Department of Computer Science

Gujarat University

Certificate

Roll No: 30 Seat No: _____

This is to certify that Mr./~~Ms.~~ Rathod Ajinkya Sreekant 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

Name: - Ajinkya Rathod

Roll No.: - 30 **Exam Seat No.: -**10029

Sr. No.	Contents	Pg. No	Date	Signature
1.	Assignment - 1		10th Dec	
	(Introduction to System Software			
	and Assembler)			
2.	Assignment - 2		10th Dec	
	(Macro Processor)			
3.	Assignment - 3		10th Dec	
	(Scanner and Parser)			
4.	Assignment 0 4		10th Dec	
	(Linkers and Loaders)			
5.	Assignment - 5		10th Dec	
	(Compilers and Interpreters)			

CELEBRO
DATE
PAGE

System Software

Name: Ajinkya Rathod

Std: MCA - III
Class

Rollwala

1. Stop:

- It is an imperative statement
- It stops execution of a program

2. ADD

- It is an imperative statement
- Its OP code is 01.
- Its adds value with register value.

3. Sub:

- It is an imperative statement
- Its OP code is 02.
- It substracts value with register value.

4. Mult:

- It is an imperative statement.
- Its OP code is 03.
- It multiplies value with register value.

MOVER

- It is imperative statement
- Its OP code is 04.
- Used to move value from memory to register.

MOVEM

- It is imperative statement.
- Its OP code is 05.
- Used to move value from register on memory.

COMP:

- It is imperative statemt.
- OP code is 06.
- Used to compare and set condition code

BC

- It is imperative statemt
- OP is 07.
- Used for Branch Condition

Read

- It is imperative statement
- OP code = 9.
- Used to read a value

PRINT

- It is imperative statement.
- OP code = 10.
- Used to print contents in register.

ORIGIN

- It is assembler directive statement.
- This directive instructs the assembler to put address given by address specification in location counter.

EQU

- It is assembler directive statement.
- In Eqn, left side is label/symbol and right side is address

PURGE

- It is an assembler directive statement.
- It is used in undefined the symbol names which are defined by ORIGIN statement.

ASSUME

- It is an assembler directive statement.
- It tells assembler that it can assume address of indicated segment to present in <Register>.

Register Segment

- It is the memory used to store three components of a program
- i.e program code, data & stack.

Segment PROC

PROC

- It indicate that it is a procedure.
- near: It indicate that whether the call to procedure is to be assumed as near call. i.e. procedure is called from segment.

FAR

It indicate whether the call to procedure is to be assembled as near call i.e the procedure is called from same segment

PUBLIC

When a symbol is define as public in a prog that means it can be refered by another progs until and unless

Extern

When an assembly module wishing to use a symbol declare in another assembly module.
The symbol should be declare as extern.

OFFSET

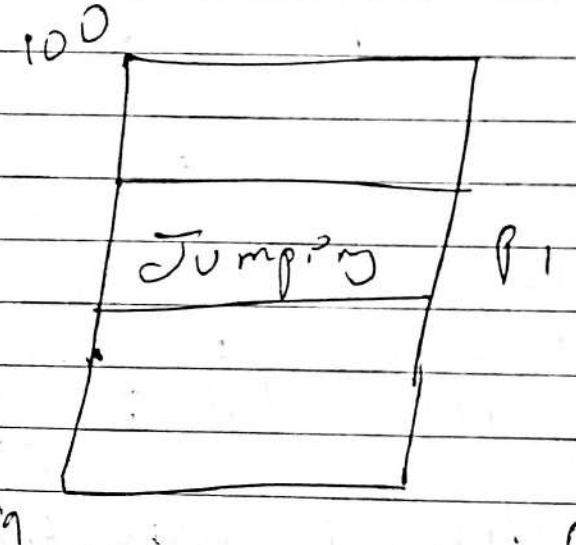
It is displacement from base address of segment.

Explain

(5.) Base Register

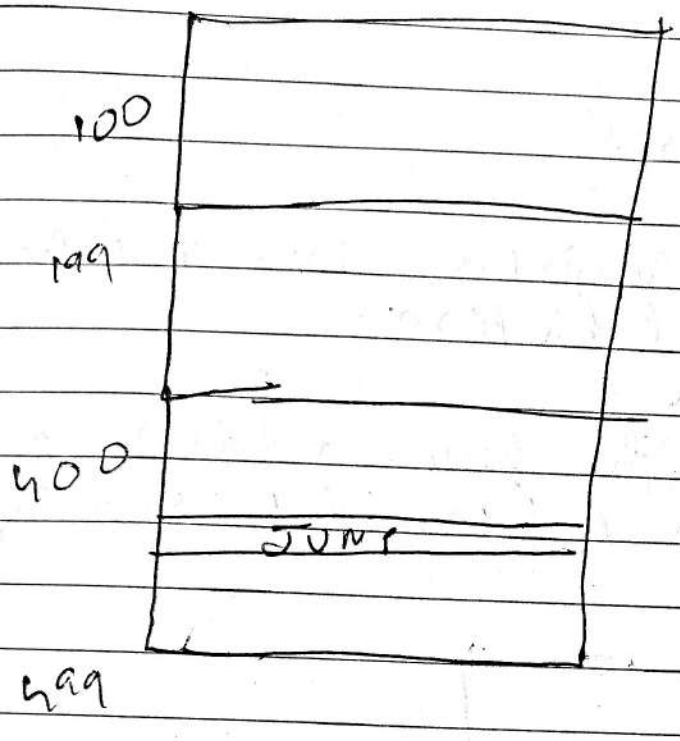
The base register is used for program relocation.

It holds the base addrss of data in RAM to be executed recently



D.1

Now the program send to secondary memory for some time then another program P2 will occupy address



D-2

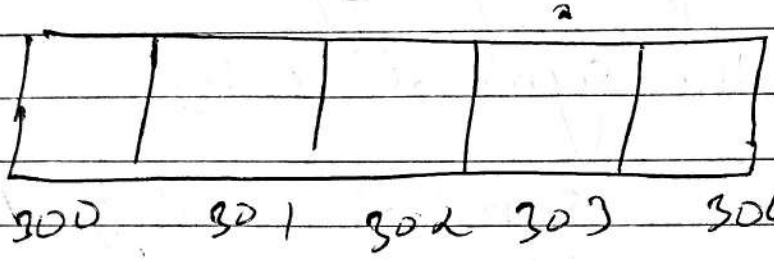
Now problem will occur as we have change the memory addrs of Progm P1. but it having an instruction JUMP 160, 60 ; it create an illegal instruction.

EA = Base Register value + displacent.

Index Registers

It Have source index of array and destination of index

e.g.



= 300+3
= 303

Symbol Table

An identifier used in source prog^n is called symbol.

Names of various functions and procedures are symbols.

It does :-

→ Add symbol
→ Add location of symbol.
→ Add length of symbol.
→ Delete the symbol entry
→ Access the symbol entry

Literal Table

It is used to collect all literals used in Program.

Ref ppt

4) Pool Table

This table contains the fields mnemonic code, opcode

Eg: Pooltab.

Literal No.
#1
#3

5) Declaration Statement

The syntax looks like

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

→ DC elem constructs a memory containing constants

Assembler Directives

A+ instructs the assembler to perform certain action in assembly program.

START

This indicates that first word of machine should be placed in memory word with address

First word of target program is stored from memory location 500 onwards.

Q3 Differences

a.

Application Domain	Execution Domain
-> In this domain designer expresses their ideas	-> In this ideas, expressed by designer are implemented
-> The code of any program is written in	-> The form of code with output is produced.

Specification Gap	Execution Gap
-> The gap between application domain and program domain is called specification gap	The gap between program domain and execution domain is called Execution gap

Assembler converts assembly language to machine language.

- P & L Types

1. Problem Oriented
2. Procedure Oriented.

OOP and Event Driven were not in picture during this time
↓
when this book was written.

Problem Oriented

My work is limited to certain thing or apps.

So, my problem.
So, developing lang. As a specifically my problem is problem Oriented

Procedure

Provides general purpose facilities in almost all domains

Problem Oriented

-> COBOL -- Specific for Business

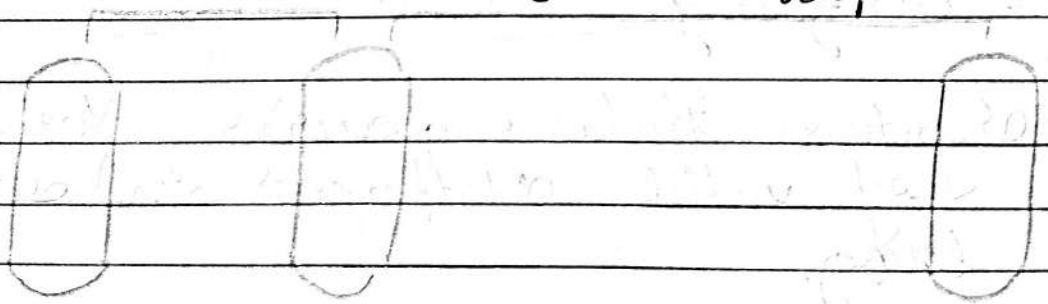
-> FORTRAN

-> Specific for mathematical & Calculations and scientific data.

Procedural

-> C.

Its used for scientific, business all level of accuracy will be less than FORTRAN. but it will be balanced.



Q. Compiler vs Interpreter

Compiler

-> Compiler converts the entire program

-> It produces optimized code

-> Analysis time is more.

-> Machine code is generated.

-> There is execution gap.

Interpreter

-> Interpreter translates code one line at a time

-> Program analysis time is less

-> No m/c level code is generated.

-> There is no execution gap

Fundamentals of Language Processing

Language Processing ≡ Analysis of SP + Synthesis of TP.

SP -> Source Prog.
TP -> Target Prog.

Synthesis -> To build something

Specification consist of 3 components

1. Lexical Rules: governs the formation of valid lexical units in source language.
2. Syntax Rules: governs formation of valid statements in source lang.
3. Semantic Rules: associate meaning with valid statement of language.

