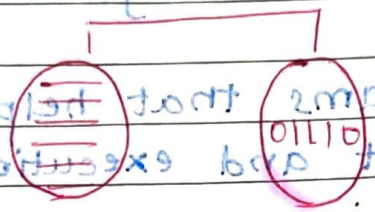
Introduction to SPCC

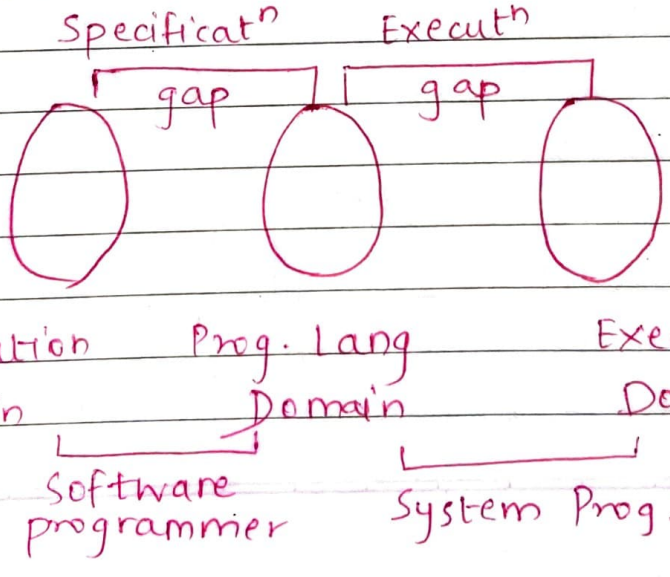
The software developer talks in terms related to application domain.
The system understands the terms related to execution domain.
The gap between the application domain and execution domain is called semantic gap.



Problems of semantic gap:

1. Large Development Time.
2. Large Dev. efforts.
3. Poor software quality.

The semantic gap was made manageable by the intro. of programming Language Domain.



Defn of Application Programs:

These are the programs used by the end-user for solving their problems.

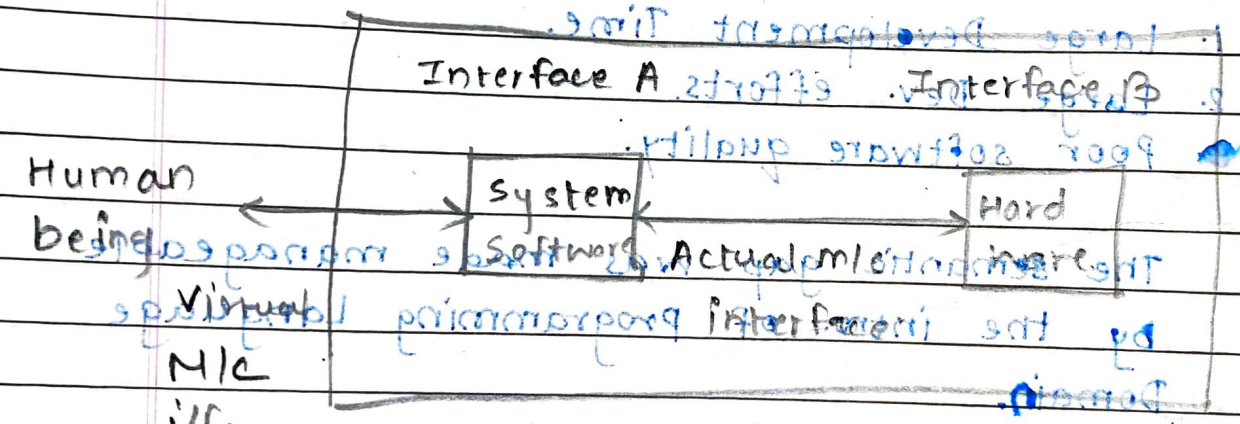
eg: Web browsers, media players, etc.

Defn of system software / Progs:

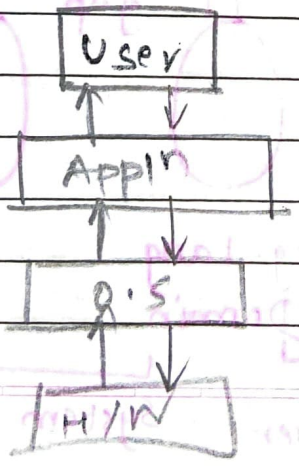
These are the programs that help in the effective development and execution of application programs.

egs: Assemblers, Macroprocessors, loaders, linkers, compilers, interpreters, editors, O.S.

It consists of variety of programs that support the operation of a comp.



Appln.



Syllabus:

1. Assemblers :- (2)

- Defn
- Types of ALP
- Features of ALP
- Statements in ALP
- Design of 2PA
- Design of 1PA

2. Macro Processors: (6)

- Defn
- Features of Macro
- Functions of M.P.
- Design of 2PMP
- Design of 1PMP
- Macro Assembler

3. Loaders And Linkers :- (3)

- Defn
- Function of loader
- Design of Absolute Loader
- Design of Dynamic Relocating Loader
- Design of MS-DOS Linker

4. Compilers

Defn

Structure of compilers

Lexical Analysis

Syntax

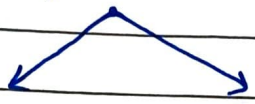
Semantic

IC G

Code Optimization

Code Generation

Parsing Techniques:



Top-Down P.T.

- Recursive Descent
- Predictive

Bottom Up P.T.

- Shift Reduce
- Operator Precedence
- Simple LR
- Canonical LR
- LALR

Ref. Books:

B1: J. Donovan (Yellow)

B2: Dhamdhere (Red)

B3: J. Ullman (Red/ Dragon Book)
purple

Assemblers

Defn of assembler:

Assembler is a lang. translator that takes as i/p assembly lang. program and generates it's m/c lang. equivalent alongwith the info. reqd by the loader.

m/c lang. equivalent + info. reqd by loader

Types of Assembler

1. Absolute / Non-relocatable Prog:

Allocation : Programmer
Loading : Loader

2. Relocatable Prog:

Allocation } Loader
Loading }

Absolute: The addr. at which the progs. need to be loaded in the memory for execution is fixed (cannot be changed).

Allocation is done by the programmer.

Loading is done by the loader.

Relocatable:

These programs can be loaded anywhere in the memory for execution. Allocation and binding could be done by the loader.

Features OF ALP:

1. Mneumonic Opcode Specification.

Instead of writing the binary opcodes, mneumonics can be specified.
It is the fxn. of assembler to replace each mneumonic opcode by it's corresponding binary opcode which is done using Machine Opcode Table.

2. Symbolic Operand Specification.

Instead of writing the addresses, symbolic operands can be specified.
It is the fxn of assembler to replace each symbol by it's address which is done using Symbol Table.

3. Storage Area Specification:

with the help of assembly lang. prog.

some portion in the memory can be kept aside for storage purpose

Statements in A.L.P.

1. Imperative statements:

These are the statements understood by the m/c and executed by the machine.

eg: Instructions.

2. Declarative statements:

These are the statements used for declaration purpose.

DC stmt: (Declare / Define Constant)

eg1:	D1	DC	F	'4'	← Data
eg2:	D2	DC	B	'7'	

DS stmt: (Declare / Define Storage)

eg1:	T1	DS	'10'	D	→ unit of collection
eg2:	T2	DS	'4'	H	

3. Assembler Directive statements:

These are the statements that direct the assembler to take necessary actions.

eg: START stmt:

eg1: PG1 START } Relocatable

eg2: PG2 START 0 }

eg3: PG3 START 4000 } Absolute

END stmt.

Design of an Assembler: (For IBM 360/370 processor)

1. Registers:

There are 16 GPRs (0-15),
4 Floating Point Reg. (FPR)
1 PSW (Program Status Word)

2. Memory:

The various units of memory available are as follows:

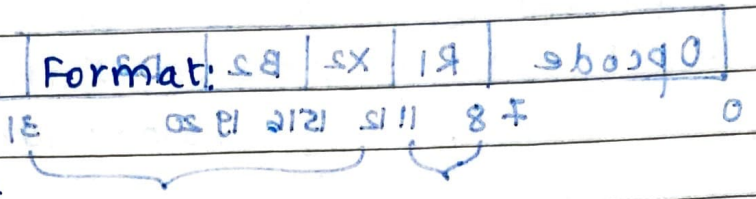
B - Byte — 8

H - Half Word — 16

F - Full Word — 32

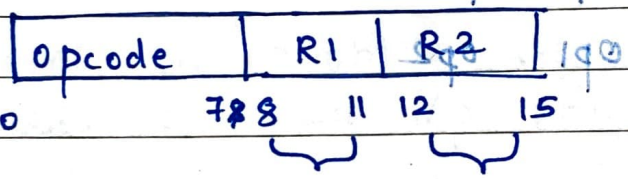
D - Double Word — 64

3. Instruction Format



1. RR format

In this format the 1st operand is in the register and the 2nd operand is also in the register.



eg1: AR 3,4

eg2: SR 4,4

op1 op2

2. Rx Format

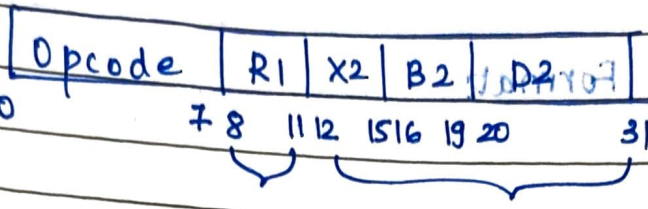
In this format the 1st operand is in the register and the 2nd operand is in the memory.

The address of the 2nd operand is:

= c(B2) + c(X2) + D2

Note: If indexing is not used then

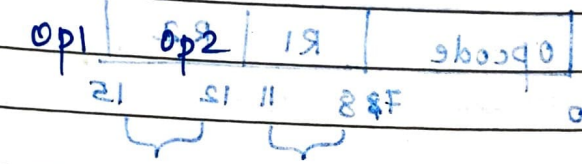
= c(B2) + D2 (X2 = 0)



Op1 Op2

eg1: A 1, D1

eg2: ST 1, T1



* USING statement:

It is an assembler directive stmt for making the base register available along with it's value.

eg: USING 100,12.

* DROP stmt:

... making B.R. unavailable

eg: DROP 12.

Forward Ref. Problem & it's solution:

1) The rules of assembly lang. program state that the symbol can be defined anywhere in the program.
Else, Hence, there may be some cases in which the ref. is made to the symbol prior to it's definition, and such a ref is called forward reference.

2) Due to forward reference assembler cannot assemble the instructions and such a problem is called forward ref. problem.

3) To solve the problem assembler will make 2 - Passes over the i/p program.

4) The purpose of Pass1 is to define the symbols and the literals encountered in the program.
The purpose of Pass 2 is to assemble the instructions and assemble the data.

E.Q.

Q.1. Using a sample assembly lang. program show the i/p & o/p of each pass of a 2 pass assembler. Clearly show the entries made in the databases. 10M.

```
                    LC
PGM  START  0
     USING  *,15    0
     L  1,FOUR      0   L  1,12(0,15)
     A  1,FIVE      4   A  1,16(0,15)
     ST 1,TEMP      8   ST 1,20(0,15)
FOUR DC F'4'        12  0100
FIVE DC F'5'        16  0101
TEMP DS F           20
     END            24
```

Pass I & Pass 2

Database Info.

MOT :- (I)

Mnemonic	Binary Opcode	Inst. length	Inst. format
L	X	✓	Rx ✓
A	✓	✓	Rx ✓
ST	...	...	Rx

POT: (D+A)

Pseudo Opcode	Address of the routine for processing pseudo opcode
DC	...
DS	
START	
END	
USING	

ST:

Symbol	Value	length	R/A
PG1	00	01	R
FOUR	12	04	R
FIVE	16	04	R
TEMP	20	04	R

LT:

Literals	Value	length	R/A

B.T:

	Availability indicator	contents of B.R.
1	'N'	
2	'N'	
:	:	...
14	'N'	
15	~~'N'~~ 'y'	00

Format of Database:

MOT (I) Machine Opcode Table.

Mnemonic Opcode	binary opcode	inst. length	inst. format

1) MOT is a fixed-length table i.e. assembler will make no entries in either of the passes.

2) In parse1 MOT is consulted to obtain instruction length (To update LC)

3) In parse 2 MOT is consulted to obtain
   i.) Binary Opcode (to replace the mnemonic Opcode).

R/A : Relative / Absolute.

ii.) Instruction length (to update LC)
iii.) Instruction format (to assemble the inst.)

POT :- (D + A)

Pseudo Opcode Table

pseudo opcode	Addr. of the routine for processing pseudo opcode

1) POT is also a fixed length table

2) In parse 1, POT is consulted for the processing of some pseudo opcodes like START, DC, DS, etc.

3) In parse 2, POT is consulted for the processing of some pseudo opcodes like USING, DROP, DC, etc.

Symbol Table

Symbol	Value	length	R/A

1) Symbol Table is used for keeping a track on the symbols that are defined in the program. (Symbol is defined whenever it appears in the label).

2) In Pass1, whenever a symbol is defined (entry for that) symbol is made in the Symbol Table.

3) In Pass2, symbol table is used for generating the address of the symbol.

Literal Table

Literals	Value	length	R/A

1) Literal Table is used for keeping track on the literals that are encountered in the program. (literals are always encountered in the operand)

2) In pass1 whenever a literal is encountered entry for that literal is made in the literal table.

3) In pass2 literal table is used for generating the address of the literals.

B.T. Base Table.

No.	Availability indicator	Contents of B.R.
0	N	
1	N	
2	N	
⋮	⋮	⋮
15	N	

(0 cannot be used as B.R)

1) B.T. is used for keeping a track on Base Registers info.

2) In parse1, Base Table is not required.

3) In parse 2, Base Table is required for the processing of USING & DROP statements

Q.2.

	LC	LC
PG2 START	0	0
USING *,BASE	0	0
L 2,D1	0 L 2,—	0 L 2,16(0,12)
A 2,D2	4 A 2,—	4 A 2,20(0,12)
A 2,D3	8 A 2,—	8 A 2,24(0,12)
ST 2,T1	12 ST 2,—	12 ST 2,28(0,12)
D1 DC F'7'	16 '7'	16 0111
D2 DC F'4'	20 '4'	20 0100
D3 DC F'8'	24 '8'	24 1000
BASE EQU 12	28	28
T1 DS 'I' F	28 —	28 —
END	32	32
	Pass 1	Pass 2

Database Info

MOT (I)

Mnemonic Opcode	binary opcode	inst. length	inst. format
		✓	
		✓	
		✓	

POT: D+A

Pseudo Opcode	Addr. of the routine for processing pseudo-opcode

Symbol

Symbol	value	length	R/A
PGMS	00	01	R
D1	16	04	R
D2	20	04	R
D3	24	04	R
BASE	12	01	R
T1	28	04	R.

Q.3.

Literal.

Literals	Value	length	R/A.
Availability indicator	Contents of B.R.		
1	'N'		
2	'N'		
:			
12	'Y'		
:			
15	'N'		

Mnemonic Opcode	Binary Opcode	Inst length	Inst Format
L			
A			
ST			
AR			
:			

Q.3.

	LC (pass 1)	LC	
PG3 START	0	0	
USING *, BASE	0	0	
L 2, D1	0 L 2,-	0	L 2, 16 (0, 12)
A 2, D2	4 A 2,-	4	A 2, 20 (0, 12)
A 2, =F'4'	8 A 2,-	8	A 2, 28 (0, 12)
ST 2, T1	12 ST 2,-	12	ST 2, 24 (0, 12)
D1 DC F'5'	16 '5'	16	0101
D2 DC F'7'	20 '7'	20	0111
BASE EQU 12	24	24	
T1 DS 1F	24 -	24	
END	28	28	0100

Pass 1 — Pass 2

Software Programmer

Assembler

MOT: (I)

Mnemonic Opcode	Binary Opcode	Inst length	Inst Format
L	✗ ✓	✓ ✓	✗ ✓
A	✗	✓	✗
ST	✗	✓	RX ✗
AR			RR
:			

POT:

Pseudo Opcode	Addr of the routine to process pseudo-op
DC	
DS	
START	
USING	

ST:

Symbol	Value	length	R/A
PGM	00	01	R
D1	16	04	R
D2	20	04	R
BASE	12	01	A

L.T.

Literal	Value	length	R/A
=F'4'	28	04	R

B.T.

	Availability indicator	Contents of B.R.
1	'N'	
2	'N'	

12	'X'	'4'
:		
15.	'N'	

Design: S1 - Sample Prog
S2 - Database
S3 - Flowchart
S4 - Explanation

Pass I dB:

1) Original source cards :

These cards contain the original source program.

2) Location Counter :

It is used for keeping a track on the locations of instructions and data.

3) Machine Opcode Table:

MOT is consulted to obtain instruction length.

4) Psuedo Opcode Table:

POT is consulted for the processing of some psuedo opcodes like START, DC, DS, etc.

5) Symbol Table:

Whenever a symbol is defined entry for that symbol is made in the symbol table.

6) Literal Table:

Whenever a literal is encountered entry for that literal is made in the literal table.

7) Copy File:

It contains the copy of the original prog. to be used by Pass 2.

Pass 2 dB:-

1) Copy File

It contains the copy of the original prog. prepared by Pass 1.

2) Location Counter:

It is used for keeping a track on the locations of instructions and data.

3) Machine Opcode Table.

MOT is consulted to obtain binary opcode, inst. length and inst. format.

4) Psuedo Opcode Table:

POT is consulted for the processing of some psuedo opcodes like USING, DROP DC, etc.

5) Symbol Table:

It is used for generating the address of the symbols.

6) Literal Table:

It is used for generating the address of the literals.

7) Base Table:

It is used for keeping a track on the basic register information.

8) INST Coorkspace.

It is used for assembling the instructions.

9) PUNCH CARD Coorkspace

It is used for punching the assembled instructions onto AOC.

POT
MOT
Length
Sym / Lit

10) PRINT LINE coorkspace.

It is used for generating a printout of assembled object program.

11) Assembled object cards (AOC)

These cards contain the object program in a format reqd by the loader.

Q.4.

	Lc	LC
PG4 START	0	0
USING *, BASE	0	0 SR 3,3
SR ACC, ACC	0 SR __,__	0 2 3,24(0,10)
L ACC, FOUR	2 L __,__	6 A 3,32(0,10)
A ACC, FIVE	6 A __,__	10 A 3,48(0,10)
A ACC, =F'5'	10 A __,__	14 A 3,52(0,10)
A ACC, =F'7'	14 A __,__	18 ST 3,40(0,10)
ST ACC, TEMP	18 ST __,__	~~22~~ 24 0100
FOUR DC F'4'	~~22~~ 24 '4'	~~28~~ 32 0101
FIVE DC D'5'	~~28~~ 32 '5'	40
ACC EQU 3	40	40
BASE EQU 10	40	40
TEMP DS 'I' D	40 —	48 0101
END	48	52 0111
	Pass 1	Pass 2

MOT: (I)

Mnemonic Opcode	Binary Opcode	Inst. length	Inst Format
L	X	✓	Rx X
A	X		Rx X
ST			RX
SR	X ✓	✓ ✓	RR ✓

POT: (D+A)

Psuedo Opcode	Addr. of the routine to process psuedo-op
DC	
DS	
START	00
USING	

ST:

Symbol	Value	length	R/A
PG4	00	01	R
FOUR	24	04	R
FIVE	32	08	R
ACC	3	01	A
BASE	10	01	A
TEMP	40	08	R

L.T.

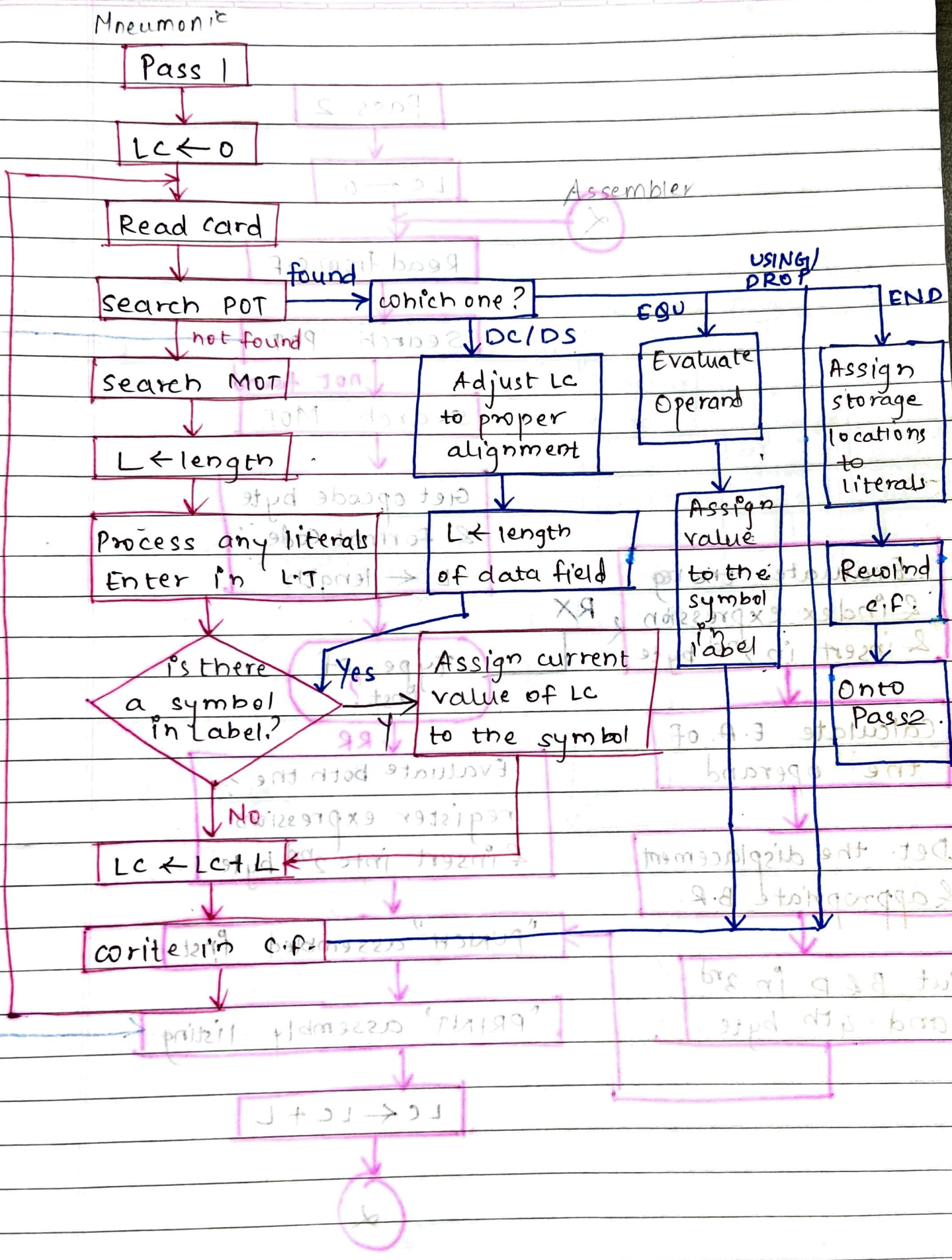
Literal	Value	length	R/A
=F'5'	48	04	R
=F'7'	52	04	R

B.T:

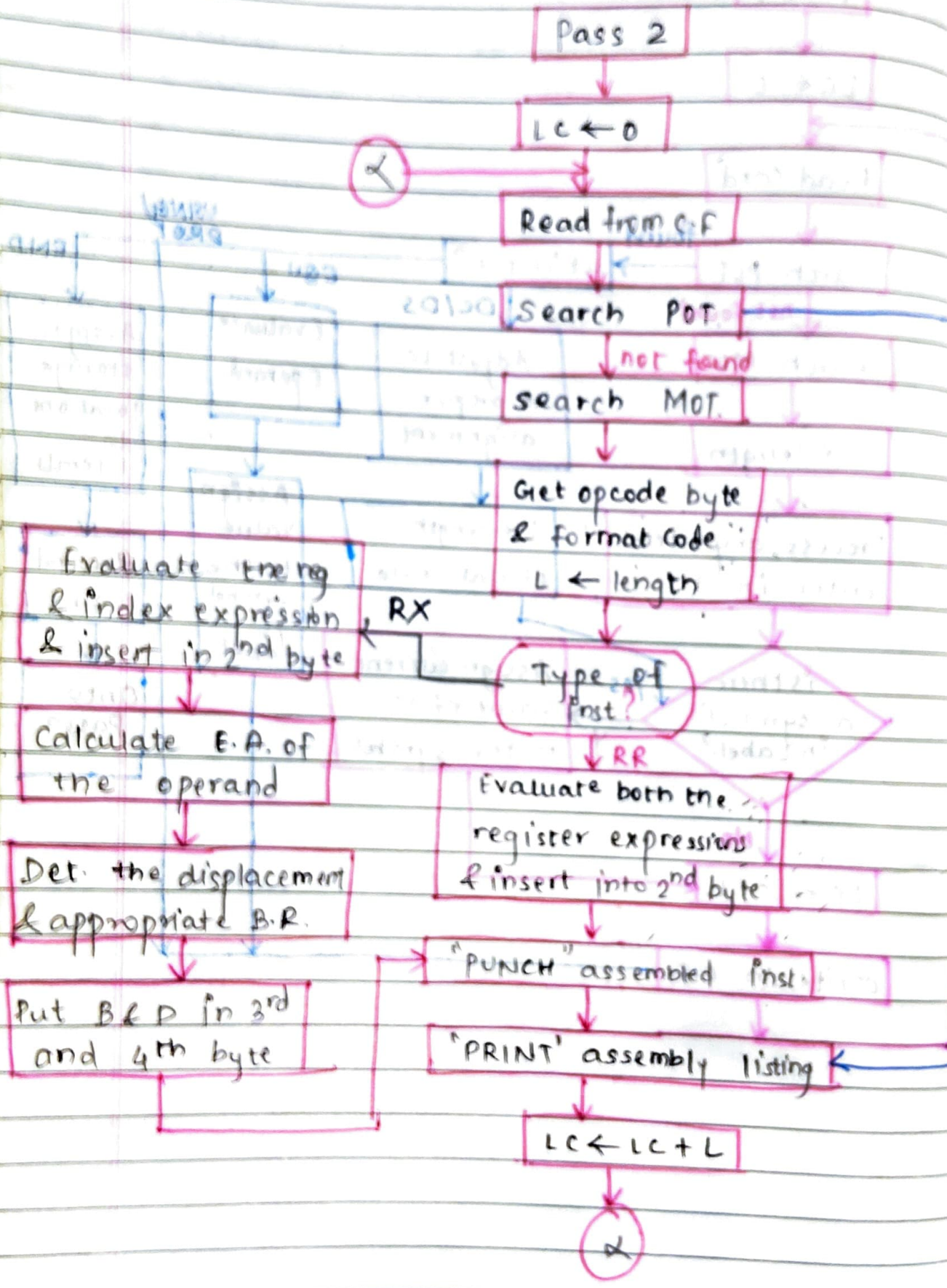
Availability of contents of indicator B.R.

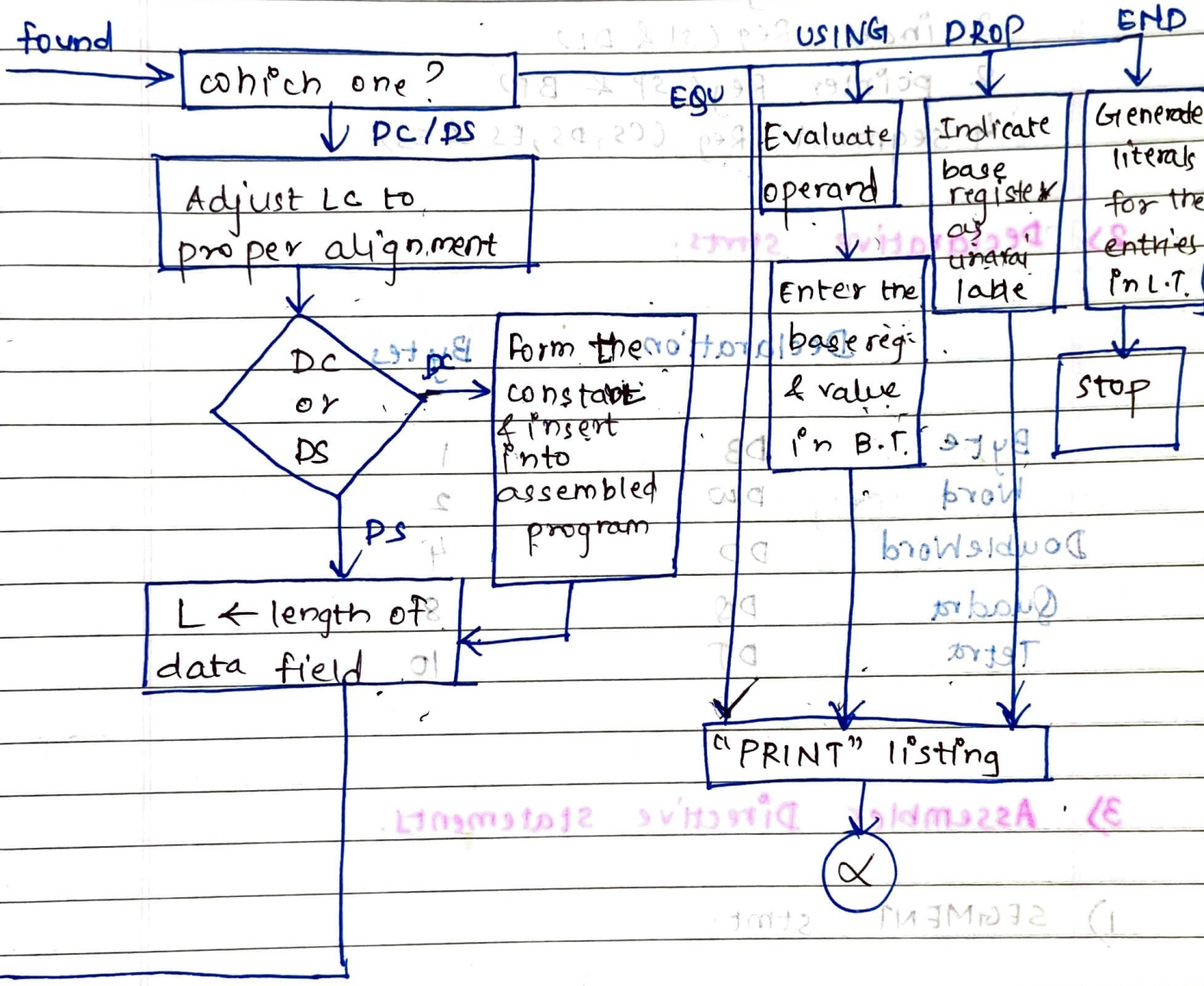
1	'N'
2	'N'
:	
10	'y'
:	
15	'N'

Detailed flow chart of Pass I.



