BANGALORE INSTITUTE OF TECHNOLOGY

K.R. Road, V.V.Puram, Bengaluru-560 004

DEPARTMENT OF COMPUTER SCIENCE & ENGG

SYSTEM SOFTWARE AND COMPILER DESIGN

NOTES

SUBJECT CODE: 15CS63

By

Mrs. Hemavathi. P
Assistant Professor
Department of CSE

SYSTEM SOFTWARE AND COMPILER DESIGN

[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2016 -2017)
SEMESTER – VI

Subject Code	15CS63	IA Marks	20
Number of Lecture Hours/Week	4	Exam Marks	80
Total Number of Lecture Hours	50	Exam Hours	03

CREDITS – 04

Course objectives: This course will enable students to

- Define System Software such as Assemblers, Loaders, Linkers and Macroprocessors
- Familiarize with source file, object file and executable file structures and libraries
- Describe the front-end and back-end phases of compiler and their importance to students

Module – 1	**Teaching Hours**
Introduction to System Software, Machine Architecture of SIC and SIC/XE. **Assemblers:** Basic assembler functions, machine dependent assembler features, machine independent assembler features, assembler design options. **Macroprocessors:** Basic macro processor functions, **Text book 1: Chapter 1: 1.1,1.2,1.3.1,1.3.2, Chapter2 : 2.1-2.4,Chapter4: 4.1.1,4.1.2**	**10 Hours**
Module – 2	
Loaders and Linkers: Basic Loader Functions**,** Machine Dependent Loader Features, Machine Independent Loader Features, Loader Design Options, Implementation Examples. **Text book 1 : Chapter 3 ,3.1 -3.5**	**10 Hours**
Module – 3	
Introduction: Language Processors, The structure of a compiler, The evaluation of programming languages, The science of building compiler, Applications of compiler technology, Programming language basics **Lexical Analysis:** The role of lexical analyzer, Input buffering, Specifications of token, recognition of tokens, lexical analyzer generator, Finite automate. **Text book 2:Chapter 1 1.1-1.6 Chapter 3 3.1 – 3.6**	**10 Hours**
Module – 4	
Syntax Analysis: Introduction, Role Of Parsers, Context Free Grammars, Writing a grammar, Top Down Parsers, Bottom-Up Parsers, Operator-Precedence Parsing **Text book 2: Chapter 4 4.1 4.2 4.3 4.4 4.5 4.6 Text book 1 : 5.1.3**	**10 Hours**
Module – 5	
Syntax Directed Translation, Intermediate code generation, Code generation **Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2**	**10 Hours**

Course outcomes: The students should be able to:

- Explain system software such as assemblers, loaders, linkers and macroprocessors
- Design and develop lexical analyzers, parsers and code generators
- Utilize lex and yacc tools for implementing different concepts of system software

Question paper pattern:

The question paper will have TEN questions.

There will be TWO questions from each module.

Each question will have questions covering all the topics under a module.

The students will have to answer FIVE full questions, selecting ONE full question from each module.

Text Books:

1. System Software by Leland. L. Beck, D Manjula, 3rd edition, 2012
2. Compilers-Principles, Techniques and Tools by Alfred V Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Pearson, 2nd edition, 2007

Reference Books:

1. Systems programming – Srimanta Pal , Oxford university press, 2016
2. System programming and Compiler Design, K C Louden, Cengage Learning
3. System software and operating system by D. M. Dhamdhere TMG
4. Compiler Design, K Muneeswaran, Oxford University Press 2013.

BANGALORE INSTITUTE OF TECHNOLOGY
K R ROAD, V V PURAM, BENGALURE-04

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

COURSE OBJECTIVES AND OUTCOMES-2015-19

Course Title : System Software and Compiler Design	Course Code : : 15CS63
No. of Lecture Hrs./Week : 04	Exam Hours : 03
Total No. of Lecture Hrs. : 52	Exam Marks : 80

Prerequisites

1. Microprocessors and Microcontrollers(15CS44)
2. Automata Theory and Computability (15CS54)

Course Learning Objectives

This course will help students to achieve the following objectives:

1. To understand the concepts of System software, Application Software and different hypothetical machine architectures.
2. Familiarize with source file, symbol table creation (pass-1), object file creation (pass-2), loaders and linkers.
3. To know the fundamental concepts of translators.
4. To identify the methods and strategies for parsing techniques.
5. Devise and perform syntax-directed translation schemes for compiler.
6. Devise intermediate code generation schemes and analyze the optimized code generated after the synthesis phase.

Course Outcomes

At the end of the course students should be able to:

1. Apply the knowledge of System Software such as Assemblers, Loaders, Linkers and Macro processors to build an application.
2. Understand the basic principles of compiler in high level programming language.
3. Analyze and design the analysis phase using different techniques.
4. Build the system software by associating synthesis phase with analysis phase for better optimization and performance.

MODULE-1

TEXTBOOK: System Software by Leland.L. Beck, D.Manjula, 3rd Edition, 2012

CHAPTER 1: **Introduction to System Software and Machine Architecture**

- 1.1 Introduction
- 1.2 System Software and Machine Architecture
- 1.3 The Simplified Instructional Computer (SIC)
  - 1.3.1 SIC Machine Architecture
  - 1.3.2 SIC/XE Machine Architecture
  - 1.3.3 SIC Programming Examples

CHAPTER 2: **Assemblers**

- 2.1 Basic Assembler Functions
  - 2.1.1 A Simple SIC Assembler
  - 2.1.2 Assembler Algorithm and Data Structures
- 2.2 Machine-Dependent Assembler Features
  - 2.2.1 Instruction Formats and Addressing Modes
  - 2.2.2 Program Relocation
- 2.3 Machine-Independent Assembler Features
  - 2.3.1 Literals
  - 2.3.2 Symbol-Defining Statements
  - 2.3.3 Expressions
  - 2.3.4 Program Blocks
  - 2.3.5 Control Sections and Program Linking
- 2.4 Assembler Design Options
  - 2.4.1 One-Pass Assemblers
  - 2.4.2 Multi-Pass Assemblers

CHAPTER 4: **Macro Processors**

- 4.1 Basic Macro Processor Functions
  - 4.1.1 Macro Definition and Expansion
  - 4.1.2 A Simple Bootstrap Loader

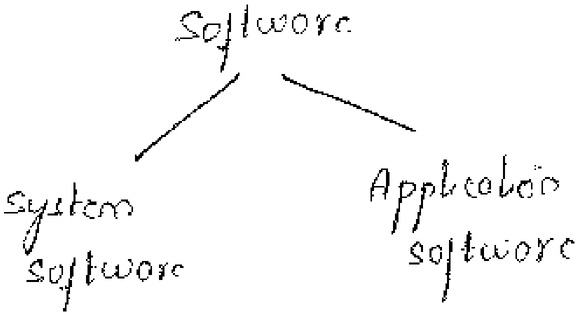
CHAPTER 1

Introduction to System Software and Machine Architecture

18/12/2017

The term "software" refers to the set of electronic program instructions or data a computer processor reads in order to perform a task.

"Hardware" refers to the physical components that you can see and touch, such as the computer hard drive, mouse and keyboard.



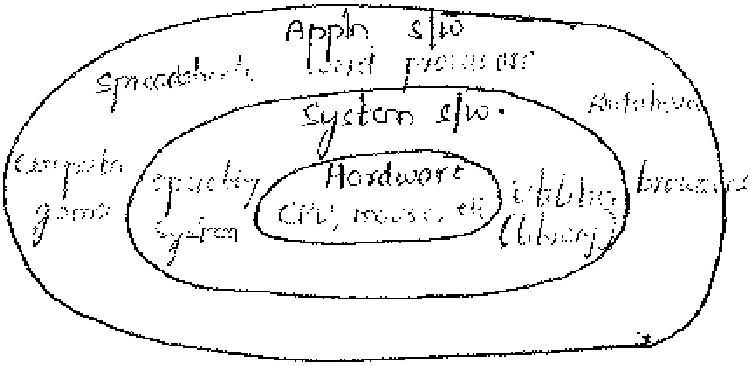


Fig: Relationship b/w system software, Appln software and Hardware

Defn: "System software" is a set of programs that are dedicated to manage the computer itself, such as operating system, file management utilities,

Application software are a set of productivity programs or end-user programs to perform their specific tasks

Difference between system software and Application software

System software	Application software
1 System software is a set of programs that are dedicated to manage the computer itself (mem. mgmt, process mgmt, protection security	Application software is a set of computer programs designed to permit the user to perform a group of functions, tasks or activities.
2 Is written in a low-level language ie assembly language	Is written in a high-level language like C, C++, Java, .net, VB etc
3. Starts running when the s/m is turned on and runs till the system is shut down	Runs as and when the user requests
4 A system is unable to run without system software	Appln. software is even not required to run the system ∴ it is user specific
5. system software is general purpose	Appln software is specific purpose software
6. Ex: operating system, assembler, compiler, loader or linker, text editor, debugger, macro processors,	Ex.. web browser, word processing, spreadsheet, database, Adobe creative suite, Audio master suite, games
7 ~~not~~ machine Dependent (machine architecture)	not machine dependent

1.2. System software and machine architecture

- Machine dependency of system software
  - → System programs are intended to support the operation and use of the computer.
  - → machine architecture differs in :
    - machine code,
    - instruction formats
    - Addressing mode
    - Registers

- machine independency of system software
  - → general design and logic is basically the same:
    - code optimization
    - subprogram linking

1.3. Simplified Instructional Computer

As we know different systems have different features and different feature are difficult to study one by one. so to avoid this problem we study Simplified Instructional Computer.

SIC is a hypothetical computer system introduced in system software. Due to the fact that most modern microprocessors include complex functions for the purpose of efficiency, it is very difficult to learn systems programming using a real-world system. The SIC solves this problem by abstracting away these complex behaviours in favour of an architecture that is clear and accessible for those wanting to learn systems programming

- SIC comes in two versions

  → standard model

  → XE version (Extra Equipment or Extra Expensive)

- The two versions has been designed to be upword compatible.

1.3.1 SIC machine architecture

Every machine architecture includes

a) memory

b) Registers

c) Data formats

d) Instruction formats

e) Addressing modes

f) Instruction set

g) Input and output

a) Memory

→ memory consists of 8 bit bytes

→ Any 3 consecutive bytes form a word (24 bits)

→ All addresses on SIC are byte addresses

→ words are addressed by the location of their lowest numbered byte

→ Total of 32,768 (2^{15}) bytes in the computer memory

∵ 15 - line address bus

b) Registers

→ Five registers, all of which have special uses

→ Each register is 24 bits in length

→ table shows the mnemonic, number and uses of each register.

Mnemonic	Number	Uses
A Accumulator	0	Used for arithmetic operations
X Index Register	1	Used for addressing
L Linkage Register	2	JSUB – Jump to subroutine instruction store return address
PC Program Counter	8	Contain the address of the next instruction to be fetched for execution
SW status word	9	Contains variety of information including a condition code in comp instruction

c) Data formats

→ Integers are stored as 24-bit binary number.

(0 - 2^{24} -1 ⇒ 0 to 16G-1)

→ Negative value are represented as 2's complement (8 bit representation)

Ex: -24 is represented as

24 = 0 0 0 1 1 0 0 0

1's complement 1 1 1 0 0 1 1 1

+ 1

1 1 1 0 1 0 0 0 → 232

→ characters are stored using their 8-bit ASCII codes

→ There is no floating-point hardware on the standard version of SIC

Ex:- 5 = 0000 0000 0000 0000 0000 0101

-5 = 1111 1111 1111 1111 1111 1011

A = 0100 0001 (65)

d) Instruction formats

→ All machine instructions on the standard version of SIC ~~are~~ have 24·bit format

8	1	15
Opcode	x	Address

x → indicates indexed addressing mode

e) Addressing modes

→ Two addressing modes based on x bit

- Direct Addressing
- Indexed addressing

mode	Indication	Target address (TA)
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (x) parentheses indicate the content of a register or memory location

→ Ex:- LDA TEN ; LDA = 00 (opcode)

8	1	15
0000 0000	0	001 0000 0000 0000
opcode	x	address

Target address / Effective address = 1000 ie contents of the address 1000 is loaded to accumulator

→ Indexed addressing mode

Ex: STCH BUFFER, X ; opcode for STCH = 54

BUFFER = 1000

0101 0100	1	001 0000 0000 0000
5 H	x	1 0 0 0
opcode		address (BUFFER)

TA = address + (x)

= 1000 + content of the index register X

ie the Accumulator content, the character is loaded to the effective address.