MOVER → Move Register
AREG → A Register
MULT → Multiply
DIV → Division

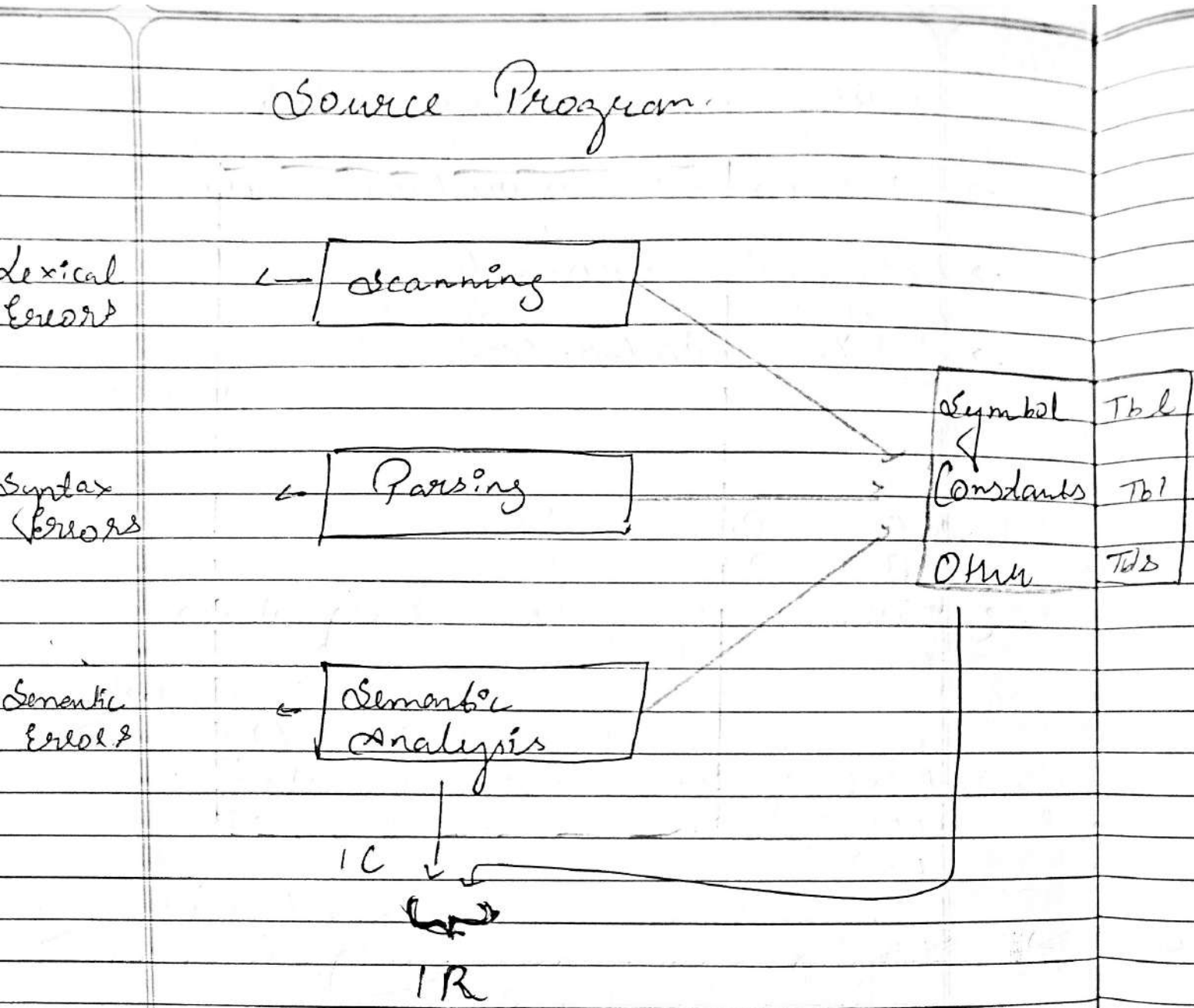
DW → single word
DD → Double word

sale price = cost_price + profit*2;

Lexical
If we are writing C.
Can't write ~~print~~ "import".
This is Java's keyword.

Syntax
1. Semicolon
2. sale_price = underscore nai hai so syntax rules.

Semantic
profit* 2 pele hoga
+ baad me.
My name is Ajinkya ✓
Ajinky my is ✗


Source Program
Lexical Errors
Scanning
Syntax Errors
Parsing
Semantic Errors
Semantic Analysis
IC
IR
Symbol Tbl
Constants Tbl
Other Tbls

CELEBRO™
DATE
PAGE

04

Single Pass Assembler

→ Scans entire source file only once.

(i) Deals with syntax

(i.) Constructs symbol table

(i.i) Creates label list

(i.v) Translates code segment, data segment, stack segment etc.

Two Pass Assembler

→ Requires two pass to scan source file

1st Pass: → For label definition

2nd pass → Translates assembly or equivalent m/c level code

Significance of Location Counter

- To implement memory allocation, data structure called location counter is introduced.
- Location counter is always used to contain the address of next memory word in target program.
- It is initialized to constant specified in START statement.
- It assists LC points to next memory word in target program even when machine instructions have different lengths.
- Eg

Symbol Table

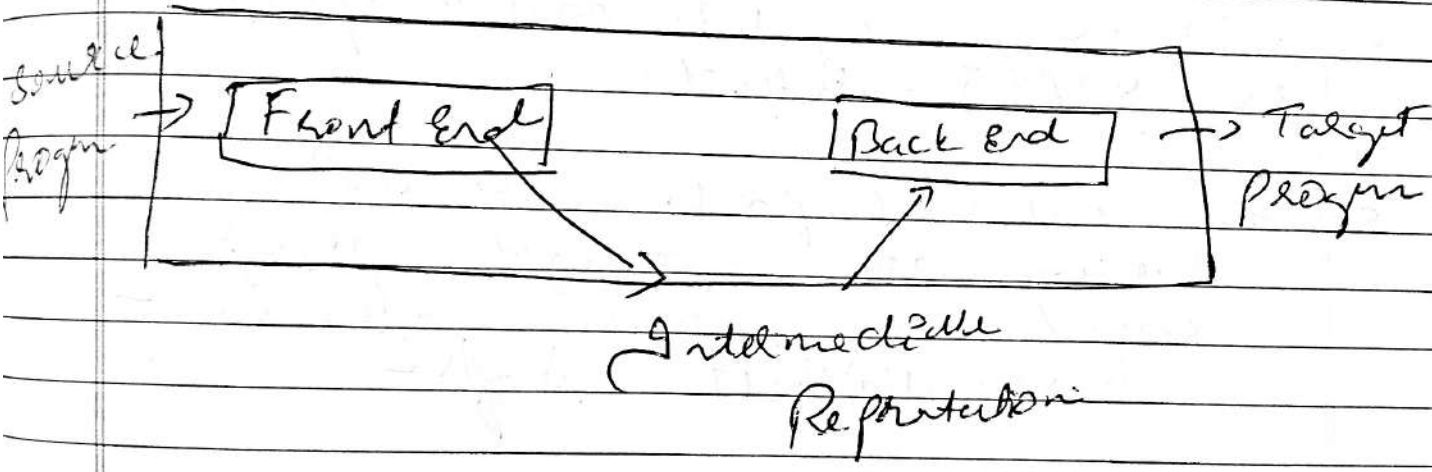
	Symbol	Address
→	Again	104
→	N	113

CELEBRO™
DATE
PAGE

04

What is intermediate representation in an Assembler?

→ An IR is a representation of source program which reflects the effect of some but not all, analysis and synthesis tasks



Properties of IR

- Ease of use
- Processing Efficiency
- Memory Efficiency

(d) Give instruction format of 8085 microprocessor.

-> The intel 8085 microprocessor supports 8 and 16 bit arithmetic and also provides special instruction for string manipulation

(a) Register / Memory to Register

(b) Immediate to Register / Memory

(c) Immediate to Accumulator

-> The direction field in instruction indicates which operand is the destination in instruction.

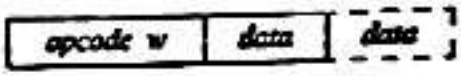
If d=0, the register / memory operand is the destination

Assembly statement	*Opcode d w*	*mod reg r/m*	*data/ displacement*
ADD AL, BL	000000 0 0	11 011 000	
ADD AL, 12H [SI]	000000 1 0	01 000 100	00010010
ADD AX, 3456H	100000 0 1	11 000 000	01010110
			00110100
ADD AX, 3456H	000001 0 1	01 010 110	00110100

Fig. 4.22 Sample instructions of 8088

adequate. The third instruction contains 16 bits of immediate data. Note that the low byte of immediate data comes first, followed by its high byte. The fourth assembly statement is identical to the third, however it has been encoded using the immediate

(c) *Immediate to Accumulator*



r/m	*mod* = 00	*mod* = 01	*mod* = 10	*mod* = 11	
				w=0	*w*=1
000	(BX) + (SI)	(BX) + (SI) + d8	Note 2	AL	AX
001	(BX) + (DI)	(BX) + (DI) + d8	Note 2	CL	CX
010	(BP) + (SI)	(BP) + (SI) + d8	Note 2	DL	DX
011	(BP) + (DI)	(BP) + (DI) + d8	Note 2	BL	BX
100	(SI)	(SI) + d8	Note 2	AH	SP
101	(DI)	(DI) + d8	Note 2	CH	BP
110	Note 1	(BP) + d8	Note 2	DH	SI
111	(BX)	(BX) + d8	Note 2	BH	DI

Note 1 : (BP) + DISP for indirect addressing, d16 for direct
Note 2 : Same as previous column, except d16 instead of d8

	Register	
reg	*8-bit* (w=0)	*16-bit* (w=1)
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Fig. 4.21 Instruction formats of Intel 8088

Explain significance of segment register.

=> Segmentation is process in which memory of PC is locally divided into each segment.

-> And all has it own base address.

Need for segmentation

The BIU contains 4 bits special purpose registers called segment registers.

Types

1. CS

Code segment Registers.
It is used for addressing memory where execute progr is stored

2. ES

Extra segment

7. DS
Data Segment.
It points to data segment of memory where data is stored.

(iv) Stack Segment

Used for addressing stack segment of the memory. It is segment in which stack data is stored.

Macros and Macro Prototype

CELEBRO
DATE
PAGE

01. Macro Prototype

-> It declares name of macro and names and kinds of its parameter.

