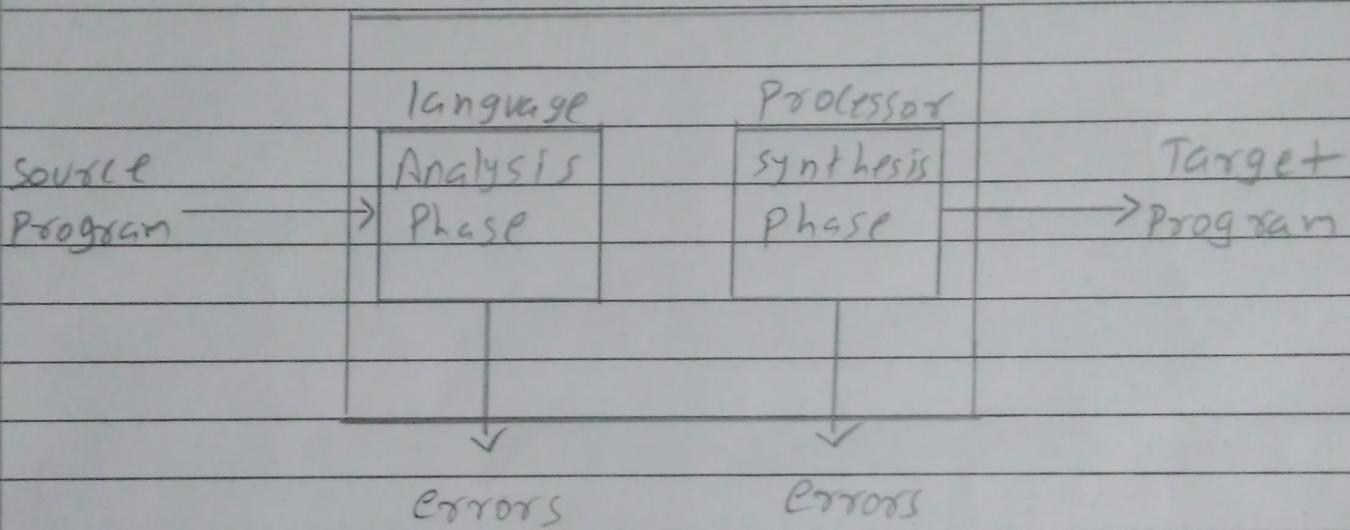
⑦

lexical, syntax & semantic rule

lexical : - It governs lexical rules in the source language (eg Identifier name is proper or not)

Syntax :- It checks validity of statement for syntax in the source language.

Syntactic analysis : It checks the semantic rules in the source language.



⑧ Analysis Phase

→ Analysis Phase use each components of source program to determine relevant information out of it.

Synthesis Phase

→ Synthesis Phase generates/ computes target language statement which have same meaning as source language statement.

→ Analysis phase consists of lexical, syntax and syntactic analysis.

→ Synthesis Phase consists of 2 activities.
(i) creating data structure in target program

	(ii) generation of target code.
→ Intermediate Representation is the O/P of Analysis Phase	→ Intermediate Representation (IR) is the I/P for synthesis phase.
→ Analysis phase is also known as Pass I.	→ Synthesis phase is also known as pass II

* Explain the following.

① - single pass assembler

→ Scans entire source file only once.

→ Generally:-

(i) Deals with syntax

(ii) constructs symbol table

(iii) creates label list

(iv) Identifies the code segments, data

segment, stack segments etc.

→ cannot resolve forward reference of class symbol.

→ No object program is written hence no loader is required.

- tends to be faster compared to two pass
- only create tables with all symbols no address of symbol is calculated.
- more memory required compare to two pass assembler.
- Intermediate code not generated.
- Two Pass assembler.
 - require two phases to scan source file
 - Along with Pass 1 Pass two is also required which:-
 - (i) generates actual opcode
 - (ii) compute actual address of every label
 - (iii) Assign code address for designing the information
 - (iv) Translate operand name to appropriate register or memory.
 - (v) Immediate value is translated binary strings.
 - can resolve forward reference of data symbol
 - Loader is required as object code is generated

- Two pass assembler requires ~~not~~ re-scanning, hence slow compared to one pass assembler.
- Address of symbols can be calculated.
- less memory required compare to single pass assembler.
- Generation of intermediate code.

② Explain significance of location counter.

- To implement memory allocation a data structure called location counter (LC) is introduced.
- The location counter is always made to contain the address of the next memory word in the target program.
- It is initialized to the constant specified in the START statement.
- Whenever the analysis phase sees a label in a assembly statement, it enters the label and contents of LC in a new entry of the symbol table then the value to the location counter is increment and it goes to the next statement.

→ It ensures that the LC points to the next memory word in the target program when machine instruction have different length.

→ To update the contents of LC analysis phase needs to know length of different instruction.
Eg.

Symbol table

Symbol	Address
AGAIN	104
N	113

Here address 13 is inserted by the value stored in the location counter.

- (3) Differentiate two variants for generation of intermediate code in two pass assemblers.

Variant 1

IS, DL, & AD all statement contain processed form.

Variant 2

OC & AD statements contain processed form while for IS statement, operand field is processed only to identify literal reference.

Extra work in Pass 1

Extra work in Pass 2

simplifies tasks in Pass 2

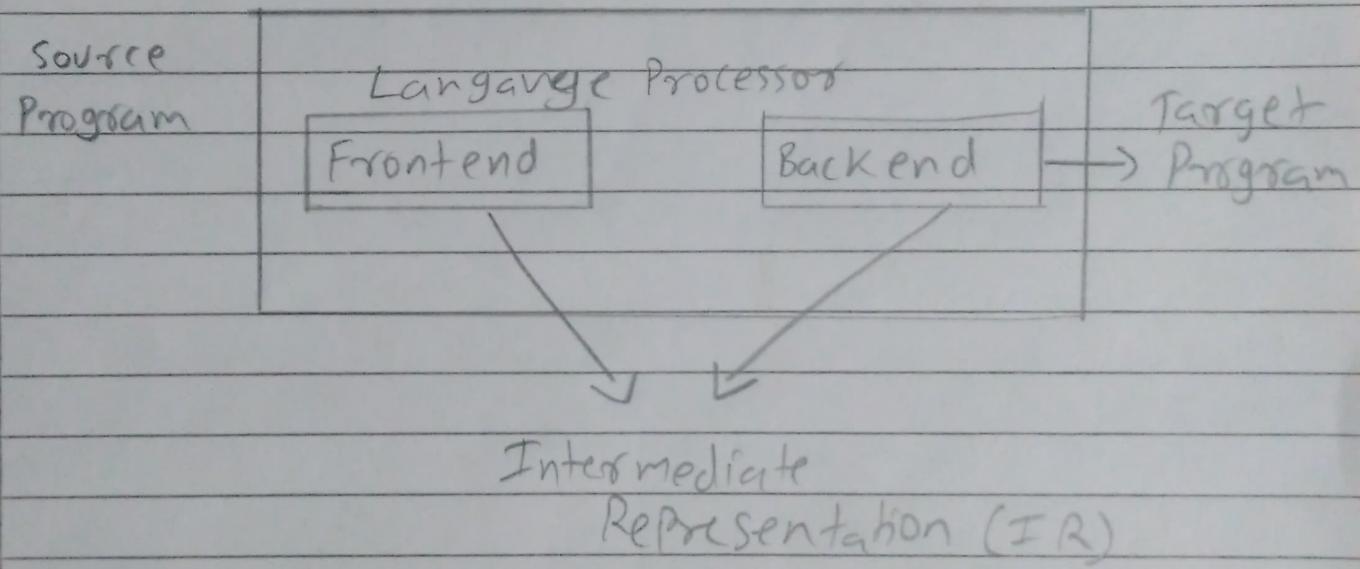
simplifies task in Pass 2

occupies more memory than Pass2

Memory utilization of two passes gets better balanced.

(ii) What is intermediate representation in an Assembler?

→ An intermediate Representation is a representation of a source program which reflects the effects of ~~the~~ source, but not all, analysis and synthesis ~~tasks~~ tasks performed during language processing.



Two Pass language processor

→ The first pass performs analysis of the source program and reflects its result in the intermediate Representation.

→ The second pass reads and analysis the IR.

instead of source program, to perform synthesis of the target program

→ This avoids repeated processing of the source program.

→ The first pass is concerned exclusively with source language issues. So it is called frontend of the language processor.

→ The second pass is concerned with program synthesis for a specific target language so it is called Backend of the language processor.

→ Properties of an IR:

Ease of use: IR should be easy to construct and analysis.

Processing efficiency: efficient algorithms must exist for constructing and analysing the IR.

Memory efficiency: It must be compact

⑤ Explain the significance of segment register

→ Segmentation is the process in which the main memory of the computer is locally divided into

different segments and each segment has its own base address.

Need of Segmentation :-

The Bus interface unit (BIU) contains four 16-bit special purpose registers called the segment registers.

Types :-

- (i) Code segment (CS) :- Code segment is used for addressing memory location in the code segment of the memory where the executable program is stored.
- (ii) Data segment (DS) :- Also refers to a segment in the memory which is another data segment in the memory.
- (iii) Stack segment (SS) :- Points to the stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

Q) Explain different registers of 8088 architecture.

→ The intel 8088 micro processor supports 8 and 16 bit ~~and~~ arithmetic, and also provides special instruction for string manipulation.

Different Registers of 8088 architectures.

- Data registers AX, BX, CX and DX
- Index registers SI and DI.
- Stack pointer registered BP and SP.
- Segment registers Code, Stack, Data and Extra.

Data Registers :-

Each data register is 16 bits in size, split into the upper and lower halves. Either half can be used for 8 bit arithmetic, where the two halves together constitute the data register for 16 bit arithmetic.

Index Register :-

The index registers SI and DI are used to index the source and destination addresses in string manipulation instruction. They are provided with the auto-increment and auto-decrement facility.

Stack pointer register :

Two Stack pointer register called SP and BP are provided to address the stack. SP points into the stack implicitly used by the architecture to store subroutines and interrupt return address. BP can be used by the programmer in any desired manner.

Segment registers :

The code stack and Data Segment registers are used to contain the start addresses of code, data and stack components. The Extra Segment register points to another memory area which can be used to store data.

② What is the meaning of segment overriding? Explain with example.

→ For arithmetic and now instructions the architecture uses the data segment by default.

→ To override this, an instruction can be preceded by a 1-byte segment override prefix with the following format:

001	seg	110	fix value.
-----	-----	-----	------------

When seg, represented in 2 bits, has the meaning.

Seg	Segment register
00	ES
01	CS
00	SS
11	DS

Example : If the code segment is to be used instead of data segment, it can be rewritten as

ADD AL, CS:12H [SF]

The assembler would encode this as,
 Segment override instruction
 001 01 110 00000 1 0 0100100
 0001000

Macros and Macro prototype

Q1 Macro Prototype

→ A macro prototype statement declares the name of a macro and the names and kinds of its parameters.

The macro prototype statement has the following syntax:

<macro name> [<format parameter spec>[, ...]]

Where <macro name> appears in the mnemonic field of an assembly statement and <format parameter> is of the form.

<Parameter name> [<Parameter kind>]

Q2 Macro Definition

A macro definition is enclosed between a macro header statement and a macro end statement.

Macro definitions are typically located at the start of a program. A macro definition consists of.

1. Macro prototype statements.
2. One or more statements.
3. Macro preprocessor statements.

→ The macro prototype statement declares the name of a macro and the names and kinds of

its parameters.

- A model statement is a statement from which an assembly language statement may be generated during macro expansion.
- A preprocessor statement is used to perform auxiliary functions during macro expansion.

Ex.

MACRO

```
INCR    &MEM_VAL, &INC_VAL, &REG  
MOVER    &REG    &MEM_VAL  
ADD     &REG    &INC_VAL  
MOVEM    &REG    &MEM_VAL  
MEND.
```

Q3. Model statements.