4) Instruction set

(i) load and store : LDA, LDX, STA, STX

(ii) Integer Arithmetic operations : ADD, SUB, MUL, DIV

- Arithmetic operation involve register A and a word in memory with the result being left in the register

Ex: ADD WORD ; A ← A + WORD

ie adds register A contents with ~~word~~ WORD and result is stored in register A

(iii) Comparison operations : COMP

- compares the value in register A with memory and sets a condition code (CC) accordingly

Ex: COMP WORD ; compares A's contents with WORD and sets CC as < = >

(iv) Conditional jump instructions : JLT, JEQ, JGT

- these instructions will test the setting of CC and jump accordingly

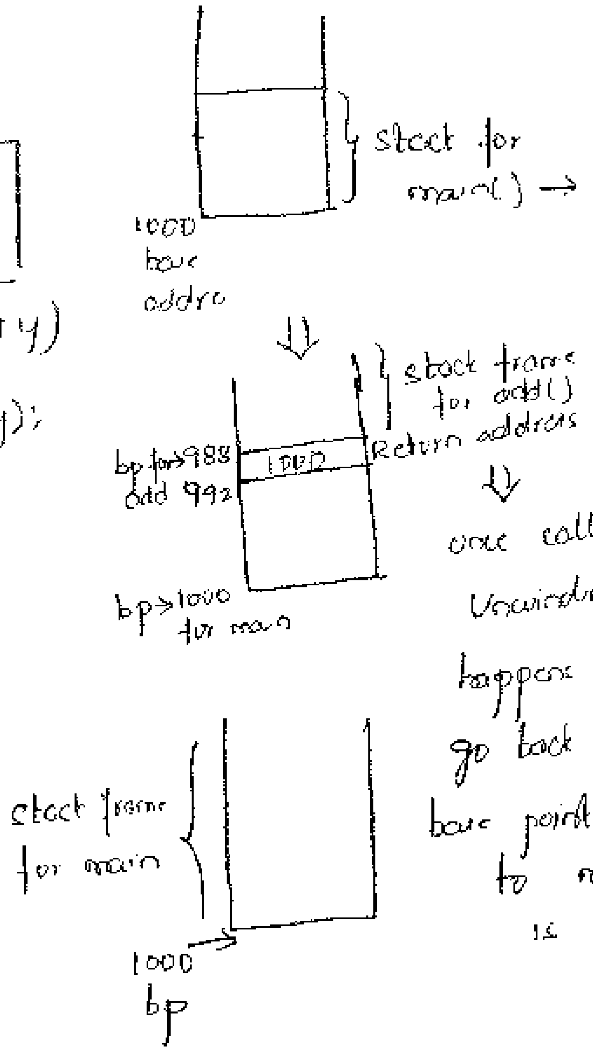
(v) subroutine linkage instructions ; JSUB, RSUB

- JSUB - jumps to the subroutine by placing the return address in register L. (Fn. call)

- RSUB - returns by jumping to the address contained in register L (Fn. return to the caller)

Ex:

```
void main()
{
    Add(1,2);
}
        Fn call
Add (int x, int y)
{ return (x+y);
Fn Return }
```



once call returns ie Unwinding of stack happens it has to go back to caller noting base pointer ~~of~~ ~~add~~ point to main so 1000 is stored.

(g) Input and Output

→ Input and output is performed by transfering 1 byte data at a time to or from the rightmost 8 bits of register A.

→ Each device is assigned a unique 8-bit code

→ There are three I/O instruction which specific the device code as an operand

(i) <u>TD (Test Device)</u> : checks whether the addressed device is ready to send or receive a byte of data. CC (condition code in SW register) is set according (< =)

< → device is ready to send/receive

= → device is not ready.

→ A program has to wait until the device is ready, then execute a Read Data (RD) and write Data (WD) instructions.

→ This sequence should be continued for each byte of data. (I/O).

→ RD : Transfers a byte of data from i/p device into rightmost byte of register A (RD INDEV STA DATA)

→ WD : Byte of data is loaded into rightmost byte of reg. A and then written to output device (LDA DATA & WD OUTDEV)

Ex: 1) SIC instructions for data movement operations (no memory - memory move instructions)

```
        LDA     FIVE        ; Load constant 5 into register A
        STA     ALPHA       ; Store in Alpha
        LDCH    CHAR2       ; load character '2' into reg A
        STCH    C1          ; store in character variable C1
        .
        .
        .
ALPHA   RESW    1           ; one-word variable
FIVE    WORD    5           ; one word constant
CHAR2   BYTE    C'2'        ; one-byte constant
C1      RESB    1           ; one-byte variable
```

⇓ Same can be written as

```
        LDX     FIVE
        STX     ALPHA

            Or

        LDL     FIVE
        STL     ALPHA
```

A — Accumulator

X — indexed register

L - linkage register

2) SIC instructions for arithmetic operations

~~Delta~~ = BETA = ALPHA + INCR − 1

DELTA = GAMMA + INCR − 1

```
        LDA     ALPHA     ; loads Alpha into register A
        ADD     INCR      ; A ← (A)+(INCR)
        SUB     ONE       ; A ← (A)−1
        STA     BETA      ; BETA ← (A)
        LDA     GAMMA     ; A ← (GAMMA)
        ADD     INCR      ; A ← (A)+(INCR)
        SUB     ONE       ; A ← (A)−1
        STA     DELTA       DELTA ← (A)
         :
ONE     WORD    1
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

3) SIC instructions for looping and indexed operations (program to copy 11-byte character string to another string)

```
// LDX  ZERO ;  initialize index register
   j=0;
 for (i=0; s1[i]!='\0'; i++)
     s2[j++] = s1[i];
     s2[j] = '\0';
```

```
        LDX    ZERO   ; initialize index register to 0
LOOP    LDCH   STR1, X ; copies the first character of str1 to reg. A
                         (TA = (address) → content of first byte
                          of str1)
        STCH   STR2, X ; store the first character into STR2
        TIX    ELEVEN ; Add 1 to index, compare to 11
                         X = 0+1 = 1 ; 1 ↔ 11 cc will
                         be set as <
        JLT    LOOP   ; repeat if index is < 11
                                      (x)
          .
          .
          .
                             space
                             ↓
STR1    BYTE   C 'HELLO WORLD'
STR2    RESB   11
ZERO    WORD   0
ELEVEN  WORD   11
```

h) Program to add 2 arrays of 100 words each and store it in another array. Each word is 3 bytes ∴ 100 words : 3 x 100 = 300 bytes

$G = A + B$;

```
ADDLOOP LDX    ~~ZE~~INDEX ; initialize index value X := 0
        LDA    ALPHA, X ; A ← (ALPHA)
        ADD    BETA, X  ; A ← (A) + (BETA) at 0th byte (index)
        STA    GAMMA, X ; G ← (A) at 0th byte (index value)
        LDA    INDEX    ; A ← 0
        ADD    THREE    ; A ← (A)+3 = 3 → m
        STA    INDEX    ; INDEX = 3
        COMP   K300     ; A ↔ K300 ie 3 ↔ 300 CC = <
        JLT    ADDLOOP  ; repeat loop, now x = 3rd byte

K300    WORD   300          THREE   WORD   3
INDEX   RESW   1
                            BETA    RESW   100
ALPHA   RESW   100
                            GAMMA   RESW   100
```

5) To read one byte of data from input device and copies it to device 05

```
INLOOP    TD     INDEV      ; Test input device
          JEQ    INLOOP     ; CC := then loop until device ready
          RD     INDEV      ; once ready, read a byte into reg A
          STCH   DATA       ; Store it in data(memory)
            :
                            ; Test output device
OUTLOOP   TD     OUTDEV     ; CC := then loop until device ready
          JEQ    OUTLOOP    ; load data byte into reg A
          LDCH   DATA       ; write one byte to output device
          WD     OUTDEV
            :

INDEV     BYTE   X'F1'
OUTDEV    BYTE   X'05'
DATA      RESB   1
```

6) Subroutine call to read a 100-byte record from an input device into memory.

```
          JSUB   READ       ; call Read subroutine wherein
            :                 it stores the return address in
                              linkage register
READ      LDX    ZERO       ; X ← 0
RLOOP     TD     INDEV      ; Test input device
          JEQ    RLOOP      ; cc:=, loop until device is ready
          RD     INDEV      ; read one byte into reg A
          STCH   RECORD,X   ; store it into RECORD at Xth address
          TIX    K100       ; X = (X)+1 & 1 <-> 100
                                              compare
          JLT    RLOOP      ; cc:< then loop back
          RSUB              ; Exit from subroutine; it
            :                 returns to the address stored in
                              linkage register
```