Detailed flow chart of pass 2:




found
which one ?
PC/DS
Adjust LC to proper alignment
DC or DS
DC
Form the constant & insert into assembled program
DS
L ← length of data field
EQU
USING
DROP
END
Evaluate operand
Indicate base register as unavailable
Generate literals for the entries in L.T.
Enter the base reg. & value in B.T.
Stop
"PRINT" listing
∝

Design of IPA:-

(For 8088 processor)

1) Registers

4 data Reg (AX, BX, CX, DX)
2 index Reg (SI & DI)
2 pointer Reg (SP & BP)
4 segment Reg (CS, DS, ES & SS)

2) Declarative stmts.

	Declaration	Bytes
Byte	DB	1
Word	DW	2
DoubleWord	DD	4
Quadra	DQ	8
Tetra	DT	10

3) Assembler Directive statements.

1) SEGMENT stmt.

It indicates the start of the segment.

2) ENDS stmt:

It indicates the end of the segment.

3) END stmt:

It indicates the end of assembly lang. program

4) ASSUME stmt:

It is used for making the segment registers available and also to make them unavailable.

5) ORG stmt:
(Origin)

It is used for manipulating the value of the Location Counter.

6) EQU stmt:

It is used for making the programs more readable.

Design of 1 P.A.
(For 8088 processor)

2 P.A. :- 360
D(I,B)

1 P.A :- 8088
[S:O]

Stmt #		Offset
1	CODE SEGMENT	000
2	ASSUME CS:CODE, DS: PATA.	
10	MOV COUNT, 00H [DS:004]	006
20	ASSUME ES: DATA, DS: NOTHING.	
30	INC COUNT ← [ES:004]	016
40	END	
41	PATA SEGMENT	000
50	COUNT DW	004
60	END	
61	END	

[DS:004]

[ES:004]

Database

NOT:-

Mnemonic Opcode	N/C Opcode	format info	Routine id
I	✓	✓	✓
D, AD	✗	✗	✓

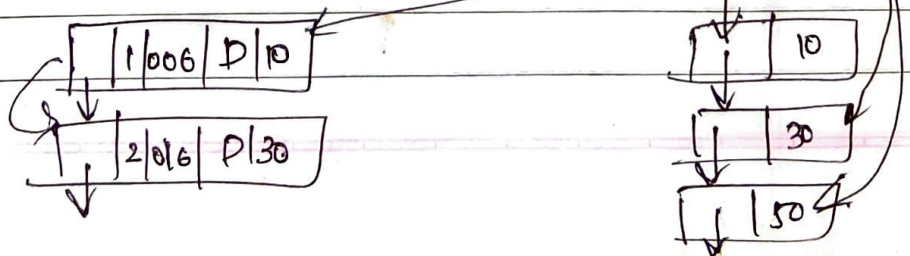
SRTAB:- SYMTAB STSRT:- SYMTAB.

Seg reg	seg name
CS	1
DS	—
ES	(2)
SS	—

Seg reg	seg name
CS	1
DS	(2)
ES	—
DS	—

SYMTAB.

symbol	EQU	Seg name	defined	type	offset in seg	owner seg	LS	source stmt no	Ptr to 1st FRT	Ptr to CRT	Ptr to CRT
1 CODE	N	Y	Y								
2 DATA	N	Y	Y								
3 COUNT	N	N	Y	...	004	2	1	2	50	41	



FRT:

D/L/S.

Pointer	SRTAB #	Instr Addr	Usage code	Source stmt #

CRT:

Pointer	Source stmt #	COUNT: 10, 30 & 80.

Stmt #		Offset
1	CODE SEGMENT	000
2	ASSUME CS: CODE, DS: DATA	
:	← [DS: 008]	
10	MOV SI, 00H	006
:		
20	MOV SI, 10H	0B
:		
30	ASSUME ES: DATA, DS: NOTHING	
:	: ← [ES: 008]	
40	INC SI	02
:	:	
50	ENDS	
51	DATA SEGMENT	000
:		:
60	SI DW ?	008
:		
70	ENDS	
71	END	

Database:-

MOT:

Mnemonic opcode	M/c Opcode	format info	Routine Id
I	✓	✓	✓
D	X	X	✓
AD			

SRTAB:

SYMTAB #

Seg reg.	Seg name
CS	1
DS	-
ES	2
SS	-

#2

STSRT:

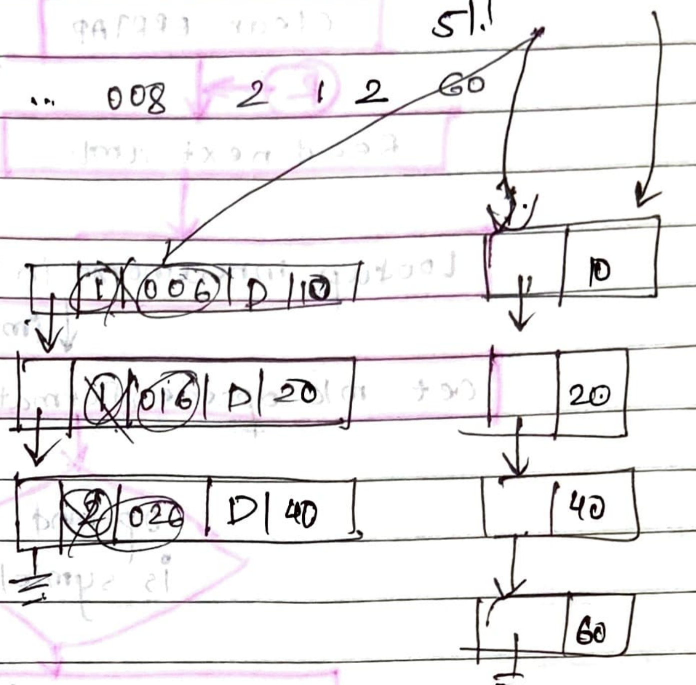
SYMTAB #

Seg reg	Seg name
CS	1
DS	2
ES	-
SS	-

#i

SYMTAB:

	Symbol	EQU ?	seg name ?	defined ?	type	offset in seg	owner seg	L	S	source stmt #	Ptr to first FRT	first CRT	last CRT
1	CODE	N	Y	Y						1			
2	DATA	N	Y	~~N~~ Y						S1.			
3	S1	N	N	~~N~~ y	...	008	2	1	2	60			



FRT: (TII) (Table of Incomplete Instruction)

D/L/S.

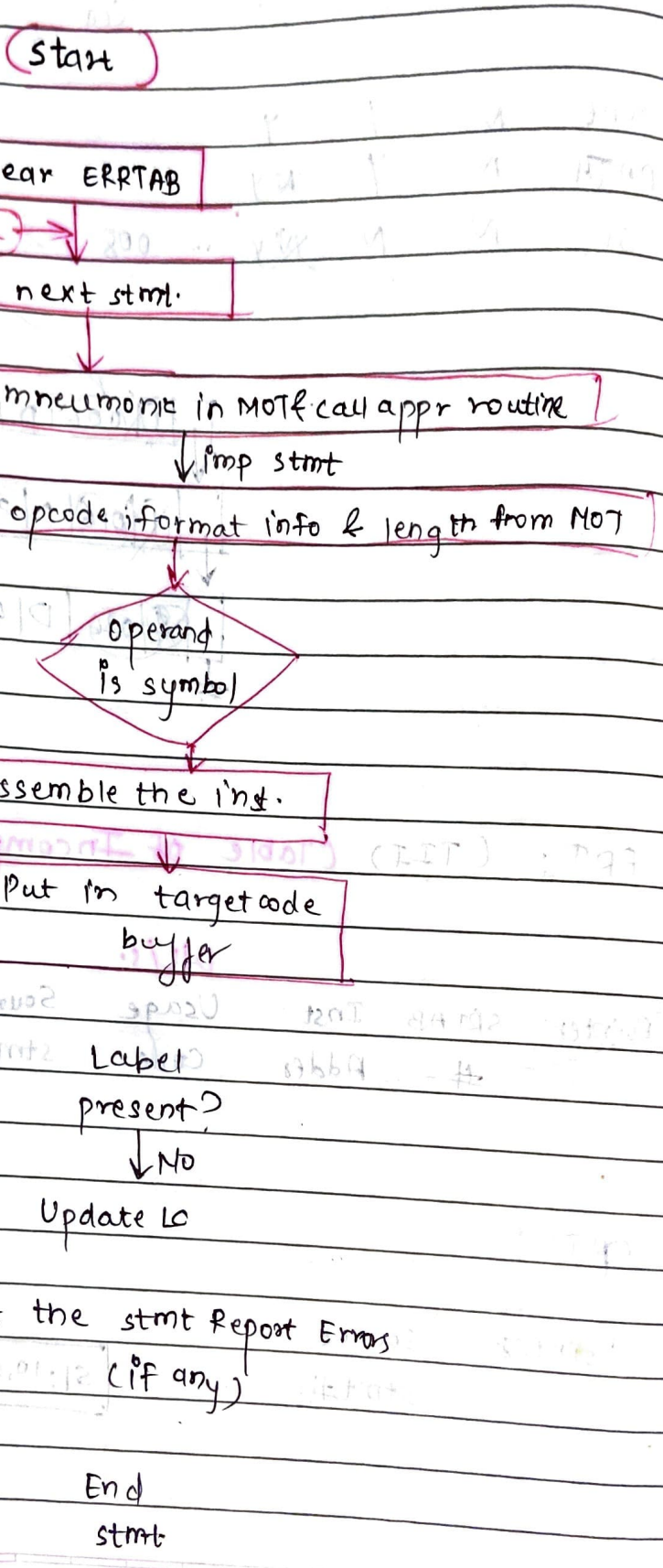
Pointer	SRTAB #	Inst Adder	Usage Code	Source stmt#

CRT:

Pointer	Source stmt#

S1: 10, 20, 40 & 60

Detailed flowchart of single parse assembler



26th Jan, 2015.

Chpt 3: Macro Processor.

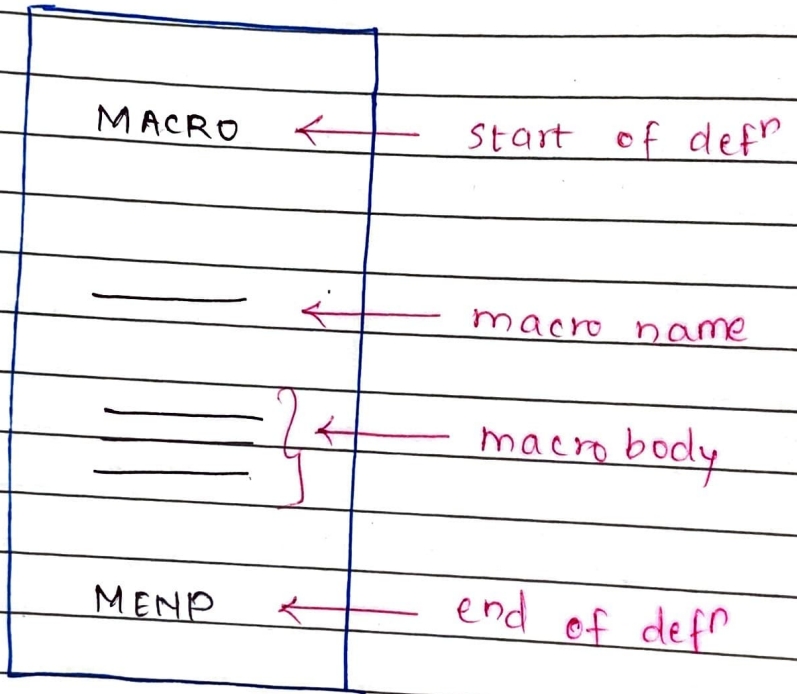
1) Defn of macro:

Macro is defined to be single line abbreviation for group of instructions.

2) Defn of macroprocessor:

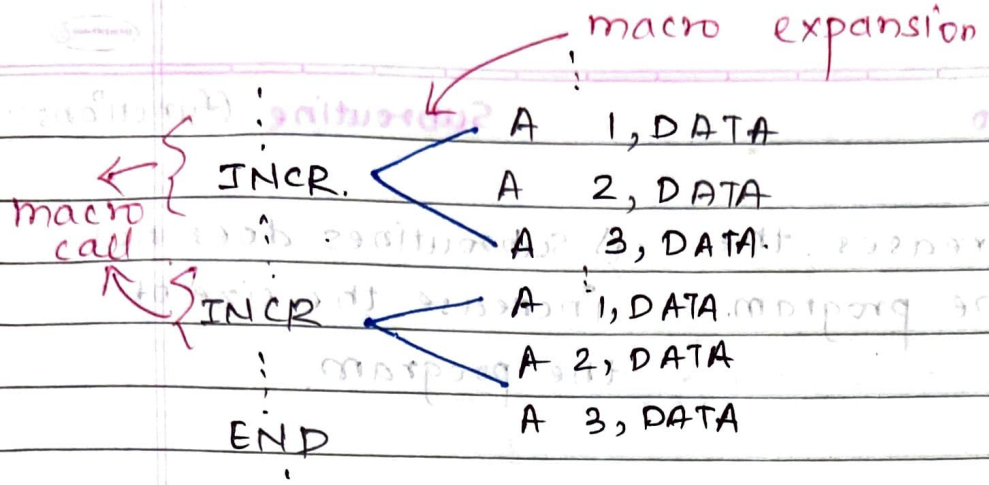
It is a program which is responsible for the processing of the macro.

Format: IBM 360



eg. macro defn

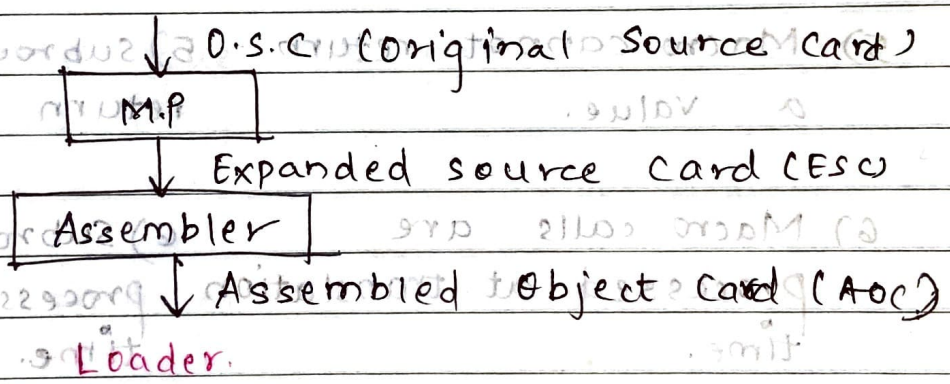
```
MACRO
INCR
A 1, DATA
A 2, DATA
A 3, DATA
MEND
```



Need for Macro Processor:

The meaning of macroname is not known to the assembler because it is programmer defined.

Hence, we need a macro-processor that takes as i/p original program and performs the macro expansion.



Macro	Subroutine (functions)
1) Macro increases the size of the program.	1) Subroutines does not increase the size of the program.
2) Macro does not alter the the flow of execution.	2) Subroutines alter the flow of execution.
3) Programs using macro coould get executed faster.	3) Programs using subroutine coould get executed comparitively slower.
4) Macro does not require any return address.	4) Subroutines require return address.
5) Macro cannot return a value.	5) subroutines can return a value.
6) Macro calls are processed at translation time.	6) Subroutine calls are processed at execution time.
7) Macro requires macro processor.	7) Subroutines require no additional processor
8) If LOC is in the range of 3 to 5 then macro usage is recommended.	8) If LOC is beyond 5 then subroutine usage is recommended

Parameters:

1) Formal Parameters:

These are the parameters that appear in the macro definition.

2) Actual Parameters:

These are the parameters that appear in the macro call.

3) Parameter Passing Techniques:

1) Positional Parameters (P.P.):-

In this technique, the parameters correspond to their positions.

2) Keyword Parameters (K.P.):-

In this technique, the parameters correspond to their keywords.

Note: If a macro contains a combination of P.P.'s and K.P.'s then all the P.P.'s must appear before the K.P.'s.

Parameters (Formal & Actual) :

eg:

```
MACRO
INCR1  &ARG
A  1, &ARG
A  2, &ARG
A  3, &ARG
MEND
:
INCR1  DATA1
:
INCR1  DATA2
:
END
```

```
A  1, DATA1
A  2, DATA1
A  3, DATA1

A  1, DATA2
A  2, DATA2
A  3, DATA2
```

Parameters Passing Techniques.

3-5.

1) Positional Parameters (P.P) :-

eg:

```
MACRO
INCR 2 &ARG1, &ARG2, &ARG3
A  1, &ARG1
A  2, &ARG2
A  3, &ARG3
MEND
:
```

In this's arguments are matched with the dummy args acc to the order in which they appear with positional parameters. The prog must be careful enough to specify the args in proper order.

```
INCR2   DATA1, DATA2, DATA3        A 1, DATA1
                                   A 2, DATA2
                                   A 3, DATA3
                                   :
INCR 2  DATA 7 , DATA 5, DATA 6    A 1, DATA7
                                   A 2, DATA5
                                   A 3, DATA6
                                   :
END
```

2. Keyword Parameters (K.P.)

eg.

```
MACRO
INCR 3  &ARG1 =, &ARG2 =, & ARG3 =
A   1, & ARG 1
A   2, & ARG 2
A   3, & ARG 3
MEND
:
INCR3   &ARG1 = DATA1, &ARG2 = DATA2, & ARG3 = DATA3
:
INCR 3  &ARG2 = DATA2, &ARG1 = DATA1, & ARG3 = DATA3
:
INCR 3  &ARG3 = DATA3, &ARG1 = DATA1, &ARG2 = DATA2
:
INCR3   DATA 1, DATA2, DATA 3
:
END
```

They allow ref. to dummy args. by name as well as by position.

Functions of Macroprocessor:

1) Recognise the macro definition. (Bth macro & mend)
2) Store the macro definition. (Entry in dB)
3) Recognise the macro call. (search in MNT)
4) Perform macro expansion.

Design of 2 Pass Macro Processor:-

1) Rules of Macro lang. state that the macro can be defined anywhere in the program.
Hence, there may be some cases in which macro call appears before the macro definition and such a call is called forward reference.

2) Due to forward ref. macro processor cannot perform the expansion and such a problem is called forward ref. problem.

3) To solve the prob. macro processor will make 2 Passes over the i/p prog.

4) Purpose of Pass1 is to recognise the macro definitions and store the macro definitions

5) Purpose of Pass 2 is to recognise the macro calls and perform macro expansion.

Format of Database:

I] Macro Name Table (MNT):

Index	Macro Name	MDT index
1		
2		
3		

1) MNT is used for storing the macroname alongwith MDT index which indicates the location in MDT where the corresponding definition is stored.

2) In pass1, MNT is used for storing the macroname alongwith MDT index.

3) In pass2, MNT is used for recognising the macro calls.

II] Macro Definition Table (MDT):

Index	Macro Definition
1	
2	
3	
4	
5	
:	

1) MDT is used for storing the macro definitio alongwith MEND statement.

2) In pass I, MDT is used for storing the macro definition.

3) In pass2, MDT is used for performing the macro expansion.

III Argument List Array (ALA)

Index	Argument
1	
2	
3	
⋮	

1) ALA is used for parameter replacement procedure.

2) In pass I, ALA is used for replacing the formal parameters by index notations.

3) In pass 2, ALA is used for replacing the index notations by actual parameters.

Pass I dB:

1) Original source cards (OSC)

These cards contain original source program. (OSC contains macro definitions, macro calls and other statements). I, D, A:

2) Macro Name Table (MNT).

It is used for storing the macroname alongwith MDT index.

3) Macro Name Table Counter (MNTC)

It indicates the available entry in MNT.

4) Macro Definition Table. (MDT)

It is used for storing the macro definition.

5) Macro Definition Table counter (MDTC)

It indicates the available entry in MDT.

6) Argument List Array (ALA)

It is used for replacing formal parameters by index notations.

P1: OSC → CF(RUM)

P2: CF → ESC(RUMC)

7) Copy File.

It is prepared by pass1 to be used by pass2 (c.f. contains macro calls and other statements).

Pass 2 dB:

1) Copy File

It is prepared by pass1 to be used by pass2.

2) Macro Name Table (MNT)

It is used for recognising the macro calls.

3) Macro Definition Table. (MDT)

It is used for performing the macro expansion

4) Macro Defn Table Pointer

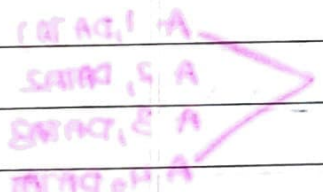
It points to the MDT stmt which is under expansion.

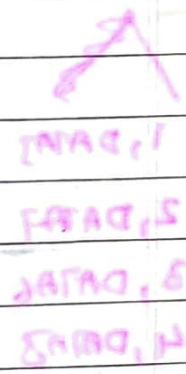
5) Argument List Array.

It is used for replacing the index notations by actual parameters.

6) Expanded source card (ESC)

These cards contain the expanded source prog. (ESC contains other stmts. plus expanded stmts.).





Sample Program:

```
:
MACRO
INCR4 &ARG1, &ARG2, &ARG3=DATA3, &ARG4=
A  1, &ARG1
A  2, &ARG2
A  3, &ARG3
A  4, &ARG4
MEND
:
:
INCR4  DATA1, DATA2, &ARG4=DATA5
:
:
:
INCR4  DATA5, DATA7, &ARG3=DATA6, &ARG4=DATA8
:
:
:
:
:
:
:
END
```

A 1, DATA1
A 2, DATA2
A 3, DATA3
A 4, DATA5

A 1, DATA5
A 2, DATA7
A 3, DATA6
A 4, DATA8

Database:

MNT:

Index	Macro Name	MDT index
1	INCR4	1
2		
3		

MNTC

MDT:

MDTP

MDTC

MDTP.

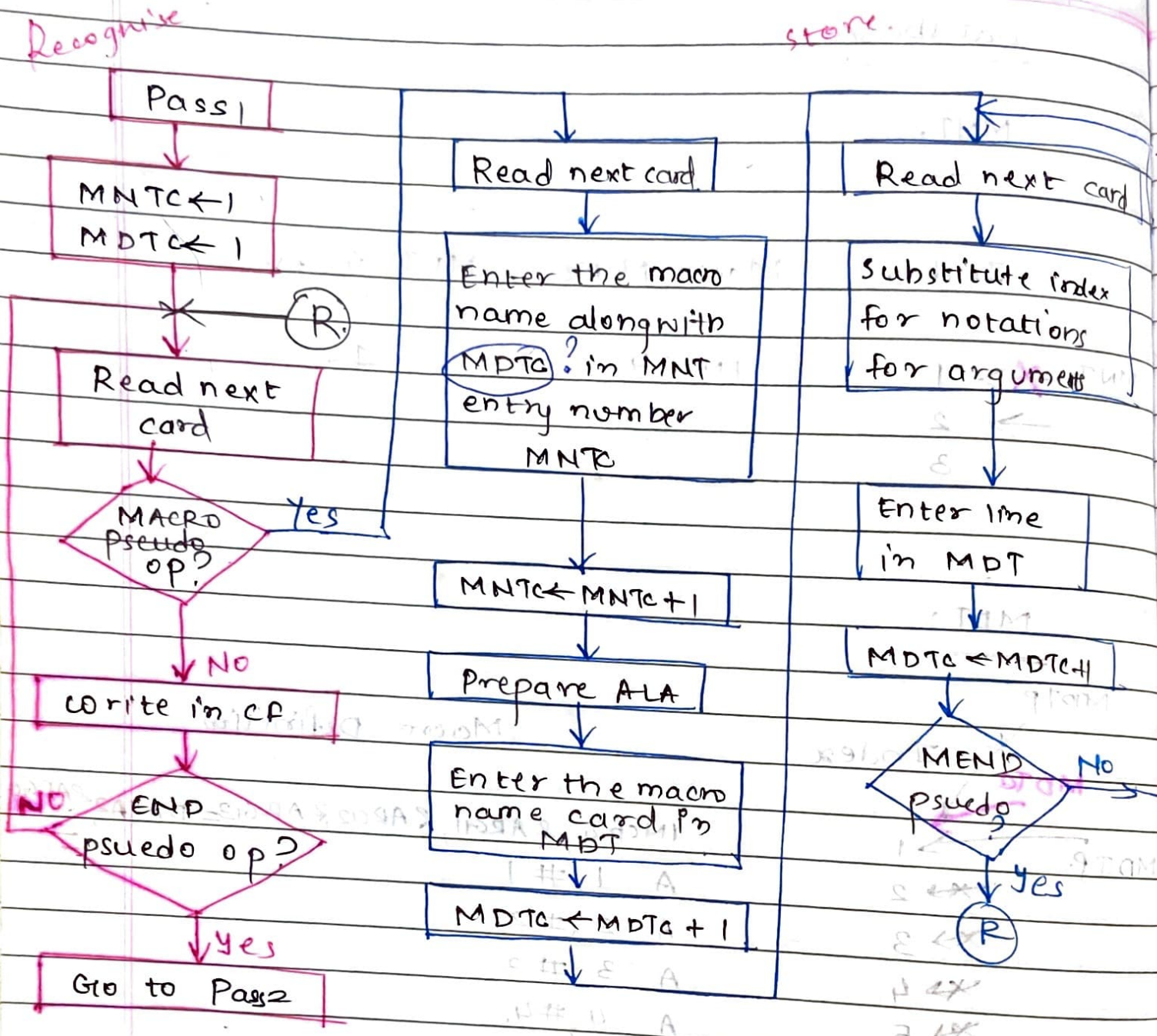
Index	Macro Definition
1	INCR4, &ARG1, &ARG2, &ARG3=DATA2, &ARG4=—
2	A 1, #1
3	A 2, #2
4	A 3, #3
5	A 4, #4.
6	MEND
7	
8	

ALA:

show the last ALA.

Index	Argument
1	DATA5
2	DATA7
3	DATA6
4	DATA8.

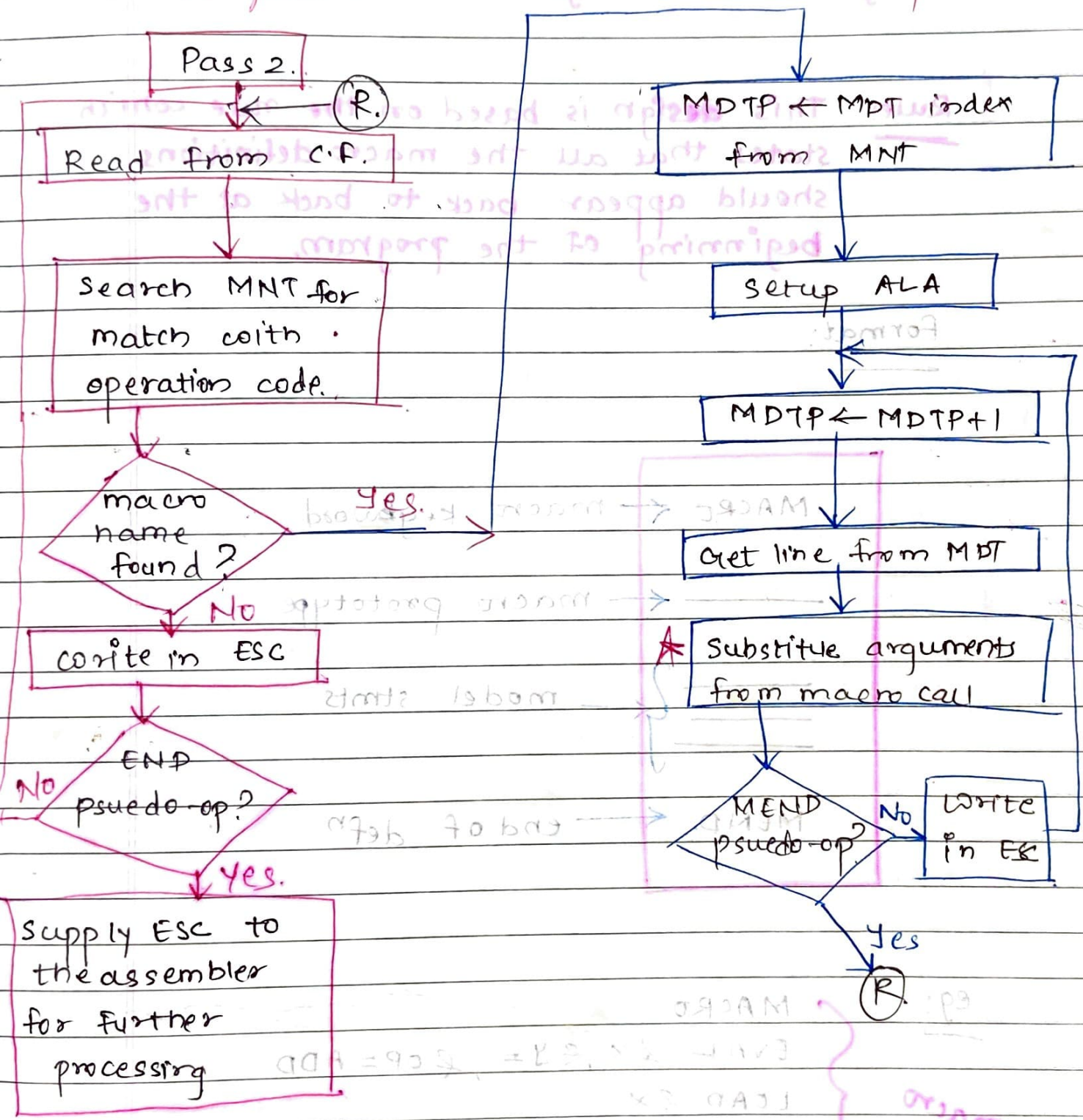
Detailed flowchart of Pass I



Detailed flowchart of Pass 2.