- A model statement is a statement from which assembly language statements may be generated during macro expansion.
- The statements between MACRO and MEND directives define the model statement of the macro and can appear in the expanded code.

Q4. Preprocessor Statement.

A Preprocessor statement is used to perform auxiliary functions during macro expansion. #If and #Else are Preprocessor statements.

Q5. Positional parameters.

→ For Positional formal Parameters, the spec. (Parameter name), e.g. &SAMPLE where SAMPLE is the name of a parameter.

→ In a call on a macro using Positional Parameters in an ordinary string

→ The value of a positional formal Parameter XYZ is determined by the rule of Positional association as follows:

* Find the ordinal of XYZ in the list of formal parameters in the macro prototype statements

* Find the actual parameter specification that occupies the same ordinal position in the list of actual parameters in the macro call statement. If it is an ordinary string ABC, the value of formal parameter XYZ would be ABC.

Q6. keyword parameter

- keyword parameters are symbolic parameters that can be specified in any order when 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 if no default value is specified and if the parameter is not given a value when the macro is called, then the parameter will be replaced by a null string.
- Each keyword parameter will have an equal sign (=) as the last character of the parameter name.

Q7. Default parameter

- A default is a standard assumption in the absence of an explicit specification by the programmer.
- When the desired value is different from the default value, the desired value can be specified explicitly in a macro call. This

Specification overrides the default value of the parameter for the duration of the call

→ If a parameter has the same value in most calls on a macro, this value can be specified as its default value in the macro definition itself.

Q8. Give two examples of nested macro calls

Ans ① MACRO

MAC-X & Par1, & Par2

MOVER REG, & Par1

MACRO

MAC-Y & Par2, REG = REG3

Add REG, & Par2

MOVEM REG, & Par2

MEND

PRINT & Par1

MEND

② MACRO

MAC-Y & Par2, REG = REG3

ADD REG, & Par2

MOVEM REG, & Par2

MEND

MACRO

MAC-X & Par1, & Par2

MOVE R1B62, Par1

MAC_Y 4 Par2, REG1=REG3 //Macro call

PRINT (Par1)

MEND.

Q9. Expansion Time Variable

- Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.
- A local EV is created for use only during a particular macro call. A global EV exists across all macro calls situation in a program and can be used in any macro which has a declaration for it.
- Local and global EV's are created through declaration statements with the following syntax:

LCL <EV spec> [, <EV spec>]

GBL <EV spec> [, <EV spec>...]

and <EV spec> has the syntax P<EV name> where <EV name> is an ordinary string.

- Values of EV's can be manipulated through Preprocessor statement. SET A SET statement is written as,

$\langle EV \cdot SPEC \rangle \quad SET \quad \langle SET - expression \rangle$

→ Where $\langle EV \cdot specification \rangle$ appears in the label field and SET in the mnemonic field A SET Statement assign the value of $\langle SET - expression \rangle$ to the EV specified in $\langle EV \cdot SPEC \rangle$.

Q10 Parameter Attributes.

Ans An attribute is written using the syntax $\langle attribute name \rangle \quad \langle formal parameter spec \rangle$ and represents information about the value of the formal parameter. The type, length and size attributes have the names T, L and S.

Q11. AIF

An AIF statement has the syntax
 $AIF \left(\langle expression \rangle \right) \quad \langle sequencing symbol \rangle$
 where $\langle expression \rangle$ is a relational expression involving ordinary strings, formal parameters and their attributes and expansion time variables.

If the relational expression evaluates to be true, expansion time control is transferred to the statement containing $\langle sequencing symbol \rangle$ in its label field.

Q12

ALO

An ALO statement has the syntax:

ALO <sequencing symbol>

It unconditionally transfers expansion time control to the statement containing <sequencing symbol> in its label field.

Q13.

ANOP

→ An ANOP statement has the syntax
<sequencing symbol> ANOP.

→ The significance of ANOP statement is to define sequencing symbol.

Q14.

REPT

→ REPT statement has the syntax:

REPT <expression>

→ <expression> should evaluate to a numerical value during macro expansion. The statement between REPT and an ENON statement would be processed for expansion, <expression> number of times.

Q15

IRP

The IRP statement.

IRP < formal parameter >, < argument - list >

The formal parameter mentioned in the statement takes successive value from the argument list for each value, the statements between the IRP and ENDM statements are expanded once.

Q16. Show combined working of macro and assembler.

Ans A macro processor is functionally independent of the assembler, and the output of the macro processor will be a part of the input into the assembler.

→ A macro processor, similar to any other assembler scans and processes statements

→ often the use of a separate macro processor for handling macro instructions leads to less efficient program translator because many functions are duplicated by the assembler and macro processor.

Q17 Macro Name table

Macro name table (MNT) has entries for all macro defined in a program.

→ Each MNT entry contains three pointers. MDTP, KPDT P and SSTP, which are pointers to MDT, KPDTAB and SSNTAB for the macro respectively.

→ MNT also contains Number of positional parameters (#PP), number of keyword parameters (#KP) and number of Expansion time variable (#EV).

Q18. Parameter name Table

→ Parameter Name Table (PNTAB) contains all the parameter name used in macro.

→ It contains all keyword, default and positional parameter names.

Q19. EV Name table

→ EV Name table contains all the expansion time variable names.

→ expansion time variables are local variables and global variable which are declared in macro.

Q20 SS NAME TABLE

→ SS name table (SSNTAB) contains the sequencing symbols.

Q21 keyword parameter Default Table

→ keyword parameter Default table (KPDTAB) contains the name of keyword parameter and it's value.

Q22 Macro Definition Table

→ Macro Definition Table (MDT) entries are constructed while processing the model statements and preprocessing statements in the macro body.

Scanner & Parser.

Q1 Difference between scanner & parser.

Ans Scanning:

- Scanning is the process of recognizing the lexical components in a source string.
- The lexical features of a language can be specified using Type-3 or regular grammars. This facilitates automatic construction of efficient recognizers for the lexical features of the language.
- Scanning uses two state FSA & DFA.

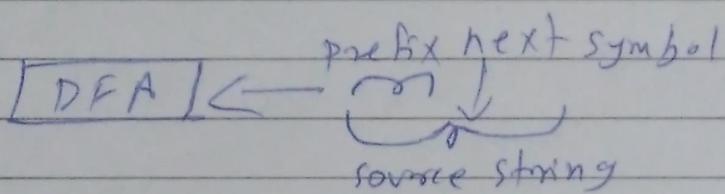
Parsing:

- Parsing is used to check the validity of a source string and to determine its Syntactic Structure.
- For an invalid string the parser issues diagnostic message reporting the cause and nature of errors in the string & for valid string it builds a parse tree to reflect the sequence of derivations or reductions performed during parsing.
- Parsing use two parse method, Top-down, Bottom up

Q2 Define and give the usage of DFA.

Ans: → It is a finite state automaton none of whose states has two or more transitions for the same source symbol. The DFA has the property that reaches a unique state for every source string input to it.

→ Transition in a DFA are deterministic.



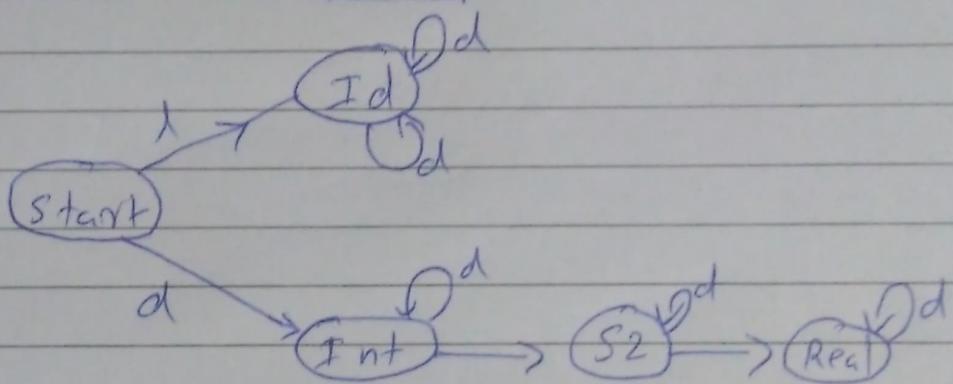
→ At any point, the DFA recognized some prefix of the source string, possibly the null string and would be poised to recognize the string pointed by the pointer next symbol.

→ The validity of a string is determined by giving it as the input of a DFA in its initial state.

→ The string is valid if and only if the DFA recognizes every symbol in the string and finds itself in a final state at the end of string.

Example of Building DFA,

State	next symbol		
	I	d	.
Start	I d	Int	
I d	I d	I d	
Int		Int	S ₂
S ₂		Real	
Real		Real	



A combined DFA for integers
real numbers & identifiers.

→ Correct input would be like '1234.56' or
'asd12' etc.

Q3 Take Example of any three valid string and constitute constant DFA.

Ans. Valid String

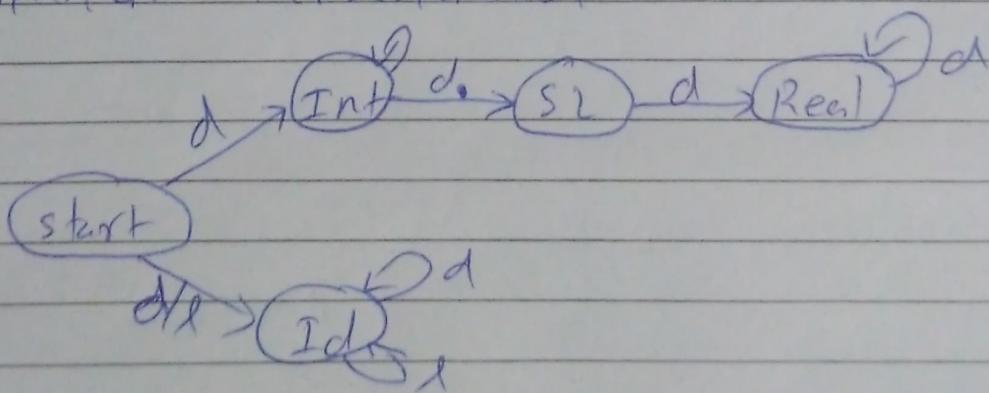
12d sf

. 234

23 .4

Start	Next Symbol	Symbol /
Start	d	d .
Start	Int	Id S2
Int	Int	S2
Id	Id	Int S2
S2	Real	
Real	Real	

* Graphical Representation



Qn Difference between top down parsing with backtracking and without backtracking

Ans Top-down Parsing with Backtracking

Top-down Parsing without Backtracking

- | | |
|--|---|
| <ol style="list-style-type: none"> 1. The leftmost non terminal in CSF is identified. So, then it matches the input string to its assigned grammar. If the check fails the predictions are discarded and it goes on to the stage before the rejected prediction was made. This parse is called backtracking. 2. The Backtracking results in problem like:
 -> semantic actions cannot be performed while making a prediction
 -> Precise error reporting is not possible
 -> Also it consumes time as it goes to previous stage | <ol style="list-style-type: none"> 1. here exclude the use of Backtracking which makes the prediction very crucial. The grammar have to be modified so that it does not need left factoring since its RHS alternatives produce unique terminal symbols, parsing is no longer by trial-and-error. It is also called Predictive Parser. 2. when removed backtracking, it resulted into several advantages -> parsing became more efficient and also be possible to perform semantic actions.
 -> Precise error reporting during Parsing |
|--|---|

for rechecking

3. Grammer used here is

$$E = T + E/T$$

$$T = v * T/V$$

$$v = \langle id \rangle$$

Parsing with Backtracking.