```
              :
INDEX     BYTE      X'F1'
RECORD    RESB      100
ZERO      WORD      0
K100      WORD      100
```

1.3.2 : SIC|XE machine Architecture

→ SIC|XE : Simple Instructional computer with Extra Equipments

a) memory

b) Registers

c) Data formats

d) Instruction formats

e) Addressing modes

f) Instruction set

g) Input and output

a) memory

→ memory consists of 8 bit bytes

→ 3 consecutive bytes form a word (24 bits)

→ All addresses are byte addresses

→ words are addressed by the location of their lowest numbered byte

→ Total of 1 MB (2^{20} bytes) in the memory. (20 $\overset{\text{line}}{\text{bit}}$ address bus) which leads to change in instruction formats and addressing modes.

b) Registers

→ There are 9 registers

→ Each register is 24 bits in length except Floating Point reg

→ The registers are A, X, L, B, S, T, F, PC & SW

Mnemonic	Number	Uses
A Accumulator	0	Used for arithmetic operations
X Index Register	1	used for addressing (indexed)
L Linkage register	2	used to store the return address for JSUB instruction
B Base register	3	used for addressing
S General Register	4	General working register - no special use
T General Register	5	General working register - no special use
F Floating Point Accumulator	6	~~General working register~~ Floating point accumulator (48 bits)
PC Program Counter	8	Contains the address of the next instruction to be fetched for execution
SW Status Word	9	Contains a variety of information including a condition code (CC)

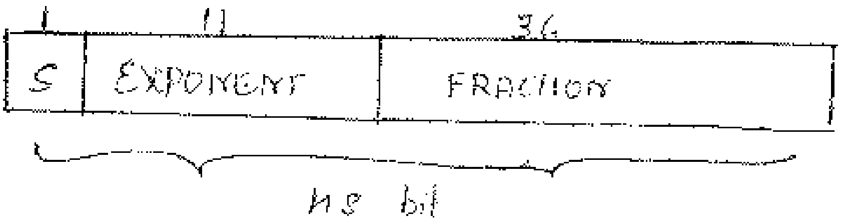
c) Data formats :

→ Integers are stored as 24-bit binary numbers

→ negative values are represented as 2's complement (1's complement +1)

→ characters are stored using their 8-bit ASCII codes

→ There is a 48 bit floating point data type



→ The fraction is interpreted as a value between 0 and 1

→ The assumed binary point is immediately before the higher order bit

→ For normalized floating point numbers, the higher order bit of the fraction must be 1

→ The exponent is interpreted as an unsigned binary number between 0 and 2047 $(0 - (2^{11} - 1))$

→ If the expression has value e, fraction f and the absolute value of number ~~is~~ represented is

$$f * 2^{(e-1024)}$$

→ The sign of floating point number is indicated by s (s = 0 (+ve) and 1 (-ve))

Ex:- 5 = 0000 0000 0000 0000 0000 0101

-5 = 1111 1111 1111 1111 1111 1011

A = 0100 0001 (65)

Ex: 4.89 representation

As we know from computer organization it is represented as $\pm m B^{e}$

↓ ↳ Base (2)

Fraction (mantissa)

1) Represent 4 in binary form → 100

2) Convert 0.89 into binary form until it repeats or until we get 36 bits which represents the fraction part

100. 111000111110101110000101000111110101 11

000010100

3) normalization has to be done but not always ∵ they have specified that binary point is immediately before the higher order bit

ie 100. 111000 . . .

$0.\underbrace{1001110001111101011100001010001111010}_{\text{fraction}} \times 2^{3}$ → Exponent

Note: for normalized floating point number it will be as $1.00111000111111 \ldots \times 2^{2}$

ie $f * 2^{e \pm 1024} = 0.100111 \ldots \times 2^{3+1024}$

$= 0.100111 \ldots \times 2^{1027}$

1	11 bit	36 bit
0	10000000011	1001110001111101011100001010001111010
S	exponent	fraction

0.89×2	→	
0.78×2	→	1 ie 1.78
0.56×2	→	1
0.12×2	→	1
0.24×2	→	0
0.48×2	→	0
0.96×2	→	0
0.92×2	→	1
0.84×2	→	1
0.68×2	→	1
0.36×2	→	1
0.72×2	→	0
0.44×2	→	1
0.88×2	→	0
0.76×2	→	1
0.52×2	→	1
0.04×2	→	1
0.08×2	→	0
0.16×2	→	0
0.32×2	→	0
0.64×2	→	0
0.28×2	→	1
0.56×2	→	0
0.12×2	→	1
0.24×2	→	0
0.48×2	→	0
0.96×2	→	continue as above

0111101011100001010

0

$(1027)_2$ → 10000000011

$\nearrow$ s = 1(negative)

c) Represent −0.000489 in binary format

Given −0.000489
↓

→ Represent 0 as binary 0

→ Represent 0.000489 as binary

= 0.0000000000100000000011000000
111100000001111110.

$\overbrace{}^{\text{fraction}}$

= .1000000000110000001111000000001
1111110x2^{-10}

⇒ $f \times 2^{e+1024} = .10000\cdots \times 2^{-10+1024}$

$= \underbrace{.10000\cdots}_{\text{fraction}} \times 2^{1014 \to E}$ $(1014)_2 = 01111110110$

.000489 X2
0.000978X2 → 0
0.001956X2 → 0
0.003912X2 → 0
0.007820X2 → 0
0.015648X2 → 0
0.031296X2 → 0
0.062592X2 → 0
0.125184X2 → 0
0.250368X2 → 0
0.500736X2 → 0
0.001472X2 — 01
0.002944X2 → 0
.
.

	11	36
1	01111110110	10000000000110000000111100000000111111
S	Exponent	fraction

10

d) Instruction formats.

→ Since the memory used by SIC/XE may be 2^{20} bytes, the instruction format of SIC is not enough.

→ There are two possible options

(i) Either use some form of relative addressing

(ii) Extend the address field to 20 bits

→ if e=0, then format 3

→ if e=4, then format 4

1) Format 1 (1 byte)

8 bits
opcode

2) Format 2 (2 byte)

8	4	4 bits
opcode	r1	r2

3) Format 3 (3 byte)

6	1	1	1	1	1	1	12 bits
opcode	n	i	x	b	p	e	dis

Note: e=0

4) Format 4 (4 byte)

6	1	1	1	1	1	1	20 bits
opcode	n	i	x	b	p	e	address

Ex: $\overset{\rightarrow 4C}{\text{RSUB}}$ (returns to subroutine)

⇒ it returns to the address stored in linkage register

0100 1100

4 C → object code

Ex: COMPR A,S (compare the contents of registers A & S)

opcode of COMPR = A0

8	4	4
1010 0000	0000	0100

A 0 0 4 → obj code

Ex: LDA #3 (load 3 to A)

6	1	1	1	1	1	1	12
000000	0	1	0	0	0	0	0000 0000 0011
opcode	n	i	x	b	p	e	0 0 3

01003 → object code

Ex: +JSUB RDREC (Jump to address 1036)

opcode JSUB — 48

	1 1 1 1 1 1	20
0100 10	11 0 0 0 1	0000 0001 0000 0011 0110
opcode	n i x b p e	address

4 B 1 0 1 0 3 6

object code is 4B101036

e) Addressing modes

	MODE	INDICATION	TARGET ADDRESS CALCULATION
1.	Base Relative	b=1, p=0	TA = (B) + displacement (0 <= disp ≤ 4095)
2.	Program Counter Relative	b=0, p=1	TA = (P) + displacement (-2048 ≤ disp ≤ 2047)
3.	Direct Addressing	b=0, p=0 (for format 3)	TA = displacement
		b=0, p=0 (for format 4)	TA = Address field

4.	Base Relative Indexed addressing	b=1, p=0
		x=1
			TA = (B) + (x) + displacement

5.	Program Relative Indexed addressing	b=0, p=1
		x=1
			TA = (P) + (x) + displacement

6.	Immediate addressing	i=1, n=0	Target address itself used as TA = operand value (No memory reference)
7	Indirect addressing	i=0, n=1	TA = displacement value

8.	Simple addressing	i=0, n=0
		OR
		i=1, n=1
			TA = location of operand

Note: → Format 3 :

↳ in Base relative, disp is interpreted as 12 bit unsigned integer (1)

↳ in PC relative disp is interpreted as 12-bit signed integer & negative numbers in 2's complete (2)

Special symbols indication

1) # : Immediate address

2) @ : Indirect address

3) + : Format 4

4) * : The current value of PC

5) c' ': character string

6) op m, x : x-denotes the index register

7) base : Base register

~~Instruction set~~ :

Note: Immediate addressing (i=1, n=0) → Target address itself is used as the operand value (no memory reference is performed)

→ Indirect addressing (i=0, n=1) → the word at the location given by the target address is fetched and the value contained in this word is then taken as the address of the operand value.

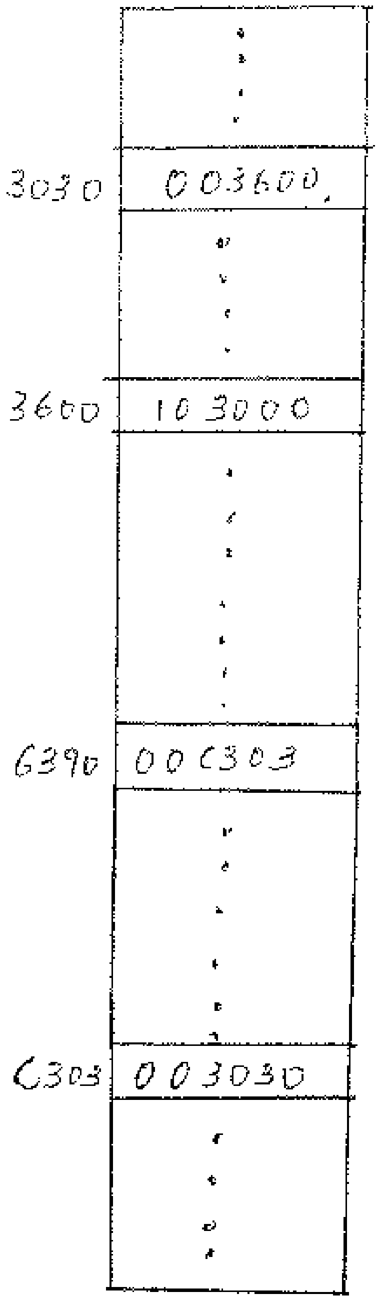
→ Indexing cannot be used with immediate or indirect addressing mode.

** → we can't set both b=1 & p=1 which is invalid instruction set.

examples of SIC/XE instructions and addressing modes

(B) = 006000
(PC) = 003000
(X) = 000090

Hex	Machine instruction										
	Binary									Value loaded into Reg. A	Mode
	6	1	1	1	1	1	1	12/20			
	opcode	n	i	x	b	p	e	disp/ address	Target address		
032600	$\underbrace{0000}_{0}00$	$\underbrace{1}_{3}$	1	0	0	$\underbrace{1}_{2}$	0	$\underbrace{0110}_{6}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}$	3600	103000	Program Counter Relative TA = (PC) + displacement = 003000 + 600 = 3600
03C300	0000 00	1	1	1	1	0	0	0011 0000 0000	6390	00C303	Base relative Indexed TA = (B) + (X) + disp = 006000 + 000090 + 300 = 6390
022030	0000 00	1	0	0	0	1	0	0000 0011 0000	3030 ↓ contain 3600) contain	103000 ↑	Indirect + Program relative TA = (PC) + disp = 003000 + 030 = 3030
010030	0000 00	0	1	0	0	0	0	0000 0011 0000	30	000030	Immediate addressing TA is used as operand value
003600	0000 00	0	0	0	0	1	1	0110 0000 0000	3600	103000	PC relative TA = (PC) + disp = 003000 + 600 = 3600
0310C303	0000 00	1	1	0	0	0	1	0000 1100 0011 0000 0011	C303	003030	Simple addressing TA = location of operand



(B) = 006000

(PC) = 003000

(X) = 000090

Fig:- contents of registers B, PC and X & memory locations

f) Instruction set

- → Load and store instruction : LDA, LDX, STA, STX, LDB, STB
- → Integer and Floating point arithmetic operations :

  ADD, SUB, MUL, DIV, ADDF, SUBF, MULF, DIVF

- → Register move instructions (Rmo) ⇒ register to- register operations such as ADDR, SUBR, MULR, DIVR
- → A special supervisor call (SVC) instruction is provided. Executing this instruction generates an interrupt that can be used for communication with the operating system.

→ Comparision instruction : COMP, COMPR, COMPF

→ Conditional jump instructions : JLT, JEQ, JGT

→ subroutine linkage instruction : JSUB, RSUB

g) Input and output

→ Input and output is performed by transferring 1 byte of data at a time to or from, the rightmost 8 bits of register A.

→ Each device is assigned a unique 8-bit code

→ Three I/O instructions which specifies the device code as an operand

(i) TD (Test Device) → Tests whether the addressed device is ready to send or receive a byte of data and sets the condition code (CC)

< : Device is ready to send/receive

= : Device is not ready

→ Test continues until the device is ready.

→ Once ready, either <u>RD (Read data)</u> : ~~read~~ Transfer data from input device or keyboard into rightmost byte of register A and stored in buffer if required (RD INDEV & STA DATA)

→ WD (write data) : a byte of data is loaded into the rightmost byte of register A and then written to the addressed device (LDA DATA & WD OUTDEV)

* → There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. This allows overlap of computing and I/O, resulting in more efficient system operation.

↳ SIO ↔ start I/O

↳ TIO ↔ Test I/O

↳ HIO ↔ Halt I/O

~~SIC~~ SIC/XE programs

1) Data movement operations

```
        LDA     #5      ; loads value 5 into register A
        STA     ALPHA   ; store in alpha : A ← (A)+(ALPHA)
                                               5+(ALPHA)
        LDA     #90     , load ascii code for 'z' into A
        STCH    C1      ; store in character variable C1
         :
ALPHA   RESW    1       ; one word variable
C1      RESB    1       ; one byte variable
```

3) Arithmetic operations (Beta = alpha + incr - 1)

```
        LDS     INCR    ; load value q incr to ~~A~~ S
        LDA     ALPHA   ; load value q alpha to A
        ADDR    S, A    ; A ← (A)+(S)
              source destination
        SUB     #1      ; A ← (A)-1
        STA     BETA    ; BETA ← (A)
         :
```

4) Looping and indexed operations

GAMMA = ALPHA + BETA where ALPHA and BETA are arrays q 100 words each. (100×3 = 300 bytes)

```
        LDS     #3        ; S = 3
        LDT     #300      ; T = 300
        LDX     #0        ; X = 0 → which specifies index value
ADDLOOP LDA     ALPHA,X   ; A ← (ALPHA) at the specified address q index register

        ADD     BETA, X   ; A ← (A)+(BETA)
        STA     GAMMA,X   ; GAMMA ← (A) at specified index value(0)
        ADDR    S,X       ; X ← (X)+(S) = 0+3 = 3 (index register address is 3)
        COMPR   X,T       ; (X) ↔ (T) compared ie 3<300, cc: <
        JLT     ADDLOOP   ; repeat loop till 300 = 300
```

```
            :
ALPHA     RESW     100
BETA      RESW     100
GAMMA     RESW     100
```

5) To read one byte of data from input device F1 and copies it to device 05

```
INLOOP    TD       INDEV
          JEQ      INLOOP
          RD       INDEV
          STCH     DATA
            :
OUTLOOP   TD       OUTDEV
          JEQ      OUTLOOP
          LDCH     DATA
          WD       OUTDEV
            :
INDEV     BYTE     X'F1'
OUTDEV    BYTE     X'05'
DATA      RESB     1
```

6) subroutine call to read 100-byte record from an input device into memory.