-> Syntax:

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

02. Macro Definition

Macro Definition is enclosed between a macro header statement and macro end statement.

-> It consists of

1. Macro Prototype statement
2. One or model statement.

CELEBRO
DATE
PAGE

03

Model Statements

-> A model statement is statement from which assembly language statements may be generated during macro expansion.

-> Statement between MACRO and MEND directives define model statements may more and can appear in expansion code.

04 Pre-processor Statement

It is used to perform auxiliary function during macro expansion.

Ex AIF
AGO.

CELEBRO™
DATE
PAGE

Q) Positional Parameters

→ The specification of <parameter kind> is simply omitted

→ <parameter name> is written as &SAMPLE when SAMPLE is name of parameter.

* Find actual parameter specification that occupies the same ordinal position in list of actual parameter in macro call statement.

Ex.

```
MACRO
M1   &P1  &P2  &P3
- - -
- - -
- - -
MEND.
```

P1, P2 and P3 are Positional Parameters

Q) Default Parameter

It is standard assumption in absence of explicit specification by programmer.

When desired value is different from default value then desire value can be specified explicitly in macro call.

Ex:

```
MACRO
M1  &P1=A   &P2=B
-  -  -
-  -
MEND
```

&P1=A
&P2=B
} → Default Parameters

CELEBRO™
DATE
PAGE

Keyword Parameter

1. They are symbolic parameter that can be specified in any order when macro is called

2. Each keyword parameter will have an = (equal sign) as last character of parameter name

Ex

```
MACRO
&P1=, &P2=, &P3=.
- - -
- - -
MEND
```

&P1=, &P2= all keyword parameter

Q Give examples of nested macro calls

1.
```
MACRO
MAC1  &PAR1, &PAR2
MOVER REG, &PAR1

    MACRO
    MAC_Y  &PAR2  PEG=REG3
    ADD  REG, &PAR2

    MOVEM  REG, &PAR2

    MEND

PRINT &PAR1

MEND
```

2.

```
MACRO
MAC_Y  &PA2  REG=REG3
ADD    REG, &PAR2
MOVEM  REG, TPAR2
MEND

MACRO
MAC_X  &Par, &Par2
MOVER  REG1, Par1
MAC_Y  &PP2, REG=REG3
PRINT  &Par1
MEND
```

Q. Parameter Attributes

-> An attribute is written using syntax

<attribute name> <formal parameter spec>

-> It represents information about value of formal parameter, i.e. about the corresponding actual parameter. The type, length and size attributes have name T, L and S.

Example

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

3 AIF

AIF statement has syntax :-

AIF (< expression >) < sequencing symbol >

where < expression is an relational expression involving ordinary strings, [illegible] and [illegible] in terms of lower [illegible]

If the relational expression evaluates to the true, expansion time control is transferred to the statement containing < sequencing symbol > in label field

4 ANOP

Syntax

<sequencing symbol> ANOP.

-) Used to define sequencing symbol.

4 REPT

It should evaluate to a numerical value during macro expansion.

The statement between REPT and ENDM statement should be processed for expansion no. of times.

Eg: MACRO

Cont to

LCL
&M SET
REPT
DC
&M SETA
ENDM
MEND

c) Macro Name Table

It is full form of added abbreviated in MNT. It has entries for all macro defined in progm.

It also contains
→ No of positional parameter (# PP)
→ No. of keyword parameters (# KP)
→ No. of expansion time variable (# EV)

d) Pointer Name Table

PNTAB contains all parameter name used in macro.

It contains all keyword, default and positional parameter name.

e) EV name Table

EVN Table contains all expansion time variable name.

3) SS Name Table

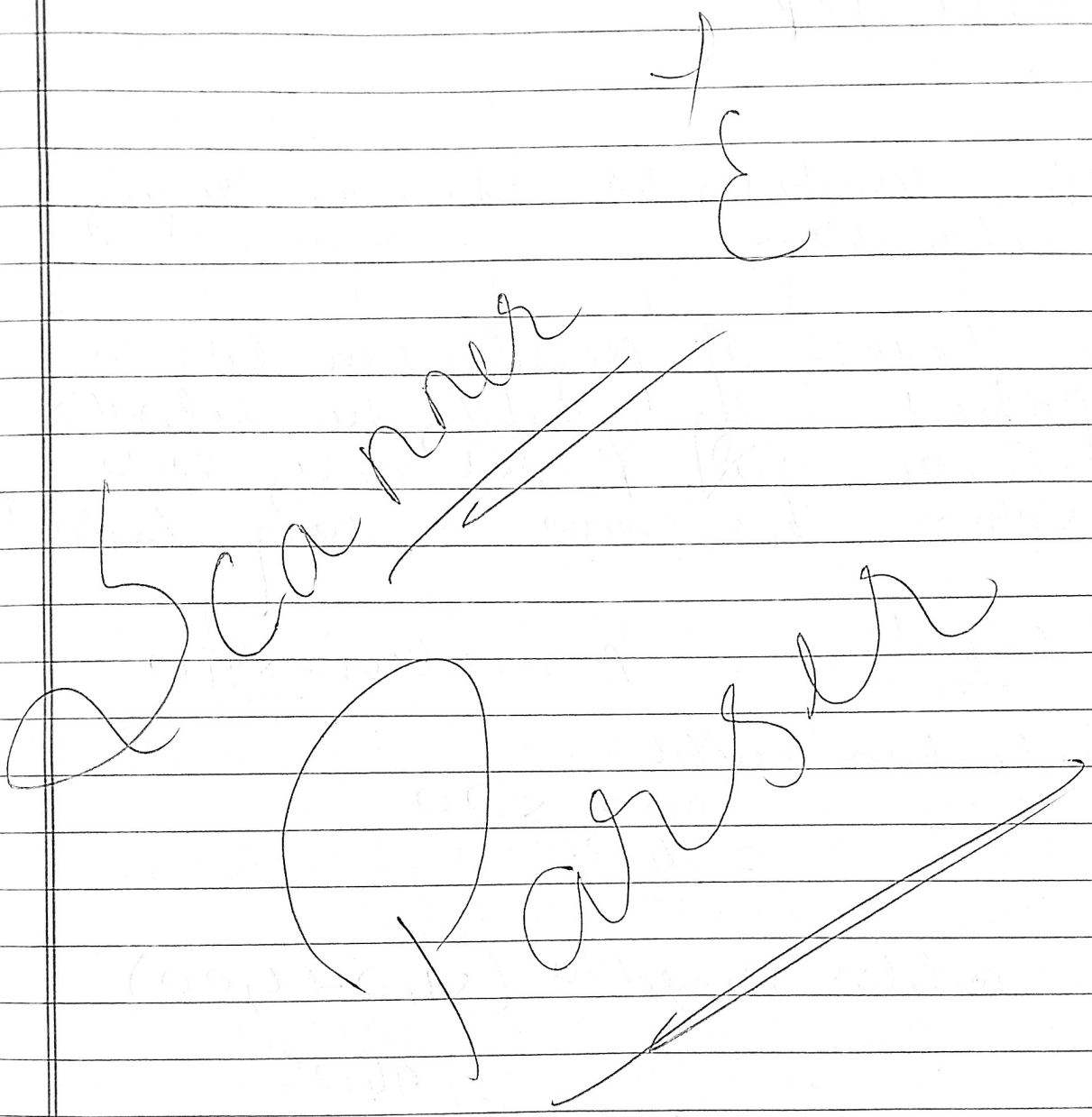
SSNTAB contains sequencing symbols

4) keyword Parameter Default Table

KPDTAB contains name of keyword parameter and its value.

5) Macro Definition Table

MDT these are constructed while processing the model statement and processor stores in macro body.



1) Diff. between scaning and parsing.

* Scaning

-> Scaning is process of recognizing the lexical components in a source string.

-> The lexical feature of language can be specified using Type-3 or regular grammars. This facilitates automatic construction of efficient recognizer for lexical fragment of language.

o Finite State

S- is finite set of state one which is initial state one or more of other one final state.

Σ - alphabet of source symbol.

T- is finite set of state transitions out of each.

CELEBRO™
DATE
PAGE

Deterministic Finite State Automata

It is an FSA such that $t_1 \in T$, T_1 indicates $\exists\, t_2, \in T$, t_2 (i.e. sigma)

In other words. It is finite state

Regular Expression

A regular expression is sequence of characters that define a search pattern, mainly use:- pattern matching.

i.e. => find and replace.
Example s.d, s.d etc...

* Parsing

Parsing is used the validity of a source string and to determine its syntactic structure.

For an invalid string we parse issues diagnostic messages depending the cause and nature of

error in string & for valid string.

→ It builds a parse-tree to reflect the sequence of derivation or reduction performed during parsing.