3. Grammer used here is

$$E = T E''$$

$$E'' = +E/\epsilon$$

$$T = VT''$$

$$T'' = *T/\epsilon$$

$$v = \langle id \rangle$$

Q5. Difference between top-down & bottom-up parser

Top-down

Bottom - UP

- | | |
|--|---|
| 1. Top down approach starts the parse tree from the root and move downwards for parsing other nodes. | 1. Bottom up approach starts evaluating the parse tree from the lowest level of the tree and move upwards for parsing the node. |
| 2. Top down parsing attempts to find the left most derivation for a given string. | 2. Bottom up Parsing attempts to reduce the input string to first symbol of the grammar. |
| 3. Top down parsing uses left most derivation | 3. Bottom up Parsing uses the right most derivation |
| 4. Top down parsing searches for a production rule to be used to construct a string | 4. Bottom up Parsing Searches for a production rule to be used to reduce a string to a glt or starting symbol of grammar. |
| 5. In this parsing technique we start parsing from top to down in top-down manner. | 5. In this Parsing technique we start parsing from bottom to up in bottom-UP manner. |

Q6

Define(a) Grammer

A Grammer G of a language L_G is a quadruple (Σ, SNT, S, P) where
 Σ is the alphabet of L_G , i.e. the set of terminal symbols.

SNT is the set of non terminal symbols
 S is the distinguished symbol
 P is the set of products.

(b) terminal symbol

A symbol in the alphabet is known as a terminal symbol

(c) Non-terminal symbol

A Non terminal symbol is the name of syntax category of a language, e.g. noun, verb etc.

(d) Production Rule

A Production Rule, also called a rewriting rule is a rule of grammar.

It has the form $A \rightarrow$ non-terminal symbol
 \rightarrow string of terminal and non-terminal symbols
where the notation \rightarrow stands for 'is defined as'.

e.g.: < Noun phrase > \rightarrow < Article > < Noun >

(e) Reduction Rule:

It is the process of replacement of string or part of string by non-terminal according to the production rule.

(f) Derivation Rule

Derivation is the replacement of non-terminal symbol in accordance with the given production rule.

(g) CSF

CSF stands for current sequential form
Let CSF be of the form $\beta A \gamma$, such that
 β is a string of TS and A is the leftmost NT in CSF. Exit with success if $CSF = \alpha$

(h) SSM

SSM stands for Source string marker.

SSM points to the first unmatched symbol in the source string.

(i) Prediction

This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in LG can be derived from so.

(j) Abstract syntax tree

An abstract syntax tree represents the structure of a source string in a more economical manner.

The word 'abstract' implies that it is a representation designed by a compiler designer for his own propose.

(k) left to left Parsing

An L(1) parser is a table driven parser for left to left Parsing

The first L indicates input is scanned from left to right. The second L means it uses left most derivation for input string.

(L) left to Right Parsing

Left to right parsing follows reverse of right most derivation

In left to right parsing input is scanned from left to right and uses right most derivation for input string.

Q2 Explain the following parsing with grammar algorithm and example.

Ans (a) Top down parser with backtracking

$$\text{Grammar} \rightarrow E = T + E / T \\ T = V * T / V \\ V = \langle \text{id} \rangle$$

source string $\rightarrow a + b * c \xrightarrow{\text{to}} \langle \text{id} \rangle + \langle \text{id} \rangle * \langle \text{id} \rangle$
 The prediction making mechanism selects the R.H.S alternative of a production in a left-to-right manner.

- 1. E
- 2. $T + E$
- 3. $V * T + F$
- 4. $\langle \text{id} \rangle * T + E$
- 5. $\langle \text{id} \rangle + T + E$
- 6. $\langle \text{id} \rangle + V * T + E$
- 7. $\langle \text{id} \rangle + \langle \text{id} \rangle * T + E$
- 8. $\langle \text{id} \rangle + \langle \text{id} \rangle * V * T + E$
- 9. $\langle \text{id} \rangle + \langle \text{id} \rangle * \langle \text{id} \rangle * T + E$
- S. $\langle \text{id} \rangle + T$
- G. $\langle \text{id} \rangle + V * T$
- F. $\langle \text{id} \rangle + \langle \text{id} \rangle * T$
- B. $\langle \text{id} \rangle + \langle \text{id} \rangle * V * T$

(b) Top-down parser without Backtracking

$$\begin{aligned}
 \text{Grammar} : \quad E &= TE'' \\
 E'' &= +E/E \\
 T &= VT'' \\
 T'' &= *T/\epsilon \\
 V &= \langle \text{id} \rangle
 \end{aligned}$$

Source string $\rightarrow a * b + c \rightarrow \langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle$

1. ~~E~~
2. TE''
3. $VT''E''$
4. $\langle \text{id} \rangle T'' E''$
5. $\langle \text{id} \rangle * TE''$
6. $\langle \text{id} \rangle * VT'' E''$
7. $\langle \text{id} \rangle * \langle \text{id} \rangle T'' E''$
8. $\langle \text{id} \rangle * \langle \text{id} \rangle E''$
9. $\langle \text{id} \rangle * \langle \text{id} \rangle + E$
10. $\langle \text{id} \rangle * \langle \text{id} \rangle + TE''$
11. $\langle \text{id} \rangle * \langle \text{id} \rangle + VT'' E''$
12. $\langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle T'' E''$
13. $\langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle E''$
14. $\langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle //$

Q8 LL(2) Parser

→ An LL(2) parser is a table driven parser for left-to-left parsing. The '2' in LL(2) indicates that the grammar uses a look-ahead of one source symbol that is the prediction to be made is determined by the next source symbol.

→ Example :

Grammar :

$$E = TE'$$

$$E' = +TE'/\epsilon$$

$$T = VT'$$

$$T' = *VT'/\epsilon$$

$$V = \langle id \rangle$$

Source string : |- <id> * <id> + <id> - |

Parser Table :

Non-terminal	Source Symbol			
	<id>	+	*	-
E	$E = TE'$		*	
E'		$E = +TE'$		$E' = S$
T	$T = VT'$			
T'		$T = \epsilon$	$T = *VT'$	$T' = \epsilon$
V	$V = \langle id \rangle$			

LL Parsing

current sentential form	Symbol	Prediction
$H E +$	$\langle \text{rd} \rangle$	$E = T E'$
$H T + E' -$	$\langle \text{rd} \rangle$	$T = VT'$
$H VT' E' -$	*	$V = \langle \text{id} \rangle$
$H \langle \text{id} \rangle T' E' -$	*	$T' \Rightarrow * VT'$
$H \langle \text{id} \rangle * VT' E' -$	$\langle \text{rd} \rangle$	$V \Rightarrow \langle \text{id} \rangle$
$H \langle \text{id} \rangle * \langle \text{id} \rangle T' E' -$	+	$T' = S$
$H \langle \text{id} \rangle * \langle \text{id} \rangle * T' E' -$	+	$E' \Rightarrow + TE'$
$H \langle \text{id} \rangle * \langle \text{id} \rangle + T' E' -$	$\langle \text{id} \rangle$	$T = VT'$
$H \langle \text{id} \rangle * \langle \text{id} \rangle + VT' E' -$	$\langle \text{id} \rangle$	$V = \langle \text{id} \rangle$
$H \langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle T' E' -$	-	$T' = \epsilon$
$H \langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle E' -$	-	$E' = \epsilon$
$H \langle \text{id} \rangle * \langle \text{id} \rangle + \langle \text{id} \rangle -$	-	

Q9.

Ans

Recursive Descent Parser.

- A recursive decent parser is a variant of top down parsing without backtracking
- It uses a set of recursive procedures to perform Parsing
- To implement recursive decent parsing, a left grammar is modified to make repeated occurrence of strings more implicit.
- Salient advantages of recursive descent parsing are its simplicity and generality
- It can be implemented in any language supporting recursive procedures.

E.g.

Recursive descent Parser.

String to be parsed $\rightarrow P + q * r$

Call Proc-E

Step 1 \rightarrow Call Proc-T(a);

in that call Proc-V(a)

in that next symbol is id

so build a tree node $\rightarrow P$ Step 5 \rightarrow Return to step 5

next symbol is not '*'

Step 2- so return to step ①
 next symbol is '+'
 so match ('+')

call proc-T(b);
 (call proc-V(b))

next symbol is ;d

so build a tree e node → ②

steps return to step 5

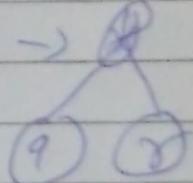
then next symbol is '*'
 so match ('*')

and call proc-V(a)

→ create tree node — ③

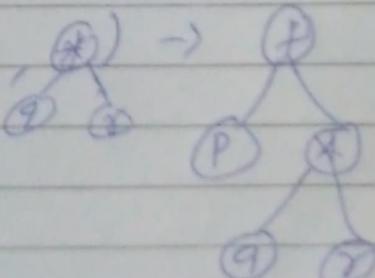
step 3. return to step ②

and build the tree ('*', a, b) → ④



Step 3- now return to step 3

and tree build ('+', a, *) → ⑤



Q10 Operator Precedence parser

Ans

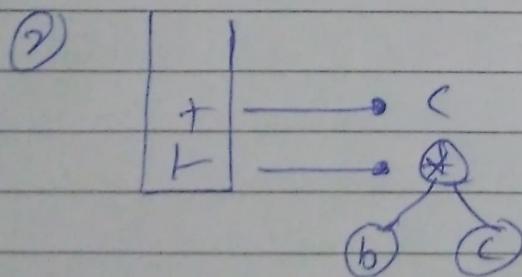
Precedence between operators a & b appearing in a sentential form aPb where P is a null string is termed as operator precedence.

Example : The parsing string is
 $a + b * c$

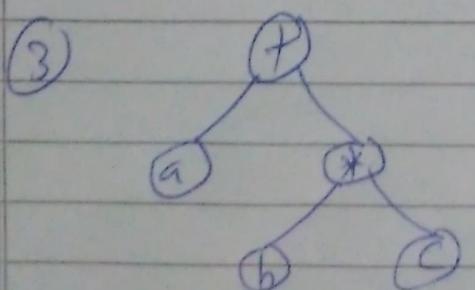
Stack

①	$\begin{array}{ c } \hline * \\ \hline \end{array}$	→ C
	$\begin{array}{ c } \hline + \\ \hline \end{array}$	→ b
	$\begin{array}{ c } \hline \cdot \\ \hline \end{array}$	→ a

So the operator precedence of * is higher
 So it will pop out.



Last remaining operator
 will pop out.



Loader & Linker.

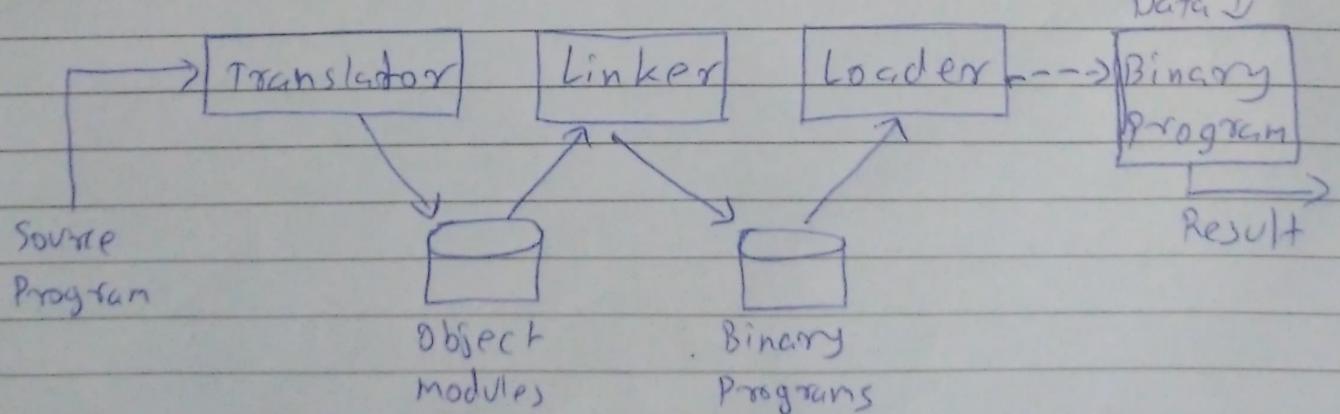
① Define Translated, linked and load time addresses.