```
          JSUB     READ
            :
READ      LDX      #0
          LDT      #100
RLOOP     TD       INDEV
          JEQ      RLOOP
          RD       INDEV
          STCH    RECORD, X
          TIXR    T
          JLT     RLOOP
          RSUB
            :
MYDEV     BYTE    X'F1'
RECORD    RESB    100
```

Write a SIC and SIC/XE program to copy 'SYSTEM SOFTWARE' to another string

a) SIC program

```
          LDX     ZERO
LOOP      LDCH    STR1, X
          STCH    STR2, X
          TIX     FIFTEEN
          JLT     LOOP
            :
STR1      BYTE    C' SYSTEM SOFTWARE'
STR2      RESB    15
ZERO      WORD    0
FIFTEEN   WORD    15
```

b) SIC/XE program

```
          LDX     #0
          LDT     #15
LOOP      LDCH    STR1, X
          STCH    STR2, X
          TIXR    T
          JLT     LOOP
            :
STR1      BYTE    C 'SYSTEM SOFTWARE'
STR2      RESB    15
```

Exercises 1.3

1 Write a sequence of instructions for SIC to set ALPHA equal to the product of BETA and GAMMA. Assume ALPHA, BETA and GAMMA are 1 word (ALPHA = BETA * GAMMA)

```
LDA    BETA
MUL    GAMMA
STA    ALPHA
 :
ALPHA   RESW  1
BETA    RESW  1
GAMMA   RESW  1
```

2. Write a sequence of instructions for SIC/XE to set ALPHA equal to 4 * BETA - 9. ALPHA, BETA and GAMMA are 1 word. Use immediate addressing for the constants (A = 4 * B - 9)

```
LDA    BETA
LDS    #4
MULR   S, A
 SUB   #9
 STA   ALPHA
 :
ALPHA   RESW  1
```

3 Write SIC instruction to swap the values of ALPHA and BETA.

```
        LDA     ALPHA
        STA     GAMMA
        LDA     BETA
        STA     ALPHA
        LDA     GAMMA
        STA     BETA
         :
ALPHA           RESW    1
BETA            RESW    1
GAMMA           RESW    1
```

4 Write a sequence of instructions for SIC to set ALPHA equal to the integer portion of BETA ÷ GAMMA. ALPHA, BETA, GAMMA are 1 word each

```
        LDA     BETA
        DIV     GAMMA
        STA     ALPHA
         :
ALPHA           RESW    1
BETA            RESW    1
GAMMA           RESW    1
```

5. Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the integer portion of the quotient and DELTA to the remainder. Use register-to-register instructions to make the calculation as efficient as possible.

Ex: B = 5
G = 2

```
LDA    BETA      ; A = 5
~~DIVF   GAMMA~~
LDS    GAMMA     ; S = 2
DIVR   S, A      ; A = A/S = 5/2 = 2
STA    ALPHA     ; A = 2
MULR   S, A      ; A = A*S = 2*2 = 4
LDS    BETA      ; S = 5
SUBR   A, S      ; S = S - A = 5 - 4 = 1
STS    DELTA     ; DELTA = 1
  :
ALPHA  RESW  1
BETA   RESW  1
GAMMA  RESW  1
DELTA  RESW  1
```

Note:

// To find the remainder

Quotient = Dividend / Divisor

Remainder = Dividend − (Quotient * Divisor)

Ex:- Divident = 10, Divisor = 3, Q = 10/3 = 3 ; R = 10 − (3 * 3) = 10 − 9 = 1

Dividend = 15, Divisor = 3, Q = 15/3 = 5 ; R = 15 − (5 * 3) = 15 − 15 = 0

6. Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the value of the quotient, rounded to the nearest integer. Use register-to-register instructions to make the calculation as efficient as possible

```
          LDF      BETA
          DIVF     GAMMA
          FIX
          STA      ALPHA
            :
          ALPHA    RESW  1
          BETA     RESW  1
          GAMMA    RESW  1
```

7. Write a sequence of instructions for SIC to clear a 20-byte string to all blanks

```
          LDX      ZERO
LOOP      LDCH     BLANK
          STCH     STR1 , X
          TIX      TWENTY        ; ADD 1 to index and compare with 20 & set CC : < = >
          JLT      LOOP          ; LOOP if index<100
            :
          STR1     RESW   20
          BLANK    BYTE   C ' '
          ZERO     WORD   0
          TWENTY   WORD   20
```

8 Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

```
          LDT     #20
          LDX     #0
CLOOP     LDCH    #0
          STCH    STRI, X
          TIXR    T
          JLT     CLOOP
            :
          STRI    RESW   20
```

9. Suppose that ALPHA is an array of 100 words. Write a sequence of SIC instruction to set all 100 elements of the array to 0.

```
          LDA     ZERO
          STA     INDEX
LOOP      LDX     INDEX
          LDA     ZERO
          STA     ALPHA, X
          LDA     INDEX
          ADD     THREE
          STA     INDEX
          COMP    K300
          TIX     TWENTY
          JLT     LOOP
```

```
     :
INDEX     RESW    1
ALPHA     RESW    100
ZERO      WORD    0
K300      WORD    100
THREE     WORD    3
```

10 ALPHA is an array of 100 words. write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0. Use immediate addressing and register-to-register instructions to make the process as efficient as possible

```
        LDS     #3
        LDT     #300
        LDX     #0
LOOP    LDA     #0
        STA     ALPHA, X
        ADDR    S, X
        COMPR   X, T
        JLT     LOOP
          :
          :
ALPHA   RESW    100
```

11 ALPHA is an array of 100 words. write a sequence of SIC/XE instructions to arrange the 100 words in ascending order and store the result in an array BETA of 100 words.

~~LDS #3~~

~~LDT #300~~

~~LDX #0~~

LOOP ~~LDA ALPHA, X~~

~~MUL #4~~

12. ALPHA and BETA are the two arrays of 100 words. Another array of GAMMA elements are obtained by multiplying the corresponding ALPHA element by 4 and adding the corresponding BETA elements. Write the SIC|XE instructions for the same.

```
                LDS     #3

                LDT     #300

                LDX     #0

      LOOP      LDA     ALPHA, X

                MUL     #4

                ADD     BETA, X

                STA     GAMMA, X

                ADDR    S, X       ; X ← X +3

                COMPR   X, T       ; X ≥ 300

                JLT     LOOP
                 :
                 :
                ALPHA     RESW   100

                BETA      RESW   100

                GAMMA     RESW   100
```

13. ALPHA is an array of 100 words. write a sequence of SIC/XE instructions to find the maximum element in the array and store results in MAX.

```
          LDS    #3
          LDT    #300
          LDX    #0
LOOP      LDA    ALPHA,X
          COMP   MAX
          JLT    NOMAX
          STA    MAX
NOMAX     ADDR   S,X
          COMPR  X,T
          JLT    LOOP
            :
ALPHA     RESW   100
MAX       WORD   -32768
```

note : COMP MAX ⇒ indicates Accumulator value is compared with MAX and set the CC (condition code) ie CC ← < = > of (A) ? (MAX). Based on cc value check the condition ie JLT, JGT, JEQ

→ COMPR X,T ⇒ Register value are compared and X : (X) ? (T) and cc is set : < = > and Jump instruction is called

Explanation

ALPHA = {10, 20, 30, 40, ... 32768, . . . }

↑ 0 ↑ 3

index

Each value is 1 word = 3 byte

100 words = 3×100 = 300 byte

index has to be incremented by 3.

ie initially x = 0, 3, 6, 9 300

According to code ; S = 3, T = 100, X = 0

1st iteration LOOP ; Accumulator (A) = 10 at 0th position

COMP MAX ; 10 ? 32768 sets CC : <

JLT NOMAX

NOMAX ; ADDR S, X ; X ← (X) + S = 0+3 → increment by 3 (next ~~element~~

element ie 20

COMPR X, T ; X ← (X) ? ( ) T

← 300 ? 300 CC ; <

⇓

To check whether the array index has come to an end.

Iteration 2) LOOP : A ← 20 which is at 3rd position

LDA ALPHA 3 → value at 3rd position

COMP MAX ; 20 ? 32768 - CC : <

JLT NOMAX

:

Continues

14. A RECORD contains a 100-byte record. write a subroutine for SIC that will write this record onto device 05.

```
                JSUB    WRREC
                  :
                  :
WRREC     LDX     ZERO          ; initialize index register = 0
LOOP      TD      OUTPUT        ; Test output device
          JEQ     WLOOP         ; Loop if device is busy
          LDCH    RECORD, X     ; load One byte to Accumulator
          WD      OUTPUT        ; write one byte to device
          TIX     LENGTH        ; Add 1 to index & compare to 100
          JLT     LOOP          ; Loop if index is < 100
          RSUB                  ; Exit from subroutine
            :
          ZERO    WORD   0
          LENGTH  WORD   1
          OUTPUT  BYTE   X '05'
          RECORD  RESB   100
```

Note: To read and write the data between the device, the device has to be ready to perform. This is done by using TD (Test device) instruction; status of the device is tested and CC is set to either {< (ready) = (not ready)} If ready then RD is executed. ⇒ reads 1 byte of data from device into rightmost byte of register A. If the input device is character-oriented (keyboard), the value placed in reg. A is the ASCII code for the character that was read. IIIly WD → o/p device byte is loaded into the rightmost byte of register A. and

15 write a subroutine for SIC/XE to write a RECORD of 100 byte onto output device 05.

```
                JSUB     WRREC    ; Jump to subroutine

WRREC           LDX      #0
                LDT      #100
LOOP            TD       OUTPUT
                JEQ      LOOP
                LDCH     RECORD, X
                WD       OUTPUT
                TIXR     T
                JLT      LOOP
                RSUB
                 :
                OUTPUT   BYTE     X '05'
                RECORD   RESB     100
```

16. write a subroutine for SIC that will read a record into a buffer. The record may be any length from 1 to 100 byte. The end of the record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH.

```
                JSUB     RDREC
                 :
                 :
RDREC           LDX      ZERO
RLOOP           TD       INDEV
```

```
            JEQ     RLOOP
            RD      INDEV
            COMP    NULL
            JEQ     EXIT
            STCH    BUFFER,X
            TIX     K100
            JLT     RLOOP
EXIT        STX     LENGTH
            RSUB
             :
            ZERO    WORD  '0
            NULL    WORD  0
            K100    WORD  1
            INDEV   BYTE  X'F1'
            LENGTH  RESW  1
            BUFFER  RESB  100
```

17) SIC|XE :

```
            JSUB    RDREC
             :
RDREC       LDX     #0
            LDT     #100
            LDS     #0
RLOOP       TD      INDEV
            JEQ     RLOOP
            RD      INDEV
            COMPR.  A,S
            JEQ     EXIT
            STCH    BUFFER,X
            TIXR    T
            JLT     RLOOP
EXIT        STX     LENGTH
            RSUB
             :
INDEV       BYTE    X'F1'
LENGTH      RESW    1
BUFFER      RESB    100
```

11 To sort an array of 10 words in on ascending order

```
OUTER    LDX    INDEX ;
         LDS    ARR1, X ;
         LDX    #0
INNER    LDT    ARR1, X
         COMR   S,T
         JLT    LOOP
         JEQ    LOOP
         RMO    S, A
         RMO    T, S
         RMO    A,T
         RMO    X, A
         LDX    INDEX
         STS    ARR1,X
         RMO    A,X
         STT    ARR1,X
LOOP     RMO    X, A
         ADD    #3
         COMP   LENGTH
         RMO    A,X
         JLT    INNER
         LDA    INDEX
         ADD    #3
         COMP   LENGTH
         STA    INDEX
         JLT    OUTER
      :
INDEX    WORD   0
ARR1     RESW   10
LENGTH   WORD   30
```

QP

1 Write a SIC program to copy string 'SYSTEM SOFTWARE to another string.