Top-down Parsing

It is parser which operates according to grammar which always to derive a string using a given string

Q. Construct DFA

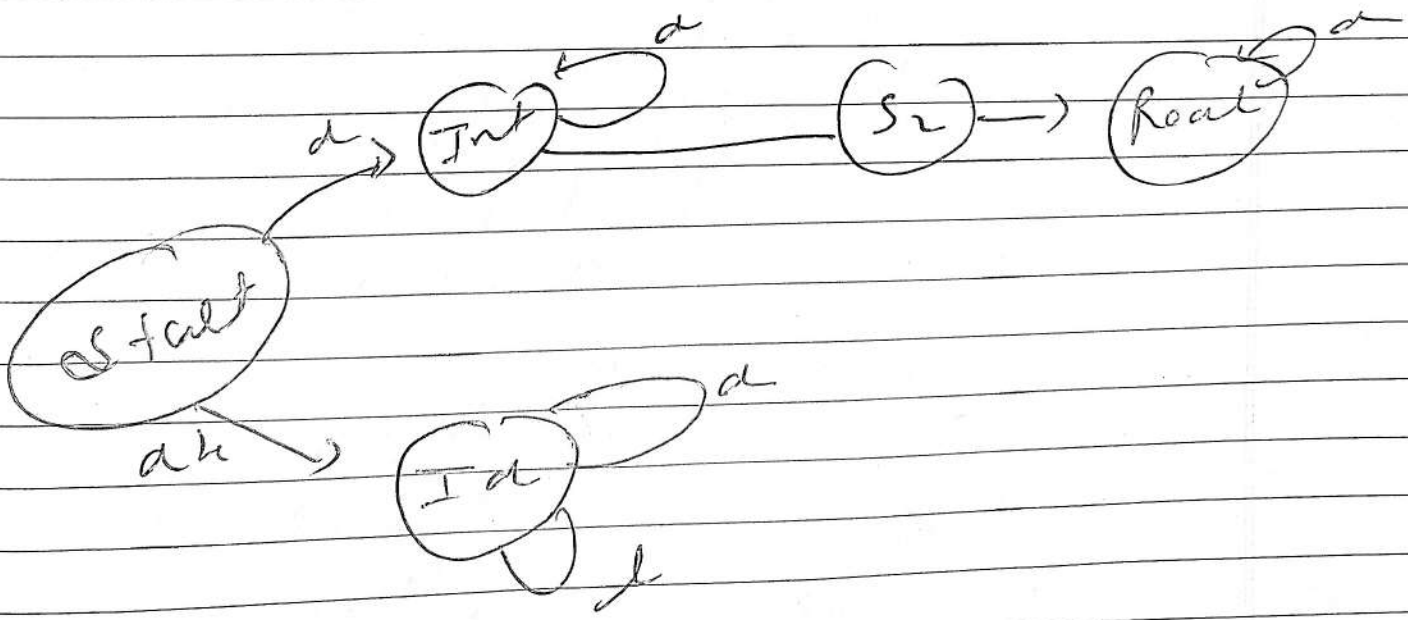
Valid →

12 asf

0 a 34

a 3.4

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



Top- Down Parser

- Top Down approach starts evaluating the parse tree from top and move down words for parsing other nodes.
- This parser uses the derivation of "left-most" derivation.
- To construct a string in this one, top down searches for production rule to be used to construct a string.
- As it uses left most derivation, thus it attempts to find left most derivation from the string.
- In this parsing technique we start parsing from top i.e. (start symbol of parse tree) to down (the leaf node of parse tree)

and this is done in top-down manner.

Bottom - Up Parser

- Bottom up approach starts evaluating the parse tree from lowest level of tree and move upwards for parsing the node.
- This parser uses the right most derivation.
- Bottom up parsing searches for production rule to be used to reduce a string to get a starting symbol of grammar.
- It attempts to reduce input string to first symbol of grammar.
- This technique we start parsing from bottom to up. i.e parsing leaf to start.

Top Down

Parse Example

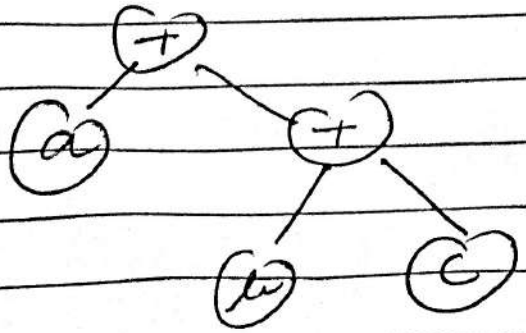
A + B + C

1. E'
2. T E'
3. V T' E'
4. <id> T' E'
5. <id> E'
6. <id> + T E'
7. <id> + T E'
8. <id> + V T' E'
9. <id> + <id> T' E'
10. <id> + <id> E'
11. <id> + <id> + T E'
12. <id> + <id> + T E'
13. <id> + <id> + V T' E'
14. <id> + <id> + <id> T' E'
15. <id> + <id> + <id> E'
16. <id> + <id> + <id>

Bottom - Up

a + b * C

⇒ a b C * +



Q. Define

(a) Grammar

The lexical and syntactic features of programming language are specified by its grammar. A grammar is set of rules which precisely specifies the syntax of prog. language.

(b) Terminal Symbol

A symbol in alphabet is called Terminal symbol.

(c) Non-Terminal symbol

A non-terminal symbol is name of syntax category of language.

e.g. → noun, verb, etc...

A non-terminal symbol is written as single, capital letter or as name enclosed between <.....>,

eg. A or <Noun>

Derivation Rule

- It is the replacement of non-terminal symbols in a accordance with given production rule.
- There are two types of derivation.
  - → Right most derivation
  - → Left most derivation.
- Let production P_1 of grammar be form

  $P_1 : A \rightarrow \alpha$ and let β be string such that $\beta \Rightarrow \gamma A \delta$ then replacement of A by α in string β constitutes a derivation according to production P_1.

  It yields string $\gamma\alpha\delta$.

Example

<Noun> =>

<Article>

<Animal phrase> =>

=> <Article> <Animal>

=> the Cow.

Production Rule

-> A production rule is also called rewriting rule.

-> It is rule of grammar

-> It has the form A non terminal symbol -> string of terminal and non-terminal symbol where the notation "->" stands for " is defined as".

Example :->

<Animal Phrase> ->

<Article> <Animal>

Linker Pass 1

- It is the first step of Program Relocation.
- "RELOCTAB" Data structure is used.
- "Translated Address" is only Relocator that has only 1 field.
- It stores the translated address of address sensitive instructions.
- Used to store address of instruction that require relocation.

Linker Pass 2

- It is step of Program Linking.
- LINKTAB Data structures is used.
- NTAB Data structures is used.

- It has 3 fields

i.e. Symbol -> To store symbolic name

- Type -> To store type of reference
- Translated address used to store translated address of symbol.

Q8 Top-Down Parsing with Back Tracking.

E = T + E/T
T = V * T/V
V = <id>

a + b * c -> <id> + <id> * id>

1. E
2. T + E
3. V * T + E
4. <id> * T + E

3. V + E
4. <id> + E
5. <id> + T + E
6. <id> + V * T + E
7. <id> + <id> * T + E
8. <id> + <id> * V * T + E
9. <id> + <id> * <id> * T * E

5. <id> + T
6. <id> + V * T
7. <id> + <id> * T
8. <id> + <id> * V * T
9. <id> + <id> * <id>

8. <id> + <id> * V
9. <id> + <id> * <id>

CELEBRO™
DATE
PAGE

without ~~Backtracking~~ Back Tracking

1. E
2. TE''
3. $VT''E$
4. $id\ T''E''$
5. $id + TE''$
6. $id * VT''E''$
7. $id\ id\ T''E''$
8. $id * id \in E''$
9. $id * id + E$
10. $id * id + TE''$
11. $id * id + VT''E''$
12. $id * id + id\ T''E''$
13. $id * id + id \in$
14. $id * id + id$

LL1 Parser

E		
TE'	<id>	E => TE'
VT'E'	<id>	T => VT'
<id> T'E'	*	V => <id>
<id> * VT'E'	*	T' => *VT'
<id> * <id> T'E'	<id>	V => <id>
<id> * <id> E'	+	T' = ε
<id> * <id> + TE'	+	E' => TE'
<id> * <id> + VT'E'	<id	T => VT'
<id> * <id> <id> T'E'	<id>	T' = ε
<id> * <id> <id> E'	+	E' => ε
id	-	—

Gramar

E => TE'
E' -> + TE' | ε
T -> VT'
T' -> * VT' | ε
V = <id>

<id> * id + id

CELEBRO™
DATE
PAGE

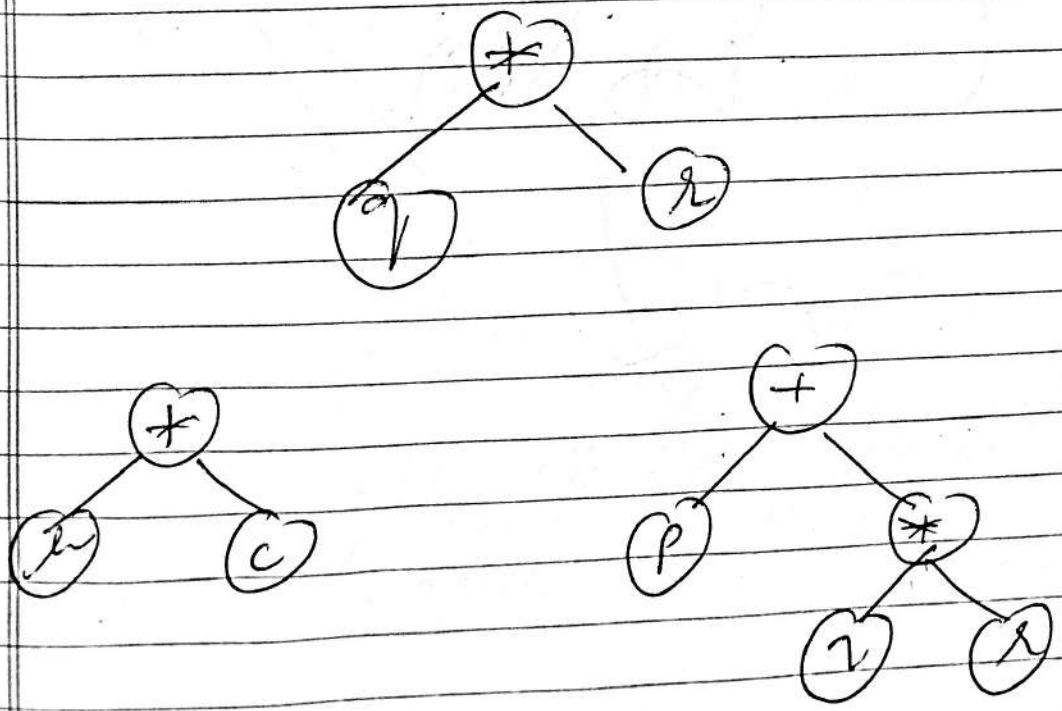
d) Recursive Descent Parser

-> A recursive descent parser is variant of top down parser without backtracking

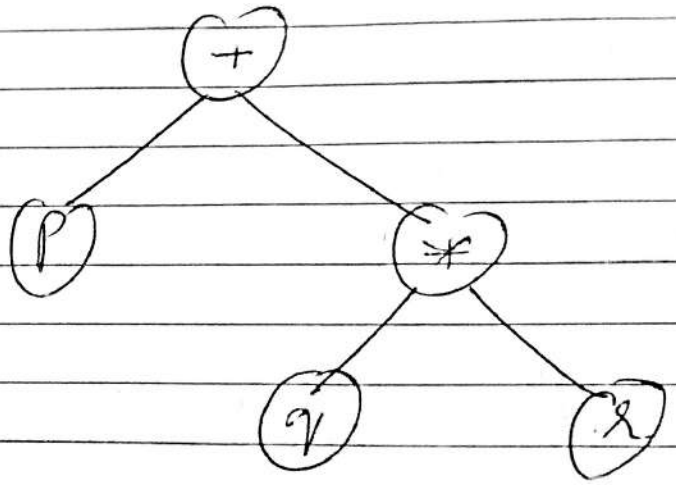
-> It uses a set of recursive procedures to perform parsing

-> Salient advantages of recursive descent parser are simplicity and generality

-> Can be implemented in any language.