Ans When a program is compiled, a translator is given an origin specification, this is called the translated origin of the program.

The translator uses the value of the translated origin to perform memory allocation for the symbols declared in program.

The execution start address or simply the start address of a program is the address of the instruction from which its execution must begin.

The start address specified by the translator is the translated start address of the program.



- The origin of a program may have to be changed by the linker or loader for any of two reasons
- First, the same set of translated addresses may have been used in different object modules constituting a program so memory allocation to such programs would conflict unless their origins are changed.

The change of origin leads to changes in the execution start address and in the addresses assigned to symbols.

1. Translation time address:

Address assigned by the translator.

2. linked address:

Address assigned by the linker.

3. Load time address:

Address assigned by the loader.

Q2

Define

(a) Program Relocation:- It is a process of modifying the address used in the address sensitive instructions such that the program can execute correctly from the designated area of memory.

- Linker performs relocation if linked origin \neq Translated origin
- Loader performs relocation if Load origin \neq Linked origin.
- In general Linker always performs relocation whereas some loaders do not.
- Absolute loaders do not perform relocation, then
 - load origin = linked origin.
 - Thus, load origin and linked origin are used interchangeable.

(b) Address sensitive instruction or IRR:-
It's a set of instructions that perform relocation in program.

- steps:
 - Calculate Relocation Factor
 - Add it to Translation Time Address

for every instruction which is member of IRR.

- e.g. relocation factor = $900 - 500 = 400$
- IRR contains translation address shot & \$38

Address is changed to shot + 400 = \$400
and \$38 + 400 = \$388.

(c) Relocation Factor :- Let the translated program P be t-origin P and l-origin P respectively.

- Consider a symbol symb in P. Let its translation time address be t_{symb} and like link time address l_{symb}.
- The relocation factor is:
$$\text{relocation-factor} = \text{l-origin}_P - \text{t-origin}_P$$
- The relocation factors can be positive, negative & zero.

(d) Public Definition :- The ENTRY statement lists the public definitions of a program unit.

(e) External Reference :- The EXTRN statement lists the symbol to which external references are made in the program unit.

Q3 Difference between Non-relocatable, Relocatable and Self-Relocatable module.

Ans. Non-relocatable : A program is one that cannot be executed in any memory area other than the area starting on its translated origin.

Relocatable :- A program that can be processed to relocate it to a desired area of memory. The difference is the availability of information concerning the address sensitive instructions.

Self-Relocatable : - A program that can perform a relocation of its own address sensitive instructions. A self Relocatable module can be executed in any area of memory.

Q4 Define: Work Area.

→ Work Area : An area of memory where the loader simply loads the binary program for the purpose of execution.

address-in-work area = address of work area
+ translated address - + origin

Q.S. Explain object module Format of MS DOS Linker.

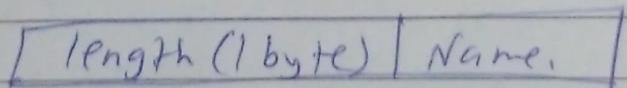
Ans An object module is a sequence of object records.

- There are 11 types of object records, in which they contain 5 kind of info:

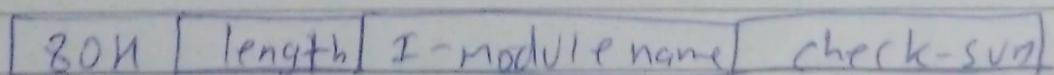
- (i) Binary image
- (ii) External reference
- (iii) Public definition
- (iv) Debugging information
- (v) Miscellaneous information

- Object Records of Intel 8088
each object record contains variable length information and may refer to the contents of previous object records.

- Each name in an object record is represented in the following format



THE ABR records:



- The module name is THEADR record is typically derived by the Translator from the source file name.
- The name is used by the Linker to report errors.
- An assembly programmer can specify the module name in the NAME directive.

L NAMES record:

96H	length	name-list	check-sum
-----	--------	-----------	-----------

- The L NAMES record lists the names for use by SEGDEF records.

SEGDEF record:

98H	length	attributes	segment	name	checksum
-----	--------	------------	---------	------	----------

- A SEGDEF record designates a segment name ~~set~~ using an index into this list.
- The attributes field of a SEGDEF record indicates whether the segment is relocatable or absolute, whether it can be combined with other segments, as also the alignment requirement of its base address.

EXTDEF and PUBDEF record.

8CH	length	ext-reference	check sum
-----	--------	---------------	-----------

90H	length	base	name	offset	check-sum
-----	--------	------	------	--------	-----------

- The EXTDEF record contains a list of external reference used by the programs of this modules.

- The PUBDEF records contains a list of public names declared in a segment of the object modules.

LEDATA record :-

1AH	length	segment	data offset	data	check
-----	--------	---------	-------------	------	-------

- An LEDATA record contains the binary image of the code generated by the language translator.

FIXUPP record

9CH	length	locat	fix	frame	target	offset	check
-----	--------	-------	-----	-------	--------	--------	-------

- A fixupp record contains information for one or more relocation and linking fixups to be performed.

MODEND record :-

[80H | length | type(1) | start addr (S) | check sum]

- The MODEND record signifies the end of the module with the type field indicating whether it is the main program.

Q6. Explain basic object module Format.

Ans

The object module of a program contains all information necessary to relocate and link the program with other program.

- The object module of a program P contains of 4 components.

① Header :- The header contains translated origin size and execution start address of P.

② Program :- This component contains the machine language program corresponding to P.

③. Relocation table (RELOCTAB) This table describes DRR. Each RELOCTAB entry contains a single field.

④ Linking table : This table contains information concerning the public definitions and external reference in P.

eg. Statement		address	code
START	S00		
ENTRY	TOTAL		
EXTRN	MAX, ALPHA		
READ	A	500	top 0 sho
LOOP		S01	
:			
MOVER	AREG, ALPHA	S18	top 1 000
BC	ANY, MAX	S19	top 6 000
:			
BC	LT, LOOP	S38	
STOP		S39	
P	DS	1	sho
TOTAL	DS	1	sh1
END			

Q7 FIXVPP codes and fix data field codes.

Ans The local field contains a numeric code called loc code to indicate the type of a fixvp.

Loc Code	Meaning
0	Low order byte fix
1	Offset fix
2	Segment fix
3	Pointer fix

- Local also contains the offset of the base location in the previous LE DATA record.
- The fix data field indicates the manner in which the target data and target offset fields are to be interpreted.

Code	Contents of target data & offset fields
0	segment index and displacement
2	external index and target displacement
4	segment index (offset field not used)
6	external index. (offset field not used)

Q8

Take 3 assembly language programs and explain the execution of MS DOS Linker.

Ans

Linker performs relocation and linking of all named object modules to produce a binary program with the specified load origin.

- Linker execution terminates when all external references have been resolved or when - The unresolved external reference cannot be found in any of the library files.
- Linker uses a two-pass strategy.
- In the first pass, the object modules are processed to collect info concerning segments and public definitions.
- The second pass performs relocation and linking
 - First pass: In the first pass Linker only processes the object records for building 'NTAB'.
 - Second pass: In the second pass, Linker builds NAMELIST, SBDDEF and EXTDEF.

Q1

Difference between static and dynamic memory allocation.

Ans

static

dynamic

- | static | dynamic |
|--|--|
| <ul style="list-style-type: none"> ① In this memory is allocated to a Variable before the execution of a program begins. | <ul style="list-style-type: none"> ① The memory bindings are established and destroyed during the execution of a program. |
| <ul style="list-style-type: none"> ② No memory allocation and deallocation action are performed during the execution of a program. | <ul style="list-style-type: none"> ② Memory is allocated and deallocation during the execution of program. |
| <ul style="list-style-type: none"> ③ Variables remain permanently allocated, allocation of variable exists even if the program unit in which it is defined is not active. | <ul style="list-style-type: none"> ③ The variable get allocated only if the program unit gets active. |
| <ul style="list-style-type: none"> ④ eg : FORTRAN | <ul style="list-style-type: none"> ④ e.g. PL/I, Pascal, Ada etc. |

Q2 Explain Scope Rules with example.

Ans. → For data declaration it is possible to use same name in many blocks of program.

→ This would establish many bindings of the form (name, var) for different values of a variable.

→ Scope rules determines which of these bindings is effective at a specific place in the program.

→ e.g.:

Consider the block structured program. The variables accessible within the various block are as follow:-

```

A [ x, y, z : integer;
    B [ g : real;
        C [ h, i, j : real;
            A [ i, j : integer;
    ]
]

```

→ Variable z_A is not accessible inside block C since C contains a ~~declaration~~ declaration using the name z . Thus z_A and z_C are two distinct variables. This would be true

even if they had identical attributes. i.e. even if 2 of c may be declared to be an integer.

Q3) Explain Memory allocation in Recursion,

- > Recursion procedure (or functions) are characterised by the fact that many invocations of a procedure coexist during the execution of a program.
- > A copy of the local variables of the procedure must be allocated for each invocation.
- > This does not pose any problem when a stack model of memory allocation is used because an activation record is created for every invocation of a procedure or function.

Program Sample (input, output);
Var

a, b : integer;

function fib(n) : integer;

var

x : integer

begin

if n > 2 then

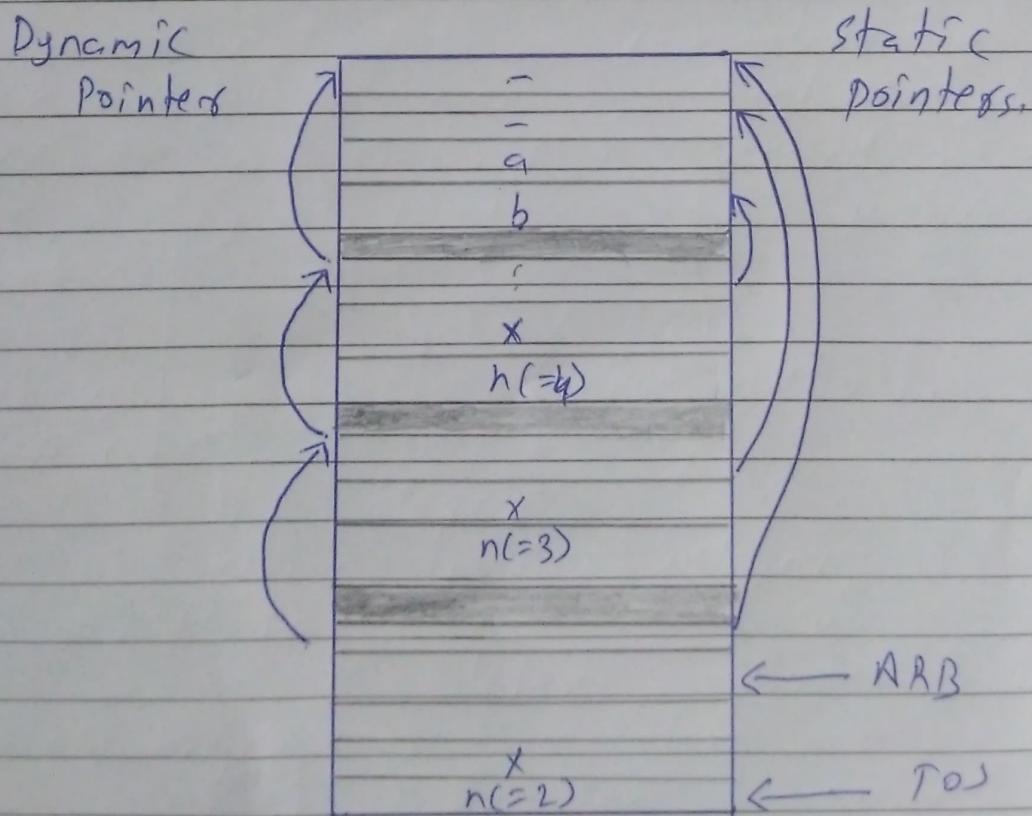
x := fib(n-1) + fib(n-2);

```

else
    x = 1;
    return (x);
end fib;
begin
    fib(y);
end.