```
        LDX    ZERO        ; Initialize x to zero
MOVECH  LDCH   STR1,X      ; x specific indexing
        STCH   STR2,X
        TIX    FIFTEEN     ; increment x and compare
                             with 15
        JLT    MOVECH

STR1    BYTE   C 'SYSTEM SOFTWARE'
STR2    RESB   15
ZERO    WORD   0
FIFTEEN WORD   15
```

Comparison Chart of SIC and SIC/XE machine

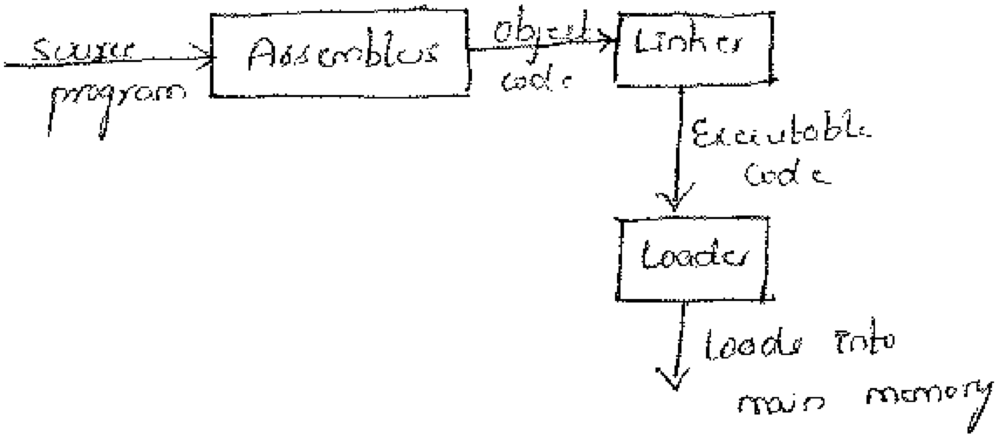
Specification	SIC	SIC/XE
Memory	• **Word size:** 3 bytes (24 bits) • **Total size:** 32,768 bytes (215). Thus any memory address will need at most 15 bits to be referenced ('almost' four hex characters).	• **Word size:** 3 bytes (24 bits) • **Total size:** 32,768 bytes (215). Thus any memory address will need at most 15 bits to be referenced ('almost' four hex characters).
Register	• **Total Registers:** 5 • **Accumulator (A):** Used for most of the operations (number 0) • **Index (X):** Used for indexed addressing (number 1) • **Linkage (L):** Stores return addresses for JSUB (number 2) • **Program Counter (PC):** Address for next instruction (number 8) • **Status Word (SW):** Information and condition codes (number 9).	• **Total Registers:** 9 , same 5 from SIC plus 4 additional ones. • **Base (B):** Used for base-relative addressing (number 3) • **General (S and T):** General use (numbers 4 and 5 resp.) • **Floating Point Accumulator (F):** Used for floating point arithmetic, 48 bits long (number 6)
Instruction Formats	• Only one instruction format of **24 bits** (3 bytes / 1 word) • Opcode: first 8 bits, direct translation from the Operation Code Table • Flag (x): next bit indicates address mode (0 direct - 1 indexed) • Address: next 15 bits, indicate address of operand according to address mode.	• Four instruction formats • **Format 1 (1 byte)**: contains only operation code (straight from table) • **Format 2** (2 bytes): first eight bits for operation code, next four for register 1 and following four for register 2. • The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5). • If the operation uses only one register the last hex digit becomes \0" (ie,TIXR T becomes B850) • **Format 3** (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. • Operation code uses only 6 bits, thus the second hex digit will be a affected by the values of the first two flags (n and i) • The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. • The last flag e indicates the instruction format (0 for 3 and 1 for 4) • **Format 4** (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented

Specification	SIC	SIC/XE
Addressing Modes	• Only two possible addressing modes • **Direct** (x = 0): operand address goes as it is Indexed (x = 1): value to be added to the value stored at the register x to obtain real address of the operand.	• five possible addressing modes plus combinations (see page 11 for examples) • **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, If set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format • **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1) • **Immediate** (i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction) • **Indirect** (i = 0, n = 1): The operand value points to an address that holds the address for the operand value • **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.
Assembler Considerations	• Operation code gets translated directly from table (no need to check other bits) • x bit dependent on the addressing mode of the operand. If indexed the code will have to indicate it with \,X" after the operand name (ie. BUFFER,X) • The last 3 hex digits of the address will remain the same, the first hex digit (leftmost) will change if the address is indexed (first bit becomes one, thus the hex digit increases by 8). Ie, if the address of the operand is 124A and the addressing is indexed, the object code will indicate 924A.	• Operation code gets translated directly from table. While the first hex digit remains the same, the second one can change according to the values of the n and i flags. Thus, we can add 1, 2 or 3 to the operation code. • **Direct addressing** is mainly used in extended format (format 4) and is indicated with a \+" before the operand (an indication that the format is 4, which will also make the e flag to be 1). • **Relative:** for Base relative, the instruction BASE will precede the current instruction. • Any other format, except immediate, will be considered Program Counter relative. If the displacement with respect to the PC does not t into the 12 bits, the assembler should try to compute the displacement with respect to the Base register. If neither case works, the instruction should be extended to format 4, where the addressing mode becomes direct.

Specification	SIC	SIC/XE
		• **Immediate addressing** will be indicated by the use of \#" before the operand name/value (ie. **#1)** • **Indirect addressing** will be indicated by adding the prex \@" to the operand name (ie. @RETADR) • **Indexed addressing will be** indicated the same way as it was for the SIC machine, \,X" after the operand name (ie. BUFFER,X) • Hex digits for the address are not affected by the content of the flags, since the first two flags affect the second digit of the operation code, and the following four make up its own hex digit.

CHAPTER 2
Assemblers

Chapter 2 : Assemblers



Assembler does two functions

1) It converts the mnemonic operation codes into their machine language equivalent

2) Converts symbolic labels into their machine address

The design of assembler can be of

1. Convert mnemonic operation codes to their machine language equivalent. Ex: Translate STL to 14

2. Convert symbolic operands to their equivalent machine addresses. Ex: Translate RETADR to 1033

3. Build the machine instructions in the proper format.

4. Convert the data constants specified in the source program into their internal machine representation.
Ex: Translate 'EOF' to 454F46

5. Write the object program and the assembly listing

Different datastructure for assemblers

1 Operation code Table (OPTAB)

2 Symbol Table (SYMTAB)

3 Location Counter Variable (LOCCTR)

I OPERATION CODE TABLE (OPTAB)

a) Contents

- Mnemonic operation codes
- machine language equivalent
- Instruction format and length

b) During pass-1

- Validates op codes
- Find instruction length to increase location counter value (LOCCTR)

c) During pass-2

- Determines the instruction format (3 or 4)
- Translates the operation codes to their machine language equivalents

d) Implementation

- static hash table, easy for searching

Mnemonic Name	op code	Format
ADD m	18	3/4
.		
.		
.		
.		

II SYMBOL TABLE (SYMTAB)

a) contents

- label name
- label address
- Flags to indicate error conditions
- Data type or length

b) During pass-1

- store label name and assigned address (from LOCCTR) in SYMTAB

c) During pass-2

- symbols used as operands are looked up in SYMTAB

d) Implementation

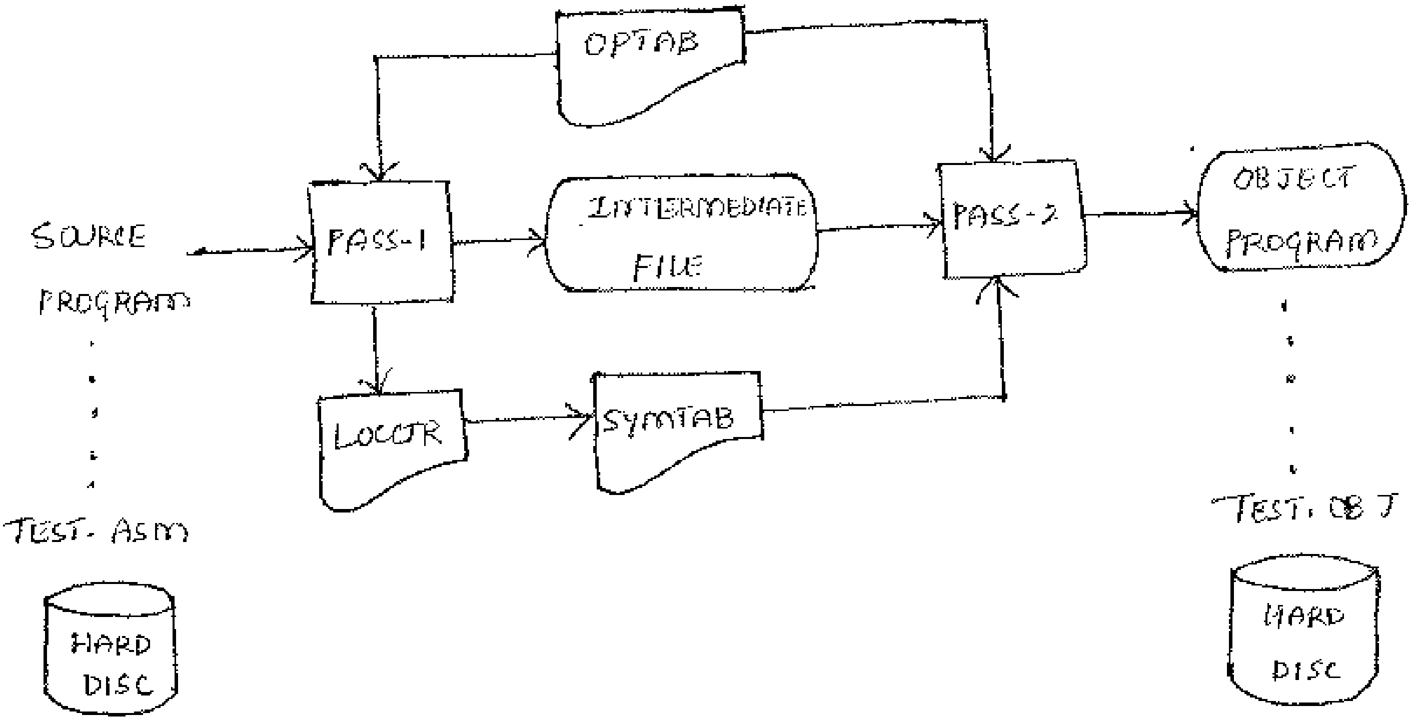
- A dynamic hash table for efficient insertion and retrieval
- should perform well with non-random keys (LOOP1, LOOP2...)

Label name	value/[illegible]	Flags	Length
CLOOP	0003		

iii LOCATION COUNTER VARIABLE (LOCCTR)

- Variable accumulated for address assignment ie LOCCTR gives the address of the associated labels
- LOCCTR is initialized to be the beginning address specified in the "START" statement
- After each source stmt is processed during pass-1, the instruction length or data area is added to LOCCTR

→ The functionality of assembler looks like this



note : During pass-1, the address of labels is not known ∵ it is defined later ie called forward reference. To resolve this we go for pass-2.

Eg JEQ RETADR

2.1 Assembler Directives

1) START	specifies name and starting address for the program
2) END	Indicates the end of the source program and optionally specify the first executable instruction in the program
3) BYTE	Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant
4) WORD	Generate one-word integer constant
5) RESB	Reserve the indicated number of bytes for a data area
6) RESW	Reserve the indicated number of words for a data area.
7) LTORG	Create a literal pool that contains all of the literal operands used since the previous LTORG or the beginning of the program
8) EQU	Establishes symbolic name that can be used for improved readability inplace of numeric values and also used to define mnemonic names for registers.

9) ORG Used to indirectly assign values to symbols

10) USE Indicates which portion of the source program belong to the various blocks and also indicates a continuation of a previously begun block

11) BASE Indicates that the base register will contain the address of operand

12) NOBASE Indicates that the contents of the base register can no longer be relied upon for addressing.

2.1.1 A simple SIC assembler

The usual (general) format to represent the assembly language program for SIC machine with generated assembly code :

LINE NO	LOCATION	SOURCE STATEMENT	OBJECT CODE

↓

LABEL	OPCODE	OPERAND	COMMENTS

where

→ LABEL : An identifier and optional. labels are used to reduce reliance upon programmers remembering where data or code is located. The length of label differs between assemblers.

Ex:- FIRST STL #4096.

→ OPCODE : is a machine code instruction. It may require additional information like operands (optional)

Ex:- COMP ZERO ; with operand

OR

RSUB ; without operands

→ OPERAND : Is an additional data or informations that the opcode requires. Operands are used to specify constants, labels, immediate data, data contained in another register, an address etc

Advantage and Disadvantage of assembly language

Advantages : → Reduced Errors

→ Faster Translation time

→ changes could be made easier and faster

Disadvantages : → many instructions are required to achieve small tasks

→ Source program tend to be large and difficult to follow

→ Programs are machine dependent, thus the complete program has to be rewritten if the hardware is changed

→ The programmer has to have the complete knowledge of the process architecture and instruction set.

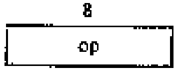
Mnemonic	Format	Opcode	Effect	Notes
ADD m	3/4	18	A ← (A) + (m..m+2)	
ADDF m	3/4	58	F ← (F) + (m..m+5)	X F
ADDR r1,r2	2	90	r2 ← (r2) + (r1)	X
AND m	3/4	40	A ← (A) & (m..m+2)	
CLEAR r1	2	B4	r1 ← 0	X
COMP m	3/4	28	(A) : (m..m+2)	C
COMPF m	3/4	88	(F) : (m..m+5)	X F C
COMPR r1,r2	2	A0	(r1) : (r2)	X C
DIV m	3/4	24	A ← (A) / (m..m+2)	
DIVF m	3/4	64	F ← (F) / (m..m+5)	X F
DIVR r1,r2	2	9C	r2 ← (r2) / (r1)	X
FIX	1	C4	A ← (F) [convert to integer]	X F
FLOAT	1	C0	F ← (A) [convert to floating]	X F
HIO	1	F4	Halt I/O channel number (A)	P X
J m	3/4	3C	PC ← m	
JEQ m	3/4	30	PC ← m if CC set to =	
JGT m	3/4	34	PC ← m if CC set to >	
JLT m	3/4	38	PC ← m if CC set to <	
JSUB m	3/4	48	L ← (PC); PC ← m	
LDA m	3/4	00	A ← (m..m+2)	
LDB m	3/4	68	B ← (m..m+2)	X
LDCH m	3/4	50	A [rightmost byte] ← (m)	
LDF m	3/4	70	F ← (m..m+5)	X F
LDL m	3/4	08	L ← (m..m+2)	
LDS m	3/4	6C	S ← (m..m+2)	X
LDT m	3/4	74	T ← (m..m+2)	X
LDX m	3/4	04	X ← (m..m+2)	
LPS m	3/4	D0	Load processor status from information beginning at address m (see Section 6.2.1)	P X
MUL m	3/4	20	A ← (A) * (m..m+2)	

Mnemonic	Format	Opcode	Effect	Notes
MULF m	3/4	60	F ← (F) * (m..m+5)	X F
MULR r1, r2	2	98	r2 ← (r2) * (r1)	X
NORM	1	C8	F ← (F) [normalized]	X F
OR m	3/4	44	A ← (A) | (m..m+2)	
RD m	3/4	D8	A [rightmost byte] ← data from device specified by (m)	P
RMO r1,r2	2	AC	r2 ← (r1)	X
RSUB	3/4	4C	PC ← (L)	
SHIFTL r1,n	2	A4	r1 ← (r1); left circular shift n bits. {In assembled instruction, r2 = n-1}	X
SHIFTR r1,n	2	A8	r1 ← (r1); right shift n bits, with vacated bit positions set equal to leftmost bit of (r1). {In assembled instruction, r2 = n-1}	X
SIO	1	F0	Start I/O channel number (A); address of channel program is given by (S)	P X
SSK m	3/4	EC	Protection key for address m ← (A) (see Section 6.2.4)	P X
STA m	3/4	0C	m..m+2 ← (A)	
STB m	3/4	78	m..m+2 ← (B)	X
STCH m	3/4	54	m ← (A) [rightmost byte]	
STF m	3/4	80	m..m+5 ← (F)	X F
STI m	3/4	D4	Interval timer value ← (m..m+2) (see Section 6.2.1)	P X
STL m	3/4	14	m..m+2 ← (L)	
STS m	3/4	7C	m..m+2 ← (S)	X
STSW m	3/4	E8	m..m+2 ← (SW)	P
STT m	3/4	84	m..m+2 ← (T)	X
STX m	3/4	10	m..m+2 ← (X)	
SUB m	3/4	1C	A ← (A) - (m..m+2)	
SUBF m	3/4	5C	F ← (F) - (m..m+5)	X F

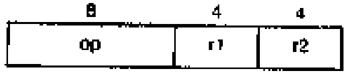
Mnemonic	Format	Opcode	Effect	Notes
SUBR r1,r2	2	94	r2 ← (r2) − (r1)	X
SVC n	2	B0	Generate SVC interrupt. [In assembled instruction, r1 = n]	X
TD m	3/4	E0	Test device specified by (m)	P C
TIO	1	F8	Test I/O channel number (A)	P X C
TIX m	3/4	2C	X ← (X) + 1; (X): (m..m+2)	C
TIXR r1	2	B8	X ← (X) + 1; (X): (r1)	X C
WD m	3/4	DC	Device specified by (m) ← (A) [rightmost byte]	P

Instruction Formats

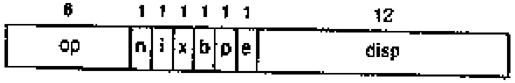
Format 1 (1 byte):



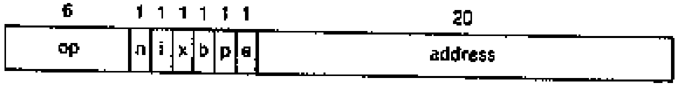
Format 2 (2 bytes):



Format 3 (3 bytes):



Format 4 (4 bytes):



Addressing Modes

The following addressing modes apply to Format 3 and 4 instructions. Combinations of addressing bits not included in this table are treated as errors by the machine. In the description of assembler language notation, *c* indicates a constant between 0 and 4095 (or a memory address known to be in this range); *m* indicates a memory address or a constant value larger than 4095. Further information can be found in Section 1.3.2.

The letters in the Notes column have the following meanings:

4	Format 4 instruction
D	Direct-addressing instruction
A	Assembler selects either program-counter relative or base-relative mode
S	Compatible with instruction format for standard SIC machine. Operand value can be between 0 and 32,767 (see Section 1.3.2 for details).

Addressing type	Flag bits n i x b p e	Assembler language notation	Calculation of target address TA	Operand	Notes
Simple	110000	op c	disp	(TA)	D
	110001	+op m	addr	(TA)	4 D
	110010	op m	(PC) + disp	(TA)	A
	110100	op m	(B) + disp	(TA)	A
	111000	op c,X	disp + (X)	(TA)	D
	111001	+op m,X	addr + (X)	(TA)	4 D
	111010	op m,X	(PC) + disp + (X)	(TA)	A
	111100	op m,X	(B) + disp + (X)	(TA)	A
	000- - -	op m	b/p/e/disp	(TA)	D S
	001- - -	op m,X	b/p/e/disp + (X)	(TA)	D S
Indirect	100000	op @c	disp	((TA))	D
	100001	+op @m	addr	((TA))	4 D
	100010	op @m	(PC) + disp	((TA))	A
	100100	op @m	(B) + disp	((TA))	A
Immediate	010000	op #c	disp	TA	D
	010001	+op #m	addr	TA	4 D
	010010	op #m	(PC) + disp	TA	A
	010100	op #m	(B) + disp	TA	A