Final Tree



2 Operator Precedence Parser

Precedence between operators a & b are appears in a sequential form and if there is null string is called operator precedence.

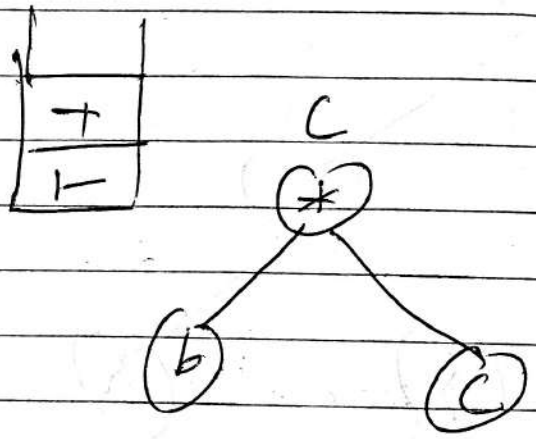
a + b * c

1. Stack

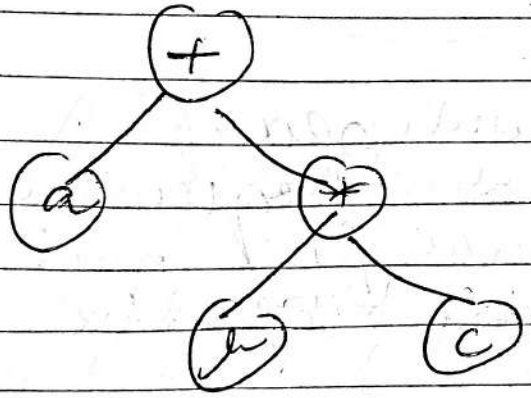
+
+
⊢

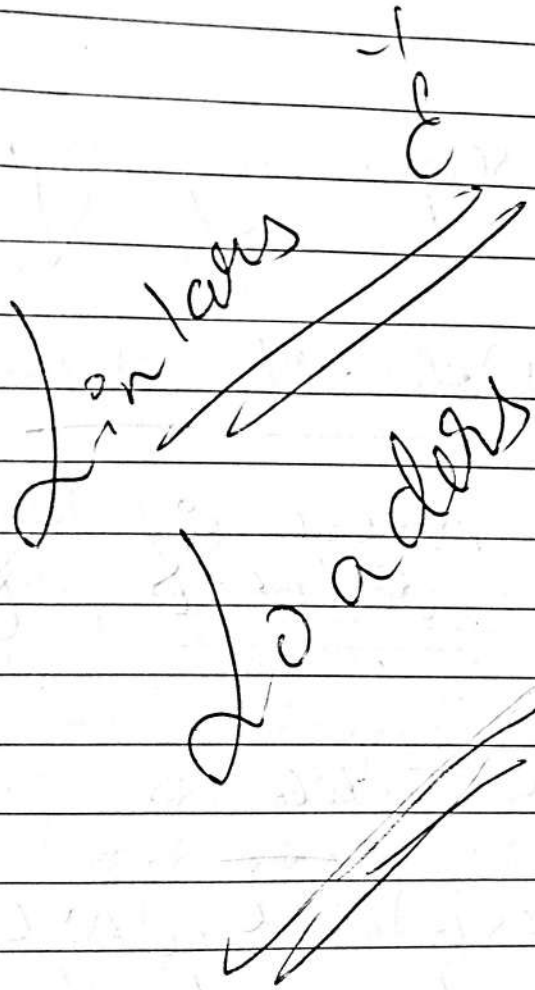
c
b
a

Operator precedence of "*" is high so use pop.



Last one will also be popped out





Linkers & Loaders

1. Translated Time address

-> Translation time addrs is used at a Translation time.

-> It uses the value of translated origin to perform memory allocation for symbols declared in P.

(ii) Load time address

It is used to at load time. This addrs is assigned by loader

(iii) Linked time address

It is used at link time. It is assigned by linker.

CELEBRO
DATE
PAGE

Program Relocation.

→ It is the process of modifying addresses used in address sensitive instructions of program such that program can execute correctly from designated area of memory.

→ If load origin ≠ linked origin
    relocation must be performed by loader

l. Let AA be set of absolute addresses used in instructions of program

→ AA includes that program uses its instructions and data to access memory words with specific addresses.

→ An address sensitive instruction

    a: ∈ AA

Q Relocation Factor

-> Let translation and linked origins of "P" be t-origin and l-origin respectively

-> Consider a symbol symb in P.

-> Relocation Factor =

$$l\text{origin} = t\text{origin} \quad \text{---} (7.1)$$

-> It can be
-> Positive
-> Negative or
-> Zero

-> tsymbol = t-origin + dsymb

-> P can be relocated by computing the relocation factor for and adding it to all relative linker address in every instruction.
: E IRP

c) Public Definition

A symbol pub- symb defined in program unit which may be referenced in other prog units

d. External Reference

A reference to ext- symbol which is not defined in prog unit containing reference

e. Basic Module Format

1. Header

→ It contains translated obj in

→ It contains size of

→ It contain execution start address of.

2. Program

→ This component contains the m/c lang. corresponding to P.

2. RELOCTAB

It contains info concerning the public definations and external refernces of P.

4. LINKTABS

Each LinkTab has 3 fields

→ Symbol

→ Type → PD/EXT

→ Translated Address

Q. Object module format of MSDOS linker

→ An Intel 8086 object module is sequence of object records, each object records.

→ There are 14 types of objects records contain the following basic category of information

i. Binary image
ii. External refrences
iii. Public definations
iv. Debugging information
v. MISC address information

Each name in object record is represented as

length (1 byte)	name

FIX UPP Record

rec ty	length	data	fix addr	frame	target	target datum	...	checksum

* MODEND Record

BAM	length	type	Start addr	check sum

Q10 Define Work Area

~~Define work area~~

- The linker uses an area of memory called the work area for constructing the binary program.
- It loads the relocatable language program stored in the program component of an object module into a work area and relocates the addresses sensitive instructions in it by processing items of

RELOCTAB.

→ The details of address computation would depend on whether we link loads and relocate one object module at a time or load all object modules that are to be linked together into core and then perform relocation

Q11) Give code for Fix up Record and for fix up data field code.

Fix up Codes

Loc Code	Means
0	low order byte
1	Offset
2	Segment pointer.
3	

Q12 Define IRR

-> IRR stands for ins intersection regarding relocation

-> It relocation of program can be performed by computing the relocation factor for P and adding it to evaluation the address in every instruction

Example A address = 540

Relocation Factor =
900 - 500
= 400

address changed to (540 + 400)
= 940

This achieves the relocation.

PAGE NO.
DATE :

Assignment - 1

Computer

PAGE NO.
DATE :

1. Diff between Static and Dynamic Memory Allocation.

Static

Ans

→ In this memory is allocated to a variable before execution of program begin.

→ No memory allocation and deallocation is done

→ Variable exists permanently allocated.

→ Eg: FORTRAN.

Dynaic

→ The memory binding are established and destroyed during execution of program

→ Memory is allocated & deallocate in

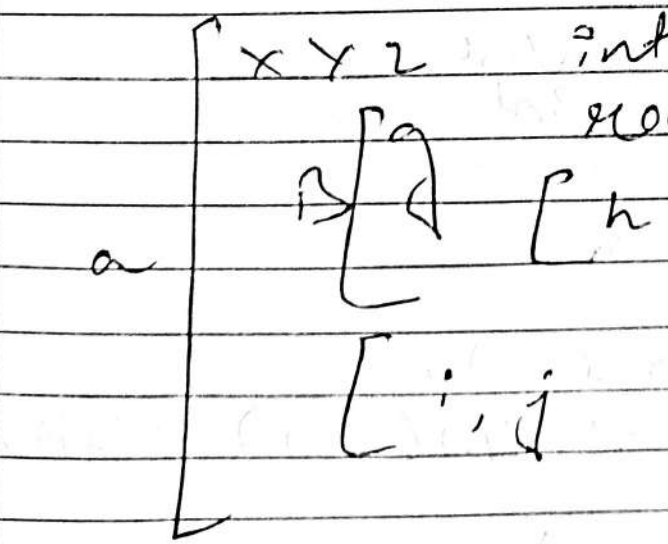
→ Eg PL/1, Pascal, Ada, etc.

Q2 Explain Scope Rule

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

→ It defines which of these bindings is effective at specific pt in prog.

Example



PAGE NO.
DATE :

Q Explain memory allocation in Recursion.

→ Recursion procedure or function are characterized by fact that many invocations of procedure coexist during the execution of program.

→ Copy of local variable of procedure must be allocated for each invocation.

Sample

```
var
    a  b      integer
    fib(n)    int
var @
    x : int
begin:
    if n > 1 then
        x = fib(n-1) + fib(n-2);
    else
        x = 1.

    Return(x)
end fib.
```

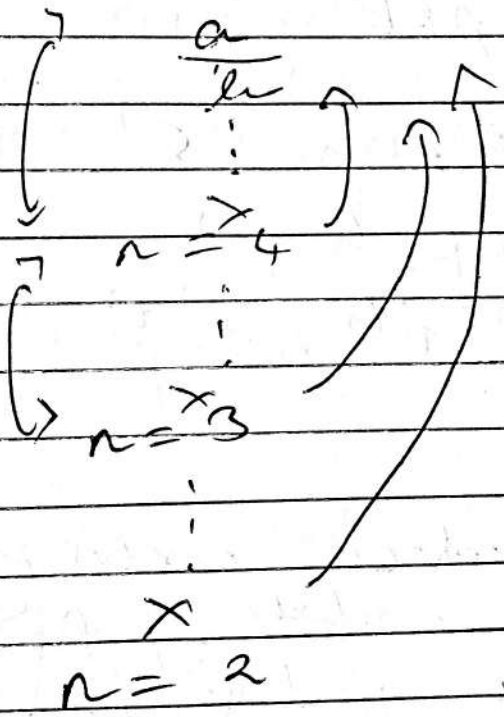
Teacher's Signature.....