```

- For the call `fib(4)` the function makes recursive calls.
- Pictorial representation of recursive call in Fibonacci Series program



Qn Define Dope vector with example.

Ans → If array dimension bounds are not known during compilation the dope vector has to exist during program execution.

- The number of dimension of an array determines the format and size of its DV.
- The Dope vector is allocated in the AR of a block and dDV, its displacement in AR, is noted in the symbol table entry of the array.
- The array is allocated dynamically by repeating step for the array.
- If start address values of l_j , u_{ij} and range j are entered in the DV
- The generated code uses dDV to access the contents of DV to check the validity of subscripts and compute the address of an array element.

Qn Define with example:-

(i) Operand descriptors:-

It has following fields:-

① Attributes :- contains the subfield type length and miscellaneous information.

② Addressability :- specifies where the operand is located and how it can be accessed. It has 2 sub fields

① Addressability code.

② Address.

→ An operand descriptor is built for every operand participating in an expression.

→ An operand descriptor is built for an id when the id is reduced during parsing

→ A partial result pri is the result of evaluating same operator opj. A descriptor is built for pri immediately after code is generated for operator opj.

→ for simplicity, assume that all operand descriptors are stored in an array called operand - descriptor.

e.g. :-

MOVER AREG, A
MULT AREG, B

(ii) Register Descriptors:-

⇒ A register descriptor has two fields.

① Status :-

contains the code free or occupied to indicate register status.

② Operand descriptor # :-

If status = occupied this field contains the descriptor # for the operand contained in the register

⇒ Register descriptors are stored in an array called Register-descriptor.

⇒ One register descriptor exists for each CPU register.

e.g. :-

The register descriptor for AREG after generating code for apb

Occupied | #3 |

This indicates that register AREG contains the operand description by descriptor #3

Q6 Write the steps for Assembly lang code generation from an expression specified in high level language.

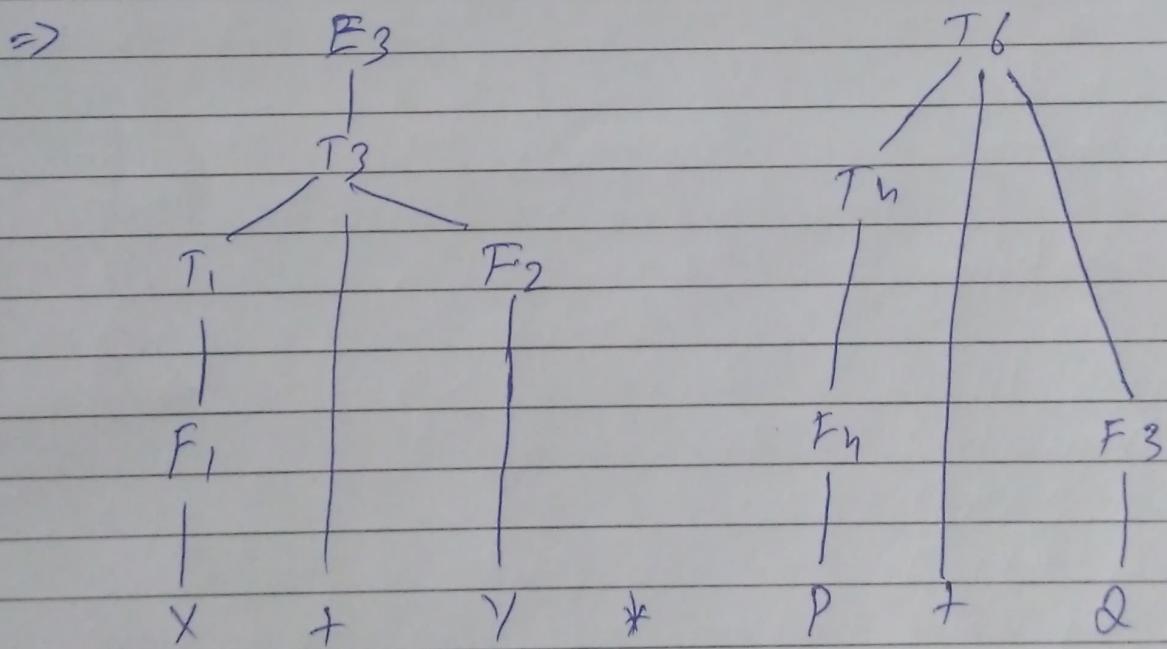
→ Expression :- $x + y * p + q$

steps :-

Step no	Parsing action	Code generation
1	$\text{<id>}x \rightarrow F^1$	Build descriptor #1
2	$F^1 \rightarrow T^1$	-
3.	$\text{<id>}y \rightarrow F^2$	Build descriptor #2
4.	$T^1 + F^2 \rightarrow T^3$	Generate MOVE R AREC
5.	$T^3 \rightarrow E^3$	-
6.	$\text{<id>}p \rightarrow F^4$	Build descriptor #3
7.	$F^4 \rightarrow T^4$	-
8.	$\text{<id>}q \rightarrow F^5$	Build descriptor #4
9.	$T^4 * F^5 \rightarrow T^6$	Generate MOVE M AREC TEMP 1 MOVE R AREC, P ADD AREC, Q Build descriptor #6
10.	$E^3 + T^6 \rightarrow E^7$	Generate MULT AREC, TEMP 1

→ Operand descriptors:-

1 (int, 1)	M, addr(x)
2 (int, 1)	M, addr(y)
3 (int, 1)	M, addr(temp1)
4 (int, 1)	M, addr(p)
5 (int, 1)	M, addr(q)
6 (int, 1)	R, addr(NRBG)

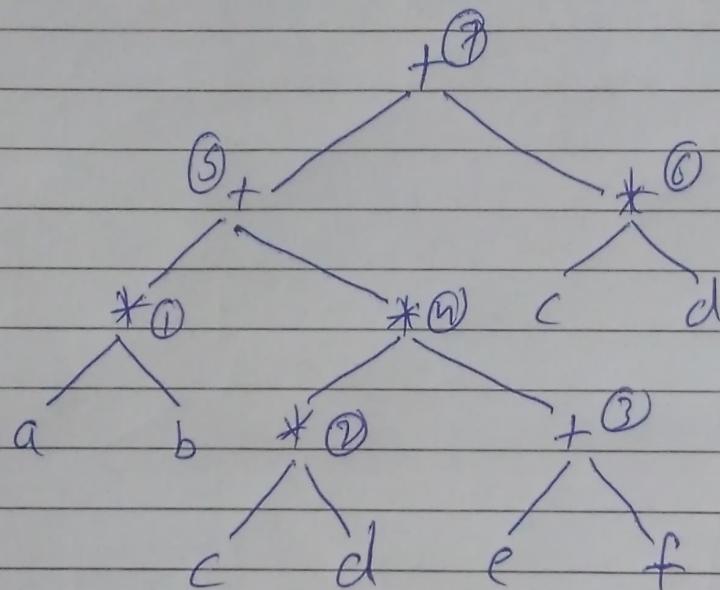


⇒ Register descriptor

[rc 16]

Q7 With an example, Explain the alternatives of assembly language code generation for expression

Ans $x * y + P * Q (R + S) + Q * Q$



MOVE R AREG, X	MULT AREG, Q
MULT AREG, S	ADD AREG, TEMP_3
MOVE M AREG, TEMP_1	
MOVE R AREG, P	
MULT AREG, Q	
MOVE M AREG, TEMP_2	
MOVE R AREG, R	
MOVE R AREG, S	
MULT AREG, TEMP_2	
ADD AREG, TEMP_3	
MOVE R AREG, P,	

Q8 How postfix string can be used for generating the intermediate code for expression

Ans → In the postfix notation each operator appears immediately after its last operand.

→ Thus a binary operator O_i appears after its second operand. If any of its operand is itself an expression involving an operator O_j , O_j must appear before O_i .

e.g.: - $t \ a \ t \ b \ * \ (\ t \ d \ * \ e \) \ f \ - \ i$
source string

Postfix string: - $t \ a \ b \ c \ * \ t \ d \ e \ f \ i \ * \ + \ -$

→ The postfix string is a popular intermediate code in non-optimizing compiler due to ease of generation and use.

→ The code generation from the postfix string using a stack of operand descriptor.

→ Operand descriptor are pushed on the stack as operand appear in string

→ When an operator with arity k appears in

The string k descriptors are popped off the stack.

- A descriptor for the partial result generated by the operator is now pushed on the stack.
- Conversion of infix to postfix is modified operands appearing in the source string are copied into the postfix string straight away.
- The TOS operator is popped into the postfix string of TOS Operator > current operator.

Q9. How triples and quadruples can be used for generating the intermediate code for expression.

Ans. → Triples: - A triples is a representation of an elementary operation in the form of a pseudo machine instruction.

Format:-

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

→ Each operand of a triple is either a variable / constant or the result of some

evaluation represented by another triple.

→ Conversion of infix string into triples can be achieved by a slight modification

e.g.

infix string:- $t = a + b * c + d * e \uparrow f -$

⇒ Quadraples:-

→ A Quadraples represents an elementary evaluation in the following format :-

Operator	Operands	Operands	Result name
----------	----------	----------	-------------

Q10. List the steps taken by the compiler at function & procedure calls.

Ans

While implementing a function call the compiler must do the following:-

① Actual parameters are accessible in the called function

② The called function is able to produce side effect according to the rules of the PL.

- ③ Control is transferred to and is returned from the called function.
- ④ The function value is returned to the calling program.
- ⑤ All other aspects of execution of a calling programs are unaffected by the function calls.
- ⑥ Compiler used a set of feature to implement function
 - (i) Parameter list
 - (ii) calling conventions
 - (iii) save area.

Q11) Different function calls.

Ans

Different types of function calls are :-

- (i) Call by value :- In this mechanism value of actual parameters are passed to the called function.
- (ii) Call by value-result :- In this mechanism the value of formal parameters is copied back into actual parameter
- (iii) Call by reference:- In this the address

of actual parameter is passed to the called function

(iv) Call by Name:- In this also the same mechanism is followed of the call by reference.

→ here also change in formal parameters affect the value of actual parameters.