Recognise

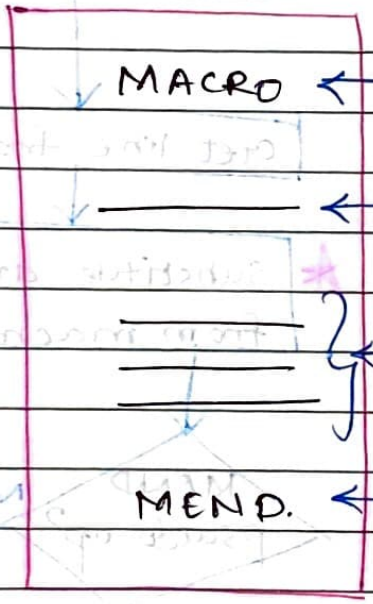
Expand.



Design of single Pass Macro Processor:

Rule: This design is based on the rule which states that all the macro definitions should appear back to back at the beginning of the program.

Format:



eg:

macro defn.

```
MACRO
EVAL &X,&Y=,&OP=ADD
LOAD &X
&OP &Y
STORE &X
MEND.
```

2PMA

MNT MDT ALA.

macro call

EVAL M, &Y=N → macro expansion: LOAD N / ADD N / STORE M

:

EVAL P, &OP= SUB, &Y=Q → LOAD P / SUB Q / STORE P

:

END.

Format of Database.

I] MNT (Macro Name Table)

Macro name	MDTP	#P.P	#K.P	KPTP	EVs
EVAL					0

EVAL → MDT

0 → KPT

1) MNT is used for storing the macroname alongwith MDTP which indicates the location in MDT where the corresponding definition is stored.

2) MNT keeps a track on the no. of P.P.s and no. of K.P.s for error handling purpose.

3) MNT also maintains a ptr. to KPT called K.P.T.P

4) MNT also keeps a track on the no. of expansion variables required by the macro.

MPTP.

II Macro Defn Table (MDT)

NEC → macro definition
EVAL &X ①, &Y = ② , &OP = ③ ADD
LOAD #1
#3 #2
STORE #1
MEND.

1) MDT is used for storing the macro defn along with MEND stmt.

2) While storing the defn, parameters and expansion variables coould be replaced by #n, &Ev #n respectively.

3) While performing the expansion #n &Ev #n coould be replaced by their values from APL and EVS respectively.

IV Keyword Parameter Table.

Keyword	default
&Y =	
&OP =	ADD.

1) KPT is used for storing the default values of keyword parameters.

EXPAND : ON/OFF.

1) The value of expand indicates whether macro expansion is ON or OFF.

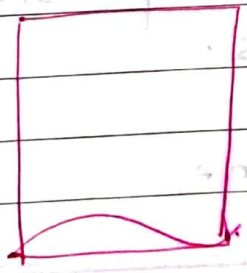
2) Macro Expansion Counter (MEC) points to the MDT stmt which is under expansion.

APL (Actual Parameter List).

#1	P
#2	Q
#3	SUB

1) APL is used for parameter replacement procedure.

EVS (Expansion Variable Storage)



1) At the time of macro expansion, EVs would be allocated depending on the no. of expansion variables required by the macro.

Feature 2: Conditional Assembly:

1. AIF stmt :-

This stmt provides conditional branching facility. The general format is:

AIF (condition) . label

2. AGO stmt:-

This stmt provides unconditional branching facility. The general format is:

AGO . label

3. LCL stmt:

This stmt is used for defining local variables. (Also called expansion variables). The general format is:

LCL &variableName

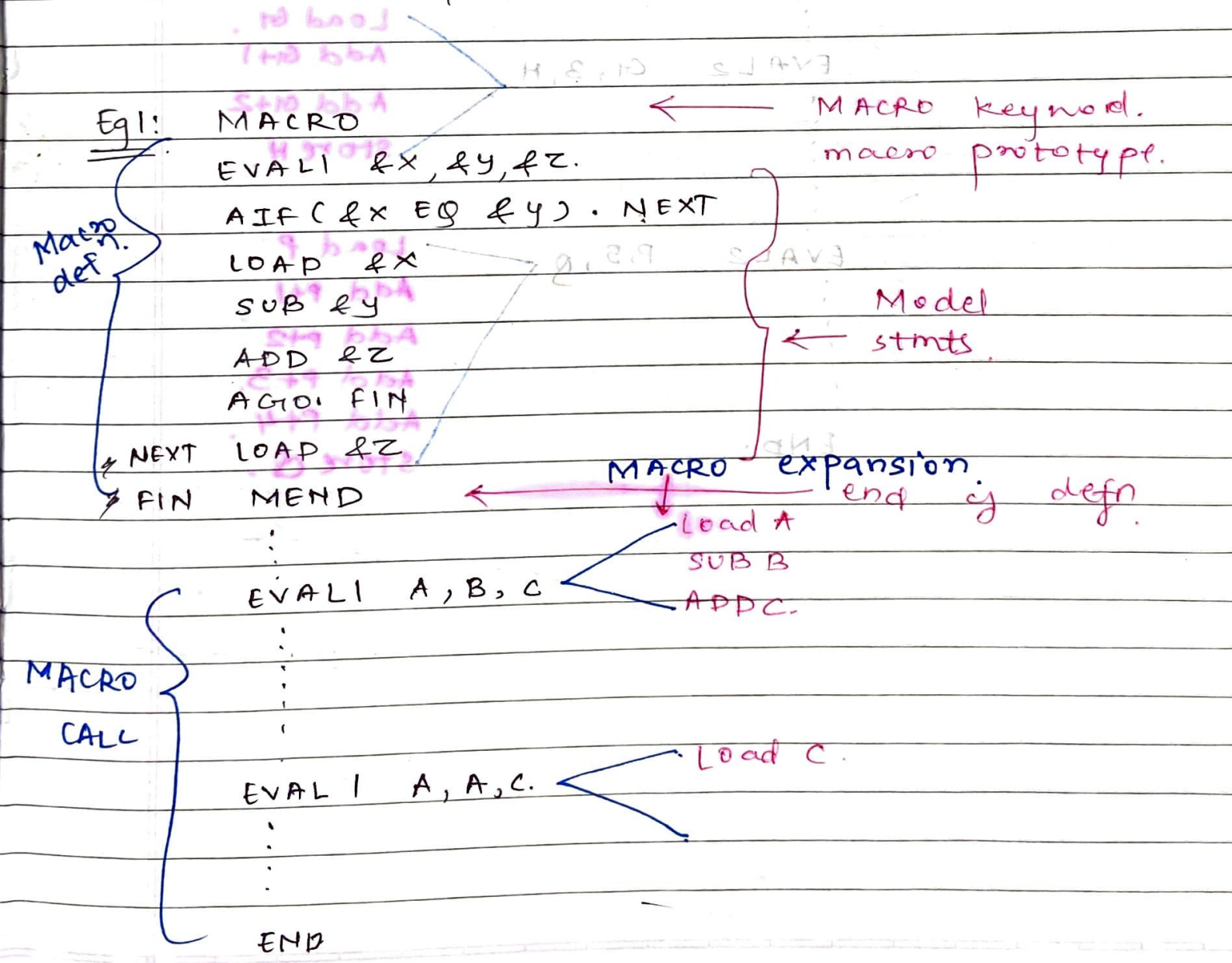
4. SET stmt.

This stmt is used for manipulating the value of expansion variables.
The general format is:

& variableName SET newValue

5. ANOP stmt.

This stmt performs NO OPERATION.

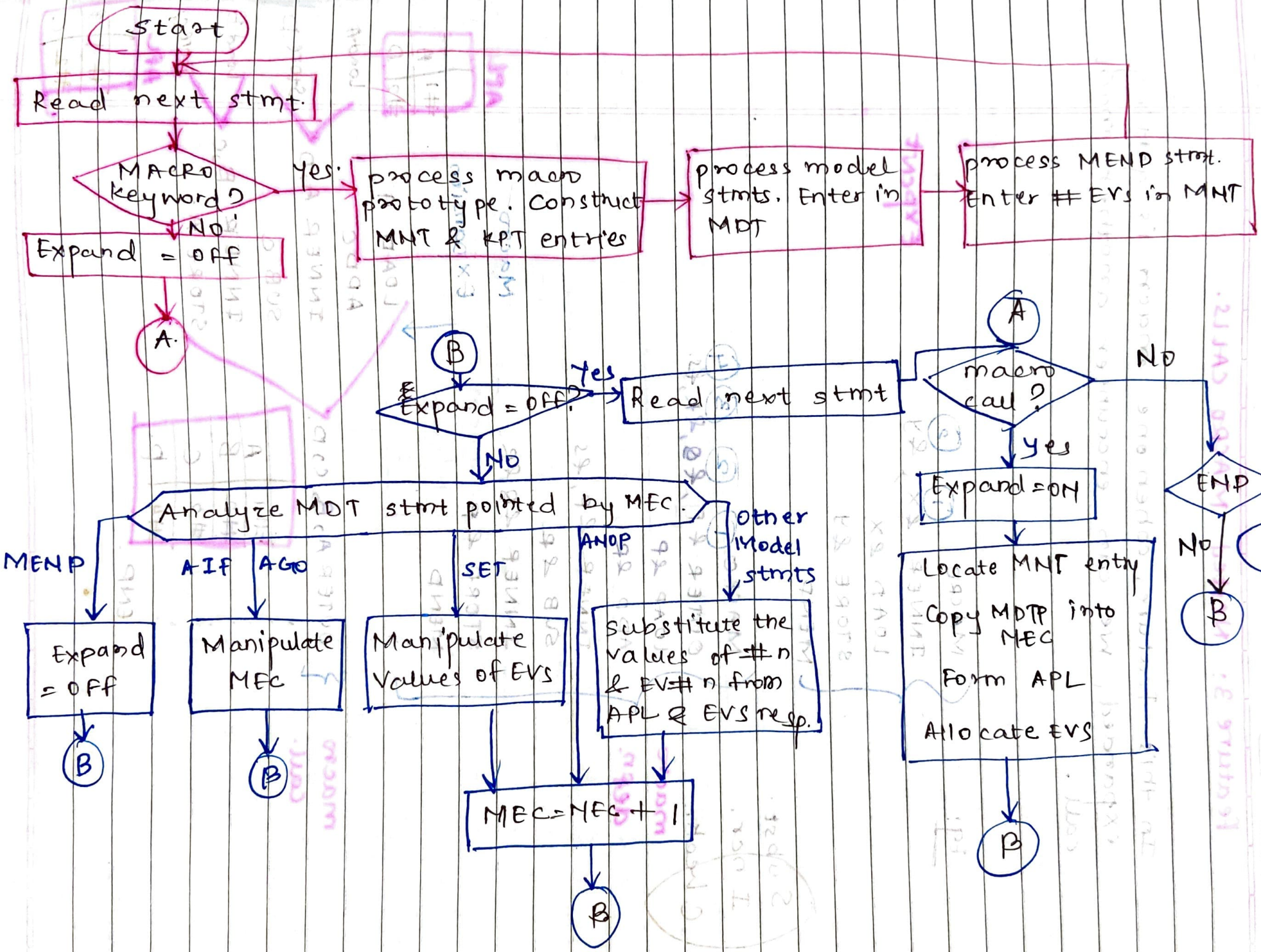


Eg2:

```
MACRO
EVAL2   &A, &N, &R.
LCL &C.
LOAD    &A
BACK    ANOP
&C      SET &C+1
        AIF(&C EQ &N).NEXT.
        ADD &A+&C.
        AGO. BACK.
NEXT    STORE &R.
        MEND
        .
        .
        .
EVAL2   G,3,H
        .
        .
        .
EVAL2   P,5,Q
        .
        .
        .
        END.
```

Load G.
Add G+1
Add G+2
Store H

Load P
Add P+1
Add P+2
Add P+3
Add P+4
Store Q.



Detailed flowchart of single Pass M.P.

Feature 3: Nested MACRO CALLS.

In this feature, when one macro is getting expanded, we can encounter another macro call.

Eg:

```
MACRO
INNER &X, &Y
LOAD &X
STORE &Y
MEND

MACRO
OUTER &P, &Q, &R, &S.
LOAD &P
ADD &R
INNER &P, &S
SUB &R
INNER &Q, &R
STORE &P
MEND
:
:
macro call.
OUTER A, B, C, D.
:
:
:
END.
```

Subst.
Incr.
Check.
macro defn.

#1	A
#2	B
#3	C
#4	D.

EXPCNT

Macro Expansion

```
LOAD A
ADD C
INNER A, D
SUB C
INNER B, C
STORE A.
```

APL

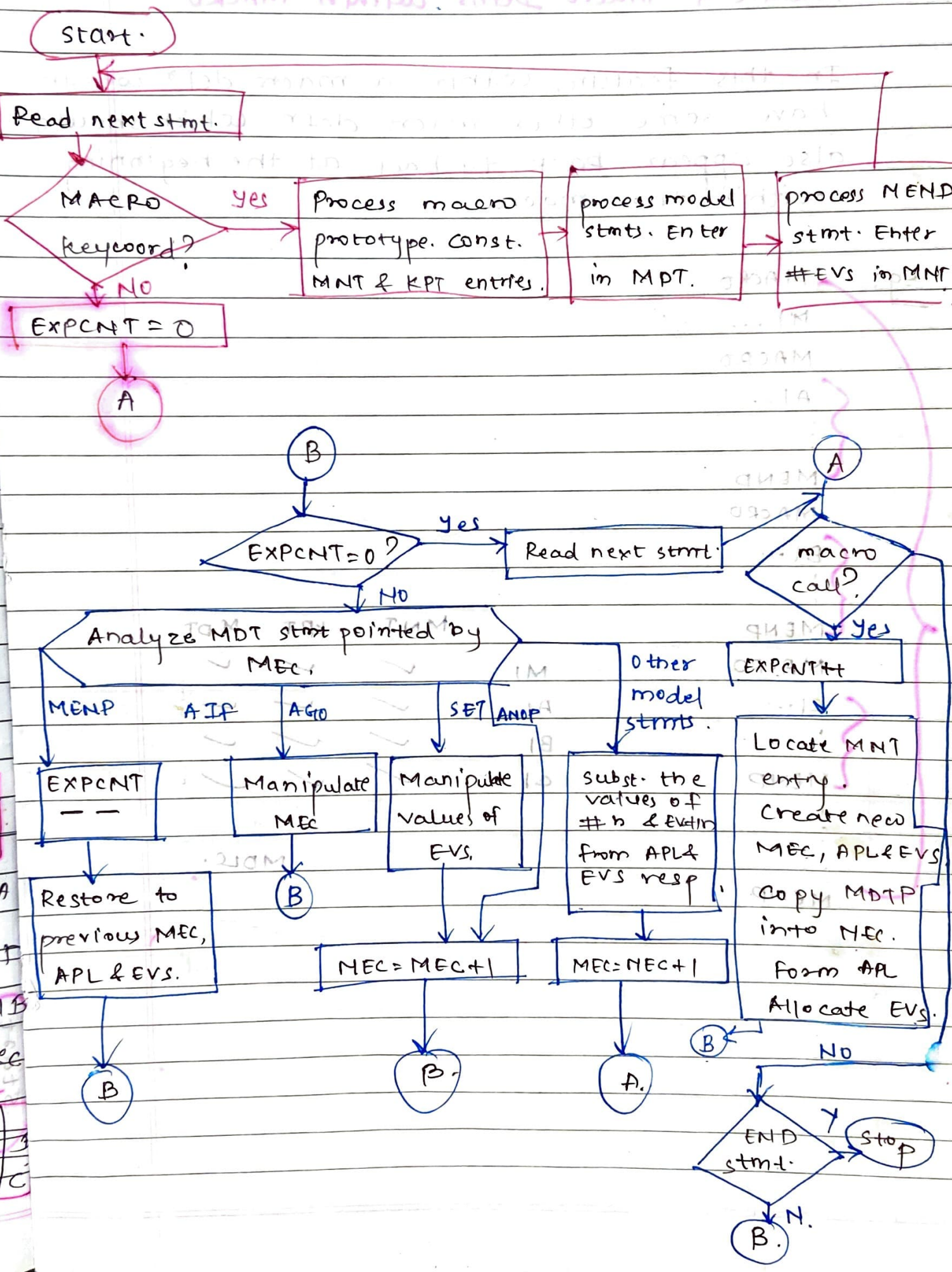
#1	A
#2	D

Load A
Store D
Load B
Store C

APL

#1	A
#2	C

Detailed flowchart of single pass M.P. to handle nested Macro CALLs.



Feature 4: Macro Defns within MACRO

In this feature, within a macro defn we can have some other macro defn which should also appear back to back at the beginning within the macro.

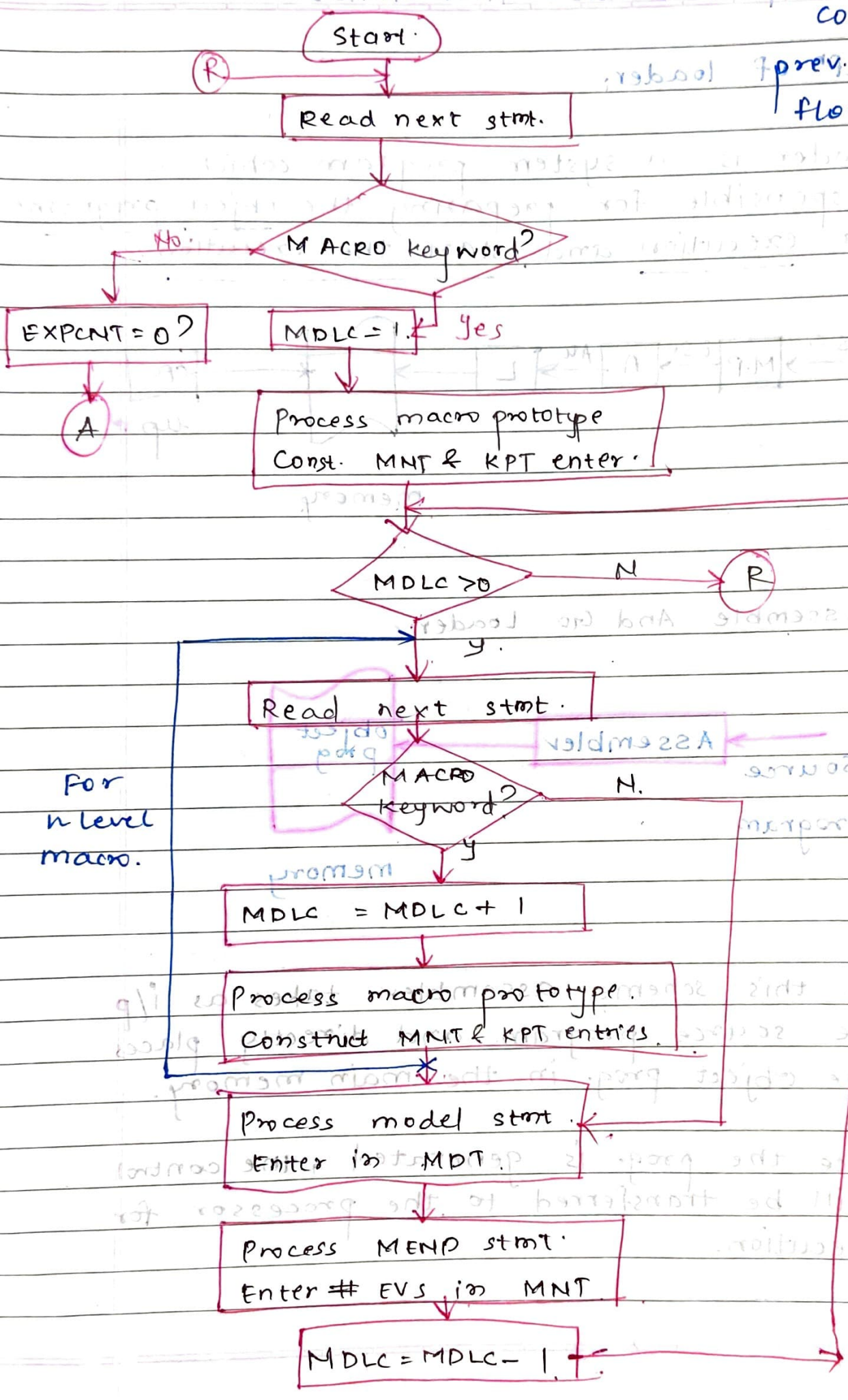
Eg:

```
MACRO
M1 ...
MACRO
A1...
:
:
MEND
MACRO
B1...
:
:
MEND
MACRO
C1...
:
:
MEND
:
:
MEND
```

	MNT	KPT	MDT
M1	✓	✓	✓
A1	✓	—	—
B1	✓	✓	—
C1	✓	✓	✓

MDLC.

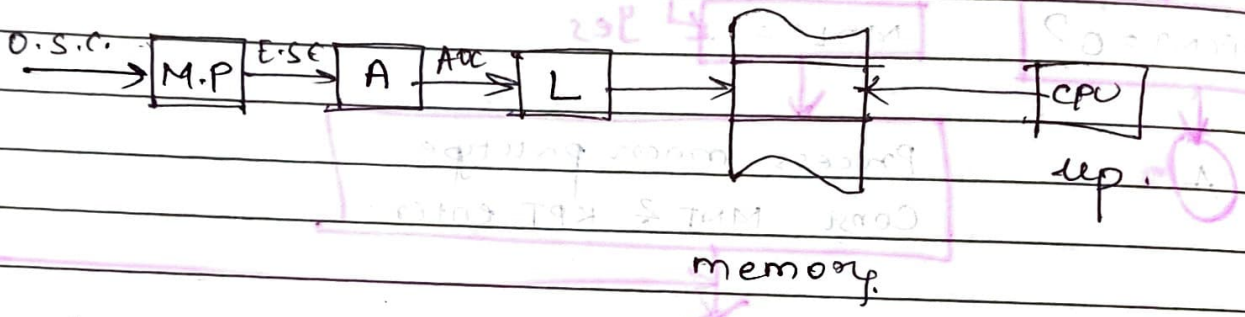
continue prev. blue flow chart.



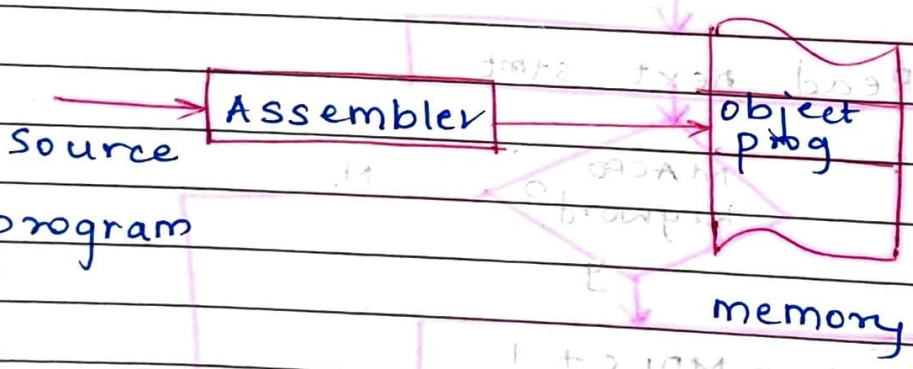
Chpt. 4 Loaders And Linkers.

Defn of loader:

Loader is a system program which is responsible for preparing the object programs for execution and start the execution.



Assemble And Go Loader:



1. In this scheme assembler takes as i/p the source program and directly places the object prog. in the main memory.
2. Once the prog. is generated, the control will be transferred to the processor for execution.

Advantage:

1. Simple to implement.

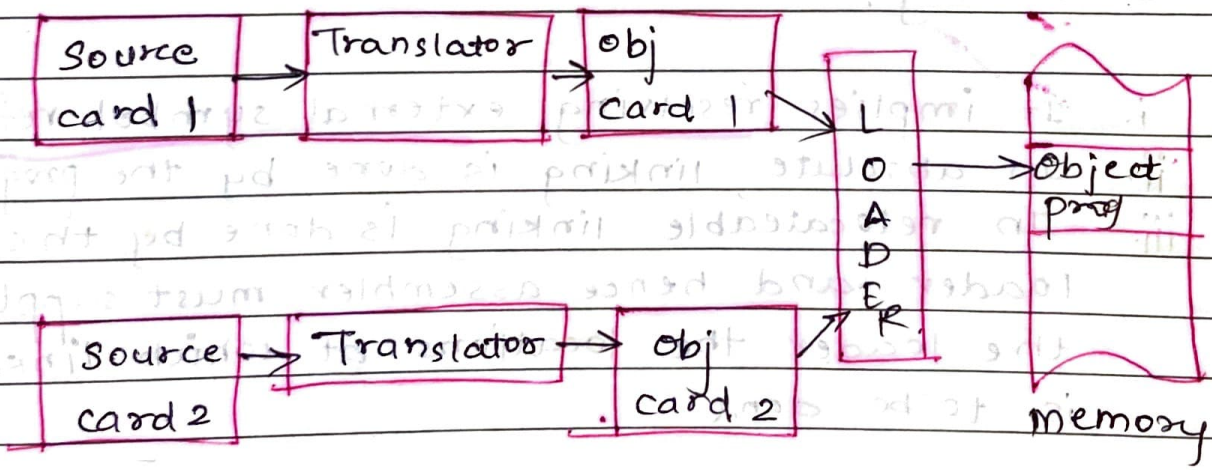
Disadvantage:

1. Scheme requires a very high execution time.

VIVA.

★ General Loading Scheme: → A.L
→ D.L.L.

This scheme was introduced to eliminate the above disadvantage.



AA

5 mks Functions of Loader:

1. Allocation.

i. It implies allocating the space in the memory where the object programs could be loaded for execution.
ii. In absolute, allocation is done by the programmer.
iii. In relocateable, allocation is done by the loader and hence assembler must supply the size of the program.

2. Linking.

i. It implies resolving external symbol references.
ii. In absolute, linking is done by the programmer.
iii. In relocateable linking is done by the loader and hence assembler must supply to the loader the locations at which linking is to be done.

3. Relocation.

i. It implies adjusting the address sensitive instructions to correspond to the allocated space.
ii. In absolute relocation is done by the assembler.

iii. In relocateable, relocation is done by the loader and hence assembler must supply to the loader the locations at which relocation is to be done.

4. Loading:

i. It implies loading the object prog. in the main memory.

ii. In both the cases, this fxn is performed by the loader and hence assembler must supply to the loader the object prog. which is to be loaded.

1. Allocation
   A → Programmer
   R → Loader (size)

2. Linking
   A → Programmer
   R → Loader (Location)

3. Relocation
   A → Assembler
   R → Loader (Location)

4. Loading
   A →
   R →
   } Loader → (obj. prog.)

```
MAIN START 100
        L 15, A(SQRT)
        BALR 14,15
        L 12, A(DATA)
SQRT    DC F'400'
DATA    DC F'4'
        END
```

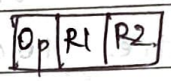
```
SQRT START 400
        BR 14
        END
```

BALR R1, R2
BR R1
A (symbol)

Design of An Absolute Loader:

Absolute loader requires the foll. types of cards from the assembler.