Begin

Fi B(4)

End.

Pictorial Representation



PAGE NO.
DATE :

MOVER AREG, I
COMP AREG, = '5'

BC GT, ERROR, RTN, Errr, ;f;>5

COMP AREG = '1'.
BC LT, Error RTN Errr ;+ ;<1

MOVER AREG, 1
COMP AREG, = '10'.

MULT AREG, '5' Err. if ; >10
ADD AREG, 1
MULT AREG, = '2'

The generated code of algo to
check the contents of reg. to
check the validity of
subscription and compute the
address of an array element

Q. Define

Ans Attributes

10 Operand Descriptor

It has following fields

a. Attributes

b. Addressability (2 subfields)
↳ Addressability code
→ Address.

The operand descriptor is used for any operand participating in an expression

Eg ⇒ MOVER AREA A

MULT AREA B.

1	(int, 1)	M, addr(a)
2	(int, 1)	M, addr(b)
3	(int, 1)	R, addr(ARG)

2. Register Description –

It has 2 fields

→ status
→ Operand Descriptor

It all stored in array called Register descriptor.

→ One register discription exist for each register

PAGE NO.
DATE :

Assembly lang. Code.

Q. Steps for Assembly lang. Code

→ Expression $x + y * p + q$

Steps

1. $\langle id \rangle \; x \rightarrow F^1$
2. $F^1 \rightarrow T^1$
3. $\langle id \rangle \; y \rightarrow F^2$
4. $T^1 + F^2 \rightarrow T^3$
8. $T^3 \rightarrow E^3$
6. $\langle id \rangle \; p \rightarrow F^5$
7. $E^4 \rightarrow T^5$
8. $\langle id \rangle \; q \; !=5$

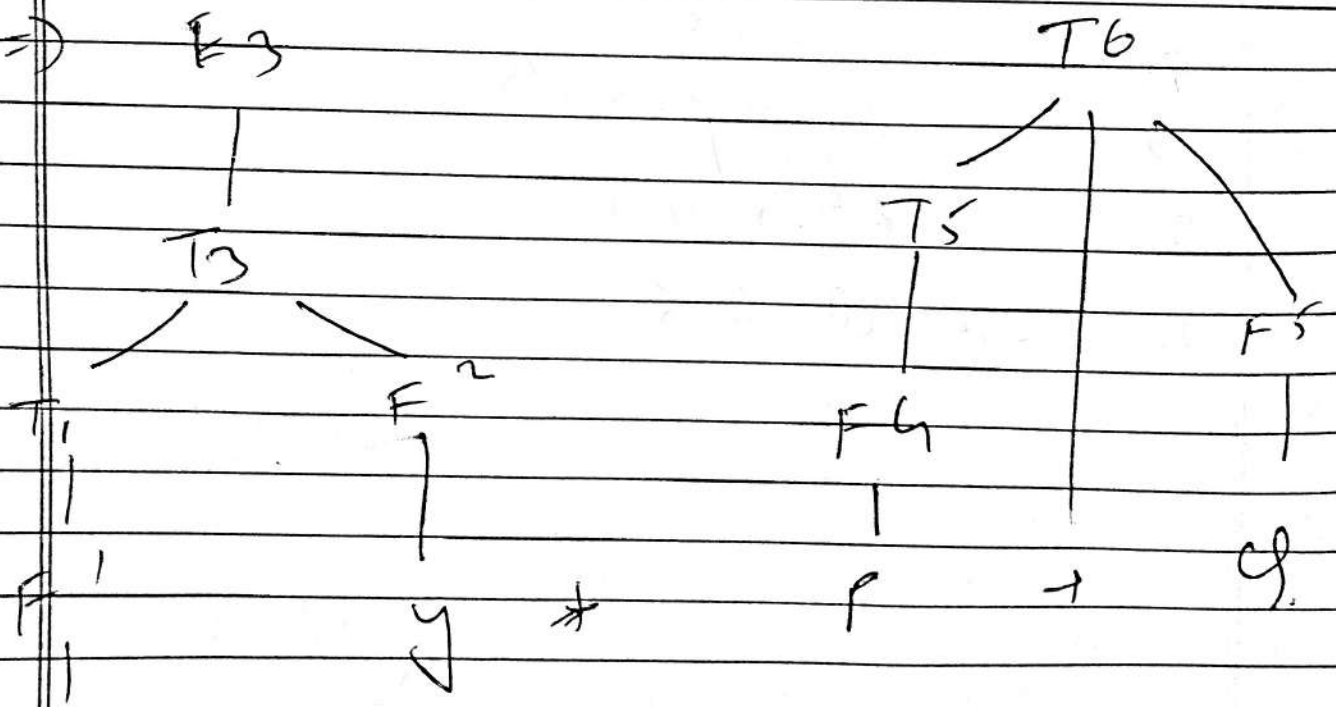
Teacher's Signature.....

DATE :

Operand descriptor

1. (int, 1) M, addr(x)
2. (int, 1) M, addr(y)
3. (int, 1) M, addr(temp[i])
4. (int, 1) M, addr(p)
5. (int, 1) M, addr(q)
6. (int, 1) R, addr(AREG)

=>



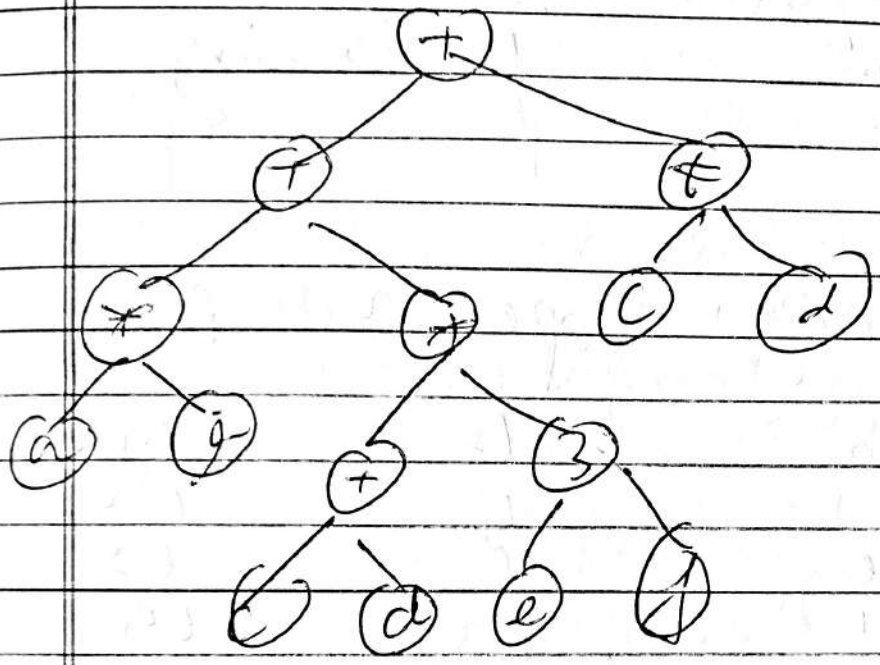
=> Register descriptor

exc	6

Teacher's Signature.....

PAGE NO.
DATE :

x + y + p * q * (r+s) + p * q.



```
MOVER    AREG, x
MULT     AREG, y
MOVEM    AREG, TEMP1
MOVER    AREG, p
MULT     AREG, q
MOVEM    AREG, TEMP2
MOVER    AREG, r
MOVER    AREG, p
MULT     AREG, q
ADD      AREG, TEMP3
```

Teacher's Signature.....

PAGE NO.
DATE :

Q Quadruples

It represents an elements evaluation in following format.

Sr. No.	Operator	Operand 1	Operand 2	Result no
1	*	b	c	t1
2	+	t1	d	t2
3	↑	e	f	t3
4	*	d	t3	t4
5	+	f	t4	t5

Q List down steps taken by compiler at function and procedure call.

1. Actual parameters are accessible in called Function.

2. The called Function is able to produce side effects according to rules of PL.

Teacher's Signature.....

c) Control is transferred to is returned from calling progr.

d) The function of execution of calling progrs are managed by function calls.

Compiler used a set of factors to implement functions

1. Parameter list
2. Calling conventions
3. Save Area.

Different Function Calls

i) Call by value

In this, value of actual parameters are passed to calling function.

In this value, changed in formal parameter does not affect the value of actual parameter.

Teacher's Signature.....

(ii) Call by value → Result

In this mechanism, the value of formal parameter is copied & back into actual parameters

→ This indicates the simplicity of call by value but increases overhead.

(ii) Call by reference

→ The address of actual parameter is passed to called function.

→ Parameter list is thus a list of address of actual parameter at every access of formal parameter in function.

Thus, the corresponding actual parameter is obtained.

PAGE NO.
DATE :

(iii) Call by name

The same mechanism of call by reference is followed in call by name.

→ The theoritical difference is every occurance of formal parameter in body of called function is replace by name of corresponding actual parameter.

→ Here, change in formal parameter affect the value of actual parameter.