System Software Practicals

Name : Pradip .S. Karmakar

Roll No : 10

Class : MCA – 3

Subject : System Software

1. Assembler

```
import java.util.StringTokenizer;

public class assembler
{
    static String[][] symbolTable=new String[10][2] ;//this is symbol table
    static String[][] litTable=new String[10][2] ;//this is literal table
    static int[] poolTable= new int[10];//this is literal table
    static int locationCounter =0;
    static int poolTabPtr = 0;//pooltable pointer
    static int litTabPtr = 0;//literaltable pointer
    static int symbolTabPtr=0;

    public static void main(String[] args)
    {
        poolTable[0]=1;
        String statements = "START 200\nREAD A\nREAD B\nMOVER AREG,A\nADD AREG
,B\nMOVEM AREG,RESULT\nPRINT A\nPRINT B\nPRINT RESULT\nA DS 0\nB DS 1\nRESULT
DS 0\nEND";

        String delimiters = "[, \n\t]"; //comma,space,new line, tab are delimi
ters
        String[] tokens = statements.split(delimiters, 0);

        String code;
        String regNO;
        int size=0;
        int i=0;
```

```

for(i=0;i<tokens.length;i++)
{
    int index=0;
    String token = tokens[i];
    String result = mnemonic(token,"type");//to find type
    code = mnemonic(token,"code");//to find code

    if(result.equals("AD"))
    {

        if(token.equals("START"))
        {
            locationCounter = Integer.parseInt(tokens[i+1]);//to go to
next token
            System.out.println("LC= "+locationCounter);
            i++;
        }
        else if(token.equals("EQU"))
        {
            index = get_symbol_index(tokens[i+1]);//for finding addres
s of loop
            System.out.println("IC=(AD,"+code+") (S,"+index+ ")");
            String address = symbolTable[index][1];
            index = get_symbol_index(tokens[i-
1]);//for finding address of equ
            symbolTable[index][1] = address;
            i++;

        }
        else if(token.equals("ORIGIN"))
        {
            index = get_symbol_index(tokens[i+1]);//for finding addres
s of loop
            int address = Integer.parseInt(symbolTable[index][1]);

            if((tokens[i+2].substring(0)).equals("+"))
                locationCounter = address + Integer.parseInt(tokens[i+
2].substring(1,(tokens[i+2].length()-1)));
            if((tokens[i+2].substring(0)).equals("-"))
                locationCounter = address - Integer.parseInt(tokens[i+
2].substring(1,(tokens[i+2].length()-1)));

            i+=2;
            System.out.println("IC=(AD,"+code+") (C,"+locationCounter+")
");
        }
        else if(token.equals("LTORG") || (token.equals("END") && i < t
okens.length))
    }
}

```

```

    {
        for(index = poolTable[poolTabPtr]-1;index<litTabPtr;index++)
        {
            litTable[index][1] = String.valueOf(locationCounter);
            System.out.println("IC=(AD,"+litTable[index][0]+") (" +locationCounter+ ")");
            locationCounter++;
        }

        poolTabPtr++;
        poolTable[poolTabPtr] = litTabPtr+1;
    }

}

else if(result == "" && !isliteral(token))//for label
{
    index=get_symbol_index(token);

    if(mnemonic(tokens[i+1],"type").equals("IS"))
    {
        if(index== -1) //label is not inserted in symbolTable
        {
            symbolTable[symbolTabPtr][0] = token;
            symbolTable[symbolTabPtr][1] = String.valueOf(locationCounter); //to insert lc in symbol table
            symbolTabPtr++;
        }
        else
        {
            symbolTable[index][1]=String.valueOf(locationCounter);
        }
    }
    else if(result.equals("IS"))
    {

        regNO= mnemonic(tokens[i+1],"code");//to find register number
eg.1 for AREG
        locationCounter++;
        String operand = tokens[i+2];
        if(token.equals("STOP")) //if stop condition
            System.out.println("IC=(IS,00)");

        else
    }
}

```

```

    {
        if(!isliteral(operand))
        {
            if(get_symbol_index(operand) == -1) //if symbol is not in symtab
            {
                symbolTable[symbolTabPtr][0] = operand;
                symbolTabPtr++;
                System.out.println("IC=(IS,"+code+") ("+regNO+) (" +symbolTabPtr+""));
            }
            else //if symbol is present in symTab
            {
                index = get_symbol_index(operand)+1;
                System.out.println("IC=(IS,"+code+") ("+regNO+) (" +index+""));
            }
        }

        else//if operand is litral
        {
            String this_litral=operand.substring(2,(operand.length()-1));
            litTable[litTabPtr][0]=String.valueOf(this_litral);

            litTabPtr++;
            System.out.println("IC=(IS,"+code+") ("+regNO+) (L, " +litTabPtr+""));
        }
        i+=2;
    }
}
else if(result.equals("DL"))
{
    index = get_symbol_index(tokens[i-1]);
    code = mnemonic(token,"code");
    size = Integer.parseInt(tokens[i+1]);
    symbolTable[index][1]=String.valueOf(locationCounter);
    System.out.println("IC=(DL,"+code+") (C, "+(index+1)+"')");
    locationCounter+=size;

    i++;
}

}
System.out.println("\n----->Literal Table");

```

```

        for(int index=0;index<litTabPtr;index++)
        {
            System.out.println(litTable[index][0]+ ":"+litTable[index][1]);
        }
        System.out.println("\n----->Symbol Table");
        for(int index=1;index<symbolTabPtr-
1;index++) //for testing values of symbol table
        {
            System.out.println(symbolTable[index][0] + " - " + symbolTable[ind
ex][1]);
        }
        System.out.println("\n----->Pool Table");
        for(int index=0;index<=poolTabPtr;index++)
        {
            System.out.println(poolTable[index]);
        }
    }

    public static String mnemonic(String token,String want)
    {
        String[][] codes = {{{"00","STOP","IS"}, {"01","ADD","IS"}, {"02","SUB","IS"}, {"03","MULT","IS"}, {"04","MOVER","IS"}, {"05","MOVEM","IS"}, {"06","COMP","IS"}, {"07","BC","IS"}, {"08","DIV","IS"}, {"09","READ","IS"}, {"10","PRINT","IS"}, {"01","DC","DL"}, {"02","DS","DL"}, {"01","START","AD"}, {"02","END","AD"}, {"03,"ORIGIN","AD"}, {"04,"EQU","AD"}, {"05,"LTORG","AD"}, {"1,"AREG","REG"}, {"2,"BREG","REG"}, {"3,"CREG","REG"}, {"4,"DREG","REG"}, {"1,"LT","FLAG"}, {"2,"LE","FLAG"}, {"3,"EQ","FLAG"}, {"4,"GT","FLAG"}, {"5,"GE","FLAG"}, {"6,"ANY","FLAG"}};
        for(String[] code : codes) //to return type or code of token
        {
            if(token.equals(code[1]))
            {
                if(want.equals("type"))
                    return code[2];
                if(want.equals("code"))
                    return code[0];
            }
        }
        return "";
    }
    //to find literals
    public static boolean isliteral(String token)
    {
        if(token.startsWith("=") || token.startsWith("\'"))
        {

```

```

        return true;
    }
    return false;
}

//tocheck already exist
public static int get_symbol_index(String token)
{
    int index;
    for(index=0;index<symbolTabPtr;index++)
    {
        if(symbolTable[index][0].equals(token))
        {
            return index;
        }
    }

    return -1;
}

```

Output :

PS D:\MCA\MCA SEM 3\SS> java .\SS_Assembler.java

LC= 200

IC=(IS,09) () (S, 1)

IC=(IS,04) (1) (S, 3)

IC=(IS,01) (1) (S, 2)

IC=(IS,05) (1) (S, 4)

IC=(IS,10) () (S, 5)

IC=(IS,10) () (S, 3)

IC=(DL,02) (C, 3)

IC=(DL,02) (C, 2)

IC=(DL,02) (C, 4)

----->**Literal Table**

----->**Symbol Table**

B - 206

A - 206

RESULT - 207

----->**Pool Table**

1

1

2. Macro Preprocessor

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.StringTokenizer;

public class Macro {

    public static void main(String[] args) throws IOException {

        List<String> input = new ArrayList<>();

        input.add("\tMACRO");
        input.add("\tCLEARMEM &X, &N, &REG=AREG");
        input.add("\tLCL &M");
        input.add("\t&M SET 0");
        input.add("\tMOVE &REG, ='0'");
        input.add(".MORE MOVEM &REG, &X + &M");
        input.add("\t&M SET &M+1");
        input.add("\tAIF (&M NE &N) .MORE");
        input.add("\tMEND");
        input.add("\tMMEND");

        System.out.println("Starting Preprocessing...");

        PreProcessor pr = new PreProcessor(input);

        pr.showCode();
        pr.analyze();
        pr.showTables();

        System.out.println("Ending Preprocessing...");
    }
}

class PreProcessor {
    private List<String> code;

    private List<String> pntab;
    private List<String> evntab;
    private List<String> ssntab;
    private List<MacroData> mnt;
    private List<String[]> kpdtab;
```

```
private List<Integer[]> sstab;
private List<String> mdt;

private int pntab_ptr;
private int evntab_ptr;
private int ssntab_ptr;
private int mnt_ptr;
private int kpdtab_ptr;
private int sstab_ptr;
private int mdt_ptr;

public PreProcessor(String filename) throws IOException {
    initialize();
    loadCode(filename);
}

public PreProcessor(List<String> code) {
    initialize();
    this.code = code;
}

private static List<String> tokenize(String line) {
    StringTokenizer st = new StringTokenizer(line, ", \t()");
    List<String> tokenized = new ArrayList<>();

    while (st.hasMoreTokens()) {
        tokenized.add(st.nextToken());
    }

    return tokenized;
}

private static String getParameterType(String parameter) {
    return parameter.indexOf('=') == -1 ? "PP" : "KP";
}

private static boolean isSequencingSymbol(String token) {
    return token.charAt(0) == '.';
}

private void initialize() {
    pntab = new ArrayList<>();
    evntab = new ArrayList<>();
    ssntab = new ArrayList<>();
    mnt = new ArrayList<>();
    kpdtab = new ArrayList<>();
    sstab = new ArrayList<>();
    mdt = new ArrayList<>();
}
```

```

        pntab_ptr = evntab_ptr = ssntab_ptr = mnt_ptr =
            kpdtab_ptr = sstab_ptr = mdt_ptr = 0;
    }

private String getIC(String data) {
    String ic = "(%s,%s)";
    int index = -1;
    int start = data.charAt(0) == '&' || data.charAt(0) == '.' ? 1 : 0;
    data = data.substring(start).toUpperCase();

    for(int i = 0; i < evntab_ptr && index == -1; i++) {
        if(evntab.get(i).toUpperCase().equals(data)) index = i;
    }
    if(index != -1) return String.format(ic, "E", ("'" + index));

    for(int i = 0; i < pntab_ptr && index == -1; i++) {
        if(pntab.get(i).toUpperCase().equals(data)) index = i;
    }
    if(index != -1) return String.format(ic, "P", ("'" + index));

    for(int i = 0; i < ssntab_ptr && index == -1; i++) {
        if(ssntab.get(i).toUpperCase().equals(data)) index = i;
    }
    if(index != -1) return String.format(ic, "S", ("'" + index));

    return null;
}

private static String removeSequencingSymbol(String line) {
    line = line.trim();
    if(line.charAt(0) == '.') {
        int indexOfSpace = line.indexOf(' ');
        line = line.substring(indexOfSpace + 1);
    }
    return line;
}

private String getLineIC(String line) {
    String lineIC = removeSequencingSymbol(line);
    List<String> tokenized = tokenize(lineIC);

    for(int i = 0; i < tokenized.size(); i++) {
        String ic = getIC(tokenized.get(i));

        if(ic != null) {
            lineIC = lineIC.replaceAll(tokenized.get(i), ic);
        }
    }
}

```

```

        }

    return lineIC;
}

private void loadCode(String filename) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    code = new ArrayList<>();

    String line;

    while ((line = reader.readLine()) != null) {
        code.add(line);
    }

    if (reader != null) reader.close();
}

public void showCode() {
    for (int i = 0; i < code.size(); i++) {
        System.out.println(code.get(i));
    }
}

public void showTables() {
    System.out.println("\n----- TABLES -----\\n");

    System.out.println("----- MNT -----");
    System.out.println("MACRONAME\t#PP\t#KP\t#EV\tMDTP\tKPDTP\tsSTP");
    System.out.println("-----");
    for (int i = 0; i < mnt_ptr; i++) {
        MacroData md = mnt.get(i);
        System.out.println(md.name + "\t" + md.pp + "\t" + md.kp + "\t" +
mnt.ev + "\t" + md.mdtp + "\t\t" + md.kpdtp + "\t\t" + md.sstp);
    }
    System.out.println("-----");

    System.out.println("\n----- PNTAB -----");
    System.out.println("Index\tName");
    System.out.println("-----");
    for (int i = 0; i < pntab_ptr; i++) {
        System.out.println(i + "\t" + pntab.get(i));
    }
    System.out.println("-----");