TXT Cards:

Card type	Count	Address	Contents
0	2	4000	

RR ,

1. Card type:
   It is always zero for text cards.

2. Count:
   It indicates in bytes the amt. of binary info. available on the card.

3. Address:
   It indicates the location to which binary info. should be loaded.

4. Contents:
   It contains the binary info. which is to be loaded (binary info. means assembled insts. and data).

Transfer card:

Card type	Count	Address	Contents.
1	0		—

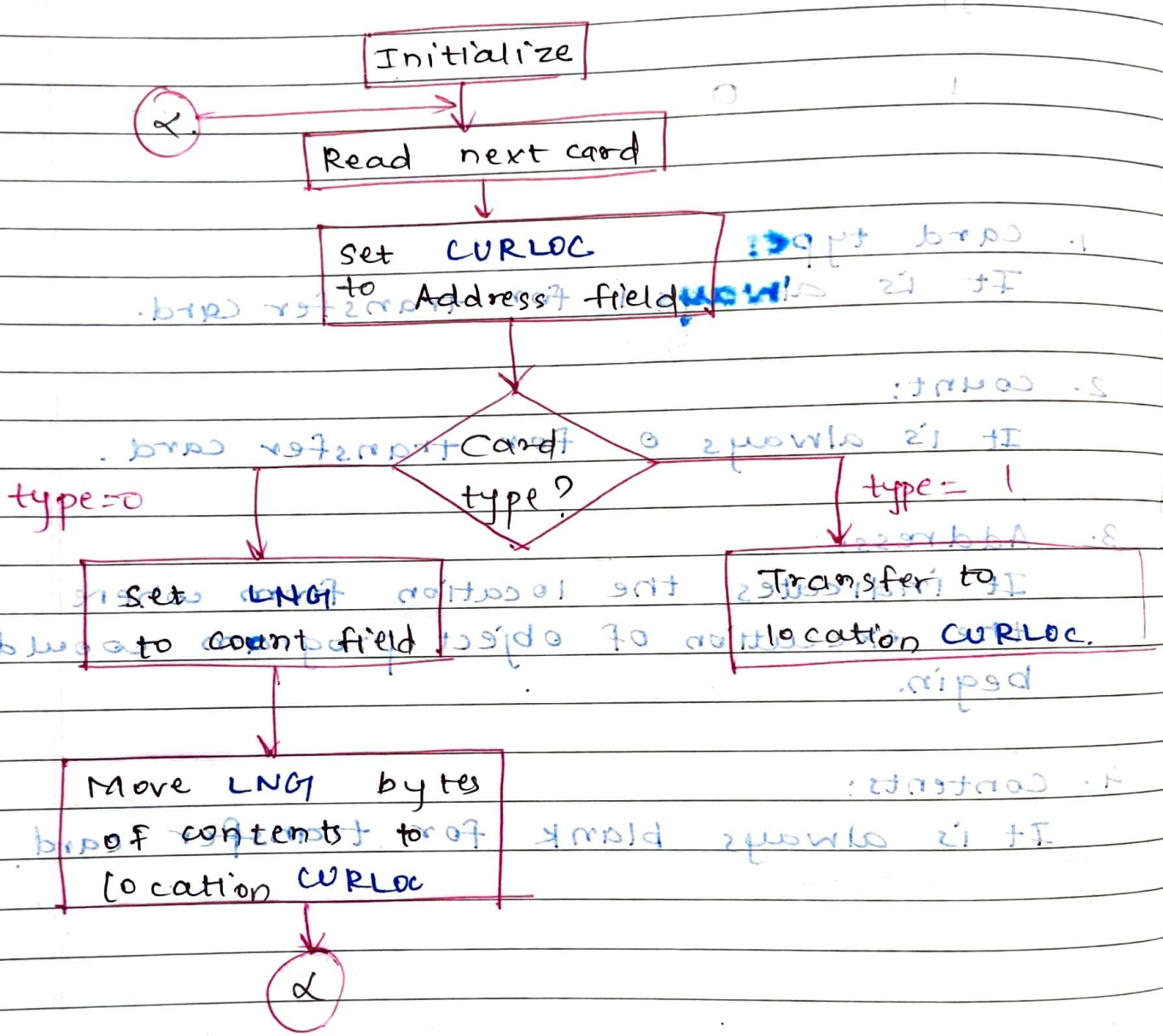
1. Card type:
   It is always 1 for transfer card.

2. Count:
   It is always 0 for transfer card.

3. Address:
   It indicates the location from where the execution of object program could begin.

4. Contents:
   It is always blank for transfer card.

Detailed flowchart of absolute loader:



CURLOC — current location

LNG — length.

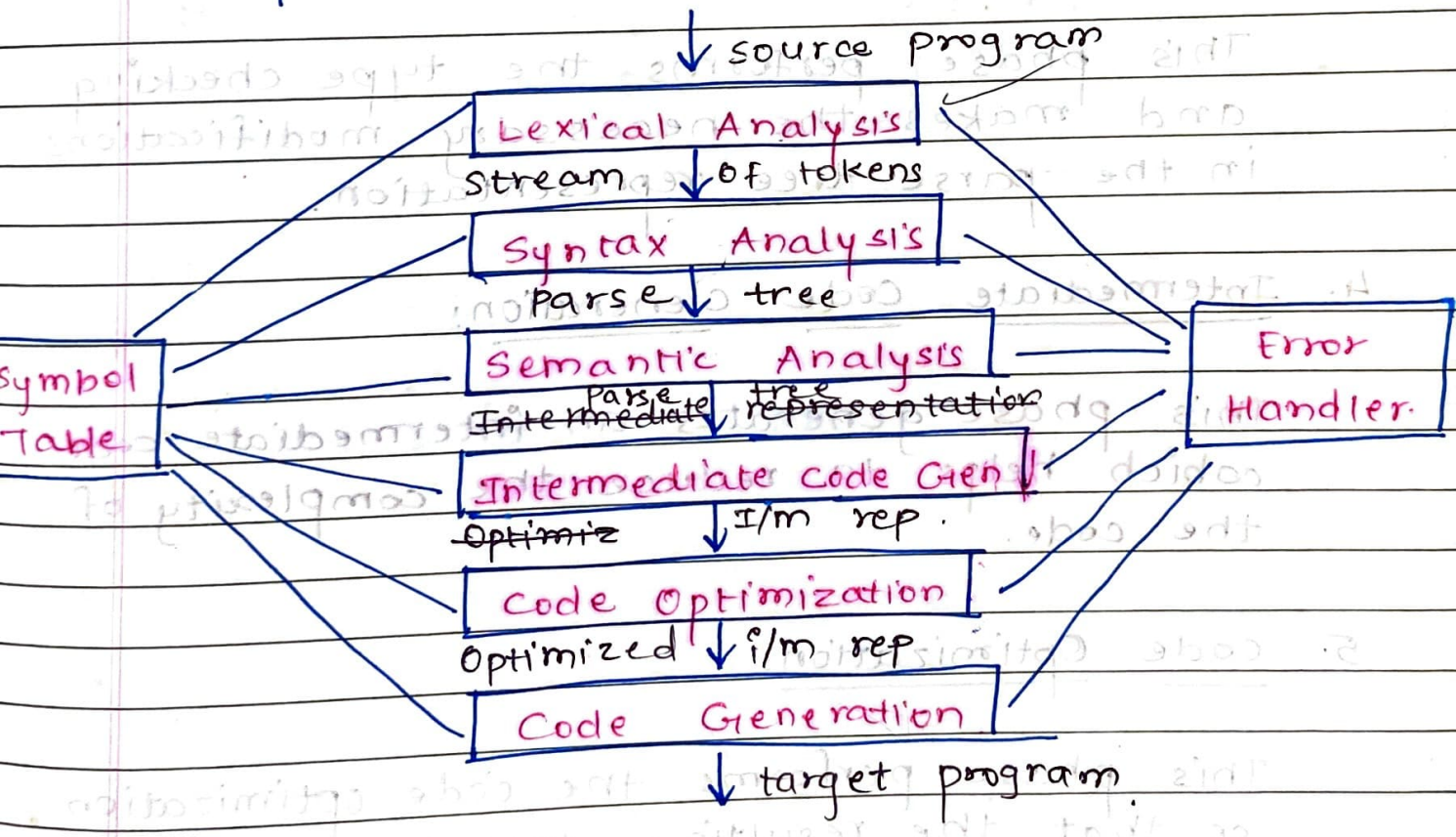
Compilers.

Defn:

Compiler is a lang. translator that takes as i/p the source prog. and generates the target program. The source prog. is a high-level lang. prog and the target prog. can be Assembly lang. or m/c lang.

✱ Structure of compiler:

Since, the process of compilation is very complex it is divided into no. of phases.



1. Lexical Analysis:

This phase takes as i/p the source program and if the elements of the prog. are correct it generates stream of tokens.

2. Syntax Analysis:

This phase takes as i/p the tokens generated by lexical analysis phase and if the syntax is correct it generates a parse tree.

3. Semantic Analysis:

This phase performs the type checking and makes the necessary modifications in the parse tree representation.

4. Intermediate Code Generation:

This phase generates an intermediate code which helps to reduce the complexity of the code.

5. Code Optimization:

This phase performs the code optimization so that the resulting target prog. would get executed faster.

6. Code Generation:

This phase is responsible for the generation of target prog.

Note: if the target prog. is assembly lang then after compiler we need an assembler to generate a machine lang. prog.

Symbol Table:

It is used for keeping a track on the symbol's info.

Error Handler:

It is responsible for handling of the errors which can occur in any of the compilation phases.

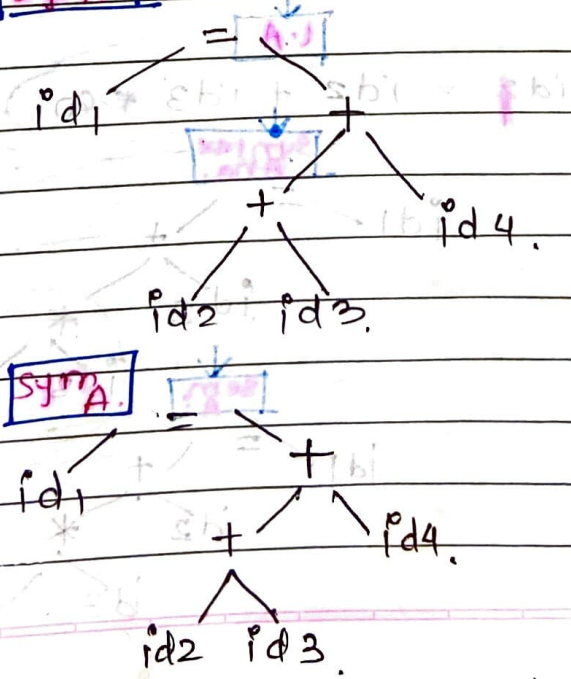
Eg 1:

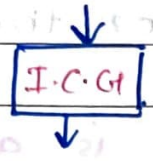
$a = b + c + d;$

L.A.

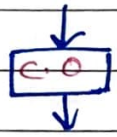
$id_1 = id_2 + id_3 + id_4$

SynA

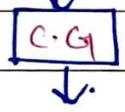




t1 = id2 + id3
t2 = t1 + id4
id1 = t2.



t1 = id2 + id3
id1 = t1 + id4

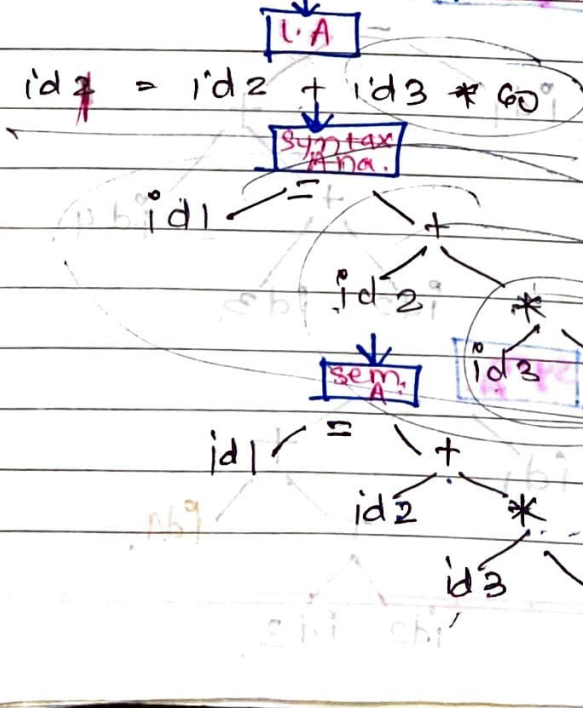



```
MOV R1, id2
MOV R2, id3
ADD R1, R2
MOV R2, id4
ADD R1, R2
MOV id1, R1
```

S.T.

ID	name	type	...
id1	a	int	...
id2	b	-	...
id3	c	-	...
id4	d	-	...
:	:	:	:

Eg.2: position = initial + rate * 60.



I.C.G

```
t1 = 60.0
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

C.O

```
t2 = id3 * 60.0
id1 = id2 + t2
```

C.G.

```
MOV R1, id3
MOV R2, 60.0
MOV R2, id2
ADD R1, R2
MOV id1, R1
```

S.T.

ID	name	type
id1	position	float
id2	initial	:
id3	rate	:

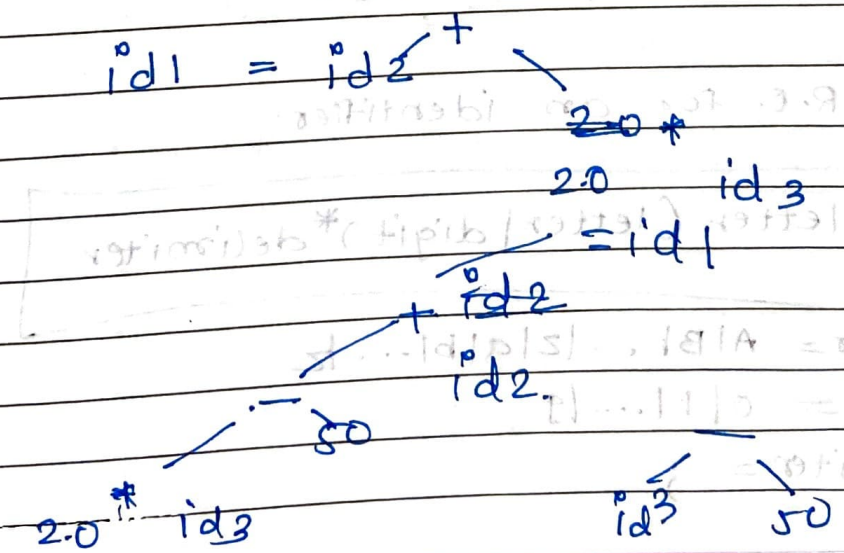
```
MOV R1, 60.0
MOV R2, id3
MUL R1, R2
MOV R3, id2
ADD R1, R3
MOV id1, R1
```

c = b + 2.0 * a - 50

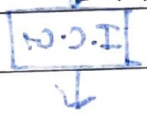
L.A.

id1 = id2 + 2.0 * id3 - 50.

S.A.



Lexical Analysis / Linear Scanning / Lexing:



1. It is the ONLY phase which is allowed to communicate with the original program.

2. The primary fxn of this phase is to generate stream of tokens provided the elements of the prog. are correct.
   eg of tokens are: Variables, operators, constants, etc.

3. The secondary fxns performed are
   → remove comments
   → remove whitespaces
   → Correlate errors.

4. Prog. that performs lexical analysis is called lexical analyser / scanner.

5. Regular Expressions are used for specifying tokens and NFA & DFA are used for recognizing tokens.

eg: R.E. for an identifier:

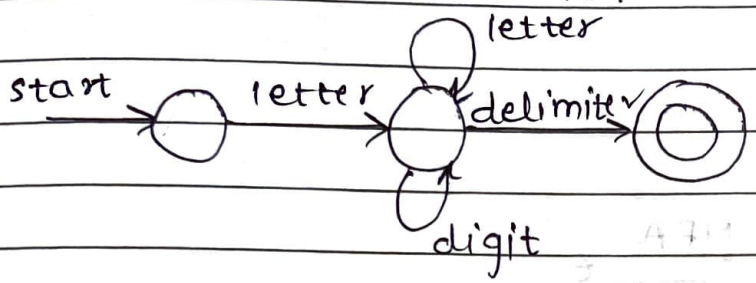
r = letter (letter | digit)* delimiter

letter = A|B|...|Z|a|b|...|z

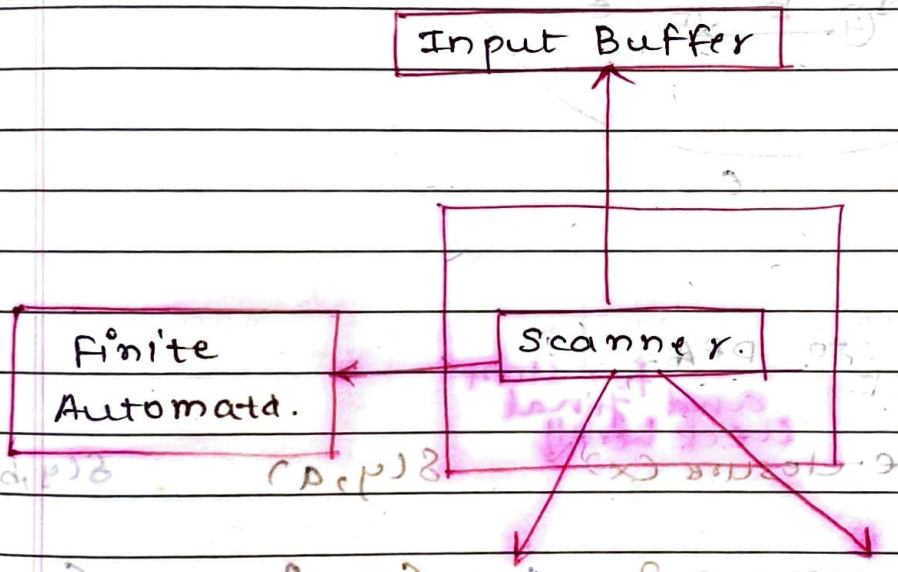
digit = 0|1|....|9

delimiter = ;

T.D. for an identifier.



Role of F.A. in compiler design / lexical Analysis



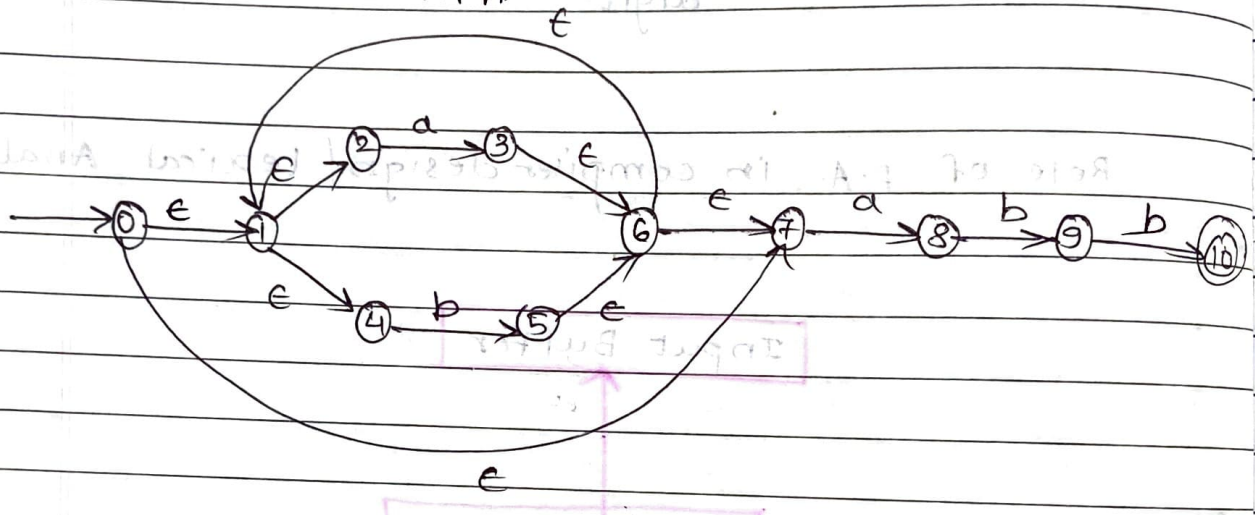
*Note: We can expect the conversion of

R.E → NFA → DFA → Min DFA

P] Design min. DFA for $r = (a+b)^* abb$

Soln:

S1: R.E to NFA



S2: NFA to DFA

X	Y = ϵ-closure(X)		δ(Y,a)	δ(Y,b)
{0}	{0,1,2,4,7}	A	{3,8}	{5}
{3,8}	{1,2,3,4,6,7,8}	B	{3,8}	{5,9}
{5}	{1,2,4,5,6,7}	C	{3,8}	{5}
{5,9}	{1,2,4,5,6,7,9}	D	{3,8}	{5,10}
{5,10}	{1,2,4,5,6,7,10}	E	{3,8}	{5}

δ:

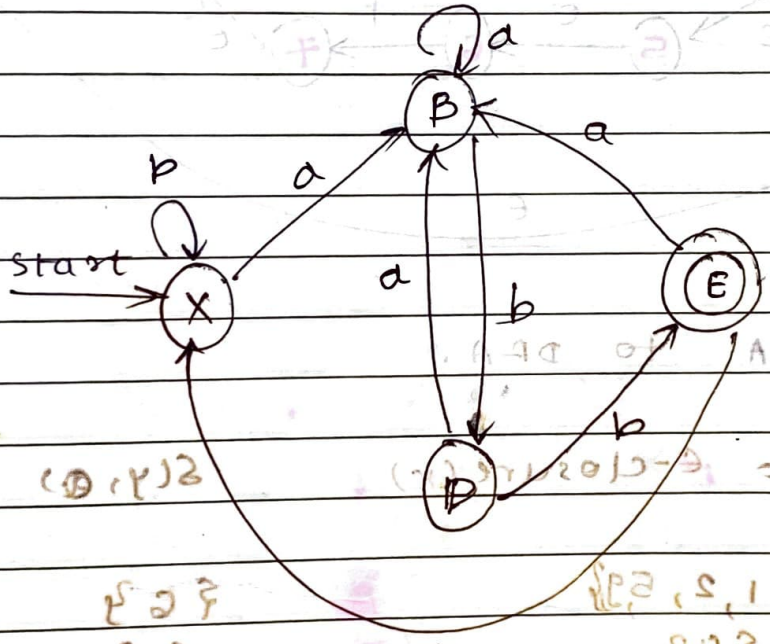
	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
E *	B	C

X = [A, C]

S3: DFA to Min DFA

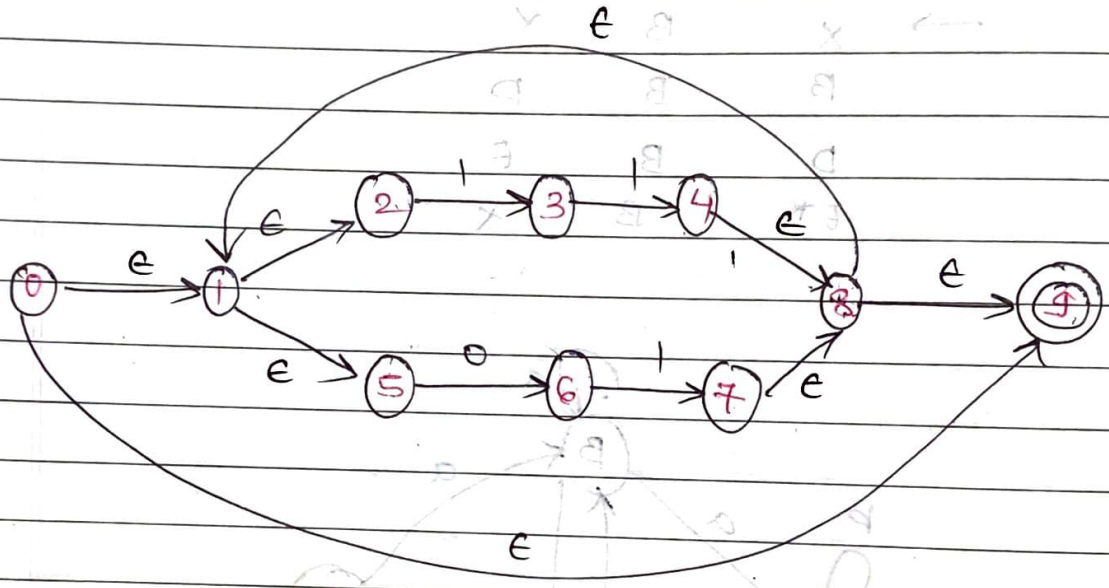
Q:

δ	a	b
→ X	B	X
B	B	D
D	B	E
E*	B	X



P] Construct F.A. for $r = (11 + 01)^*$

Soln: S1] R.E. to NFA.



S2] NFA to DFA.

x	y = ϵ-closure(x)		δ(y, 0)	δ(y, 1)
{0}	{0, 1, 2, 5, 9}	A	{6}	{3}
{6}	{6}	B	{4}	{7}
{3}	{3}	C	{}	{4}
{7}	{7, 8, 9, 1, 2, 5}	D*	{6}	{3}
{4}	{4, 8, 1, 2, 5, 9}	E*	{6}	{3}
{}	{}	F.	{}	{}

S:

Q\Σ	0	1
→ A*	B	C
B	F	D
C	F	E
D*	B	C
E*	B	C
F	F	F

A, D, E = X

{D, E} = X

S3: Minimized DFA.

Q\Σ	0	1
A	B	C
B	F	X
C	F	X
X*	B	C
F	F	F

{B, C} = Y

Q\Σ	0	1
→ A	Y	Y
Y	F	X
X*	Y	Y
F	F	F

Diag.

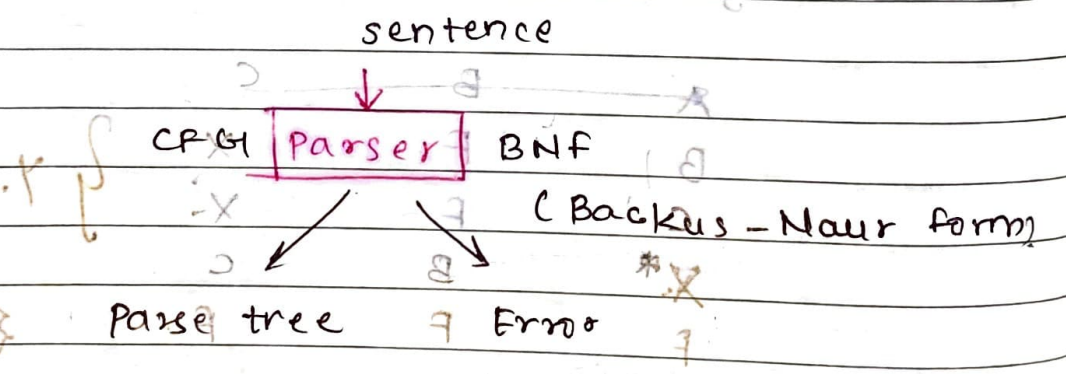
Syntax Analysis / Hierarchical Scanning / Parsing

1) This phase performs the check on the syntax of the statements written in the program.

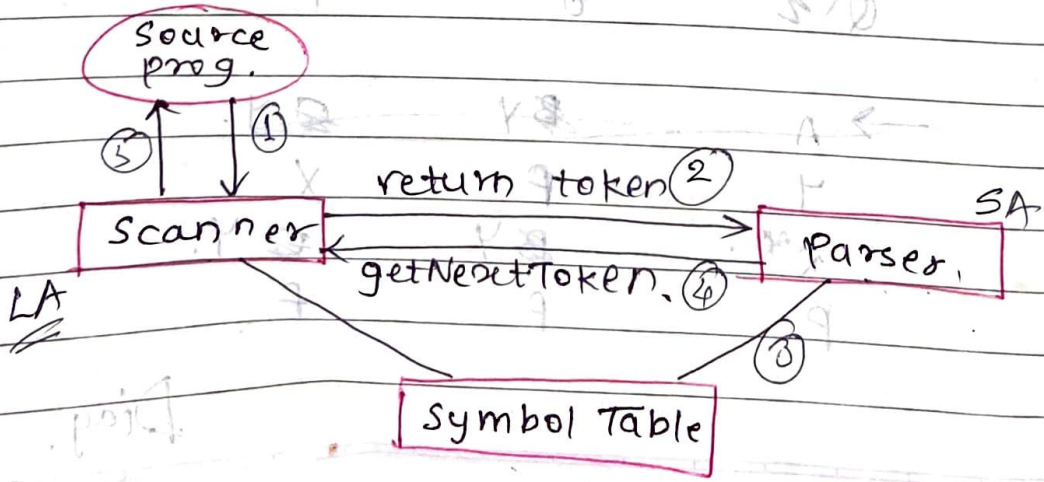
2) Prog. that performs syntax Analysis is called syntax analyser / Parser.

3) Defn of Parser:

★ Parser is a prog. that takes as i/p a sentence and if the grammar can derive the sentence it generates a parse tree. ~~Hence~~ else it generates an error.



Communication between Scanner & Parser



V - capital letter

PAGE No.
DATE

Classification of Grammar / Types of Grammar / Chomsky's Hierarchy:

As per chomsky there are 4 different types of Grammar.

Type	Grammar	Restriction
0	Unrestricted Grammar	$\alpha \rightarrow \beta$
1	Context Sensitive Grammar	$\lvert\alpha\rvert \leq \lvert\beta\rvert$
2	Context Free Grammar	$A \rightarrow \alpha$
3	Regular Grammar	A → any no. of Terminals & atmost 1V.

Parser 1:

Recursive Descent Parser

1. This parser follows top-down parsing technique.

2. It is called recursive because it is implemented with the help of functions which may work recursively.

3. It is called descent because it follows the top-down approach.

4. Parser will generate the parse tree if the following 2 conditions are satisfied:

   i.) I/P should be completely scanned.

   ii.) No fxn should call error fxn

Q] Design Recursive Descent Parser for the foll. grammar.

$S \rightarrow AB$

$A \rightarrow aA \mid b$

$B \rightarrow bB \mid a.$

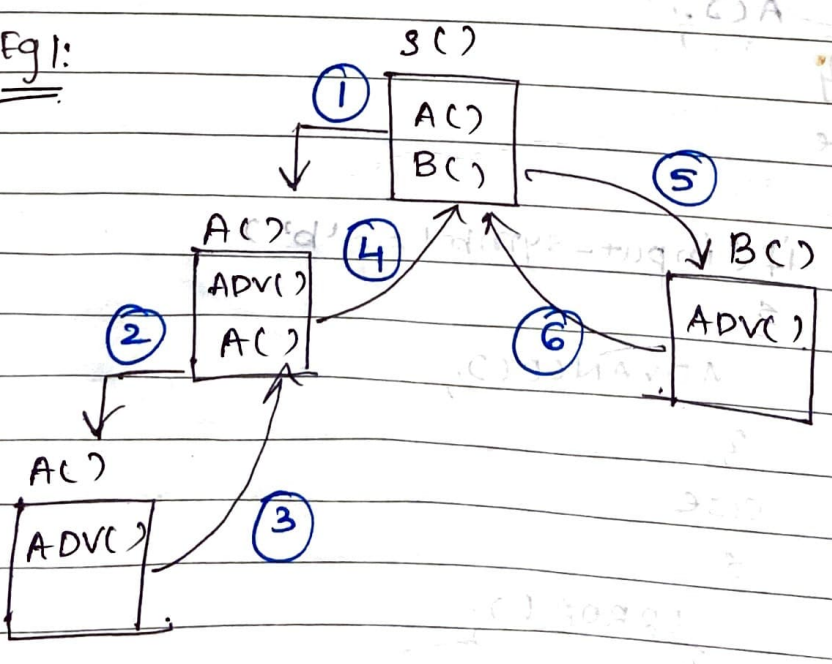
Soln:

```
function S()
{
    A();
    B();
}

function A()
{
  if (input.symbol == 'a').
    {
      ADVANCE();
      A();
    }
  else
    {
      if (input-symbol == 'b')
      {
        ADVANCE();
      }
      else
      {
        ERROR();
      }
    }
}
```

```
function B()
{
  if(input.symbol == 'b')
    {
      ADVANCE();
      B();
    }
  else
    {
      if (input.symbol == 'a')
       {
         ADVANCE();
       }
      else
       {
         ERROR();
       }
    }
}
```

Eg1:

aba.

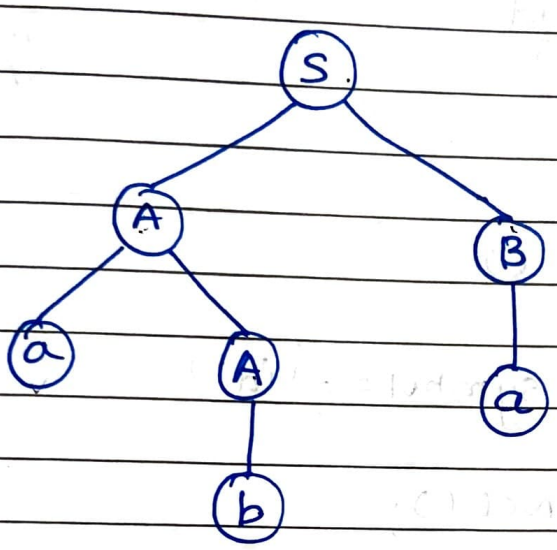


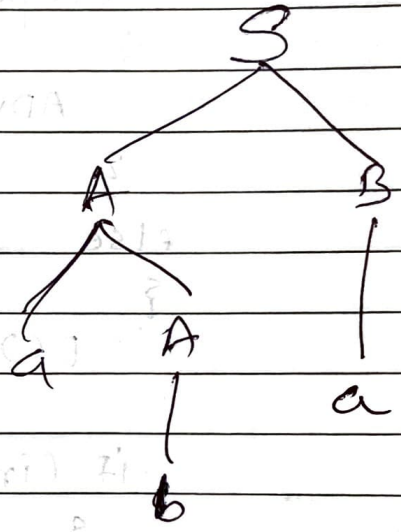
A B
a AB

PAGE No.
DATE

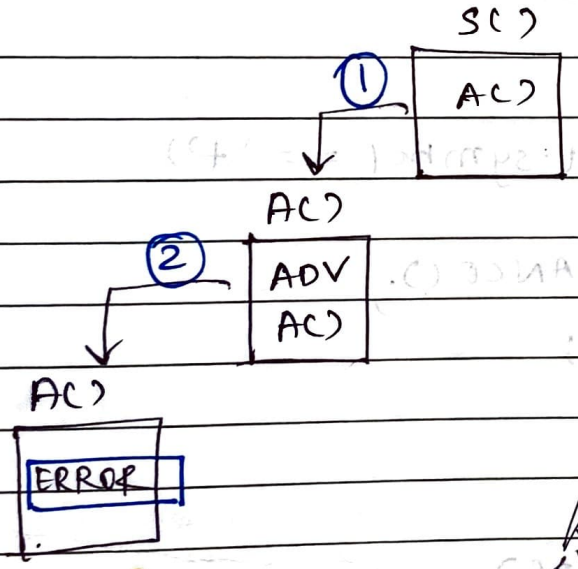
aba

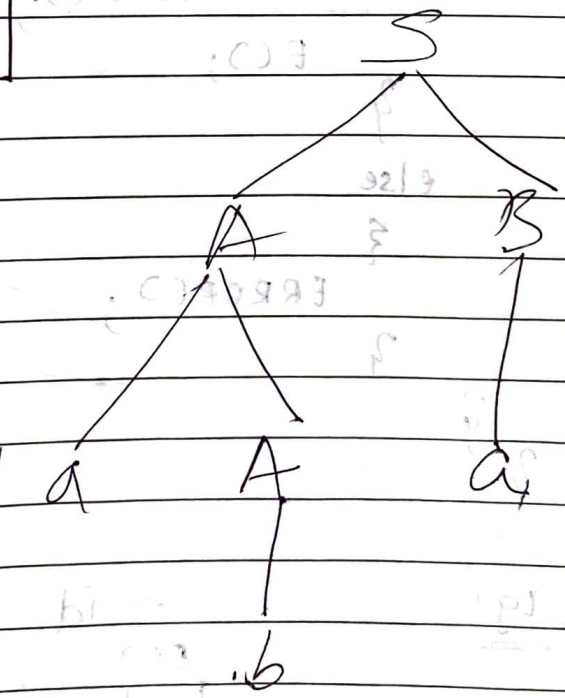
S → AB
→ aAB
→ abB
→ aba





Eg 2:





$E \rightarrow id\,E'$

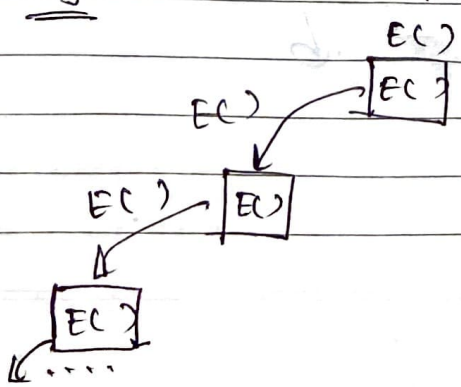
$E' \rightarrow +E\,E' \mid \epsilon$

P] $E \rightarrow \underbrace{E + E}_{\alpha} \mid \underbrace{id}_{\beta}$

Soln:

```
function E()
{
  if (input.symbol == 'id')
  {
     ADVANCE();
  }
  else
  {
     E();
     if (input.symbol == '+')
     {
        ADVANCE();
        E();
     }
     else
     {
        ERROR();
     }
  }
}
```

Eg: + id

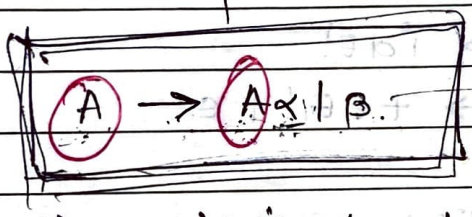


$A \rightarrow aW / a\beta$

The parser goes into an infinite loop because the given Grammar is left Recursive and hence not suitable for parser design.

* Left Recursion of it's Elimination.

Defn: Grammar is said to be left recursive if there is a production of the form:



Elimination: To eliminate L.R. we must recorite L.R. prodn as:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon.$

General Format:

L.R: $A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | \alpha_r.$

$A \rightarrow \beta_1 | \beta_2 | \ldots | \beta_r.$

E.R: $A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_3 A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_r A' | \epsilon.$

P] $E \rightarrow E+E \mid id$ (α = $+E$, β = id) ⟹ $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$.

Soln:

$E \rightarrow E+E \mid id$

$A \rightarrow A\alpha \mid \beta$

After elimination of L.R. :-

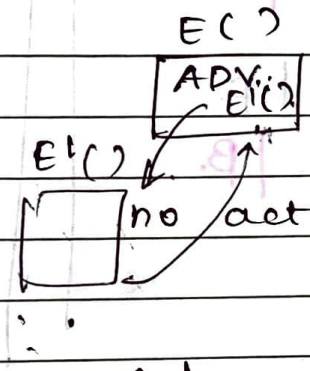
$E \rightarrow idE'$

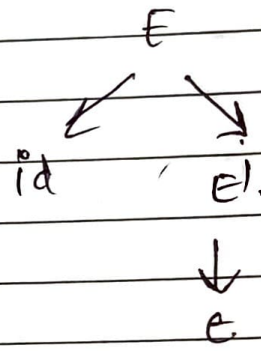
$E' \rightarrow +EE' \mid \epsilon$

```
function E()
{
  if (input symbol == 'id')
    {
      ADVANCE ();
      EDASH();
    }
  else
    {
      ERROR();
    }

function EDASH()
{
  if (input symbol == '+')
    {
      ADVANCE();
      E();
      EDASH();
    }
```

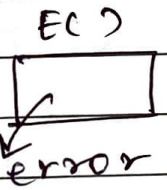
```
else
{
    /* no action */
}
}.
```

Eg: id

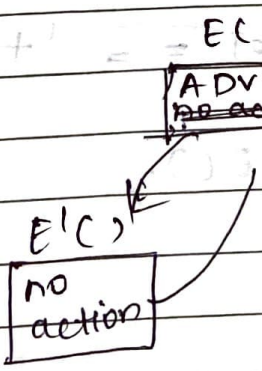




Eg 2: +id.



Eg 3: idid.



$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$.

P] $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$.

Soln:

$E \rightarrow E+T \mid T$.

$A \rightarrow A\alpha \mid \beta$.

$T \rightarrow T*F \mid F$.

$A \rightarrow A\alpha \mid \beta$.

$F \rightarrow id$.

Elim:

$E \rightarrow TE'$.

$E' \rightarrow +TE' \mid \epsilon$.

$T \rightarrow FT'$.

$T' \rightarrow *FT' \mid \epsilon$

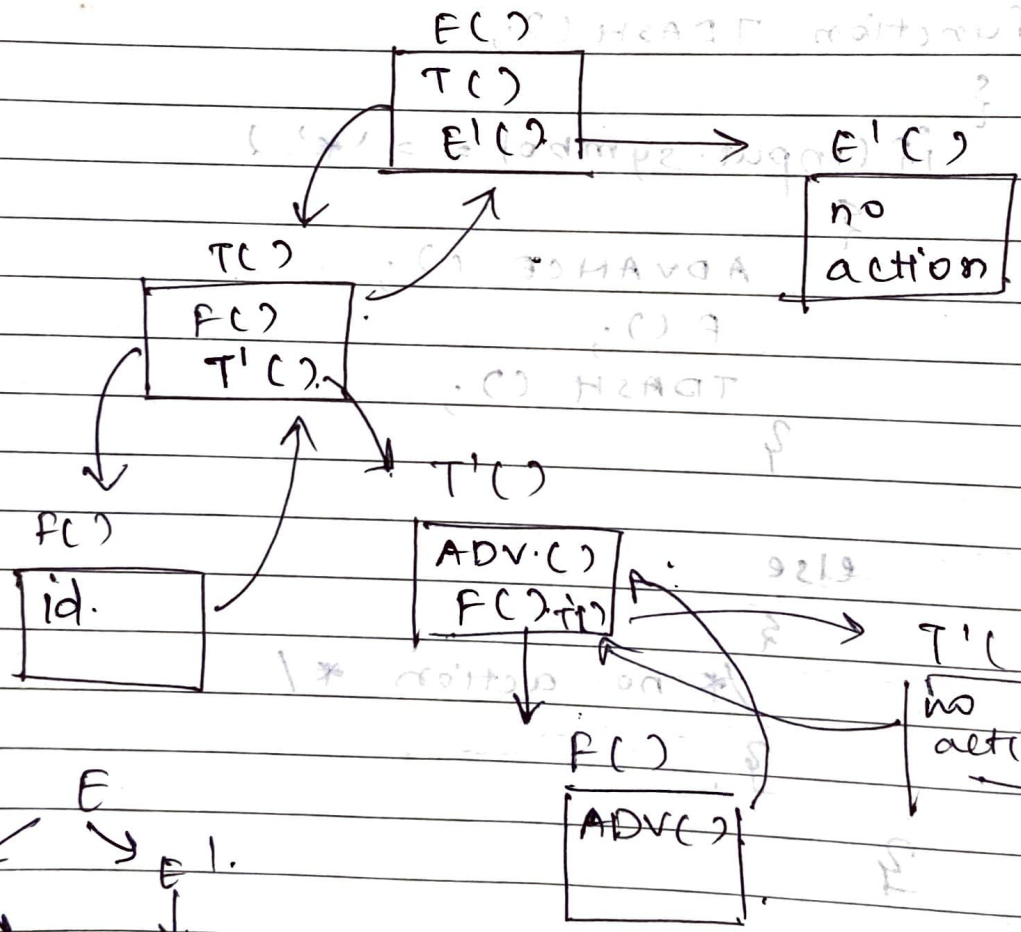
$F \rightarrow id$.

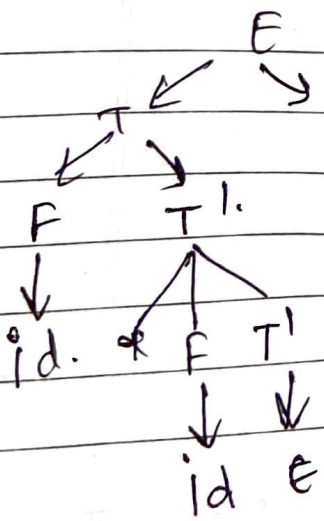
```
function E()
{
    T();
    E'();
}

function EDASH()
{
  if (input symbol == '+')
    {
      ADVANCE();
      T();
      E'();
    }
else
    {
```

```
*/ no action/*
    }.
}.

function T ().
{
    F ().
    T' ();
}.

function TDASH ();
{
  if (input.symbol == = '*')
      {
        ADVANCE ().
        F ().
        TDASH ().
      }

    else
      {
        /* no action */
      }.

  }

function F ()
{
  if (input. symbol = = 'id')
      {
        ADVANCE ();
```

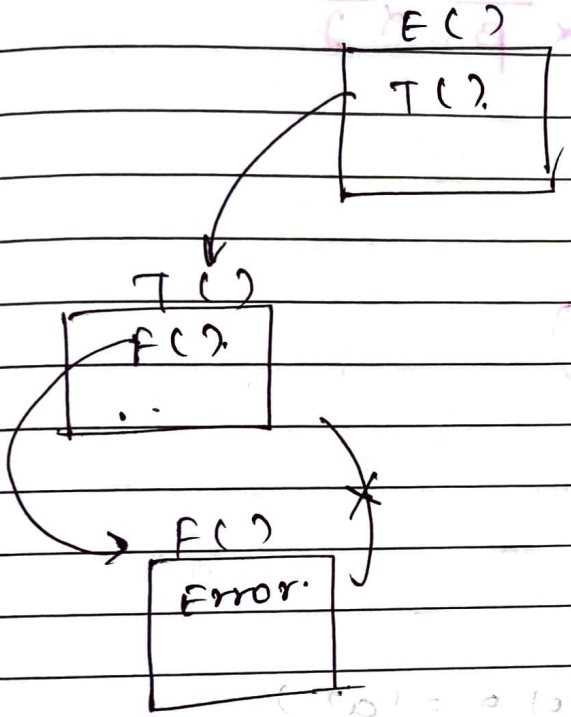
```
}
else
{
    ERROR
};
}
```

Eg 1: id * id.





Eg2: + id.



Left Factoring.

It implies factoring out of the common prefixes.

Suppose the prodn of A are:

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

the above can be left factored as:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

P] $S \rightarrow asb \mid ab$

Soln: $S \rightarrow a\underline{Sb} \mid \underline{ab}$

$A \rightarrow \alpha\ \beta \mid \alpha$

After L.F.

$S \rightarrow as'$

$S' \rightarrow Sb \mid b$

```
function S()
{

if (input symbol == 'a')
 {
    ADVANCE();
    S'();
 }

else
 {
    error();
 }
}

function S'()
{
   S();
if (input symbol == 'b')
   { ADVANCE();
   }
```

start with Terminal when u have such option