D E P A R T M E N T O F C O M P U T E R S C I E N C E
R O L L W A L A C O M P U T E R C E N T R E
G U J A R A T U N I V E R S I T Y
M.C.A. – III

R O L L NO : 30
N A M E : Ajinkya Rathod
S U B J E C T : System Software

NO.	TITLE	PAGE NO.	DATE	SIGN
1	**Assembler Pass 1**		**10/12/2020**	
2	**Assembler Pass 2**		**10/12/2020**	
3	**Macro Pass 1**		**10/12/2020**	
4	**Macro pass 2**		**10/12/2020**	
5	**Scanner**		**10/12/2020**	
6	**Parser Without Backtracking**		**10/12/2020**	
7	**LL1 Parser**		**10/12/2020**	
8	**Recursive Decent Parser**		**10/12/2020**	
9	**Operator Precedence Parser**		**10/12/2020**	

SS Practical Programs

Q. Assember Pass 1 & 2

```
import java.util.*;
import java.io.*;

class AssemblerTable {
    private List<String[]> list;

    public AssemblerTable() {
        list = new ArrayList<>();
    }

    public int indexOf(String data, int start, int end) {
        data = data.toUpperCase();

        for (int i = start; i < end; i++) {
            String[] next = list.get(i);
            if (next[0].equals(data))
                return i;
        }

        return -1;
    }

    public int insert(String data, String address) {
        String[] element = { data, address };
        list.add(element);
        return (list.size() - 1);
    }
```

```
    public void modifyAddress(int index, String address)
{
        String[] tableData = list.get(index);
        String[] updatedData =
{ tableData[0].toUpperCase(), address };
        list.set(index, updatedData);
    }

    public boolean isEmpty() {
        return (list.size() == 0);
    }

    public String getAddress(int index) {
        return list.get(index)[1];
    }

    public void print() {
        if (isEmpty())
            System.out.println("Assembly Table is empty!");
        else {
            int i = 0;
            System.out.println("Index\tData\tAddress");
            for (String[] item : this.list)
                System.out.println(i++ + "\t" + item[0] + "\t"
+ item[1]);
        }
    }
}

class PoolTable {
    private List<Integer> list;

    public PoolTable() {
        list = new ArrayList<>();
    }

    public int indexOf(Integer data) {
```

```
        int i = 0;

        for (Integer item : this.list) {
            if (item.equals(data))
                return i;
            i++;
        }

        return -1;
    }

    public int insert(Integer data) {
        list.add(data);
        return (list.size() - 1);
    }

    public void print() {
        int i = 0;
        System.out.println("Index\tPool");

        for (Integer item : this.list)
            System.out.println(i++ + "\t#" + item);
    }

    public int get(int index) {
        return list.get(index);
    }
}

class OpTable {
    private static String[][] codesArray = { { "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" }, };

   static List<String[]> codes =
Arrays.asList(codesArray);

   public static String getType(String statement) {
      statement = statement.toUpperCase();

      for (String[] code : codes) {
         String st = code[1];
         if (st.equals(statement)) {
            return code[2];
         }
      }

      return null;
   }

   public static String getOpCode(String statement) {
      statement = statement.toUpperCase();

      for (String[] code : codes) {
         if (code[1].equals(statement))
            return code[0];
      }
      return null;
   }
}
```

```
public class Assembler {
    private AssemblerTable symbolTable;
    private AssemblerTable literalTable;
    private PoolTable poolTable;
    private List<String> code;
    private List<String> ic;

    int locationCounter, literalTablePtr, symbolTablePtr,
poolTablePtr;

    public Assembler(String filename) throws
IOException {
        symbolTable = new AssemblerTable();
        literalTable = new AssemblerTable();
        poolTable = new PoolTable();

        locationCounter = literalTablePtr =
symbolTablePtr = poolTablePtr = 0;
        poolTable.insert(1);
        ic = new ArrayList<String>();

        loadCode(filename);
    }

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

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

        if (input != null)
            input.close();
```

```
    }

    public void showCode() {
        this.code.forEach(System.out::println);
    }

    public void print() {
        System.out.println("----- SYMBOL TABLE -----");
        symbolTable.print();

        System.out.println("----- LITERAL TABLE -----");
        literalTable.print();

        System.out.println("----- POOL TABLE -----");
        poolTable.print();
    }

    private List<String> tokenize(String statement,
String delimiters) {
        StringTokenizer st = new
StringTokenizer(statement, delimiters);
        List<String> tokenizedStatements = new
ArrayList<String>();

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

        return tokenizedStatements;
    }

    private String getStatementType(List<String>
statements) {
        Iterator<String> itr = statements.iterator();

        while (itr.hasNext()) {
```

```
            String statement = itr.next();
            String type = OpTable.getType(statement);
            if (type != null)
                return type;
        }
        return "";
    }

    private boolean isLabel(String token) {
        return (!isLiteral(token) &&
OpTable.getOpCode(token) == null);
    }

    private boolean isLiteral(String token) {
        return token.charAt(0) == '=' || token.charAt(0)
== '\'';
    }

    private String parseLiteral(String token) {
        String literal = "";
        int i = token.charAt(0) == '=' ? 2 : 1;
        for (; i < token.length() - 1; i++)
            literal = literal + token.charAt(i);
        return literal;
    }

    private String parseOperand(String operand) {
        char firstChar = operand.charAt(0);

        if (firstChar >= '0' && firstChar <= '9')
            return operand;
        else {
            int index = symbolTable.indexOf(operand, 0,
symbolTablePtr);
            String parsedOperand =
symbolTable.getAddress(index);
            return parsedOperand;
```

```
        }
    }

    private String getIC(String classCode, String
indexInClass, String midValue, String symbolType,
String symbolCode) {
        String ic = String.format("(%s,%s)", classCode,
indexInClass);

        if (midValue != null)
            ic = ic + String.format("(%s)", midValue);

        if (symbolType != null && symbolCode != null)
            ic = ic + String.format("(%s,%s)", symbolType,
symbolCode);

        return ic;
    }

    public void printIC() {
        System.out.println("\n----- IC -----");
        this.ic.forEach(System.out::println);
    }

    public void getIntermediateCode() {
        Iterator<String> codeReader = code.iterator();

        while (codeReader.hasNext()) {
            String currentStatement = codeReader.next();
            List<String> tokens =
tokenize(currentStatement, ", \t");

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

            String statementType =
getStatementType(tokens);
```

```
            boolean hasLabel = isLabel(tokens.get(0));

            if (statementType.equals("AD")) {
                String token = tokens.get(0).toUpperCase();

                if (token.equals("START") ||
token.equals("ORIGIN")) {
                    String operand =
parseOperand(tokens.get(1));
                    locationCounter =
Integer.parseInt(operand);

                    String code = OpTable.getOpCode(token);
                    String ic = getIC("AD", code, null, "C", ("" 
+ locationCounter));
                    this.ic.add(ic);

                } else if (token.equals("LTORG") ||
token.equals("END")) {
                    int pptr = poolTable.get(poolTablePtr) - 1;

                    String adStatement =
tokens.get(0).toUpperCase();
                    String code =
OpTable.getOpCode(adStatement);

                    for (int i = pptr; i < literalTablePtr; i++) {
                        literalTable.modifyAddress(i, ("" +
locationCounter));
                        String ic = getIC("AD", code, null, "C",
("" + locationCounter));
                        locationCounter++;
                        this.ic.add(ic);
                    }
                    poolTablePtr++;
                    poolTable.insert(literalTablePtr + 1);
```

```
            } else if
(tokens.get(1).toUpperCase().equals("EQU")) {

                int indexToChange =
symbolTable.indexOf(tokens.get(0), 0, symbolTablePtr);
                int indexOfAddress =
symbolTable.indexOf(tokens.get(2), 0, symbolTablePtr);
                String addressToAssign =
symbolTable.getAddress(indexOfAddress);

symbolTable.modifyAddress(indexToChange,
addressToAssign);

                String code = OpTable.getOpCode("EQU");
                String ic = getIC("AD", code, null, "C", "" +
indexOfAddress);
                this.ic.add(ic);
            }

        } else if (statementType.equals("DL")) {
            String symbol = tokens.get(0);
            int index = symbolTable.indexOf(symbol, 0,
symbolTablePtr);
            symbolTable.modifyAddress(index, ("" +
locationCounter));

            String code =
OpTable.getOpCode(tokens.get(1));
            String value = tokens.get(2);

            if (tokens.get(1).toUpperCase().equals("DS"))
{
                int size = Integer.parseInt(value);
                locationCounter += size;
            } else {
                if (isLiteral(value))
                    value = parseLiteral(value);
```

```
                locationCounter++;
            }

            String ic = getIC("DL", code, null, "C", value);
            this.ic.add(ic);
        } else if (statementType.equals("IS")) {
            if
(tokens.get(0).toUpperCase().equals("STOP")) {
                String ic = getIC("IS",
OpTable.getOpCode("STOP"), null, null, null);
                this.ic.add(ic);
                continue;
            }

            String code = null;
            String midToken = null;
            String symbolType = null, symbolCode = null;

            if (hasLabel) {
                symbolTable.insert(tokens.get(0), ("" +
locationCounter));
                symbolTablePtr++;
                code = OpTable.getOpCode(tokens.get(1));
                if (tokens.size() > 3)
                    midToken =
OpTable.getOpCode(tokens.get(2));
            } else {
                code = OpTable.getOpCode(tokens.get(0));
                if (tokens.size() > 2)
                    midToken =
OpTable.getOpCode(tokens.get(1));
            }

            int size = tokens.size();
            String operand = tokens.get(size - 1);

            if (isLiteral(operand)) {
```

```
                int pptr = poolTable.get(poolTablePtr) - 1;
                int index =
literalTable.indexOf(parseLiteral(operand), pptr,
literalTablePtr);
                if (index == -1) {
                    literalTable.insert(parseLiteral(operand),
"");
                    index = literalTablePtr++;
                }
                symbolType = "L";
                symbolCode = ("" + index);
            } else {
                int index = symbolTable.indexOf(operand,
0, symbolTablePtr);
                if (index == -1) {
                    symbolTable.insert(operand, "");
                    index = symbolTablePtr++;
                }
                symbolType = "S";
                symbolCode = ("" + index);
            }
            String ic = getIC("IS", code, midToken,
symbolType, symbolCode);
            this.ic.add(ic);
            locationCounter++;
        }
    }
}

public void getMachineCode() {
    System.out.println("\n----- Machine Code -----");
    for (String s : this.ic) {
        List<String> tokens = this.tokenize(s, "(),");
        if (tokens.get(0).equals("AD"))
            continue;

        if (tokens.get(0).equals("IS")) {
```

```
                boolean hasMid = tokens.size() == 5;

                String firstToken = tokens.get(1);
                String mid = hasMid ? tokens.get(2) : "0";
                String table = hasMid ? tokens.get(3) :
tokens.get(2);
                int index =
Integer.parseInt(tokens.get(tokens.size() - 1));
                String lastValue = "";

                if (table.equals("S"))
                    lastValue =
symbolTable.getAddress(index);
                else
                    lastValue = literalTable.getAddress(index);

                System.out.println(firstToken + " " + mid + "
" + lastValue);
            }
        }
    }

    public static void main(String[] args) throws
IOException {
        System.out.println("Starting Compilation...\n");

        Assembler assembler = new
Assembler("input.txt");
        assembler.getIntermediateCode();
        assembler.print();
        assembler.printIC();
        assembler.getMachineCode();

        System.out.println("\nEnding Compilation...");
    }
}
```