    System.out.println("\n----- EVNTAB -----");
    System.out.println("Index\tName");
    System.out.println("-----");
    for (int i = 0; i < evntab_ptr; i++) {

```

```

        System.out.println(i + "\t\t" + evntab.get(i));
    }
    System.out.println("-----");

    System.out.println("\n----- SSNTAB -----");
    System.out.println("Index\tName");
    System.out.println("-----");
    for (int i = 0; i < ssntab_ptr; i++) {
        System.out.println(i + "\t\t" + ssntab.get(i));
    }
    System.out.println("-----");

    System.out.println("\n----- SSTAB -----");
    System.out.println("Index\tValue\tValue");
    System.out.println("-----");
    for (int i = 0; i < sstab_ptr; i++) {
        System.out.println(i + "\t\t" + sstab.get(i)[0] + "\t\t" + sstab.get(i)[1]);
    }
    System.out.println("-----");

    System.out.println("\n----- KPDTAB -----");
    System.out.println("Index\tName\tDefault");
    System.out.println("-----");
    for (int i = 0; i < kpdtab_ptr; i++) {
        System.out.println(i + "\t\t" + kpdtab.get(i)[0] + "\t\t" + kpdtab.get(i)[1]);
    }
    System.out.println("-----");

    System.out.println("\n----- MDT -----");
    System.out.println("Index\tIC");
    System.out.println("-----");
    for (int i = 0; i < mdt_ptr; i++) {
        System.out.println(i + "\t\t" + mdt.get(i));
    }
    System.out.println("-----");
}

public void analyze() {

    List<String> tokenized;
    MacroData md = new MacroData();

    String prototype = code.get(1);
}

```

```

tokenized = tokenize(prototype);

md.name = tokenized.get(0);
md.kpdtb_ptr = kpdtab_ptr;

for (int i = 1; i < tokenized.size(); i++) {
    String parameter = tokenized.get(i);
    if (getParameterType(parameter).equals("PP")) {
        System.out.println(parameter + " is PP");
        pntab.add(parameter.substring(1));
        pntab_ptr++;
        md.pp++;
    } else {
        System.out.println(parameter + " is KP");
        int index = parameter.indexOf('=');
        String parameterName = parameter.substring(1, index);
        String defaultValue = parameter.substring(index + 1);
        String[] kpdtab_entry = {parameterName, defaultValue};

        kpdtab.add(kpdtab_entry);
        pntab.add(parameterName);

        kpdtab_ptr++;
        pntab_ptr++;
        md.kp++;
    }
}

md.mdtb = mdt_ptr;
md.ev = 0;
md.sstb_ptr = sstab_ptr;

for (int i = 2; i < code.size(); i++) {
    String currentLine = code.get(i);
    tokenized = tokenize(currentLine);

    if(tokenized.size() < 1) continue;

    boolean hasSequencingSymbol = isSequencingSymbol(tokenized.get(0))
;

    if(hasSequencingSymbol) {
        ssntab.add(tokenized.get(0).substring(1));
        int index = ssntab_ptr++;

        Integer[] data = {index, mdt_ptr};
        sstab.add(data);
    }
}

```

```

System.out.println("CurrentLine: " + currentLine);
if (tokenized.get(0).toUpperCase().equals("LCL")) {
    int start = tokenized.get(1).charAt(0) == '&'amp; 1 : 0;
    String variable = tokenized.get(1).substring(start);
    evntab.add(variable);
    evntab_ptr++;
    md.ev++;

    String lineIC = getLineIC(currentLine);
    System.out.print(lineIC);
    mdt.add(lineIC);
    mdt_ptr++;
}
else if(tokenized.size() > 1 && tokenized.get(1).toUpperCase().equals("SET")) {
    String lineIC = getLineIC(currentLine);
    System.out.println("IC-> "+lineIC);
    mdt.add(lineIC);
    mdt_ptr++;
}
else if(tokenized.get(0).toUpperCase().equals("AIF") || tokenized.get(0).toUpperCase().equals("AGO")) {
    String sequencingSymbol = tokenized.get(tokenized.size() - 1).
substring(1);

    int index = ssntab.indexOf(sequencingSymbol);

    if(index == -1) {
        ssntab.add(sequencingSymbol);
        index = ssntab_ptr++;
    }

    String lineIC = getLineIC(currentLine);
    System.out.println(lineIC);
    mdt.add(lineIC);
    mdt_ptr++;
}
else if (tokenized.get(0).toUpperCase().equals("MEND")) {
    if(ssntab_ptr == 0) md.sstp = 0;
    else sstab_ptr = sstab_ptr + ssntab_ptr;
    break;
}
else {
    String lineIC = getLineIC(currentLine);
    System.out.print(lineIC);
    mdt.add(lineIC);
    mdt_ptr++;
}

```

```

        }
    }

    mnt.add(md);
    mnt_ptr++;
}

}

class MacroData {
    String name;
    int pp, kp, ev, mdtp, kpdtp, sstp;

    MacroData() {
        name = "";
        pp = kp = ev = mdtp = kpdtp = sstp = 0;
    }
}

```

Output :

PS D:\MCA\MCA SEM 3\SS> java .\Macro.java

Starting Preprocessing...

MACRO

CLEARMEM &X, &N, ®=AREG

LCL &M

&M SET 0

MOVER ®, ='0'

.MORE MOVEM ®, &X + &M

&M SET &M+1

AIF (&M NE &N) .MORE

MEND

MMEND

&X is PP

&N is PP

®=AREG is KP

CurrentLine: LCL &M

LCL (E,0)CurrentLine: &M SET 0

IC-> (E,0) SET 0

CurrentLine: MOVER ®, ='0'

MOVER (P,2), ='0'CurrentLine: .MORE MOVEM ®, &X + &M
MOVEM (P,2), (P,0) + (E,0)CurrentLine: &M SET &M+1
IC-> (E,0) SET (E,0)+1
CurrentLine: AIF (&M NE &N) .MORE
AIF ((E,0) NE (P,1)) (S,0)
CurrentLine: MEND

----- TABLES -----

----- MNT -----

MACRONAME	#PP	#KP	#EV	MDTP	KPDTP	SSTP
CLEARMEM	2	1	1	0	0	0

----- PNTAB -----

Index Name

0	X
1	N
2	REG

----- EVNTAB -----

Index Name

0	M
---	---

----- SSNTAB -----

Index Name

0	MORE
---	------

----- SSTAB -----

Index	Value	Value
-------	-------	-------

0	0	3
---	---	---

----- KPDTAB -----

Index	Name	Default
-------	------	---------

0	REG	AREG
---	-----	------

----- MDT -----

Index	IC
-------	----

0	LCL (E,0)
1	(E,0) SET 0
2	MOVER (P,2), ='0'
3	MOVEM (P,2), (P,0) + (E,0)
4	(E,0) SET (E,0)+1
5	AIF ((E,0) NE (P,1)) (S,0)

Ending Preprocessing...

3. Top Down Without Backtracking

```
public class TopDown {
    public static void main(String[] args) {
        System.out.println("TopDownWithoutBackTrack");
        TopDownWithoutBackTrack a = new TopDownWithoutBackTrack();

        String parsed = a.parse("a + b * c * d + e");

        System.out.println("Parsed: " + parsed);

        // System.out.println(a.replaceAt(1, "TE'", "+E", 3));
    }
}

class TopDownWithoutBackTrack {
    private static final String EPSILON = "";

    private static String replaceAt(int index, String subject, String replacement, int size) {
        return subject.substring(0, index) + replacement + subject.substring(index + size);
    }

    public String parse(String equation) {
        System.out.println("Steps: ");

        String parsed = "E";
        int indexInEquation = 0, index = 0, count = 0;
        equation = equation.replaceAll(" ", "");

        while (index < parsed.length()) {
            count++;
            System.out.println(String.format("%2d", count) + ": " + parsed);

            if (parsed.charAt(index) == 'E') {
                // E'
                if (index < parsed.length() - 2 && parsed.charAt(index + 1) == '\''
                    && parsed.charAt(index + 2) == '\'') {
                    if (indexInEquation < equation.length() && equation.charAt(indexInEquation) == '+') {
                        parsed = replaceAt(index, parsed, "+E", 3);
                        indexInEquation++;
                    } else
                        parsed = replaceAt(index, parsed, EPSILON, 3);
                }
            }
        }
    }
}
```

```

        // E
        else {
            parsed = replaceAt(index, parsed, "TE''", 1);
        }
    } else if (parsed.charAt(index) == 'T') {
        // T'
        if (index < parsed.length() - 2 && parsed.charAt(index + 1) ==
        '\'
            && parsed.charAt(index + 2) == '\') {
            if (indexInEquation < equation.length() && equation.charAt(
            indexInEquation) == '*') {
                parsed = replaceAt(index, parsed, "*T", 3);
                indexInEquation++;
            } else
                parsed = replaceAt(index, parsed, EPSILON, 3);
        }
        // T
        else {
            parsed = replaceAt(index, parsed, "VT''", 1);
        }
    } else if (parsed.charAt(index) == 'V') {
        parsed = replaceAt(index, parsed, "<id>", 1);
        indexInEquation++;
        index += 4;
    } else
        index++;
}
System.out.println(String.format("%2d", ++count) + ": " + parsed);
System.out.println("Completed in " + count + " steps.");
return parsed;
}
}

class TreeNode {
    private char expression;
    private TreeNode leftNode, rightNode;

    public TreeNode() {
    }

    public TreeNode(char expression, TreeNode leftNode, TreeNode rightNode) {
        this.expression = expression;
        this.leftNode = leftNode;
        this.rightNode = rightNode;
    }

    public void postOrderTraversal() {
        if (this.leftNode != null)

```

```

        leftNode.postOrderTraversal();

        if (this.rightNode != null)
            rightNode.postOrderTraversal();

        System.out.print(this.expression);
    }
}

```

Output :

PS D:\MCA\MCA SEM 3\SS\Parsers> java .\TopDown.java
TopDownWithoutBackTrack

Steps:

- 1: E
- 2: TE"
- 3: VT"E"
- 4: <id>T"E"
- 5: <id>E"
- 6: <id>+E
- 7: <id>+E
- 8: <id>+TE"
- 9: <id>+VT"E"
- 10: <id>+<id>T"E"
- 11: <id>+<id>*TE"
- 12: <id>+<id>*TE"
- 13: <id>+<id>*VT"E"
- 14: <id>+<id>*<id>T"E"
- 15: <id>+<id>*<id>*TE"
- 16: <id>+<id>*<id>*TE"
- 17: <id>+<id>*<id>*VT"E"
- 18: <id>+<id>*<id>*<id>T"E"
- 19: <id>+<id>*<id>*<id>E"
- 20: <id>+<id>*<id>*<id>+E
- 21: <id>+<id>*<id>*<id>+E
- 22: <id>+<id>*<id>*<id>+TE"
- 23: <id>+<id>*<id>*<id>+VT"E"

24: <id>+<id>*<id>*<id>+<id>T"E"

25: <id>+<id>*<id>*<id>+<id>E"

26: <id>+<id>*<id>*<id>+<id>

Completed in 26 steps.

Parsed: <id>+<id>*<id>*<id>+<id>

4 . Recursive Decent Parser

```
import java.util.Scanner;

public class RD {
    public static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.print("Enter the Expression: ");
        String expression = scanner.nextLine();
        RecursiveDescentParser recursiveDescentParsing = new RecursiveDescentParser(expression);
        TreeNode rootNode;

        rootNode = recursiveDescentParsing.proc_E();

        if (rootNode != null) {
            rootNode.postOrderTraversal();
        }
    }
}

class RecursiveDescentParser {
    private String expressionString;
    private int indexInEquation = 0;

    public RecursiveDescentParser(String expressionString) {
        this.expressionString = expressionString;
        this.indexInEquation = 0;
    }

    public TreeNode proc_E() {
        TreeNode leftNode = null, rightNode = null;
        leftNode = proc_T();

        while (indexInEquation < expressionString.length() && expressionString.charAt(indexInEquation) == '+') {
            this.indexInEquation++;
            rightNode = proc_T();

            if (rightNode == null)
                return null;

            leftNode = new TreeNode('+', leftNode, rightNode);
        }
        return leftNode;
    }
}
```