```
Line       Source statement

  5     COPY      START    1000         COPY FILE FROM INPUT TO OUTPUT
 10     FIRST     STL      RETADR       SAVE RETURN ADDRESS
 15     CLOOP     JSUB     RDREC        READ INPUT RECORD
 20               LDA      LENGTH       TEST FOR EOF (LENGTH = 0)
 25               COMP     ZERO
 30               JEQ      ENDFIL       EXIT IF EOF FOUND
 35               JSUB     WRREC        WRITE OUTPUT RECORD
 40               J        CLOOP        LOOP
 45     ENDFIL    LDA      EOF          INSERT END OF FILE MARKER
 50               STA      BUFFER
 55               LDA      THREE        SET LENGTH = 3
 60               STA      LENGTH
 65               JSUB     WRREC        WRITE EOF
 70               LDL      RETADR       GET RETURN ADDRESS
 75               RSUB                  RETURN TO CALLER
 80     EOF       BYTE     C'EOF'
 85     THREE     WORD     3
 90     ZERO      WORD     0
 95     RETADR    RESW     1
100     LENGTH    RESW     1            LENGTH OF RECORD
105     BUFFER    RESB     4096         4096-BYTE BUFFER AREA
110     .
115     .         SUBROUTINE TO READ RECORD INTO BUFFER
120     .
125     RDREC     LDX      ZERO         CLEAR LOOP COUNTER
130               LDA      ZERO         CLEAR A TO ZERO
135     RLOOP     TD       INPUT        TEST INPUT DEVICE
140               JEQ      RLOOP        LOOP UNTIL READY
145               RD       INPUT        READ CHARACTER INTO REGISTER A
150               COMP     ZERO         TEST FOR END OF RECORD (X'00')
155               JEQ      EXIT         EXIT LOOP IF EOR
160               STCH     BUFFER,X     STORE CHARACTER IN BUFFER
165               TIX      MAXLEN       LOOP UNLESS MAX LENGTH
170               JLT      RLOOP          HAS BEEN REACHED
175     EXIT      STX      LENGTH       SAVE RECORD LENGTH
180               RSUB                  RETURN TO CALLER
185     INPUT     BYTE     X'F1'        CODE FOR INPUT DEVICE
190     MAXLEN    WORD     4096
195     .
200     .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC     LDX      ZERO         CLEAR LOOP COUNTER
215     WLOOP     TD       OUTPUT       TEST OUTPUT DEVICE
220               JEQ      WLOOP        LOOP UNTIL READY
225               LDCH     BUFFER,X     GET CHARACTER FROM BUFFER
230               WD       OUTPUT       WRITE CHARACTER
235               TIX      LENGTH       LOOP UNTIL ALL CHARACTERS
240               JLT      WLOOP          HAVE BEEN WRITTEN
245               RSUB                  RETURN TO CALLER
250     OUTPUT    BYTE     X'05'        CODE FOR OUTPUT DEVICE
255               END      FIRST
```

Figure 2.1 Example of a SIC assembler language program.

ASCII CODE
A : 65
a : 97