```
else
{
    S();
    if (input.symbol == 'b')
    {
        ADVANCE();
    }
    else
    {
        ERROR();
    }
}
}
```

5

FIRST (α):

It is defined as the set of terminals that begin with strings derived from α (α - some sentential form).

R1: If $X \to a\alpha$, then FIRST (X) = {a}.

R2: If $X \to \epsilon$, then FIRST (X) = {ε}

R3: If a is a terminal, FIRST (a) = {a}

R4. If $X \to Y_1 Y_2 \dots Y_k$ then FIRST(X) = FIRST (Y1)

* If FIRST (Y1) contains ε, then FIRST (X) = FIRST(Y1) − {ε} + FIRST(Y2).

Note: If FIRST (Y_j) contain ε, then FIRST(X) = {ε}, j = 1 to k.

a, b, c, ϵ

Eg:

S → ABC
A → a | ϵ
B → b | ϵ
C → c | ϵ

A → aAB | bBA | G

A → aA | B

A → aA

A → B

S = {a, b, c, ϵ}

first(S) = first(A) = {a, ϵ}

= f(A) − ϵ + f(B)

= {a, b, ϵ}

Union of B and A?

FIRST(a) = {a}

FIRST(B A) = {b, a, ϵ}

" (b) = {b}

" (A C) = {a, c, ϵ}

" (c) = {c}

(B a) = {b, a}

" (A) = {a, ϵ}

(a B) = {a}

" (B) = {b, ϵ}

(b A c) = {b}

" (c) = {c, ϵ}

(S) = {a, b, c, ϵ}

★ FOLLOW(A):

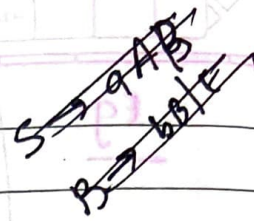
It is defined as the set of terminals that can appear on the R.H.S. of A.
(A - variable)

R1: If y is start variable then
FOLLOW(y) = {$}

R2: If X → xY then
FOLLOW(Y) = FOLLOW(X)

S → aAB
A → aA | a
B → bB | ϵ

RB: If $X \to \alpha Y \beta$ then
FOLLOW(Y) = FIRST(β)



Note: If FIRST(β) contains ϵ, then
FOLLOW(Y) = FIRST(β) − {ϵ} + FOLLOW(X)

Eg: S → ABC
A → a | ϵ
B → b
C → c | ϵ

{a, ϵ} − {ϵ} + {b}
a, b

FIRST(S) = {a, b}
" (A) = {a, ϵ}
" (B) = {b}
" (C) = {c, ϵ}

FOLLOW(S) = {$}
FOLLOW(C) = FOLLOW(S)
S → ABc = {$}
X → α Y
FOLLOW(B) = FIRST(C)
S → ABC = {c, ϵ} − {ϵ} + FOLLOW(S)
X → α Y B
= {c, $}
FOLLOW(A) = FIRST(BC)
S → ABc = {b}
X → Y B

FOLLOW(A) = {b}
FOLLOW(B) = FIRST(C) = {c, ϵ}
= {c} ∪ {ϵ} × FOLLOW(S)
ϵ
{c}

), $ | + , ε
+ ,) , $

Eg: $E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow (E) \mid id$

FIRST (T) = { (, id }
FIRST (E') = { +, ε }
FIRST (E) = FIRST T
= { (, id }

FOLLOW (T) = FIRST (E')
= { +, ε } − ε + FW (E)
+ ,) , $

FIRST (E) = { (, id }

" (E') = { +, ε }

" (T) = { (, id }

FOLLOW (E) = { $,) }
$T \rightarrow (E)$.
$X \rightarrow \alpha Y \beta$.

FOLLOW (E') = { $,) }
$E \rightarrow TE'$
$X \rightarrow \alpha Y$

FOLLOW (T) = { +, $,) }
$E' \rightarrow TE'$
$X \rightarrow Y\beta$.

$E' \rightarrow +TE'$
$X \rightarrow \alpha Y$. ?

$E' \rightarrow +TE'$
$X \rightarrow \alpha Y \beta$.
(+, $,))

P1] Eg. $S \rightarrow ABC$

$A \rightarrow a$.

$B \rightarrow b \mid \epsilon$

$C \rightarrow c$

Soln.

FIRST(S) = {a}

" (A) = {a}

" (B) = {b, ε}

" (C) = {c}.

FOLLOW(S) = {$}

" (C) = {$}.

$S \rightarrow ABC$

$X \rightarrow \alpha y$.

FOLLOW (B) = {c}

$S \rightarrow ABC$:

$X \rightarrow \alpha y \beta$

FOLLOW (A) = {b, c}.

$S \rightarrow ABC$.

$X \rightarrow y\beta$

$S \rightarrow ABC$

$X \rightarrow \alpha \beta$

= FIRST (BC)

= b, ε → ε + c

= {b, c}

Q7.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

$*, \epsilon$

$+, \epsilon,), \$$

Soln.

FIRST (E) = { (, id }.

" (E') = { +, ∈ }.

" (T') = { *, ∈ }

" (F) = { (, id }

FIRST (T) = { (, id }

FOLLOW (E) = {), $ }

$F \rightarrow (E)$

$X \rightarrow \alpha Y \beta$.

" (E') = { $,) }.

$E \rightarrow TE'$

$X \rightarrow \alpha Y$

$E' \rightarrow +TE'$.

$X \rightarrow \alpha Y$.

(T) = { +, $,) }.

$E \rightarrow TE'$

$X \rightarrow YB$

$E' \rightarrow +TE'$.

$X \rightarrow \alpha Y B$.

(T') = { +, $,) }.

?? (F) = { *, +, $,) }

+, -, $,)

β3].

E → TQ
T → FR
Q → +TQ | −TQ | ϵ
R → *FR | /FR | ϵ
F → (E) | id

T ⇒ (+, −,), $)

Soln:

FIRST (E) = { (, id }.

FIRST (T) = { (, id }.

FIRST (Q) = { +, −, ϵ }.

FIRST (R) = { *, /, ϵ }.

FIRST (F) = { (, id }

FOLLOW (E) = { $,) }

E → (E) $
∝ γ β.

FOLLOW (T) = { +, −,), $ }

~~T → FR~~

E → TQ.
X → γ . β

Q → + T Q. | − T Q.
X → . ∝ γ β ∝ γ β.

FOLLOW (T) = { +, −,), $ }

FOLLOW (Q) = {), $ }

E → TQ
X → ~~γ β~~ ∝ . γ

Q → + T Q | − T Q
X → ~~∝ β~~ γ . ∝ γ .

FOLLOW (R) = { +, −,), $ }

T → FR.
X → ∝ γ

R → * F R | / F R.
. X → ∝ γ ∝ . γ .

FOLLOW (F) = { *, /, +, −,), $ }

T → F R.
X → ∝ γ β

R → * F R | / F R.
∝ γ β ∝ γ . β

= { }

Parser No. 2: Predictive Parser.

1. This parser follows top-down parsing technique.
2. The model of predictive parser is as shown.

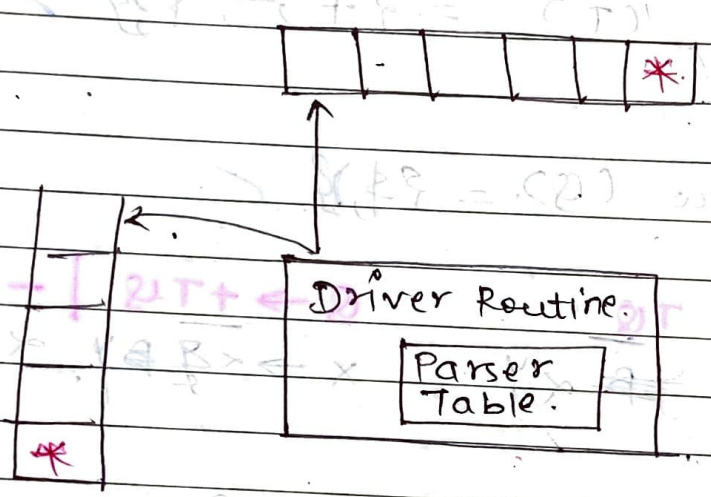


Fig: Model of Predictive Parser.

1. Input buffer
   It contains a $ to indicate the end of i/p.
2. stack.
   It contains a $ to indicate the bottom of the stack.

3. Parser Table.
It is a 2-D table which indicates the actions to be carried out by the parser.

4. Driver Routine.
It is the function that drives the parser.

Note: Predictive Parser is also called.

LL(1) parser.

- L → left to right scanning
- L → L.M.D.
- (1) → number of lookahead symbols.

Steps for Designing Predictive Parser:

S1: Perform elimination of Left Recursion and perform left factoring

S2: find FIRST & FOLLOW of all variables.

S3: Design Predictive Parser Matrix/Table

S4: Write Predictive Parser Algorithm.

S5: Give some Examples.
(1- valid & 1- invalid)

P] Design P.P. for:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to (E) \mid id$

FIRST	FOLLOW
FIRST(E) = {(, id}	FOLLOW(E) = {), $}
FIRST(E') = {+, ε}	FOLLOW(E') = {), $}
FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}

$id + TE'$

$id + id\ E'$

$id + id + TE'$

$id + id + \epsilon$

id+id+id

Note:

S1: L.R: $A \to A\alpha \mid \beta$ ✗

L.F: $A \to \alpha\beta \mid \alpha\gamma$ ✓

(done).

S2:

FIRST (E) = { (, id }

FOLLOW (E) = { $,) }

" (E') = { +, ε }

" (E') = { $,) }

" (T) = { (, id }

" (T) = { +, $,) }

S3:

V \ T	id	+	(	)	$
E	$E \to TE'$		$E \to TE'$		
E'		$E' \to +TE'$		$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to id$		$T \to (E)$		

S4: repeat forever. Algo

{ let 'X' be the stacktop symbol and let 'a' be the i/p symbol.

if X = a = $ then Accept & break.

else if X = a ≠ $ then Pop 'X' & remove 'a'

else if M[X, a] = X → Y1 Y2 ... Yk then Pop 'X'.
and Push Yk Yk-1 ... Y1

else ERROR()

}

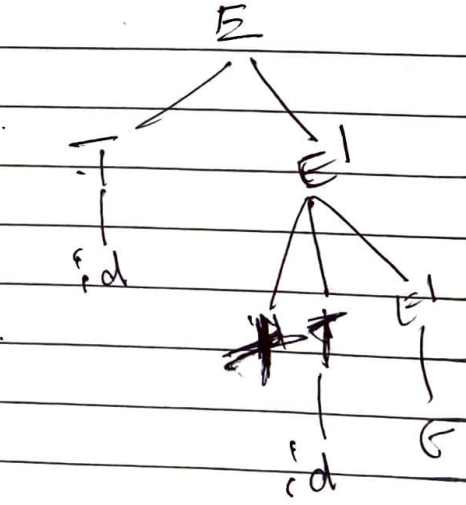
S5: Eg:

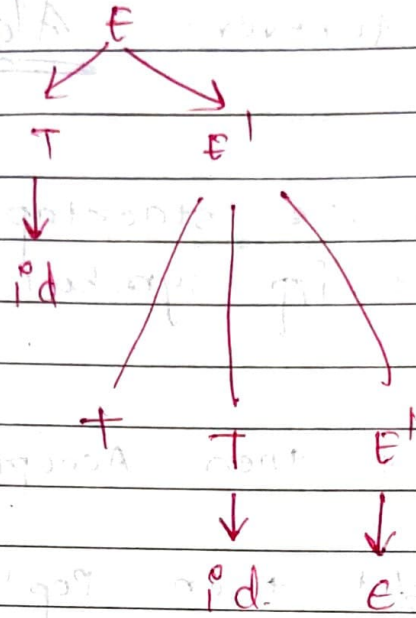
Stack	Input
$ E	id + id $
$ E' T	id + id $
$ E' id	id + id $
$ E'	+ id $
$ E' T +	+ id $
$ E' T	id $
$ E' id	id $
$ E'	$
$	$

Accept.

id + id

E → T E'
→ id E'
→ id + T E'
→ id + id E'
→ id + id
→ id + id



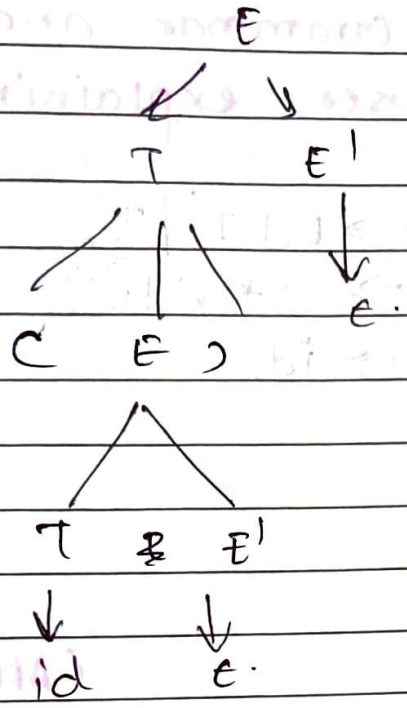


eg2:

stack	Input
$E	+id $
	Error.

eg 3:

stack	Input
$ E	id > $
$ E' T	id > $
$ E' > F C	id > $
$ E' > E	id > $
$ E' > F' T	id > $
$ E' > E id	id > $
$ E' > E'	> $
$ E' >	> $
$ E'	$
$	$
	Accept



= (id ε) ε. = (id)

Stack	Input
$ E	id + id + id $
$ E'T	id + id + id $
$ E' id	id + id + id $
$ E'	+ id + id $
$ E'T +	+ id + id $
$ E'T	id + id $
$ E' id	id + id $
$ E'	+ id $
$ E'T +	+ id $
$ E'T	id $
$ E' id	id $
$ E'	$
$ ε	$
$	$

Dec 2012 [10M]

P] Modify the Grammar and construct predictive Parser explaining each step:

$E \rightarrow E+T \mid T$ (α = $+T$, β = T)
$T \rightarrow T*V \mid V$ (α = $*V$, β = V)
$V \rightarrow id$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow VT'$
$T' \rightarrow *VT' \mid \epsilon$
$V \rightarrow id$

Solution:

S1:

(Elim. of L.R)

$E \rightarrow E+T \mid T$
$A \rightarrow A\alpha \mid \beta$

$T \rightarrow T*V \mid V$
$A \rightarrow A\alpha \mid \beta$

$V \rightarrow id$

(After elim. of L.R.)

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$.

$T \rightarrow VT'$
$T' \rightarrow *VT' \mid \epsilon$
$V \rightarrow id$.

S2:

FIRST(E) = {id}

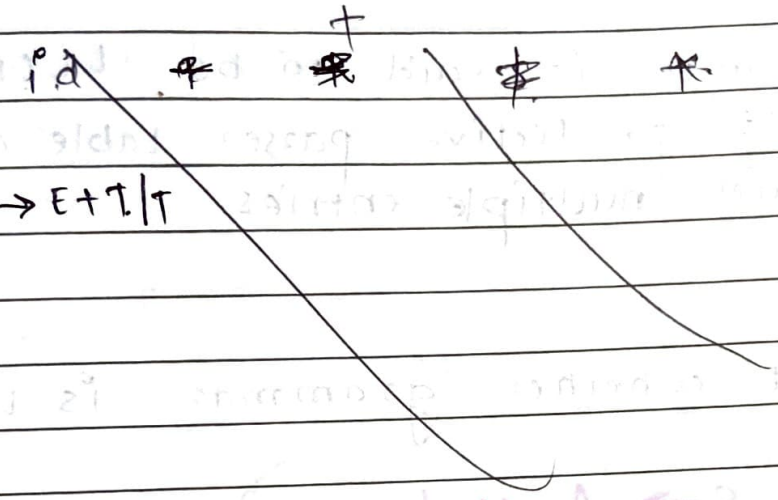
FOLLOW(E) = {$}

(E') = {+, ε} → (E') = {$}

(T) = {*, ε}
(T') = {id}
(V) = {id}

(T) = {+, $}
(V) = {*, +, $}

S3:



	id	+	*	$
E	$E \to E+T \mid T$			
E'				
T				
T'				
V				

	id	+	*	$
E	$E \to TE'$	~~+TE'~~		~~+TE'~~
E'		$+TE'$		$E' \to \epsilon$ ~~+TE'~~
T	$T \to VT'$	~~VT'~~	~~VT'~~	VT'
T'		$T' \to \epsilon$ ~~*VT'~~	$T' \to *VT'$	$T' \to \epsilon$
V	$V \to id$			

Note:

Grammar is said to be LL(1) grammar if it's predictive parser table does not contain multiple entries.

P] Test whether grammar is LL(1).

$S \to AaAb \mid BaBb$

$A \to \epsilon$

$B \to \epsilon$

$S \to AaAb$

$S \to \alpha\ \beta$

Follow(A) = FIRST(aAb)

= a

$S \to A\,a\,A\,b$

$\alpha\ \beta$

Follow(A) = b

S1: (done)

S2: FIRST(S) = {a}

" (A) = {ε} → (A) = {a, b}

(B) = {ε} → (B) = {a, b}

FOLLOW(S) = {$}

S3:

	a	b	$
S	S → AaAb S → BaBb		
A	A → ε	A → ε.	
B	B → ε	B → ε.	

Since, it contains multiple entries, it is not LL1.

[10M]

P] Test whether Grammar is LL(1).

S → AB | gDa
A → ab | c
B → dC
C → gC | g
D → fD | g

Soln:

(No left recursion)

S → AB | gDa.
A → ab | c
B → dC.

C → gC | g.
A → αβ | αγ.

(By left factoring)
C → g.C1.
C1 → .C | ε

D → fD | g.

S2: FIRST(S): {a, c, g}.

FIRST(A): {a, c}.

(B): {d}.

(C): {g, ε}

(D): {f, g}.

FOLLOW.

S: {$}

A: {d}
S → AB.
X → αYB

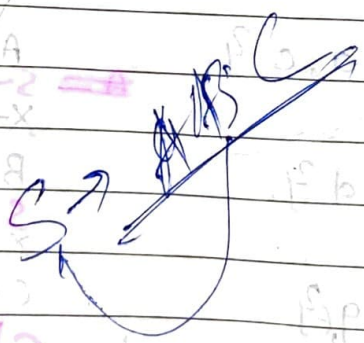
B: {$}.
S → AB
X → αY

C: {$}.
C1 → C | ε
αYβ.

D: {$}.
S → gDa, fD
X → αY

C1: {$}.

	a	b	c	d	f	g	$
S	AB S→gDa		AB S→gDa			S→gDa.	
A	A→ab		A→c.				
B		∅		B→dc.			
C						C→gc'	
c'				C'→c.			c'→$.
D.					D→fD	D→g.	



(Def 13 – 5M).

Parser 3: Shift Reduce Parser:

This parser follows the bottom-parsing technique.

Defn of handle:

It is a string that appears on the R.H.S. of the prodn and can be replaced by the L.H.S. of the prodn in a process moving towards start variable.

Defn of handle pruning:

It is the process of selecting and reducing the handle.

Actions in Shift Reduce Parser:

1. **Shift:** shift i/p terminal on stack.
2. **Reduce:** Replace R.H.S. by L.H.S. of prodn.
3. **Accept:** I/P is valid and hence parse tree.
4. **Reject:** I/P is invalid and hence ERROR.

Shift Reduce Parser delivers the string using R.M.D. in reverse which is called canonical derivation

Initial and Final conditions.

	Stack	Input
Initial	$	w $
Final	$ S	$

Shift Reducing Parsing Algo:

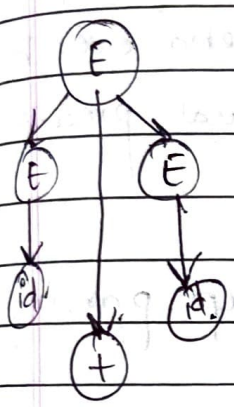
→ shift zero/more terminals on the stack until a handle is found.

→ Reduce the handle.

→ Repeat the above steps. Until the i/p is completely scanned and stack contains only start variable.

OR. Error.

Eg: $E \rightarrow E+E \mid E*E \mid id$



Stack	Input	Actions
$	id + id $	Shift id.
$ id	+ id $	Reducing $E \rightarrow id$.
$ E	+ id $	Shift +
$ E +	id $	Shift id
$ E + id	$	R U $E \rightarrow id$
$ E + E	$	R U $E \rightarrow E+E$
$ E	$	Accept.

(2)

Stack	Input	Action
$	* id $	S *
$ *	id $	S id.
$ * id.	$	R U $E \rightarrow id$
$ * E	$	Reject.

X

stack	Input	Actions
$	id + id $	Shift id
$ id	+ id $	Red. using $E \rightarrow id$
$ E	+ id $	Shift +
$ E +	id $	shift id
$ E + id	$	Red. using $E \rightarrow id$
$ E + E	$	Red. using $E \rightarrow E+E$
$ E	$	Accept.

Parser No. 4:

P4: Operator Precedency Parser: (OPP).

Relations	Hoco to read it.
$a \lessdot b$	a gives precedence to b
$a \gtrdot b$	a takes precedence over b
$a \doteq b$	a & b have equal precedence.

- This parser follows bottom-up parsing technique.
- Defn of operator grammar:

Operator grammar is a CFG that satisfies the foll. two conditions:

1) It does not contain any null production.
2) Two variables should not appear together.
3) The Parser works on the foll. precedence relations.

Relations.	Hoco to read it.
$a \lessdot b$	a gives precedence to b
$a \gtrdot b$	a takes precedence over b
$a \doteq b$	a & b have equal precedence.

Parser No-4:

Steps for designing OPP:

1. Design Precedence Reln table (Matrix) for the given operator grammar.
2. Write Operator precedence parsing algorithm.
3. Give some examples.

Q. Design OPP for:

$E \rightarrow E+E \mid E*E \mid id.$

Sl:

	id	+	*	$
id	—	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	Accept.

id * id

$

S2:

repeat forever.
{

if i/p is completely scanned
and stack contain only
start variable
then accept & break.
else

let 'a' be the topmost stack terminal
&
let 'b' be the i/p terminal
if $a \lessdot b$ OR $a \doteq b$ then SHIFT
else if $a \gtrdot b$ then REDUCE
else ERROR ()
}.

S3: Eg:

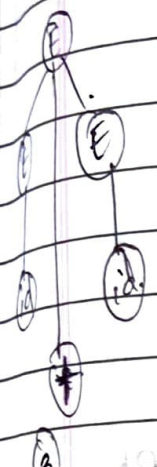
(i)

Stack	Input	a	b	Action
$	id * id $	$	id	S id.
$ id	* id $	id	*	R U E→id.
$ E	* id $	$	*	S *.
$ E *	id $	*	id	S id
$ E * id	$	id	$	R U E→id.
$ E * E	$	*	$	R U E→E*E
$ E.	$	$	$	Accept.

Shift → Terminal.

Reduce → Variable.

Stack.	Input.	a	b	Action.
$	id id $	$	id	s id.
$ id.	id $	id.	id.	Reject.



stack	Input	a	b	Action.
$	id+id * id $	$	id	s id.
$ id	+ id x id $	id	+	R U E * id.
$ E	+ id x id $	$	+	s +
$ E +	id * id $	+	id	s id.
$ E + id	* id $	id	*	R U E → id.
$ E + E	* id $	+	*	s *.
$ E + E	id $	*	id	s id.
$ E + E * id	$	id	$	R U E → id
$ E + E * E	$	*	$	R U E → E * E
$ E + E	$	+	$	R U E → E + E.
$ E	$	$	$	Accept.

P] $E \rightarrow E+E \mid E*E \mid id$

Soln:

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	Accept.

P] $E \rightarrow E+E \mid E-E \mid E \uparrow E \mid id.$

Soln:

	id	+	−	↑	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
−	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
↑	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	Accept.

p] $E \rightarrow E + E \mid (E) \mid id$

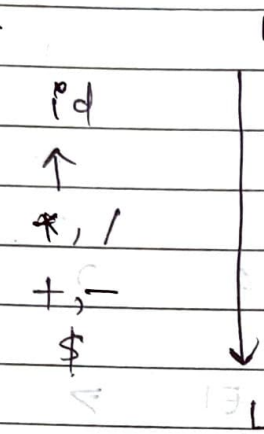
soln:

	id	+	(	)	$
id	E1	>	E1	>	>
+	<	>	<	>	>
(	<	<	<	=	E2.
)	E1	>	E1	>	>
$	<	<	<	E2	Accept.

p] $E \rightarrow E * E \mid E / E \mid (E) \mid id.$

	id	*	/	(	)	$.
id	E1	>	>	E1	>	>
*	<	>	>	<	>	>
/	<	>	>	<	>	>
(	<	<	<	<	=	E2.
)	E1	>	>	E1	>	>
$	<	<	<	<	E2	Accept.

Note:



Error Description.

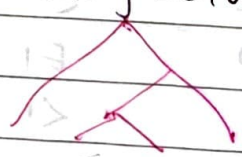
E1 → Missing Operator.

E2 → Missing Paranthesis.

Q.] With the help of full. Grammar. Explain the role of opp.

E → E + T | T
T → T * V | V.
V → a | b | c | d.

May 2010 / 10M.



string to be parsed :- " a + b * c * d "

soln:

For simplicity, we recorite the above Grammar by considering operator only.

$E \rightarrow E+E \mid E*E \mid a \mid b \mid c \mid d$.

S1:

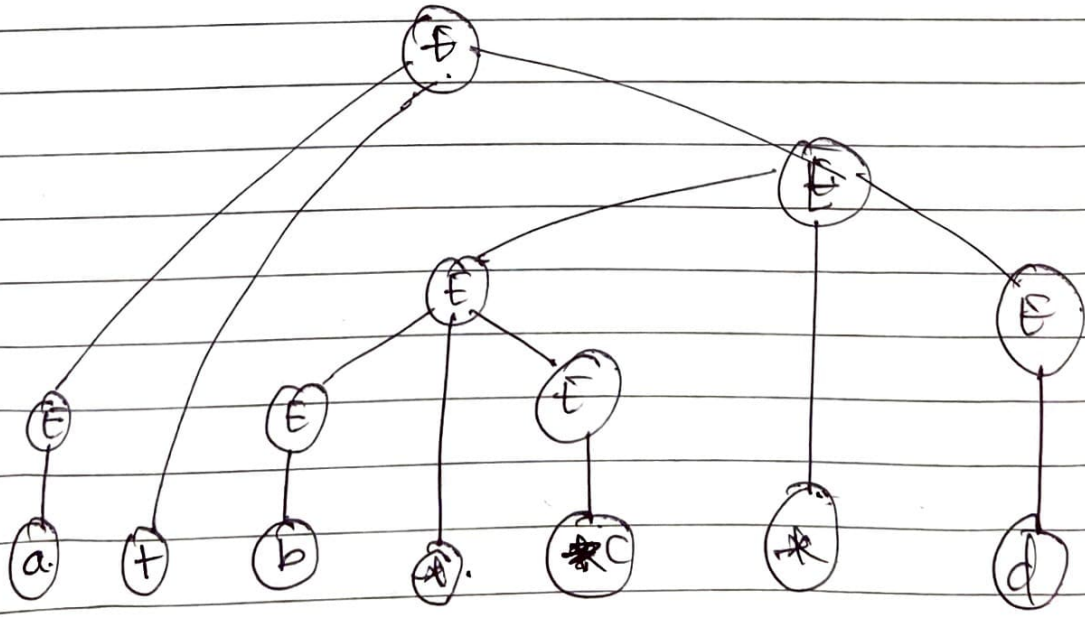
	a	b	c	d	+	*	$
a					>	>	>
b					>	>	>
c					>	>	>
d					>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
$	<	<	<	<	<	<	Accept

E1

S2:

S3:

	stack	Input	a	b	Action.
1.	$	a+b*c*d$	$	a	shift a.
2.	$ a	+b*c*d$	a	+	Reduce using E→a.
3.					
3.	$ E	+b*c*d$	$	+	shift +
4.	$ E +	b*c*d$	+	b	S b
5.	$ E+b	*c*d$	b	*	R U E→b
6.	$E+E	*c*d$	+	*	S *.
7.	$E+E*	c*d$	*	c	S c.
8.	$ E+E*c	*d$	c	*	R U E→c.
9.	$E+E*E	*d$	*	*	R U E→E*E.
10.	$E+E	*d$	+	*	S *.
11.	$E+E*	$	*	$	R U E→d.
12.	$E+E*d.	$	*d	$	R U E→E*E
13.	$E+E	*E.	$	+	$. R U E→E+E.
14.	$ E+E.				
15.	$E.	$	$	$.	



Module 6: Compilers

Introduction:

Compiler is a translator that takes as input the source language that is written in High Level Language and produces as output an assembly language or machine language.

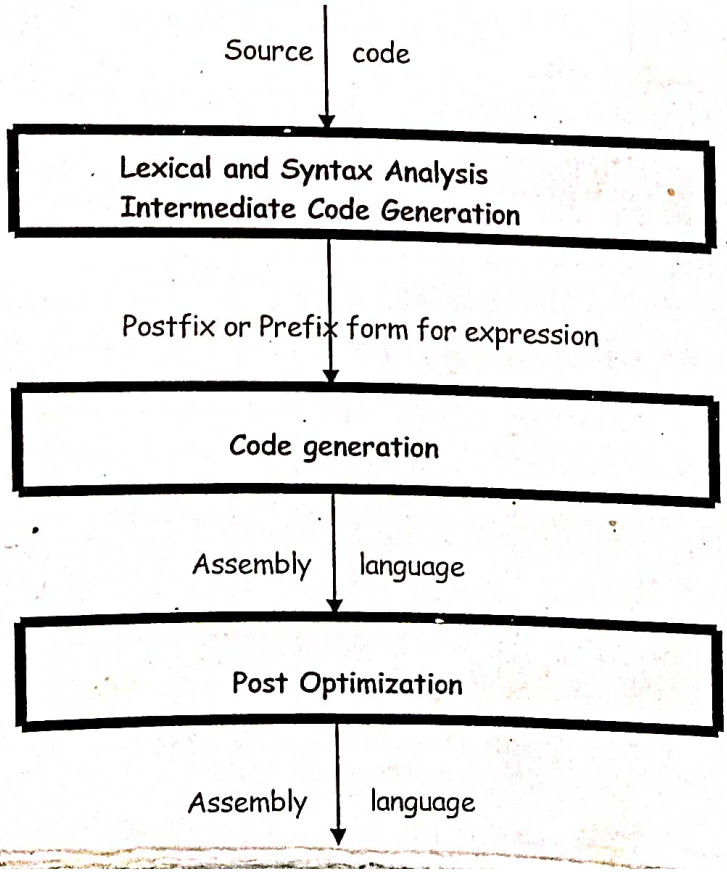
Structure of Compiler:

The process of compilation is very complex and hence it is partitioned into a series of sub processes called phases.

(Refer Class Notes)

The C Compiler:

C is a general purpose programming language designed by D.M Ritchie and is used as the primary programming language on the UNIX O.S. UNIX itself is written in C and has been moved to a number of machines, ranging from micro-processors to large main-frames, by first moving a C Compiler.
The following is the overall structure of the compiler for PDP-11 by Ritchie



The PDP-11 compiler works in two-passes:

Pass I of compiler does *lexical analysis, syntax analysis* and *intermediate code generation.* The PDP-11 compiler uses *recursive descent parser* everything except expressions, for which *operator precedence parser* is used. The intermediate code consists for *postfix notation* for expressions and assembly code for control flow statements. The *storage allocation* for local names is done during the first pass.

Pass II of compiler does *code generation* and PDP-11 has an optional third pass that does optimization on the assembly language output.

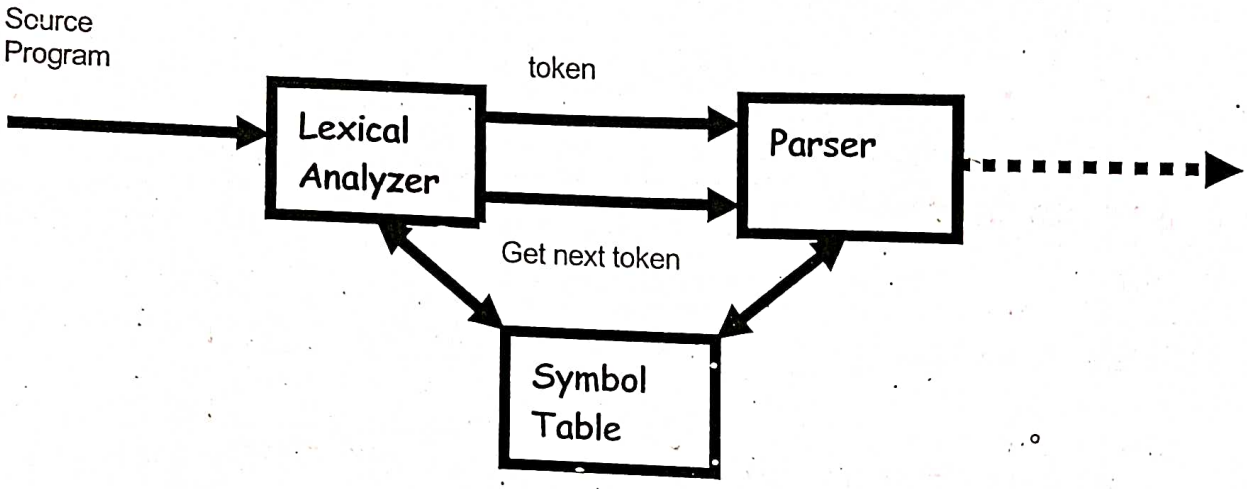
Interpreter:

- An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input.
- The process of interpretation can be carried out in the following phases:
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Direct Execution
- Examples of Interpreted languages are: BASIC, SNOBOL, LISP.
- Advantages:
  - Modification of user program can be easily made and implemented as execution proceeds.
  - Type of object that denotes a variable may change dynamically
  - Debugging a program and finding errors is simplified task for a program used for interpretation
- Disadvantages;
  - The execution of the program is slower.
  - Memory consumption is more.

Module 7: Lexical Analysis

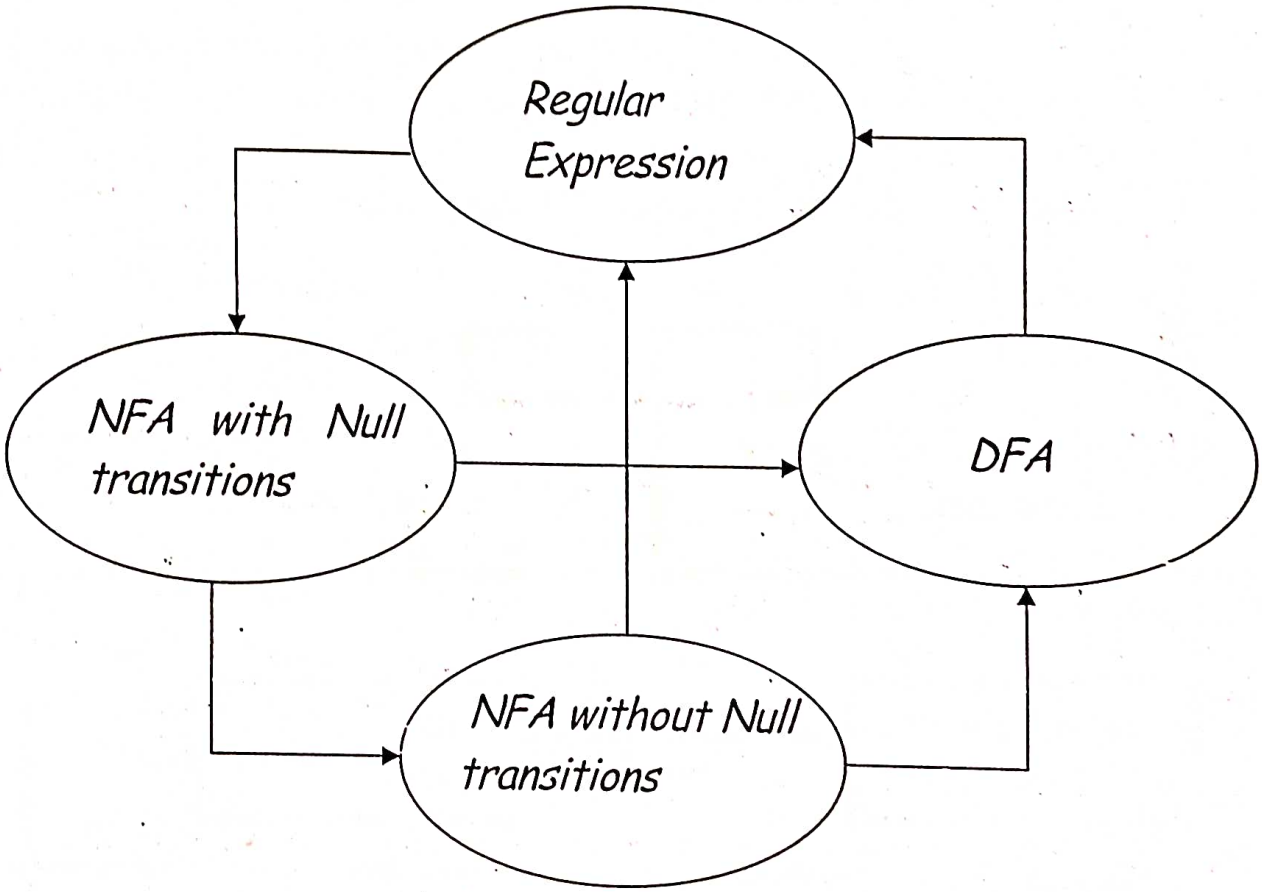
Introduction:

- The lexical analyzer (scanner) is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- The interaction between lexical analyzer and syntax analyzer is summarized as follows:



- Although the task of the scanner is to convert the entire source program into a sequence of tokens, the scanner will rarely do this all at once.
- The lexical analyzer is a subroutine of the parser i.e. scanner will operate under the control of the parser.
- Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.
- Other functions performed by Lexical Analyzer are:
  - Removal of comments
  - Case conversion
  - removal of white spaces
  - Communication with symbol table i.e. storing information regarding an identifier in the symbol table.
- Regular Expressions are used to specify the tokens.
- The advantage of using Regular Expression is that a recognizer for the tokens, called Finite Automaton could be easily constructed.

- Two classes of Finite Automaton are:
  1) Non Deterministic (NFA)
  2) Deterministic (DFA)

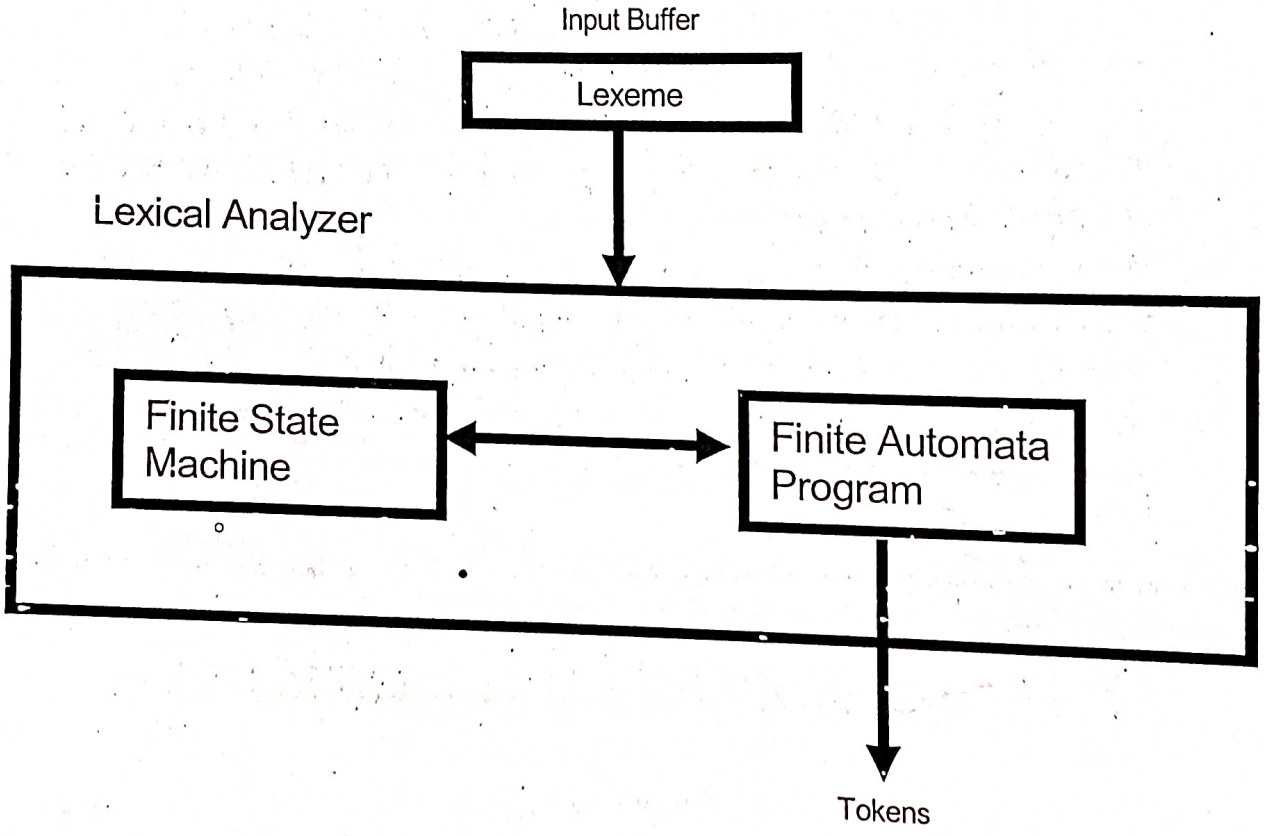


(refer TCS notes for this topic)

R.E. → NFA → DFA → Minimized DFA

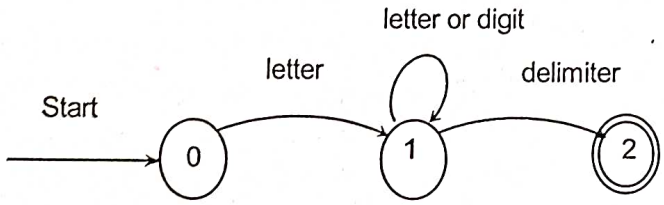
Role of Finite State Automata (FA) in Compiler Design:

- Compiler is a translator that takes as input the source language that is written in High Level Language and produces as output an assembly language or machine language.
- The first phase of the compilation process is called Lexical Analysis.
- Lexical Analysis is a process of recognizing tokens from the input source program.
- The block diagram for this process is shown below:



- While constructing the lexical analyzer we first design the regular expression for the token.
- A method to convert a regular expression into its recognizer is to construct a generalized transition diagram from that expression is called Finite Automata.
- Finite Automata (FA) is a recognizer for a language L that takes as input a string x and answers "yes" if x is a sentence of L and "no" otherwise.
- Two classes of Finite Automaton are:
    - Non Deterministic (NFA)
    - Deterministic (DFA)

- Example to illustrate the above concept:
- r.e. for an identifier : r = letter (letter | digit)* delimiter
- transition diagram for an identifier:



- The starting state of the transition diagram is state 0, the edge from which indicates that the first input character must be a letter.
- If this is the case, we enter state1 and look at the next input character. If that is a letter or digit, we re-enter state 1 and look at the input character after that.
- We continue this was, reading letters and digits and making transitions from state1 to itself, until the next input character is a delimiter for an identifier. On reading the delimiter we enter state2.
- State2 indicates an identifier has been found and hence a token is generated and returned to the parser.

Related University Questions:

May 2010

1) iii)	Explain the role of the Finite Automata in Compiler Theory. **(Refer Printed Notes)**	(5)

Dec 2010

1) d)	Explain the role of Finite State Automata in Compiler design. **(Refer Printed Notes)**	(5)
5) c)	Explain role of Lexical Analyzer. **(Refer Printed Notes)**	(5)

Module 8: Syntax Analysis

Introduction:

Syntax Analyzer analyzes the syntactic structure of the program and its components and checks for the errors.

Parser (Syntax Analyzer):

Parser for grammar G is a program that takes as input a string w and produces as output either a parser tree for w, if w is a sentence of G or an error message indication that w is not a sentence of G.

Parsing Strategies:

1) Top-Down Parsing:

- The parser builds the parse tree from top to bottom i.e. from root to the leaves.
- Uses Leftmost Derivation
- Backtracking is required

Examples:

- ✓ Recursive Decent Parser
- ✓ Predictive Parser (LL(1))

2) Bottom-Up Parsing:

- The parser builds the parse tree from bottom to top i.e. from the leaf to the root
- Uses Right Derivation
- No Back tracking is required

Examples:

- ✓ Shift Reduce Parser
- ✓ Operator Precedence Parser
- ✓ Simple LR Parser (SLR)
- ✓ Canonical Parser (Canonical)
- ✓ Lookahead LR Parser (LALR)

Top down Parsing:

This parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Recursive-Descent Parser:

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser.

Predictive Parser:

Predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly.

LR Parsers:

LR parsers scan
derivation in r

LR par

Bottom Up Parsing:

This parsing method is bottom-up because it attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Shift Reduce Parser:

A convenient way to implement a shift-reduce parser is to use a stack and an input buffer. We use $ to mark the bottom of the stack and the right end of the input.

Operator Precedence Parser:

In operator precedence parsing, three disjoint precedence relations < , = and > .between pairs of terminals. These precedence relations guide the selection of handles.

LR Parsers:

LR parsers scan the input from Left to Right and construct a right most derivation in reverse.

LR parsers are attractive because of the following reasons:

- LR parsers can be constructed to recognize virtually all programming language constructs for which CFG can be written.
- LR parsing method is more common than operator precedence or any of the other common shift-reduce techniques as discussed.
- LR parsers can detect syntactic errors as soon as it is possible to do so on a left-to-right scan of the input.

Step 1: Augment the given grammar:

If G is a grammar with start symbol S, the G' then the augmented grammar for G, is G with a new start symbol S' and production S' → S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce the acceptance of the input. This would occur when the parser was about to reduce by S' → S.

Step 2: Constructing Canonical Collection of LR (0) Items:

```
procedure CLOSURE(I);
begin
        repeat
                for each item A → α.Bβ and each production
                B→ γ in G such that B →. Γ is not in I
                do add B→ .γ to I
        until no more items can be added to I;
        return I;
end
```

```
procedure GOTO(I,X);
begin
        let J be the set of items [A → αX.β], such that
        [A→ α.Xβ] is in I;
        return CLOSURE(J);
end;
```

```
procedure ITEMS(G');
begin
        C:= { CLOSURE ( { S'→ .S } ) };
        repeat
                for each set of items I in C and each grammar symbol X such that
                GOTO(I,X) is not empty and is not in C
                do add GOTO(I,X) to C
        until no more sets of items can be added to C.
end
```

Fig: The sets of items Construction

Step 3: Constructing of an SLR parsing table:

Input: C, the canonical collection of sets of items for an augmented grammar G'

Output: If possible, an LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method: Let $C = \{ I_0, I_1, \dots, I_n \}$. The states of parser are 0, 1, ..., n, state I being constructed form I_i.

The parsing actions for state I are determined as follows:

1) If $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, then set ACTION(i,a) to "shift j" here a is terminal.

2) If $[A \rightarrow \alpha.]$ is in I_i, then set ACTION[i,a] to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A).

3) If $[S' \rightarrow S.]$ is in I_i, the set ACTION [i,$] to "accept"

if any conflicting actions are generated by the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a valid parser in this case.

The goto transitions for state i are constructed using the rule:

4. If GOTO(I_i, A) = I_j, the GOTO [i, A] = j

5. All entries not defined by rules(1) through (4) are made "error".

6. The initial state of the parser is one constructed from the set of items containing [S' → .S]

Step 4: Example of moves of LR parser:

(Refer Class Notes)

Canonical LR Parser:

Step 1: Augment the given grammar:

If G is a grammar with start symbol S, the G' then the augmented grammar for G, is G with a new start symbol S' and production S' → S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce the acceptance of the input. This would occur when the parser was about to reduce by S' → S.

Step 2: Constructing Canonical Collection of LR(1) Items:

```
procedure CLOSURE(I);
begin
        repeat
                for each item [A → α.Bβ , a] inI and each production
                B→ γ , and each terminal b in FIRST(βa)
                such that [B →. Γ, b] is not in I
                do add B→ .γ to I
        until no more items can be added to I;
        return I;
end
```

```
procedure GOTO(I,X);
begin
        let J be the set of items [A → αX.β, a], such that
        [A→ αXβ, a] is in I;
        return CLOSURE(J);
end;
```

```
procedure ITEMS(G');
begin
        C:= { CLOSURE ( { S'→ .S } , $) };
        repeat
                for each set of items I in C and each grammar symbol X such that
                GOTO(I,X) is not empty and is not in C
                do add GOTO(I,X) to C
        until no more sets of items can be added to C.
end
```

Fig: The sets of items Construction

Step3 : Constructing of a canonical LR parsing table:

Input: A grammar G augmented by production $S' \rightarrow S$.

Output: If possible, a canonical LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method: Let $C = \{ I_0, I_1, \ldots, I_n \}$. The states of parser are 0, 1, ..., n, state I being constructed form I_i.

The parsing actions for state I are determined as follows:

1) If $[A \rightarrow \alpha.a\beta, b]]$ is in I_i, and $GOTO(I_i, a) = I_j$, then set ACTION(i,a) to "shift j"
here a is terminal.
2) If $[A \rightarrow \alpha., a]$ is in I_i, then set ACTION[i,a] to "reduce $A \rightarrow \alpha$" .
3) If $[S' \rightarrow S., \$]$ is in I_i, the set ACTION [i,$] to "accept"

if any conflicting actions are generated by the above rules, we say the grammar is not LR(1). The algorithm fails to produce a valid parser in this case.

The goto transitions for state i are constructed using the rule:

4. If $GOTO(I_i, A) = I_j$, the GOTO [i, A] = j
5. All entries not defined by rules(1) through (4) are made "error".
6. The initial state of the parser is one constructed from the set of items containing $[S' \rightarrow .S,, \$]$

LALR Parser:

(Refer Class Notes)

Top Down Parsing V/S Bottom Up Parsing

	Top Down Parsing (LL parsing)	Bottom Up Parsing (LR Parsing)
1	In this techniques const. of parse tree begins at the root and works down towards the leaves.	In this technique the const. of parse tree begins at the leaves and works up towards the root.
2	Left recursion can make the top down parser go into an infinite loop.	Left recursion causes no such problem.
3	Left factoring required.	Left factoring not required.
4	No concept of handle selection.	Selection of handle is required.
5	In top-down parser the actions are pop and remove.	In bottom-up parser derive the string using RMD in reverse which is called canonical derivation.
6	Top down parsers derive the string using leftmost derivation.	Bottom up parsers ~~derive the string using RMD in~~ actions are shift & reduce.
7	Eg: 1) Recursive Descent Parser 2) Predictive Parser.	Eg: 1) Shift Reduce Parser. 2) Operator Precedence Parser. 3) Simple LR Parser. 4) Canonical LR Parser. 5) LALR Parser.

Module 9: Intermediate Code Generation (ICG)

P) Write the po

1) Infix

Intermediate Languages:

- In many compilers the source code is translated into a language which is intermediate in complexity between a (high level) programming language and machine code.
- Such a language is therefore called intermediate code or intermediate text.
- The following are the intermediate codes are often used:
  - Postfix Notation
  - Syntax Trees
  - Directed Acyclic Graph (DAG)
  - Three Address Code Form (3ACF)

Postfix Notation:

- The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a + b.
- The postfix notation for the same expression places the operator at the right end, as ab+.
- In general, if e_1 and e_2 are any postfix expressions and θ is any binary operator, the result of applying θ to the values denoted by e_1 and e_2 is indicated in postfix notation by $e_1e_2\theta$.

Eg:

Infix	Postfix
$E \rightarrow E^{(1)}$ op $E^{(2)}$	
$E \rightarrow$ id	

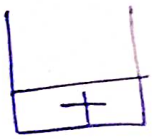
Advantage: No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permits only one way to decode

a + bc* .
abc*+ .

P) Write the postfix notation for the following:

1) Infix a + b * c

Postfix abc * +



coming Precedence > Already stack.

if greater then 1st pop then push

2) Infix a * b + c

ab*+c
abc*+

Postfix ab * c +

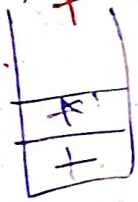
a+bc*-d.
abc*+d-.

3) Infix a + b * c - d

a+bc*-d
abc*+-d
abc*+d-

Postfix abc * +d -

ab*+cd/
+cd/
ab*
ab*cd/+



4) Infix a * b + c / d

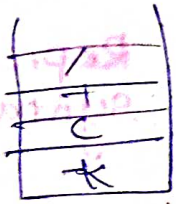
Postfix ab * cd/+

5) Infix a * (b + c / d)

Postfix abcd/+ *

6) Infix a * (b + c) / d

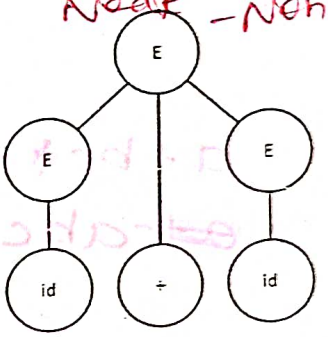
Postfix abc + * d /



Syntax Tree:

- The parse tree is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to be extensively restructured.
- A parse tree often contains redundant information which can be eliminated, thus producing a more economical representation of the source program.
- Syntax tree is a tree in which leaf represents an operand and each interior node an operation.

Eg:



Parse tree

Syntax tree

- **Advantage:** Redundant information in the parse tree is avoided in the syntax tree.

	Parse Tree	Syntax Tree
1	Node represent Non-terminals & leaves represent terminals.	Nodes represent operator and leaves represent operands
2	Contains a lot of redundant info. & hence not compact.	Does not contain any redundant info. and hence compact.
3	Rarely used as a data structure.	Widely used for representing a prog. in a tree structure
4	Represents the concrete syntax of a prog.	Represents abstract syntax of a prog.

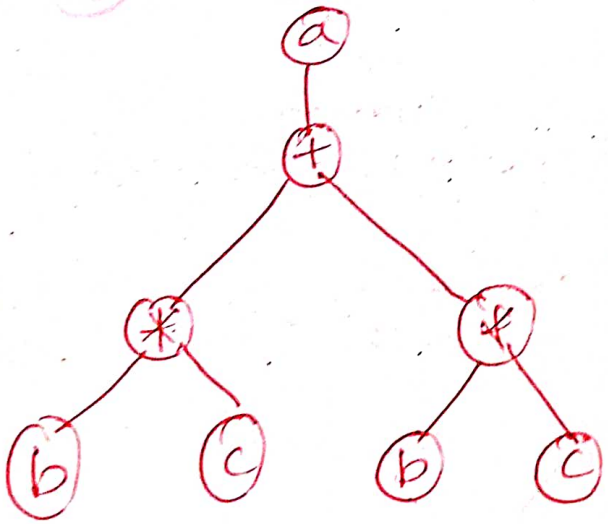
Handling common sub-expressions.

Directed Acyclic Graph (DAG):

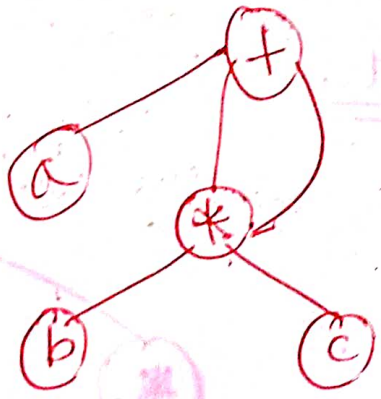
- DAG for an expression identifies the common subexpression in the expression.
- Like, syntax tree, a DAG has a node for every subexpression of the expression, an interior node represents an operator and its children represent its operands.
- The difference is that a node in a DAG representing a common subexpression has more than one "parent" in a syntax tree, the common subexpression would be represented as a duplicated subtree.

May 2013

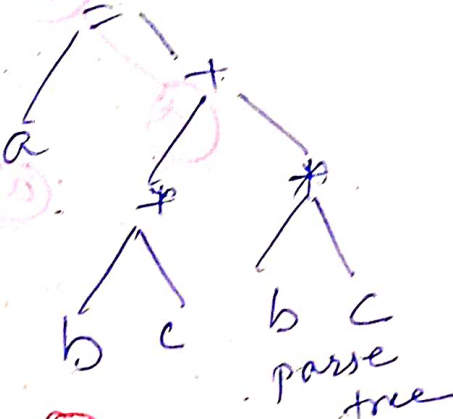
Eg: a = b * c + b * c



Syntax tree



DAG



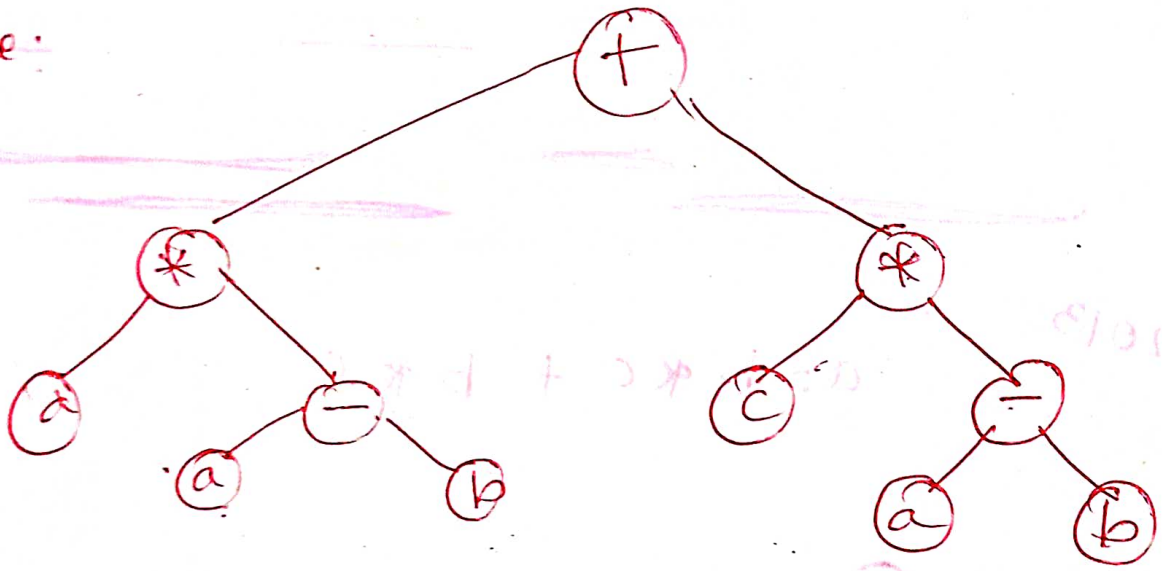
parse tree

- Advantage: Common subexpression can be handled more efficiently.

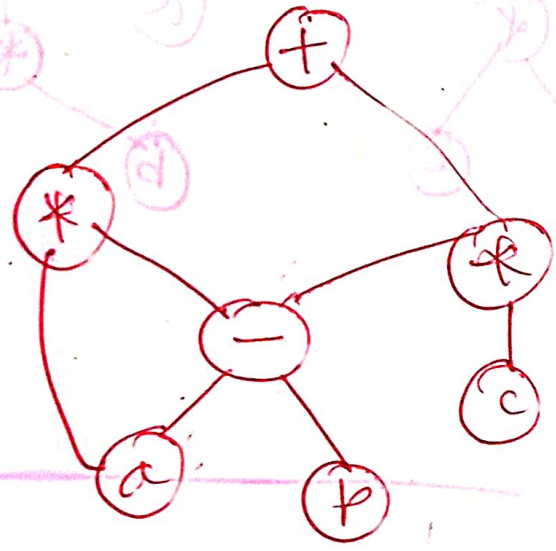
Draw the Syntax tree and DAG for the following:

P1) $a * (a-b) + c * (a-b)$

Syntax Tree:



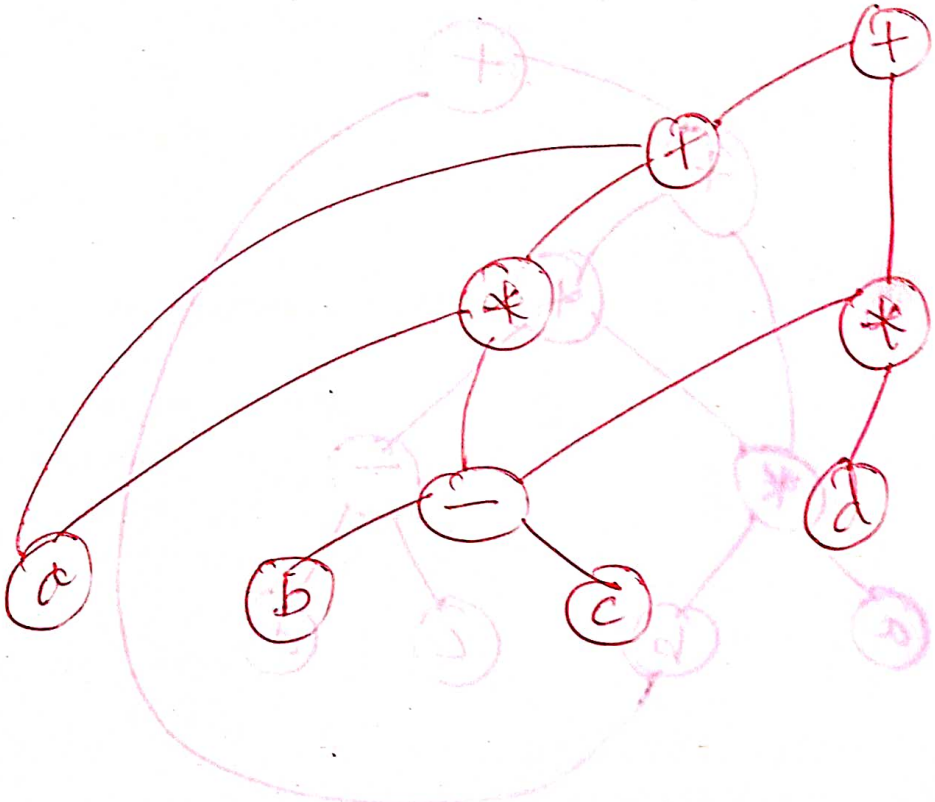
DAG



P2) a + a * (b - c) + (b - c) * d

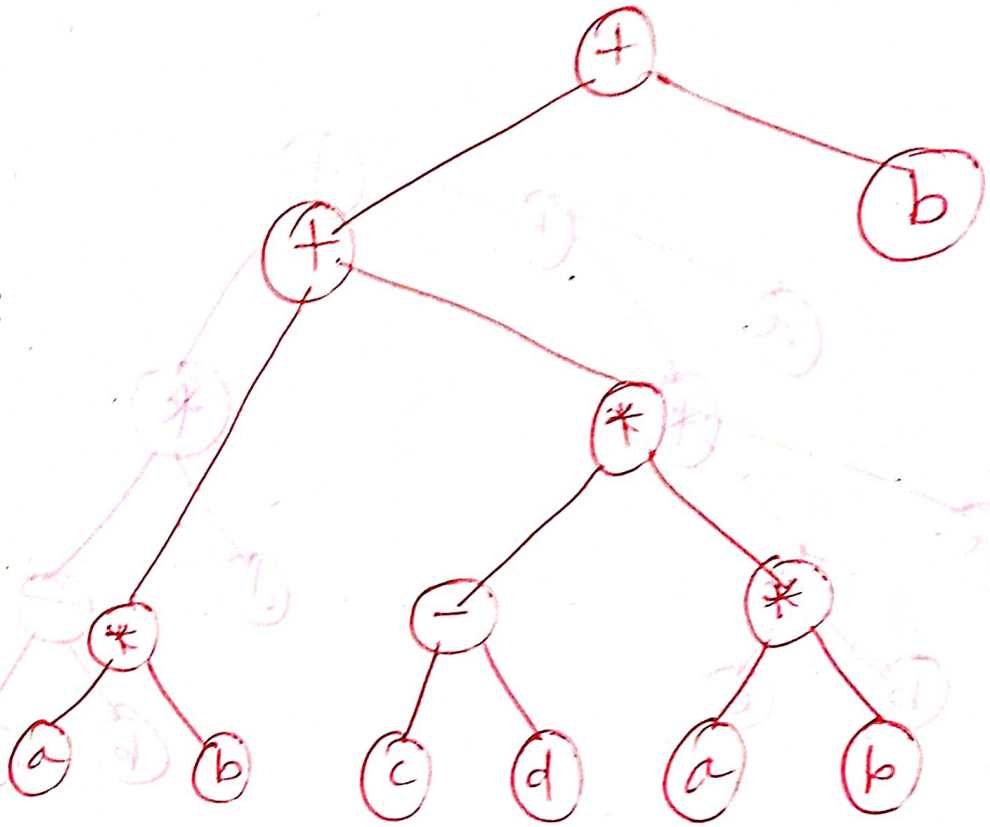
Syntax Tree.

DAG:

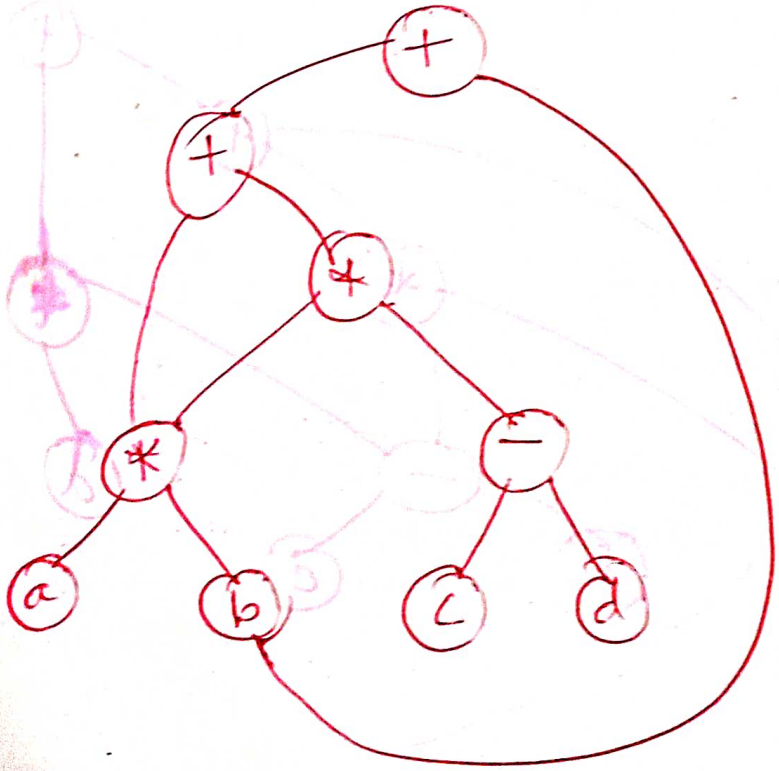


P3) (a*b)+(c-d)*(a*b)+b

Syntax tree



DAG:



Three-address code:

- Three address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.
- Three -address code is a sequence of statements of the form

  A:= B op C,

  where A, B and C are either programmer-defined names constants or compiler-generated temporary name and op stands for any operator.
- The reason for the name "three-address code' is that each statement usually contains three addresses, two for the operands and one for the result.

Eg:

HLL statement	3ACF
a = (e-b) * (c+d)	t1 = e-b t2 = c+d t3 = t1 * t2 a = t3

Types of Three-address Statements:

The following are the different three-address statements

1. Assignment statements of the form x = y op z

   where, op is a binary arithmetic or logical operation
2. Assignments instructions of the form x = op z,

   where, op is a unary operation.
3. Copy statements of the form x = y

   where the value of y is assigned to x.

Implementation of Three Address Statements:

- The three address statement is an abstract form of intermediate code. In actual compiler these statements can be implemented in one of the following ways:
  - Quadruples
  - Triples

Quadruples:

- Quadruple is a record structure with four fields, OP, ARG1, ARG2 and RESULT.
- The contents of fields ARG1, ARG2 and RESULT are normally pointers to the symbol-table entries for the names represented by these fields.

Eg:

3ACF

```
t1 = e - b
t2 = c + d
t3 = t1 * t2
a = t3
```

Quadruple

	OP	ARG1	ARG2	Result
100	−	e	b	t1
101	+	c	d	t2
102	*	t1	t2	t3
103	=	t3	—	a

Advantage: Simple to Implement.

Disadvantage: Requires a lot of temporary variables.

Triples:

- To avoid entering temporary names into the symbol table, one can allow the statement computing a temporary value to represent that value.
- If we do so, three-address statements are representable by a structure with only three fields OP, ARG1 and ARG2, the arguments of OP, are either pointers to the symbols table (for programmer defined names or constants) or pointers into the structure itself (for temporary values).
- Since three fields are used, this intermediate code format is known as triples.
- Parenthesize numbers are used to represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves.

Eg:

3ACF

```
t1 = e - b
t2 = c + d
t3 = t1 * t2
a = t3
```

Triple

	OP	ARG1	ARG2
100	—	e	b
101	+	c	d
102	*	(100)	(101)
103	=	a	(102)

Advantage: Does not require any temporary variables.

Disadvantage: Complex to implement.

Some Problems based on three address statements:

P1) a = b * c + d * (f-e);

Solution:

3 address statements.