```
public TreeNode proc_T() {
    TreeNode leftNode = null, rightNode = null;
    leftNode = proc_V();

    while (indexInEquation < expressionString.length() && expressionString.charAt(indexInEquation) == '*') {
        this.indexInEquation++;
        rightNode = proc_V();

        if (rightNode == null)
            return null;

        leftNode = new TreeNode('*', leftNode, rightNode);
    }
    return leftNode;
}

public TreeNode proc_V() {
    if (indexInEquation < expressionString.length() && expressionString.charAt(indexInEquation) != '*'
        && expressionString.charAt(indexInEquation) != '+')
        return new TreeNode(expressionString.charAt(indexInEquation++), null, null);

    else {
        System.out.println("\nInvalid Expression!");
        return null;
    }
}
}

class TreeNode {
    private char expression;
    private TreeNode leftNode, rightNode;

    public TreeNode() {
    }

    public TreeNode(char expression, TreeNode leftNode, TreeNode rightNode) {
        this.expression = expression;
        this.leftNode = leftNode;
        this.rightNode = rightNode;
    }

    public void postOrderTraversal() {
        if (this.leftNode != null)
            leftNode.postOrderTraversal();
    }
}
```

```
        if (this.rightNode != null)
            rightNode.postOrderTraversal();

        System.out.print(this.expression);
    }
}
```

Output:

PS D:\MCA\MCA SEM 3\SS\Parsers> java .\RD.java

Enter the Expression: x+x*x

xxx*+

5 . Operator Precedence Parser

```
import java.util.Stack;

public class OP {
    public static void main(String[] args) {
        String equation = "x + x * x";
        OperatorPrecedenceParser a = new OperatorPrecedenceParser();
        OperatorPrecedenceParser.TreeNode tree = a.parse(equation);

        System.out.println("Equation: " + equation);
        System.out.print("InOrder Traversal: ");
        OperatorPrecedenceParser.inOrder(tree);

        System.out.print("\nPostOrder Traversal: ");
        OperatorPrecedenceParser.postOrder(tree);
        System.out.println();

    }
}

class OperatorPrecedenceParser {
    public static class TreeNode {
        char data;
        TreeNode left, right;

        TreeNode(char value) {
            data = value;
            left = right = null;
        }
    }

    private static short getPriority(char op) {
        switch (op) {
            case '+':
            case '-':
                return 1;

            case '/':
            case '*':
                return 2;

            default:
                return 0;
        }
    }
}
```

```

private static boolean isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

private static boolean isOperand(char ch) {
    return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'));
}

private static boolean isOpeningBracket(char ch) {
    return (ch == '(' || ch == '{' || ch == '[');
}

private static boolean isClosingBracket(char ch) {
    return (ch == ')' || ch == '}' || ch == ']');
}

private static char getPair(char bracket) {
    switch (bracket) {
        case '(':
            return ')';
        case '{':
            return '}';
        case '[':
            return ']';
        case ')':
            return '(';
        case '}':
            return '{';
        case ']':
            return '[';
        default:
            return (char) 0;
    }
}

private static String toPostFix(String equation) {
    Stack<Character> operators = new Stack<>();
    String postfix = "";
    for (int i = 0; i < equation.length(); i++) {
        char ch = equation.charAt(i);

        if (isOpeningBracket(ch))
            operators.push(ch);
        else if (isClosingBracket(ch)) {
            char op = operators.pop();
            char openingPair = getPair(ch);

            while (op != openingPair) {

```

```

        postfix += op;
        op = operators.pop();
    }
} else if (isOperator(ch)) {
    short previousPriority = operators.isEmpty() ? 0 : getPriority(operators.peek());
    short currentPriority = getPriority(ch);

    while (previousPriority != 0 && previousPriority >= currentPriority) {
        postfix += operators.pop();
        previousPriority = operators.isEmpty() ? 0 : getPriority(operators.peek());
    }

    operators.push(ch);
} else if (isOperand(ch))
    postfix += ch;
}

while (!operators.isEmpty())
    postfix += operators.pop();

return postfix;
}

private static TreeNode getExpressionTree(String equation) {
    Stack<TreeNode> stack = new Stack<>();
    for (int i = 0; i < equation.length(); i++) {
        char ch = equation.charAt(i);

        if (isOperator(ch)) {
            TreeNode operand2 = stack.pop();
            TreeNode operand1 = stack.pop();
            TreeNode parentNode = new TreeNode(ch);
            parentNode.left = operand1;
            parentNode.right = operand2;

            stack.push(parentNode);
        } else if (isOperand(ch))
            stack.push(new TreeNode(ch));
    }

    return stack.pop();
}

public static void inOrder(TreeNode root) {
    if (root == null)

```

```
        return;

        inOrder(root.left);
        System.out.print(root.data);
        inOrder(root.right);
    }

    public static void postOrder(TreeNode root) {
        if (root == null)
            return;

        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data);
    }

    public TreeNode parse(String equation) {
        return getExpressionTree(toPostFix(equation));
    }
}
```

Output:

PS D:\MCA\MCA SEM 3\SS\Parsers> java .\OP.java

Equation: x + x * x

InOrder Traversal: x+x*x

PostOrder Traversal: xxx*+

6. LL1 Parser

```
public class LL1 {
    public static void main(String[] args) {
        System.out.println("LL1Parser");
        LL1Parser a = new LL1Parser();

        String parsed = a.parse("a * b + c");

        System.out.println("Parsed: " + parsed);

        // System.out.println(a.replaceAt(1, "TE'", "+E", 3));
    }
}

class LL1Parser {
    private static final String EPSILON = "";

    private static String replaceAt(int index, String subject, String replacement, int size) {
        return subject.substring(0, index) + replacement + subject.substring(index + size);
    }

    public String parse(String equation) {
        System.out.println("Steps: ");

        String parsed = "E";
        int indexInEquation = 0, index = 0, count = 0;
        equation = equation.replaceAll(" ", "");

        while (index < parsed.length()) {
            count++;
            System.out.println(String.format("%2d", count) + ". " + parsed);
            if (parsed.charAt(index) == 'E') {
                // E'
                if (index < parsed.length() - 1 && parsed.charAt(index + 1) == '\'') {
                    if (indexInEquation < equation.length() && equation.charAt(indexInEquation) == '+') {
                        parsed = replaceAt(index, parsed, "+TE'", 2);
                        indexInEquation++;
                    } else
                        parsed = replaceAt(index, parsed, EPSILON, 2);
                }
                // E
            } else {
                parsed = replaceAt(index, parsed, "TE'", 1);
            }
        }
    }
}
```

```

        }
    } else if (parsed.charAt(index) == 'T') {
        // T
        if (index < parsed.length() - 1 && parsed.charAt(index + 1) ==
        '\'' ) {
            if (indexInEquation < equation.length() && equation.charAt(
            indexInEquation) == '*') {
                parsed = replaceAt(index, parsed, "*VT'", 2);
                indexInEquation++;
            } else
                parsed = replaceAt(index, parsed, EPSILON, 2);
        }
        // T
        else {
            parsed = replaceAt(index, parsed, "VT'", 1);
        }
    } else if (parsed.charAt(index) == 'V') {
        parsed = replaceAt(index, parsed, "<id>", 1);
        indexInEquation++;
        index += 4;
    } else
        index++;
    }
    System.out.println(String.format("%2d", ++count) + ". " + parsed);
    System.out.println("Completed in " + count + " steps.");
    return parsed;
}
}

```

Output :

PS D:\MCA\MCA SEM 3\SS\Parsers > java .\LL1.java

LL1Parser

Steps:

1. E
2. TE'
3. VT'E'
4. <id>T'E'
5. <id>*VT'E'
6. <id>*VT'E'
7. <id>*<id>T'E'
8. <id>*<id>E'
9. <id>*<id>+TE'

10. $\langle \text{id} \rangle^* \langle \text{id} \rangle + T E'$

11. $\langle \text{id} \rangle^* \langle \text{id} \rangle + V T' E'$

12. $\langle \text{id} \rangle^* \langle \text{id} \rangle + \langle \text{id} \rangle T' E'$

13. $\langle \text{id} \rangle^* \langle \text{id} \rangle + \langle \text{id} \rangle E'$

14. $\langle \text{id} \rangle^* \langle \text{id} \rangle + \langle \text{id} \rangle$

Completed in 14 steps.

Parsed: $\langle \text{id} \rangle^* \langle \text{id} \rangle + \langle \text{id} \rangle$

7. Scanner

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class ScannerDemo {
    public static void main(String[] args) {
        String[] valids = { "aaabbccddd", "aaabcddd", "abcd", "aaaaabbbbddddd",
", "abd" };
        MyScanner sc = new MyScanner(valids);
        boolean check6 = sc.check("cccddd");
        System.out.println("aaaaccddd is " + (check6 ? " valid" : " not valid
."));
        System.out.println();
    }
}

class State {
    char symbol;
    List<Character> nextStates;

    State(char state) {
        this.symbol = state;
        nextStates = new ArrayList<>();
    }

    boolean hasNextState(char state) {
        return this.nextStates.stream().anyMatch(ch -> ch == state);
    }

    @Override
    public String toString() {
        String state = "State: " + (symbol == (int) 0 ? "start" : symbol) + ", ";
        Next States: ";

        for (char ch : nextStates)
            state += ch + ", ";

        return state.substring(0, state.length() - 2);
    }
}

class MyScanner {
    State start;
    List<State> states;
```

```

public MyScanner() {
    this.initialize();
    this.createDFA();
    this.displayStates();
}

public MyScanner(String[] valids) {
    this.initialize(valids);
    this.displayStates();
}

private void initialize() {
    start = new State((char) 0);
    State[] list = new State[] { new State('a'), new State('b'), new State
('c'), new State('d') };
    states = Arrays.asList(list);
}

private void initialize(String[] valids) {
    this.states = new ArrayList<>();
    this.start = new State((char) 0);

    for (String valid : valids) {
        State current = this.start;

        for (int i = 0; i < valid.length(); i++) {
            char ch = valid.charAt(i);

            if (this.getState(ch) == null)
                this.states.add(new State(ch));

            if (!current.hasNextState(ch))
                current.nextStates.add(ch);

            current = this.getState(ch);
        }
    }
}

private State getState(char value) {

    return this.states.stream().filter(state -
> state.symbol == value).findAny().orElse(null);
}

private void createDFA() {
    start.nextStates.add('a');
}

```

```

        State a = this.getState('a');
        a.nextStates.add('a');
        a.nextStates.add('b');

        State b = this.getState('b');
        b.nextStates.add('b');
        b.nextStates.add('c');
        b.nextStates.add('d');

        State c = this.getState('c');
        c.nextStates.add('c');
        c.nextStates.add('d');

        State d = this.getState('d');
        d.nextStates.add('d');
    }

    private void displayStates() {
        System.out.println(start);

        this.states.forEach(System.out::println);
    }

    public boolean check(String expression) {

        State current = start;

        for (int i = 0; i < expression.length(); i++) {
            char symbol = expression.charAt(i);

            if (current.hasNextState(symbol)) {
                System.out
                    .println((current.symbol == (int) 0 ? "start" : current.symbol) + " has next state " + symbol);
                current = this.getState(symbol);
            }
            else
                return false;
        }
        return true;
    }
}

```

Output :

PS D:\MCA\MCA SEM 3\SS\scanner> java .\ScannerDemo.java

State: start, Next States: a

State: a, Next States: a, b

State: b, Next States: b, c, d

State: c, Next States: c, d

State: d, Next States: d

aaaacccddd is not valid.