```
Line   Loc   Length  Source statement                          Object code
                     LABEL     OPCODE    OPERAND
  5    1000          COPY      START     1000
 10    1000   3      FIRST     STL       RETADR                   141033
 15    1003   3      CLOOP     JSUB      RDREC                    482039
 20    1006   3                LDA       LENGTH                   001036
 25    1009   3                COMP      ZERO                     281030
 30    100C   3                JEQ       ENDFIL                   301015
 35    100F   3                JSUB      WRREC                    482061
 40    1012   3                J         CLOOP                    3C1003
 45    1015   3      ENDFIL    LDA       EOF                      00102A
 50    1018   3                STA       BUFFER                   0C1039
 55    101B   3                LDA       THREE                    00102D
 60    101E   3                STA       LENGTH                   0C1036
 65    1021   3                JSUB      WRREC                    482061
 70    1024   3                LDL       RETADR                   081033
 75    1027   3                RSUB                               4C0000
 80    102A   3      EOF       BYTE      C'EOF'                   454F46
 85    102D   3      THREE     WORD      3      3Byte             000003
 90    1030   3      ZERO      WORD      0                        000000
 95    1033   3      RETADR    RESW      1 → 3BYTE
100    1036   3      LENGTH    RESW      1
105    1039   1000   BUFFER    RESB      4096   (1000 in hexadecimal)
110    1000 {        .
115    difference    .         SUBROUTINE TO READ RECORD INTO BUFFER
120                  .
125    2039   3      RDREC     LDX       ZERO                     041030
130    203C   3                LDA       ZERO                     001030
135    203F   3      RLOOP     TD        INPUT                    E0205D
140    2042   3                JEQ       RLOOP                    30203F
145    2045   3                RD        INPUT                    D8205D
150    2048   3                COMP      ZERO                     281030
155    204B   3                JEQ       EXIT                     302057
160    204E   3                STCH      BUFFER,X                 549039
165    2051   3                TIX       MAXLEN                   2C205E
170    2054   3                JLT       RLOOP                    38203F
175    2057   3      EXIT      STX       LENGTH                   101036
180    205A   3                RSUB                               4C0000
185    205D   1      INPUT     BYTE      X'F1'                    F1
190    205E   3      MAXLEN    WORD      4096                     001000
195                  .           4 needs 3 byte
200                  .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205                  .
210    2061   3      WRREC     LDX       ZERO                     041030
215    2064   3      WLOOP     TD        OUTPUT                   E02079
220    2067   3                JEQ       WLOOP                    302064
225    206A   3                LDCH      BUFFER,X                 509039
230    206D   3                WD        OUTPUT                   DC2079
235    2070   3                TIX       LENGTH                   2C1036
240    2073   3                JLT       WLOOP                    382064
245    2076   3                RSUB                               4C0000
250    2079   1      OUTPUT    BYTE      X'05'                    05
255    207A                    END       FIRST                    1Byte
```

Figure 2.2 Program from Fig. 2.1 with object code.

The following program contains a main routine that reads records from an input device (code : F1) and copies them to output device (code : 05).

main function calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write record from the buffer to output device.

Each subroutine transfers ~~one~~ record one character at a time because only I/O instructions available are RD and WD.

Since the I/O rates of two devices (disk and a printing terminal) may be differente, a buffer is used. The end of each record is marked with a null character is 00 (in hexadecimal). If a record is longer than length of buffer (4096 bytes) than only the first 4096 bytes are copied. The end of file to be copied is indicated by a zero length record.

The program indicate EOF (End of File) on output device when the zero length record (ie end of file) is detected. The program terminate by executing the RSUB instruction since it was called by JSUB instruction.

Procedure to generate object code and object Program (Intermediate File) .

Note : we have assumed that the program starts at address 1000 .

1) First and foremost write the LOCCTR address

- → START 1000
- → Add 3 byte for each instructions. (∵ instruction format for SIC m/c is 24bits . ie 3 bytes)
- → BYTE C 'EOF' : count the length of constant and add those many byte
- → RESW 2000 : then it should be 2000x3 bytes = 6000B = 1770(H) added to previous address
- → RESW 1 : add just 3 bytes
- → RESB 2000 : convert 2000 to hexadecimal ~~(7D0~~ ie 7D0 byte and add

  RESB 4096 : 4096 → $1000_{(H)}$ is added to previous value

- → WORD 3 or WORD 0 → 3 byte added .

2) Start creating the object code .

- → Convert mnemonic operation codes to their machine language equivalent Ex:- STL to 14
- → Convert symbolic operands to their equivalent machine address Ex RETADR to 1033 (forward reference)

→ Build machine instruction in proper format

a) Direct addressing : x = 0 : TA = address

b) Indexed addressing : x = 1 : TA = address + (X)

→ indicated by symbol 'X

Ex: STCH BUFFER, X :→ line NO. 160

→ Convert the data constants into their machine representation. Ex:- EOF to 454F46 (line no 80)

(A = 65 , a = 97)

↳ $(41)_{16}$ ↳ $(61)_{16}$

3) Write the object program (Intermediate File)

→ object program contains three type of records

a) Header Record b) Text Record c) End Record.

a) **Header Record** : Contains program name, starting address and length of program.

column 1	H
col. 2 - 7	Program name
col. 8 - 13	starting address of object Program (Hexadecimal
col. 14 - 19	Length of object program in bytes (Hexadecimal

Ex:- 5 COPY START 1000

→ name of program

↳ starting address

⋮

255 207A END

Length of program = last address - startingaddress

= 207A - 1000 = 107A

ie H^COPY^001000^00107A (Header Record)

b) Text Record :

Text record contains the translated instructions (machine code) and data of the program together with an indication of addresses where these are to be loaded.

col. 1	T
col. 2-7	starting address for object code in this record (Hexadecimal)
col 8-9	length of object code in this record in bytes (hexadecimal)
col 10-69	object code represented in hexadecimal (2 columns per byte of object code)

↓ note :

60 columns

⇒ 10 words ⇒ 30 bytes ⇒ $(1E)_{16}$

length of object code

Ex: 10 10000 FIRST STL RETADDR 141033

3 byte each } 10 words

⋮

55 101B LDA THREE 00102D

Text record →

T^001000^1E^141033^ . . . . ^00102D

↳ marker for separation

c) End Record :

End record marks the end of the object program and specifies the address in the program where execution is to begin. If no operand is ~~used~~ specified then the address of the first executable instruction is used.

col 1	E
col 2-7	Address of first Executable instruction in object program (hexadecimal)

```
Ex:  10   1000   FIRST   STL   RETADR   141033
            :
            :
     255         END     FIRST
```

End record → E^001000

Let us start for the given program in Fig. 2.1

Given opcodes

STL - 14
JSUB - 48
LDA - 00
COMP - 28
JEQ - 30

J - 3C
STA - 0C
STX - 10
LDL - 08
RSUB - 4C

LDX - 04
TD - E0
RD - D8
STCH - 54
TIX - 2C

JLT - 38
LDCH - 50
WD - DC
E - 45
O - 4F

F - 46

① start incrementing LOCCTR

Initially it is 1000.

→ start adding 3 byte each time from 1000 line no. 5 to 105

→ 105 1039 BUFFER RESB 4096

↳ convert to Hexadecimal ie

$(4096)_{16} = 1000$

∴ add 1000 byte to 1039 = 2039

∴ line no. 125 starts at 2039th address continue till line no. 185.

→ 185 205D INPUT BYTE X 'F1' F1 1 Byte

∴ add only 1 byte to 205D ⇒ 205E at line no 190.

→ 190 205E MAXLEN WORD 4096 001000

↳ word so 3 byte not 1000 byte

∴ 3 byte added to 205E ⇒ 2061 at line No. 210

→ 210 2061 WRREC LDX ZERO 041030.

continue the same till end.

→ 255 207A END FIRST

③ object code for each line.

⟼ Every line is direct addressing except line no 160 and 225

	line	LOC CTR	label	opcode	operand	object code
(i)	5	1000	COPY	START	1000	
	10	1000	FIRST	STL	RETADR	14 1033
	⋮					
	95	1033	RETADR	RESW	1	

mnemonic code

~~(ii)~~ ~~40~~ ~~1012~~ 7

(ii) 75 1027 RSUB . . . HC0000

no operand.

mnemonic code

(iii) 160 2045 STCH BUFFER, X ↳ indicate indexed addressing

STCH → 54

BUFFER ↓ address of buffer = 1039

8	1	15
opcode	x	address

0101 0100	1	001 0000 0011 1001
		1 0 3 9

⇓

5 4 9 0 3 9

∴ 160 2045 STCH BUFFER, X 549039

(iv) Same for line number 225

225 206A LDCH BUFFER, X

LDCH ↓ 50

BUFFER ↓ 1039

opcode	X	Address
0101 0000	1	001 0000 0011 1001
5 0	9	0 3 9

∴ 225 206A LDCH BUFFER, X 509039

③ object program for Fig 2.2

```
H^COPY. ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^ ··· ^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^ ·· ^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^ ··· ^2C1036
T^002073^07^382064^4C0000^05
E^001000.
```

②

SYMBOL TABLE

Symbol name	Value of the symbol
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	102A
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039
RLOOP	203F

symbol name	Value
EXIT	2057
INPUT	205D
MAXLEN	205E
WRREC	2061
WLOOP	2064
OUTPUT	2079

loader loads into main memory

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000																	
0010																	
⋮																	
⋮																	
1000	14	10	33	48	20	39	00	10	36	28	10	30	30	10	15	48	①
1010	20	61	3C	10	03	00	10	2A	0C	10	39	00	10	2D	0C	10	②
1020	36	48	20	61	08	10	33	4C	00	00	45	4F	46	00	00	08	
1030	00	00	00	RETADR			LENGTH										
1040																	
1050																	
⋮						BUFFER											
⋮																	
2030										04	10	30	00	10	30	E0	③
2040	20	5D	30	20	3F	D8	20	5D	28	10	30	30	20	57	54	90	
2050	39	2C	20	5E	38	20	3F	10	10	36	4C	00	00	F1	00	10	
2060	00	04	10	30	E0	20	79	30	20	64	50	90	39	DC	20	79	④
2070	2C	10	36	38	20	64	4C	00	00	05	⑤						
2080																	

Functions of Pass-I and Pass-II

Pass 1:

→ Assign addresses to all statements in the program

→ Save the values (addresses) assigned to all labels for use in Pass 2

→ Perform some processing of assembler directives (includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW etc)

Pass 2:

→ Assemble instructions (Translating operation codes and looking up addresses)

→ Generate data values defined by BYTE, WORD etc

→ Perform processing of assembler directives not done during pass-1

→ Write the object program and the assembly listing.

Pass 1:

```
begin
   read first input line
   if OPCODE = 'START' then
      begin
         save #[OPERAND] as starting address
         initialize LOCCTR to starting address
         write line to intermediate file
         read next input line
      end {if START}
   else
      initialize LOCCTR to 0
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               if there is a symbol in the LABEL field then
                  begin
                     search SYMTAB for LABEL
                     if found then
                        set error flag (duplicate symbol)
                     else
                        insert (LABEL,LOCCTR) into SYMTAB
                  end {if symbol}
               search OPTAB for OPCODE
               if found then
                  add 3 {instruction length} to LOCCTR
               else if OPCODE = 'WORD' then
                  add 3 to LOCCTR
               else if OPCODE = 'RESW' then
                  add 3 * #[OPERAND] to LOCCTR
               else if OPCODE = 'RESB' then
                  add #[OPERAND] to LOCCTR
               else if OPCODE  = 'BYTE' then
                  begin
                     find length of constant in bytes
                     add length to LOCCTR
                  end {if BYTE}
               else
                  set error flag (invalid operation code)
            end {if not a comment}
         write line to intermediate file
         read next input line
      end {while not END}
   write last line to intermediate file
   save (LOCCTR - starting address) as program length
end {Pass 1}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

Pass 2:

```
begin
   read first input line {from intermediate file}
   if OPCODE = 'START' then
       begin
           write listing line
           read next input line
       end {if START}
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
       begin
           if this is not a comment line then
               begin
                   search OPTAB for OPCODE
                   if found then
                       begin
                           if there is a symbol in OPERAND field then
                               begin
                                   search SYMTAB for OPERAND
                                   if found then
                                       store symbol value as operand address
                                   else
                                       begin
                                           store 0 as operand address
                                           set error flag (undefined symbol)
                                       end
                               end {if symbol}
                           else
                               store 0 as operand address
                           assemble the object code instruction
                       end {if opcode found}
                   else if OPCODE = 'BYTE' or 'WORD' then
                       convert constant to object code
                   if object code will not fit into the current Text record then
                       begin
                           write Text record to object program
                           initialize new Text record
                       end
                   add object code to Text record
               end {if not comment}
           write listing line
           read next input line
       end {while not END}
   write last Text record to object program
   write End record to object program
   write last listing line
end {Pass 2}
```

Figure 2.4(b) Algorithm for Pass 2 of assembler.

2.3. Machine Dependent Assembler Features

- Here we consider an example of SIC/XE machine

→ As we know already, SIC/XE has

a) Registers : A X L B S T F PC SW
(0 1 2 3 4 5 6 8 9)

b) Dataformats : Integer : 3 bytes
Character : 1 byte
Float : 6 bytes

c) Instruction Formats :

Format 1 : 1 byte

8
opcode

Ex: FLOAT, FIX

Format 2 : 2 bytes

8	4	4
opcode	r1	r2

Ex: ADDR A,X

Format 3 : 3 byte

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

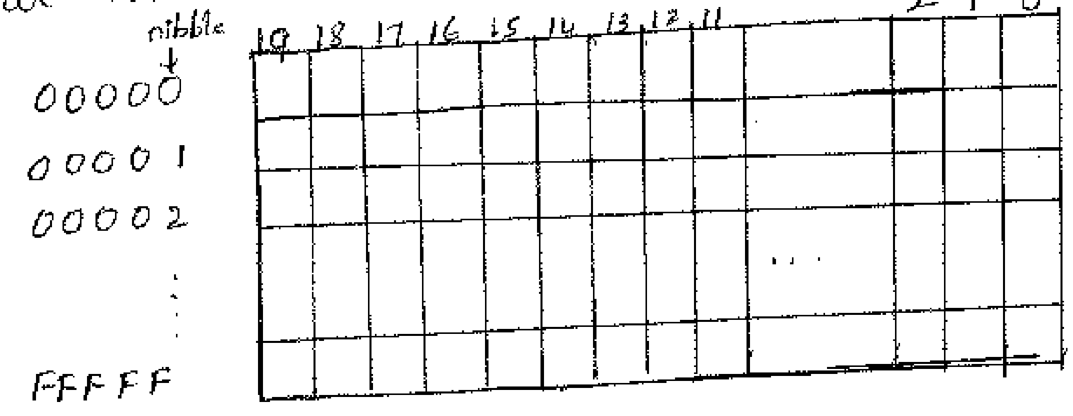
Ex:- STL RETADR

Format 4 : 4 bytes

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

Ex:- +JSUB RDREC

→ we have 20 address lines ∴ we can have 2^{20} addresses



d) Addressing mode are determined based on 6 bits

n i x b p e

(i) →

	n	i	x	Addressing mode
	1	0		Indirect addressing
	0	1		Immediate
**	1	1		not immediate, not indirect
	0	0		Simple addressing
			1	Indexed addressing
			0	Direct addressing

(ii)

	b	p	e	Addressing mode
	0	1		Program Counter Relative
	1	0		Base relative
**	1	1		Invalid (can't be set)
	0	0		NO PC relative, no base relative
			1	Format 4 instruction
			0	Format 3 instructions

Different addressing mode <u>notations</u>

1) Indirect Addressing : @

2) Immediate Addressing : #

3) Extended Format : +

4) Indexed Addressing : operand, X

5) character string : C ' '

6) Base - Register : BASE

7) Current value of PC : *

→ The addressing priority are as follows

a) PC relative addressing : $-2048 \le disp \le 2047$
($FFFFF800 \le disp \le 7FF$)

b) Base relative addressing : $0 \le disp \le 4095$
($0 \le disp \le FFF$)

c) Extended Instruction Format :

note: negative numbers are represented in 2's compliment

Procedure to create object code for SIC/XE program

1) Write the LOCCTR address for each instruction in the program.