```
100  t1 = f - e
101  t2 = b * c
102  t3 = d * t1
103  t4 = t2 + t3
104  a = t4
```

Quadruple:-

	OP	ARG1	ARG2	Result
100	-	f	e	t1
101	*	b	c	t2
102	*	d	t1	t3
103	+	t2	t3	t4
104	=	t4	—	a

P2) if (x < y)
a = b + c * 3;
else
p = q + r;

100 if(x<y) then go to 102
101 go to 105
102 t1 = c*3
103 t2 = b+t1
104 a = t2
105 t4 = q+r
106 p = t4
107 stop

Solution:

3 address statements.

100 if(x<y) then goto 102
101 goto 106
102 t1 = c*3
103 t2 = b+t1
104 a = t2
105 goto 109
106 t3 = q+r
107 p = t3
108 goto 109
109 stop

a=b+c*3;

Quadruple:

	OP	ARG1	ARG2	Result
100	<	x	y	goto 102
101	—	—	—	goto 106
102	*	c	3	t1
103	+	b	t1	t2
104	=	t2	—	a
105	—	—	—	goto 109
106	+	q	r	t3

```
while (i<=n)
{
  sum = sum+i;
  i++;
}
```

P4)

soln:

```
100 if(i<=n) then goto 102
101 goto 107
102 t1 = sum+i
103 sum = t1
104 t2 = i+1
105 i = t2
106 goto 100
107
108
```

P5)

```
do
{
  sum = sum+i;
  i++;
} while (i<=n);
```

```
100 t1 = sum+i
101 sum = t1
102 t2 = i+1
103 i = t2
104 if(i<=n) then goto 100
105 goto 106
106
107
```

```
for (i=1; i<=n; i++)
    f = f * i;
```

P6)

```
100    t1 = 1
101    i = t1
102    if (i<=n) then goto 104
103    goto
104    t2 = f * i
105    f = t2
106    t3 = i+1
107    i = t3
108    goto 102
109
```

P7) while (A<B) do
if (C<D)
then X = Y+Z

```
100   if (A<B) then goto 102
101   goto 107
102   if (C<D) then goto 104
103   goto 100/106
104   t1 = Y+Z
105   X = t1
106   goto 100
107
108
```

```
100  if (A<B) then goto 102
101  goto 106
102  if (C<D) then goto 104
103  goto
104  t1 = y+z
105  X = t1
106  goto 100.
```

```
while (a<b) do
if (c<d) then
    x = y+z
else
    x = y-z.
```

P8)

soln:

```
100   if (a<b) goto 102
101   goto 110
102.  if (c<d) goto 104.
103   goto 107
104   t1 = y+z
105   x = t1
106   goto 100
107   t2 = y-z
108   x = t2.
109   goto. 100
110
```

Module 10: Syntax Directed Translation (SDT)

Introduction:

- Syntax Directed Translation is a notational framework for intermediate code generation which allows subroutines or "semantic actions" to be attached to the productions of the context-free grammar.
- Syntax directed translation is useful because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.

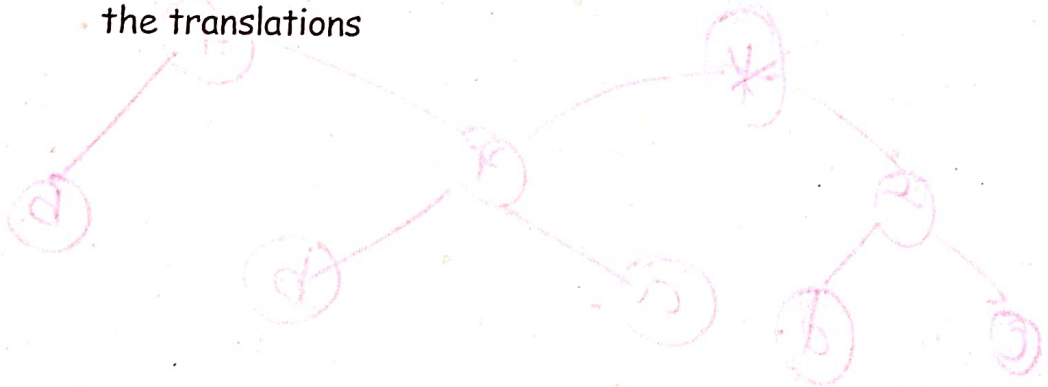
Semantic Actions:

Output Action

- Syntax directed translation scheme is a context free grammar in which a program fragment called an output action (semantic action or semantic rule) is associated with each production.
- The output action may involve the computation of values for variable belonging to the compiler, the generation of intermediate code, the printing of an error diagnostic or the placement of some value in a table.

Implementation of Syntax Directed Translators:

- A Syntax Directed translation scheme is a convenient description of what we would like to be done.
- The output defined is independent of the kind of the parser used to construct the parse tress or the kind of mechanism used to compute the translations



Syntax Directed Construction of Syntax Tree:

Like postfix code, it is easy to define either a parse tree or a syntax tree in terms of Syntax Directed Translation scheme.

SDT for Syntax Tree:

Production	Semantic Action
$E \rightarrow E_1 + T$	$\{E.\text{ptr} = \text{mknode}(\text{'+'}, E_1.\text{ptr}, T.\text{ptr})\}$
$E \rightarrow T$	$\{E.\text{ptr} = T.\text{ptr}\}$
$T \rightarrow T_1 * F$	$\{T.\text{ptr} = \text{mknode}(\text{'*'}, T_1.\text{ptr}, F.\text{ptr})$
$T \rightarrow F$	$\{T.\text{ptr} = F.\text{ptr}\}$
$F \rightarrow \text{id}$	$\{F.\text{ptr} = \text{mkleaf}(\text{id.place})\}$

- where ptr is pointer value attribute used to link the pointer to a node in the syntax tree, and place is pointer value attribute used to link the pointer to the symbol record that contains the name of the identifier.
- The mkleaf generates leaf nodes, and mknode generates intermediate nodes.

Eg1:

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → id

LR Parser

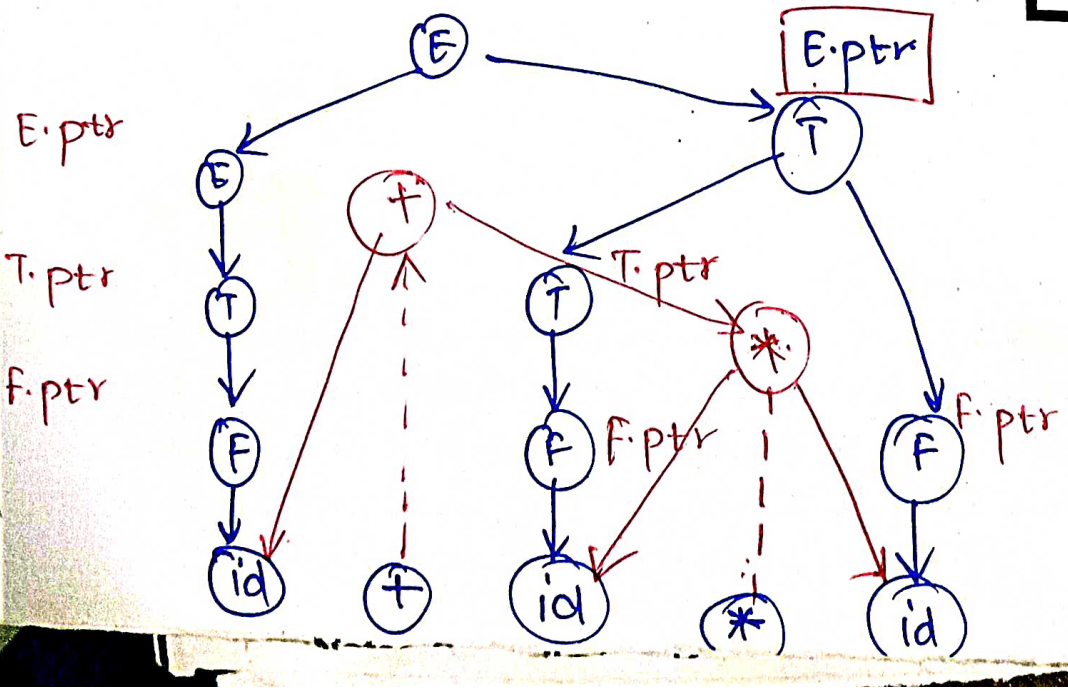
	Actions				Goto		
	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		Acc			
2		R2	S6	R2			
3		R4	R4	R4			
4		R5	R5	R5			
5	S4					7	3
6	S4						8
7		R1	S6	R1			
8		R3	R3	R3			

Stack	Input	Actions
$ 0	id + id * id $	Shift id
$ 0 id 4	+ id * id $	Reduce F → id
$ 0 F 3	+ id * id $	Reduce T → F
$ 0 T 2	+ id * id $	Reduce E → T
$ 0 E 1	+ id * id $	Shift +
$ 0 E 1 + 5	id * id $	Shift id
$ 0 E 1 + 5 id 4	* id $	Reduce F → id
$ 0 E 1 + 5 F 3	* id $	Reduce T → F
$ 0 E 1 + 5 T 7	* id $	Shift *
$ 0 E 1 + 5 T 7 * 5	id $	Shift id
$ 0 E 1 + 5 T 7 * 6 id 4	$	Reduce F → id
$ 0 E 1 + 5 T 7 * 6 F 8	$	Reduce T → T * F
$ 0 E 1 + 5 T 7	$	Reduce E → E + T
$ 0 E 1	$	Accept

Eg2:

SDT for Syntax Tree

Production	Semantic Action
$E \rightarrow E_1 + T$	{E.ptr = mknode ('+', E_1.ptr, T.ptr)}
$E \rightarrow T$	{E.ptr = T.ptr}
$T \rightarrow T_1 * F$	{T.ptr = mknode ('*', T_1.ptr, F.ptr)}
$T \rightarrow F$	{T.ptr = F.ptr}
$F \rightarrow id$	{F.ptr = mkleaf (id.place)}



Eg2:

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → id _

LR Parser

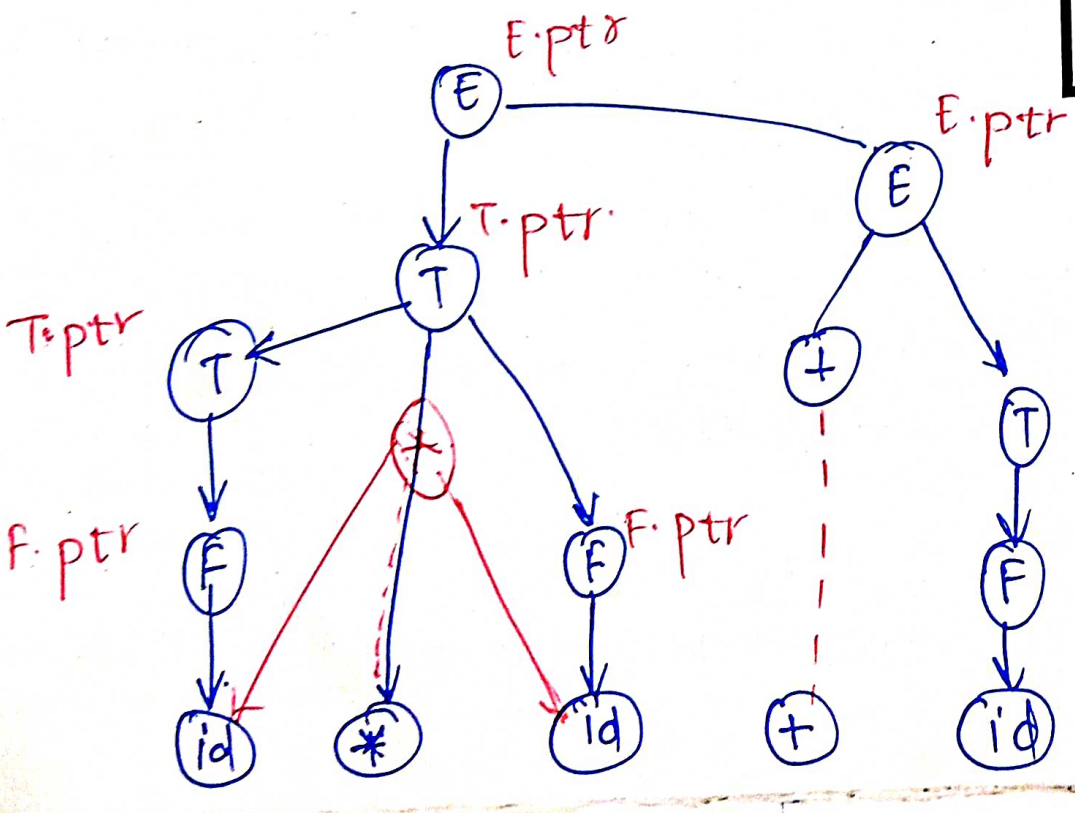
Actions | Goto

	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		Acc			
2		R2	S6	R2			
3		R4	R4	R4			
4		R5	R5	R5			
5	S4					7	3
6	S4						8
7		R1	S6	R1			
8		R3	R3	R3			

Stack	Input	Actions
$ 0	id * id + id $	Shift id
$ 0 id 4	* id + id $	Reduce F → id
$ 0 F 3	* id + id $	Reduce T → F
$ 0 T 2	* id + id $	Shift *
$ 0 T 2 * 6	id + id $	Shift id
$ 0 T 2 * 6 id 4	+ id $	Reduce F → id
$ 0 T 2 * 6 F 8	+ id $	Reduce T → T * F
$ 0 T 2	+ id $	Reduce E → T
$ 0 E 1	+ id $	Shift +
$ 0 E 1 + 5	id $	Shift id
$ 0 E 1 + 5 id 4	$	Reduce F → id
$ 0 E 1 + 5 F 3	$	Reduce T → F
$ 0 E 1 + 5 T 7	$	Reduce E → E + T
$ 0 E 1	$	Accept

SDT for Syntax Tree

Production	Semantic Action
$E \rightarrow E_1 + T$	{E.ptr = mknode ('+', E_1.ptr, T.ptr)}
$E \rightarrow T$	{E.ptr = T.ptr}
$T \rightarrow T_1 * F$	{T.ptr = mknode ('*', T_1 ptr, F.ptr)
$T \rightarrow F$	{T.ptr = F.ptr}
$F \rightarrow$ id	{F.ptr = mkleaf (id.place)}



Module 11: Code Optimization

Introduction:

The code optimization phase attempts to improve the intermediate so that a faster running machine could be generated.

Code Optimization is performed to:

- Minimize the time taken to execute a program.
- Minimize the amount of memory occupied.

Machine Dependent Optimization:

1) Machine instructions that use registers as operands must be selected since such instructions are faster that the instructions that refer to locations in memory
2) Special control instructions or addressing modes could be used to generate efficient object code.
3) Consecutive instructions that involve different functional units could be executed at the same time.

Machine Independent Optimization:

There are two classes of Machine Independent Optimizations

(Sources of optimization):

1. Local Code Optimization (Function preserving transformations)
2. Global Code Optimization (Loop Optimizations)

Local Code Optimization:

- Local Optimization performs optimization local to a function definition.
- Therefore it is also called **Function Preserving Transformation**.
- There are number of ways in which a compiler can improve a program without changing the function it computes. Examples of Function preserving transformations are:
  - ❖ Algebraic Simplifications
  - ❖ Constant Folding
  - ❖ Constant Propagation
  - ❖ Common Sub-expression Elimination
  - ❖ Dead Code Elimination

Algebraic Simplifications;

- Some statements can be deleted while some can be simplified.
- Eg:

Before Optimization	After Optimization
Eg1: x = x + 0;	Can be deleted
Eg2: x = x * 1;	Can be deleted
Eg3: x = x * 0;	Can be simplified as x=0.

Constant Folding:

- Operations on constants can be computed at compile time.
- Expressions values can be pre-computed at the compilation time, hence requiring less execution time.
- Eg:

Before Optimization	After Optimization
Eg1: area = (22.0/7.0)*r*r;	area = 3.14286 * r * r.
Eg2: a = 3 * 2;	a = 6

Constant Propagation

- Replace a variable with constant which has been assigned to it earlier.
- Advantage : Smaller code and Fewer registers.
- Eg:

Before Optimization	After Optimization
PI = 3.14286; area = PI*r*r;	area = 3.14286 * r * r;

Common Sub-expression Elimination:

- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
- Common Sub expressions would be evaluated once and not repeatedly.
- Eg:

Before Optimization	After Optimization

:
a = 10 + b * c;
x = b * c + 5;
:
	temp = b * c;
	a = 10 + temp;
	x = temp + 5;

Dead Code Elimination:

- Dead Code are portion of the program which will not be executed in any path of the program.
- Eg:

Before Optimization	After Optimization

DEBUG = false;
if (DEBUG)
{
...
}
	DEBUG = false;

Loop Optimizations: (Global Code Optimization):

The running time of a program can be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Examples of Loop Optimization techniques are:

- Code Motion
- Strength Reduction
- Loop Fission or Loop Distribution
- Loop Unwinding

Code Motion

- Moving code from one part of the program to other without modifying the algorithm.
- Advantage: Reduce execution frequency of the code subjected to movement.
- Eg:

Before Optimization	After Optimization

while (i < (max * 25))
{
...
...
}
	t= max * 25;
	while (i<t)
	{
	
	}

Strength Reduction

- Replacement of an operator with a less costly one.
- Multiplication or division by powers of two can be replaced by adds or shifts.
- Advantage; Replacing operations that require more number of CPU cycles with the ones that require less number of CPU cycles will lead to a fast running code.
- Eg:

Before Optimization	After Optimization
r1 = r2 x 2; r1 = r2 / 2;	r1 = r2+r2; OR r1 = r2<<1; r1 = r2>>1;

1442

Loop Fission or Loop Distribution:

- Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body.
- This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.
- Eg:

Before Optimization	After Optimization

int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
a[i] = 1;
b[i] = 2;
}
	int i, a[100], b[100];
	for (i=0; i<100; i++)
	{ a[i] = 1;
	}
	for (i=0; i<100; i++)
	{ b[i] = 2;
	}

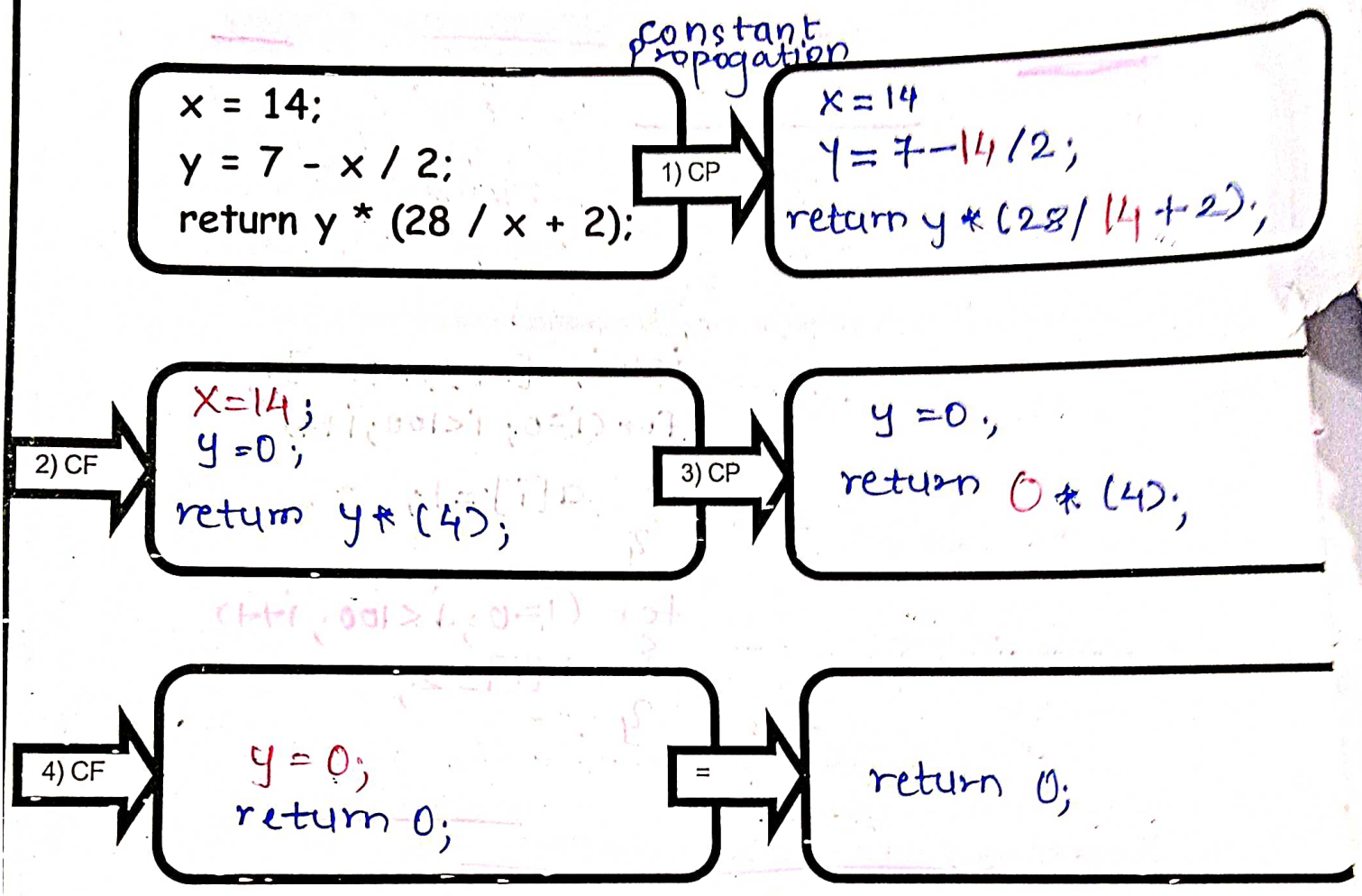
Loop Unwinding:

- Loop unwinding, also known as loop unrolling, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size
- Eg:

Before Optimization	After Optimization

int x;
for (x = 0; x < 100; x++)
{
delete(x);
}
	int x;
	for (x=0; x<100; x+=5)
	{
	delete(x);
	delete (x+1);
	delete (x+2);
	delete (x+3);
	delete (x+4);
	}

P1) Optimize the following code:



Syntax Directed Translation to Postfix Code:

The production of postfix intermediate code for expressions is described by the syntax directed translation scheme as follows:

SDT for Postfix Code:

Production	Semantic Action
$E \rightarrow E_1 + T$	$\{E.\text{code} = \text{concate}(E_1.\text{code}, T.\text{code}, \text{"+"});\}$
$E \rightarrow T$	$\{E.\text{code} = T.\text{code};\}$
$T \rightarrow T_1 * F$	$\{T.\text{code} = \text{concate}(T_1.\text{code}, F.\text{code}, \text{"*"});\}$
$T \rightarrow F$	$\{T.\text{code} = F.\text{code};\}$
$F \rightarrow id$	$\{F.\text{code} = \text{getname}(\text{id.place});\}$

- where code is a string value attribute used to hold the postfix expression, and place is pointer value attribute used to link the pointer to the symbol record that contains the name of the identifier.
- The label getname returns the name of the identifier from the symbol table record that is pointed to by ptr, and concate(s_1, s_2, s_3) returns the concatenation of the strings s_1, s_2, and s_3, respectively.

P) Design LR(0) Parser for the following:

E → E + T | T
T → T * F | F
F → id

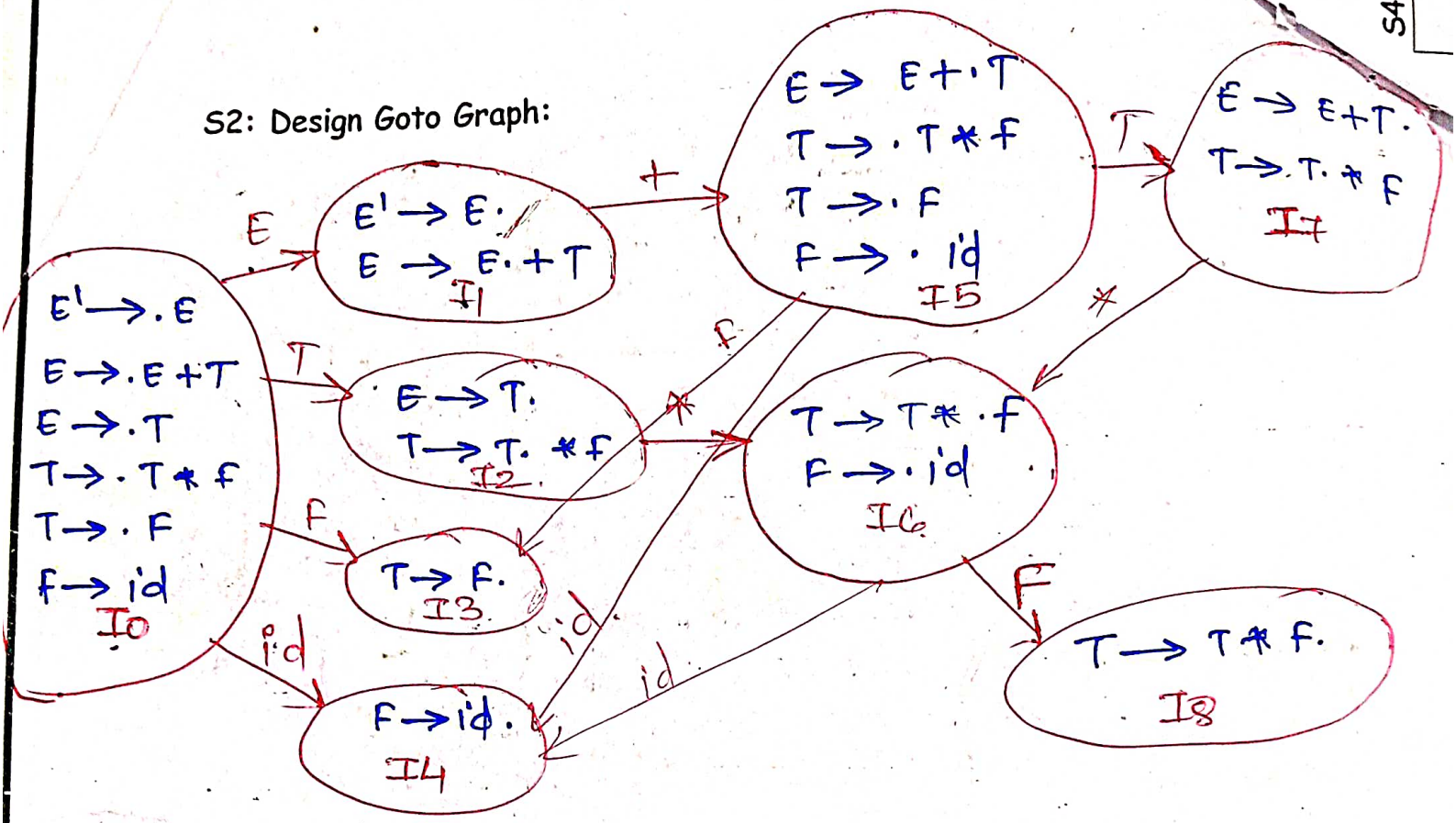
Solution:

S1: Augment the given Grammar:

E' → E

1) E → E+T
2) E → T
3) T → T * F
4) T → F
5) F → id

S2: Design Goto Graph:



S3): Design LR(0) Parser Table:

	Actions				Goto		
	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		Acc			
2		r2	S6	r2			
3		r4	r4	r4			
4		r5	r5	r5			
5	S4					7	3
6	S4						8
7		r1	S6	r1			
8		r3	r3	r3			

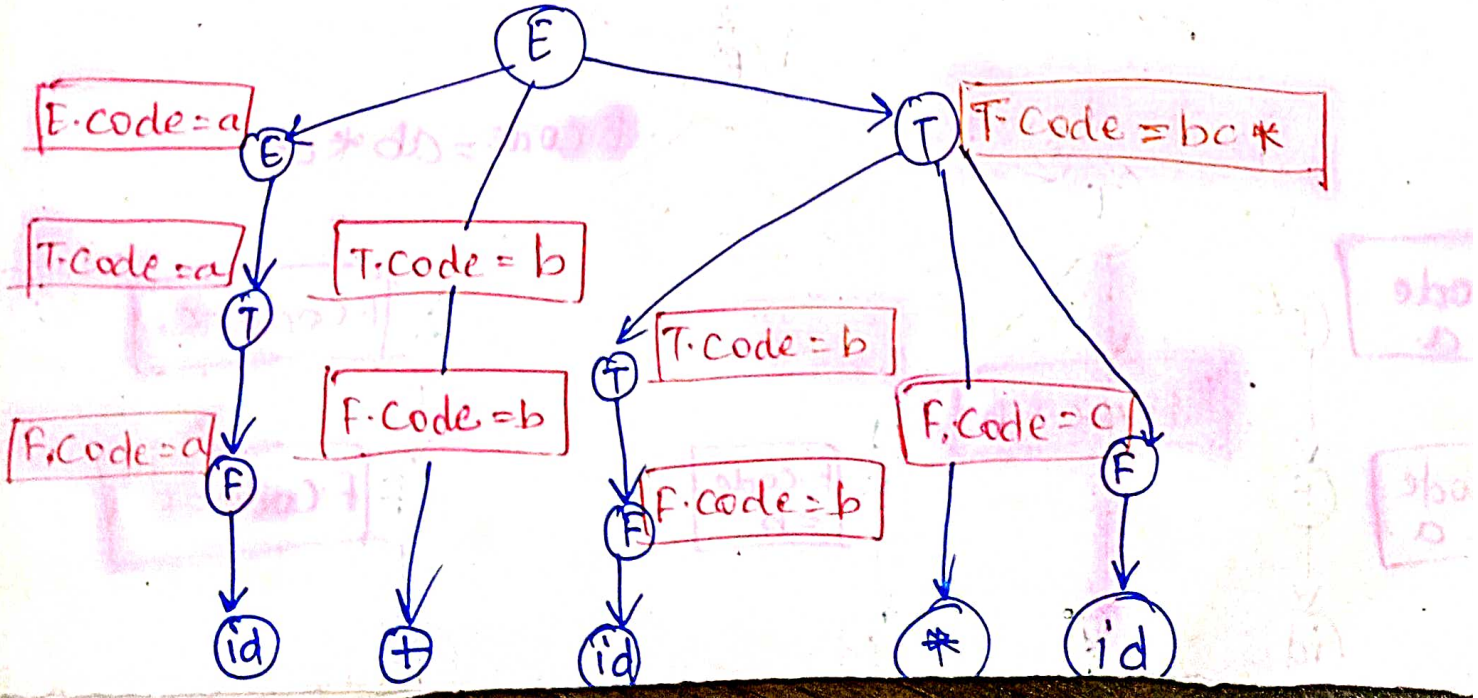
Follow (E') = {$}
Follow(E) = {+, $}
Follow(T) = {+, *, $}
Follow(F) = {+, * $}

S4: Eg1:

Stack	Input	Actions
$ 0	id + id * id $	shift id
$ 0 id 4	+ id * id $	Reduce F → id
$ 0 F 3	+ id * id $	Reduce T → F
$ 0 T 2	+ id * id $	Reduce E → T
$ 0 E 1	+ id * id $	shift +
$ 0 E 1 + 5	id * id $	shift id
$ 0 E 1 + 5 id 4	* id $	Reduce F → id
$ 0 E 1 + 5 F 3	* id $	Reduce T → F
$ 0 E 1 + 5 T 7	* id $	shift *
$ 0 E 1 + 5 T 7 * C	id $	shift id
$ 0 E 1 + 5 T 7 * 6 id 4	$	Reduce F → id
$ 0 E 1 + 5 T 7 * 6 F 8	$	Reduce T → T * F
$ 0 E 1 + 5 T 7	$	Reduce E → E + T
$ 0 E 1	$	Accept

SDT for Postfix Code

Production	Semantic Action
$E \to E_1 + T$	{E.code = concate(E_1.code, T.code,"+");}
$E \to T$	{E.code = T.code;}
$T \to T_1 * F$	{T.code = concate(T_1.code, F.code,"*");}
$T \to F$	{T.code = F.code;}
$F \to id$	{F.code = getname(id.place);}



S4: Eg2:

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → id

LR Parser

	Actions				Goto		
	id	+	*	$	E	T	F
0	S4				1	2	3
1		S5		Acc			
2		R2	S6	R2			
3		R4	R4	R4			
4		R5	R5	R5			
5	S4					7	3
6	S4						8
7		R1	S6	R1			
8		R3	R3	R3			

Stack	Input	Actions
$ 0	id * id + id $	Shift id
$ 0 id 4	* id + id $	Reduce F → id
$ 0 F 3	* id + id $	Reduce T → F
$ 0 T 2	* id + id $	Shift *
$ 0 T 2 * 6	id + id $	Shift id
$ 0 T 2 * 6 id 4	+ id $	Reduce F → id
$ 0 T 2 * 6 F 8	+ id $	Reduce T → T * F
$ 0 T 2	+ id $	Reduce E → T
$ 0 E 1	+ id $	Shift +
$ 0 E 1 + 5	id $	Shift id
$ 0 E 1 + 5 id 4	$	Reduce F → id
$ 0 E 1 + 5 F 3	$	Reduce T → F
$ 0 E 1 + 5 T 7	$	Reduce E → E + T
$ 0 E 1	$	Accept

SDT for Postfix Code

Production	Semantic Action
$E \to E_1 + T$	{E.code = concate(E1.code, T.code, "+")}
$E \to T$	{E.code = T.code;}
$T \to T_1 * F$	{T.code = concate(T1.code, F.code, "*");}
$T \to F$	{T.code = F.code;}
$F \to id$	{F.code = getname(id.place);}