OUTPUT

```
javac -d bin/ Assembler.java && java -cp bin/
Assembler
Starting Compilation...

----- SYMBOL TABLE -----
Index   Data    Address
0       A       207
1       B       208
2       SUM     209
----- LITERAL TABLE -----
Index   Data    Address
0       2       210
----- POOL TABLE -----
Index   Pool
0       #1
1       #2

----- IC -----
(AD,01)(C,200)
(IS,09)(S,0)
(IS,09)(S,1)
(IS,04)(1)(S,0)
(IS,01)(1)(S,1)
(IS,03)(1)(L,0)
(IS,05)(1)(S,2)
(IS,10)(S,2)
(DL,02)(C,1)
(DL,02)(C,1)
(DL,02)(C,1)
(AD,02)(C,210)

----- Machine Code -----
09 0 207
09 0 208
04 1 207
```

```
01 1 208
03 1 210
05 1 209
10 0 209

Ending Compilation...
```

Q. Macro Pass 1 & 2

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

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

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

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" + md.ev + "\t" + md.mdtp + "\t" +
md.kpdtp
                    + "\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" + 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" + 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" + sstab.get(i)[0] + "\
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" + kpdtab.get(i)[0] +
"\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" + mdt.get(i));
        }

System.out.println("----------------------------------------------
-------");
    }

    public void analyze() {

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

        // MACRO PROTOTYPE PROCESSING STARTS HERE
        String prototype = code.get(1);
        tokenized = tokenize(prototype);

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

        for (int i = 1; i < tokenized.size(); i++) {
            String parameter = tokenized.get(i);
            if (getParameterType(parameter).equals("PP"))
{
                pntab.add(parameter.substring(1));
                pntab_ptr++;
                md.pp++;
            } else {
```

```
                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.mdtp = mdt_ptr;
        md.ev = 0;
        md.sstp = sstab_ptr;

        // MACRO PROTOTYPE PROCESSING ENDS HERE

        // MACRO MODEL STATEMENTS PROCESSING
STARTS HERE

        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);
            }

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

                String lineIC = getLineIC(currentLine);
                mdt.add(lineIC);
                mdt_ptr++;
            } else if (tokenized.size() > 1 &&
tokenized.get(1).toUpperCase().equals("SET")) {
                String lineIC = getLineIC(currentLine);
                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);
            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);
            mdt.add(lineIC);
            mdt_ptr++;
        }
    }

    // MACRO MODEL STATEMENTS PROCESSING ENDS
HERE

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

}

public class MacroDemo {
  public static void main(String[] args) throws
IOException {
    List<String> code = new ArrayList<>();

    code.add("\tMACRO");
    code.add("\tCLEARMEM &X, &N, &REG=AREG");
    code.add("\tLCL &M");
    code.add("\t&M SET 0");
```

```
        code.add("\tMOVER &REG, ='0'");
        code.add(".MORE MOVEM &REG, &X + &M");
        code.add("\t&M SET &M+1");
        code.add("\tAIF (&M NE &N) .MORE");
        code.add("\tMEND");
        code.add("\tMMEND");

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

        PreProcessor pr = new PreProcessor(code);

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

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

OUTPUT

```
$ javac MacroDemo.java && java MacroDemo
Starting Preprocessing...
        MACRO
        CLEARMEM &X, &N, &REG=AREG
        LCL &M
        &M SET 0
        MOVER &REG, ='0'
.MORE MOVEM &REG, &X + &M
        &M SET &M+1
        AIF (&M NE &N) .MORE
        MEND
        MMEND

------- 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...
```

Q. Scanner / DFA

MyScanner.java

```
package scanner;

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

// class State to store the state and its next states
class State {
    char symbol; // symbol for the current state
    List<Character> nextStates; // list of symbols which
are the next states

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

    // returns true if current state has a next state with
given value
    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);
    }
}

public class MyScanner {
    State start;
    // State[] states; // changes from array to List
    List<State> states;

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

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

    // Initializes the start and other states - default
values
    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);
    }

    // Initializes the start and other states from list of
valid strings
    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);
            }
        }

    }

    // returns the State object with a given value
    private State getState(char value) {
        return this.states.stream().filter(state ->
state.symbol == value).findAny().orElse(null);
    }

    // create the DFA/state table
    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');
    }

    // display all the states in DFA with their next states
    private void displayStates() {
        System.out.println(start);
        this.states.forEach(System.out::println);
    }

    // checks the expression and returns true/false
accordingly
    public boolean check(String expression) {
        // initializes the current state from start state
        State current = start;

        // for each character in expression string
        for (int i = 0; i < expression.length(); i++) {
            char symbol = expression.charAt(i);

            // if the current state has the next state with the
current symbol
            // then the expression is going fine...
            if (current.hasNextState(symbol)) {
                current = this.getState(symbol);
            }

            // if current state doesnt have next state with
given value
            // expression is not valid
            else
                return false;
        }
```

```
        char currentSymbol = current.symbol;

        for (char ch : current.nextStates) {
            if (ch != currentSymbol)
                return false;
        }

        return true;
    }
}
```

ScannerDemo.java

```
package test;

import java.util.Scanner;

import scanner.MyScanner;

public class ScannerDemo {
    public static void main(String[] args) {
        String[] valids = { "aaabbbcccddd", "aaabcddd",
"abcd", "aaaaabbbbdddd", "abd" };
        MyScanner sc = new MyScanner(valids);
        Scanner s = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String str = s.nextLine();
        boolean check = sc.check(str);
        System.out.println(str + " is " + (check ? " valid" :
" not valid."));
        System.out.println();
    }
}
```

OUTPUT

```
$ javac -d ../bin scanner/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.ScannerDemo
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
Enter a string:
abd
abd is  valid

$ javac -d ../bin scanner/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.ScannerDemo
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
Enter a string:
abc
abc is  not valid.
```

Q. Top Down Parser without Backtracking

```
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;
    }
}

import java.util.Scanner;

public class TopDownTest {
    public static void main(String[] args) {
        System.out.println("TopDownWithoutBackTrack");
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String str = sc.nextLine();
        TopDownWithoutBackTrack a = new
TopDownWithoutBackTrack();
        String parsed = a.parse(str);
        System.out.println("Parsed: " + parsed);
    }
}
```

OUTPUT

```
$ javac -d ../bin parser/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.TopDownTest
TopDownWithoutBackTrack
Enter a string:
a + b * c * d + e
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>
```

Q. LL1 Parser

```
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;
    }
}
```

```
import java.util.Scanner;

public class LL1Test {
    public static void main(String[] args) {
        System.out.println("LL1Parser");
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String str = sc.nextLine();
        LL1Parser a = new LL1Parser();

        String parsed = a.parse(str);

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

OUTPUT

```
$ javac -d ../bin parser/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.LL1Test
LL1Parser
Enter a string:
a + b * c * d + e
Steps:
 1. E
 2. TE'
 3. VT'E'
 4. <id>T'E'
 5. <id>E'
 6. <id>+TE'
 7. <id>+TE'
 8. <id>+VT'E'
 9. <id>+<id>T'E'
10. <id>+<id>*VT'E'
11. <id>+<id>*VT'E'
12. <id>+<id>*<id>T'E'
13. <id>+<id>*<id>*VT'E'
```

14. <id>+<id>*<id>*VT'E'
15. <id>+<id>*<id>*<id>T'E'
16. <id>+<id>*<id>*<id>E'
17. <id>+<id>*<id>*<id>+TE'
18. <id>+<id>*<id>*<id>+TE'
19. <id>+<id>*<id>*<id>+VT'E'
20. <id>+<id>*<id>*<id>+<id>T'E'
21. <id>+<id>*<id>*<id>+<id>E'
22. <id>+<id>*<id>*<id>+<id>

Completed in 22 steps.
Parsed: <id>+<id>*<id>*<id>+<id>

Q. Recusrive Descent Parser

```
package parser;

public 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 preOrderTraversal() {
        System.out.print(this.expression);

        if (this.leftNode != null)
            leftNode.preOrderTraversal();

        if (this.rightNode != null)
            rightNode.preOrderTraversal();
    }

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

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

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

    }

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

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

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

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;
```

```
        }
    }
}

import java.util.Scanner;

public class RDTest {
    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) {
            System.out.print("Pre-Order:");
            rootNode.preOrderTraversal();
            System.out.println();

            System.out.print("In-Order:");
            rootNode.inOrderTraversal();
            System.out.println();

            System.out.print("Post-Order:");
            rootNode.postOrderTraversal();
            System.out.println();
        }
    }
}
```

OUTPUT

```
$ javac -d ../bin parser/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.RDTest
Enter the Expression: a+b*c*d+e
Pre-Order:++a**bcde
In-Order:a+b*c*d+e
Post-Order:abc*d*+e+
```

Q. Operator Precendence Parser

```
import java.util.Stack;

public 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 preOrder(TreeNode root) {
        if (root == null)
            return;

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

    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));
    }
}

import java.util.Scanner;

public class OPTest {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String str = sc.nextLine();

        OperatorPrecedenceParser a = new
OperatorPrecedenceParser();
        OperatorPrecedenceParser.TreeNode tree =
a.parse(str);

        System.out.println("Equation: " + str);

        System.out.print("InOrder Traversal: ");
        OperatorPrecedenceParser.inOrder(tree);

        System.out.print("\nPreOrder Traversal: ");
        OperatorPrecedenceParser.preOrder(tree);
```

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

OUTPUT

```
$ javac -d ../bin parser/*.java && javac -d ../bin
test/*.java && java -cp ../bin test.OPTest
Enter a string:
a+b*c*d+e
Equation: a+b*c*d+e
InOrder Traversal: a+b*c*d+e
PreOrder Traversal: ++a**bcde
PostOrder Traversal: abc*d*+e+
```