→ if operand field is

(i) memory address → Format 3 ⇒ Add 3 bytes

(ii) Register - Register → Format 2 ⇒ Add 2 bytes

(iii) + before operand → Format 4 ⇒ Add 4 bytes

→ if it is RESW 2000

. 2000 X 3 bytes $= (6000)_d = (1770)_H$ ⇒ Add these many bytes to previous address.

. multiplication by 3 ∵ each word is 3 bytes

→ RESW 1 ⇒ Add just 3 bytes

→ RESB 2000 ⇒ Add 2000 bytes in hexadecimal

ie $(2000)_d = (7D0)_H$

→ RESB 4096

$(4096)_d = (1000)_H$ ⇒ Add 1000 bytes

→ BYTE C 'EOF' ⇒ Count the length of constant and add those many bytes

→ Enter the labels onto SYMTAB (pass 1)

2) Once we are done with LOCCTR calculation and then finding program length = EndAddress − startaddress

3) now start creating the object code (Pass 2) based on different addressing mode and set corresponding bits and calculate displacement

→ For extended format, displacement = address

→ For Reg-to-Reg instruction, write the opcode address followed by register numbers.

Ex: CLEAR X ⇒ B410 (Format 2)

(1) → Number of X register in the list

→ For PC relative, disp = TA − PC

→ For Base relative, disp = TA − (B)

```
Line        Source statement

  5     COPY      START   0              COPY FILE FROM INPUT TO OUTPUT
 10     FIRST     STL     RETADR         SAVE RETURN ADDRESS
 12               LDB    #LENGTH         ESTABLISH BASE REGISTER
 13               BASE    LENGTH
 15     CLOOP    +JSUB    RDREC          READ INPUT RECORD
 20               LDA     LENGTH         TEST FOR EOF (LENGTH = 0)
 25               COMP   #0
 30               JEQ     ENDFIL         EXIT IF EOF FOUND
 35              +JSUB    WRREC          WRITE OUTPUT RECORD
 40               J       CLOOP          LOOP
 45     ENDFIL    LDA     EOF            INSERT END OF FILE MARKER
 50               STA     BUFFER
 55               LDA    #3              SET LENGTH = 3
 60               STA     LENGTH
 65              +JSUB    WRREC          WRITE EOF
 70               J      @RETADR         RETURN TO CALLER
 80     EOF       BYTE   C'EOF'
 95     RETADR    RESW    1
100     LENGTH    RESW    1              LENGTH OF RECORD
105     BUFFER    RESB    4096           4096-BYTE BUFFER AREA
110     .
115     .         SUBROUTINE TO READ RECORD INTO BUFFER
120     .
125     RDREC     CLEAR   X              CLEAR LOOP COUNTER
130               CLEAR   A              CLEAR A TO ZERO
132               CLEAR   S              CLEAR S TO ZERO
133              +LDT    #4096
135     RLOOP     TD      INPUT          TEST INPUT DEVICE
140               JEQ     RLOOP          LOOP UNTIL READY
145               RD      INPUT          READ CHARACTER INTO REGISTER A
150               COMPR   A,S            TEST FOR END OF RECORD (X'00')
155               JEQ     EXIT           EXIT LOOP IF EOR
160               STCH    BUFFER,X       STORE CHARACTER IN BUFFER
165               TIXR    T              LOOP UNLESS MAX LENGTH
170               JLT     RLOOP            HAS BEEN REACHED
175     EXIT      STX     LENGTH         SAVE RECORD LENGTH
180               RSUB                   RETURN TO CALLER
185     INPUT     BYTE    X'F1'          CODE FOR INPUT DEVICE
195     .
200     .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC     CLEAR   X              CLEAR LOOP COUNTER
212               LDT     LENGTH
215     WLOOP     TD      OUTPUT         TEST OUTPUT DEVICE
220               JEQ     WLOOP          LOOP UNTIL READY
225               LDCH    BUFFER,X       GET CHARACTER FROM BUFFER
230               WD      OUTPUT         WRITE CHARACTER
235               TIXR    T              LOOP UNTIL ALL CHARACTERS
240               JLT     WLOOP            HAVE BEEN WRITTEN
245               RSUB                   RETURN TO CALLER
250     OUTPUT    BYTE    X'05'          CODE FOR OUTPUT DEVICE
255               END     FIRST
```

Figure 2.5 Example of a SIC/XE program.

```
Line    Loc          Source statement                  Object code

  5     0000    COPY     START    0
 10     0000    FIRST    STL      RETADR               17202D
 12     0003             LDB      #LENGTH              69202D
 13                      BASE     LENGTH
 15     0006    CLOOP   +JSUB     RDREC                4B101036
 20     000A             LDA      LENGTH               032026
 25     000D             COMP     #0                   290000
 30     0010             JEQ      ENDFIL               332007
 35     0013            +JSUB     WRREC                4B10105D
 40     0017             J        CLOOP                3F2FEC
 45     001A    ENDFIL   LDA      EOF                  032010
 50     001D             STA      BUFFER               0F2016
 55     0020             LDA      #3                   010003
 60     0023             STA      LENGTH               0F200D
 65     0026            +JSUB     WRREC                4B10105D
 70     002A             J        @RETADR              3E2003
 80     002D    EOF      BYTE     C'EOF'               454F46
 95     0030    RETADR   RESW     1
100     0033    LENGTH   RESW     1
105     0036    BUFFER   RESB     4096
110             .
115             .        SUBROUTINE TO READ RECORD INTO BUFFER
120             .
125     1036    RDREC    CLEAR    X                    B410
130     1038             CLEAR    A                    B400
132     103A             CLEAR    S                    B440
133     103C            +LDT      #4096                75101000
135     1040    RLOOP    TD       INPUT                E32019
140     1043             JEQ      RLOOP                332FFA
145     1046             RD       INPUT                DB2013
150     1049             COMPR    A,S                  A004
155     104B             JEQ      EXIT                 332008
160     104E             STCH     BUFFER,X             57C003
165     1051             TIXR     T                    B850
170     1053             JLT      RLOOP                3B2FEA
175     1056    EXIT     STX      LENGTH               134000
180     1059             RSUB                          4F0000
185     105C    INPUT    BYTE     X'F1'                F1
195             .
200             .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205             .
210     105D    WRREC    CLEAR    X                    B410
212     105F             LDT      LENGTH               774000
215     1062    WLOOP    TD       OUTPUT               E32011
220     1065             JEQ      WLOOP                332FFA
225     1068             LDCH     BUFFER,X             53C003
230     106B             WD       OUTPUT               DF2008
235     106E             TIXR     T                    B850
240     1070             JLT      WLOOP                3B2FEF
245     1073             RSUB                          4F0000
250     1076    OUTPUT   BYTE     X'05'                05
255     1077             END      FIRST
```

Figure 2.6 Program from Fig. 2.5 with object code.

Consider the example of figure 2.5

1) Add the length of each instruction and add it to LOCCTR and find the program length

Program length = ~~Start~~ End address - start address

= 1077 - 0000 = 1077

pass 1

2) Create the symbol table

Symbol name	PC value
FIRST	0000
CLOOP	0006
ENDFIL	001A
EOF	002D
RETADR	0030
LENGTH	0033
BUFFER	0036
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WRREC	105D
WLOOP	1062
OUTPUT	1076

3) start creating object code (pass - 2)

10 0000 FIRST $\underbrace{\text{STL}}_{14}$ $\underbrace{\text{RETADR}}_{0030}$ (Format 3 ∵ oprd is memory address)

By default assembler uses PC relative addressing

opcode		n	i	x	b	p	e	disp
1	01	1	1	0	0	1	0	02D

→ TA = PC + disp

↳ Displacement = TA − PC

= RETADR − LOCCTR (location of next instn to be executed)

= 0030 − 0003 = $\cancel{0}$02D

∵ format 3 displacement is 12 bits

↳ 02D is within range of $-2048 \le disp \le 2047$

↳ opcode is 6 bits (4 + 2 ⇒ 1 nibble + 2 - bits)

⇒ last 2 bits can be represented by 4 bits

but always last 2 bits are "zero"

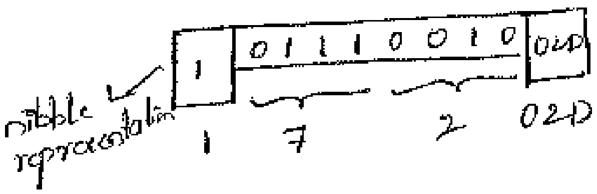
Ex:- 4 ⇒ $\underbrace{01}_{\text{only 2 bits}}\cancel{0}\cancel{0}$

C ⇒ $\underbrace{11}_{C}\cancel{0}\cancel{0}$

↳ not immediate, not indirect so set n=1, i=1

↳ not indexed x=0, not base relative b=0 but it is pc relative p=1, not format 4 e=0

↳ write the whole instruction's object code ie



STL RETADR 17202D

13 0003 $\underbrace{\text{LDB}}_{68}$ #$\underbrace{\text{LENGTH}}_{0033}$

→ opcode for LDB = 68

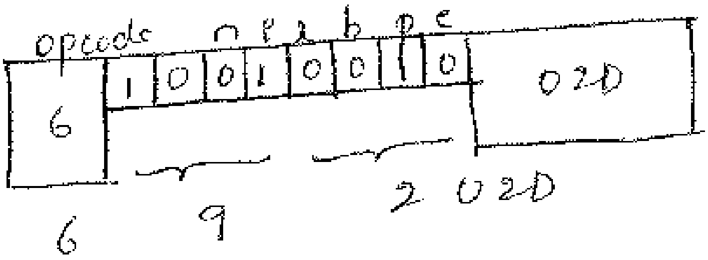
→ ~~it~~ is imm calculate disp.

TA = PC + disp

disp = TA − PC = 0033 − 0006 = ∅02D

→ PC relative ∵ operand is memory address

→ it is immediate so i = 1



LDB #LENGTH ⟹ 69202D

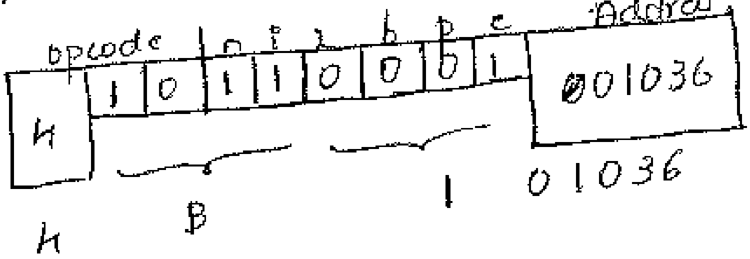
15 0006 $\underbrace{\text{CLOOP}}$ $\underbrace{\text{+JSUB}}_{48}$ $\underbrace{\text{RDREC}}_{0036}$ → Format 4

→ disp = (operand) ∵ it is extended format (F4)

=∅01036 (20 bits)

→ not immediate, not indirect n=1, i=1.

→ extended e=1



CLOOP +JSUB RDREC ⟹ 4B101036

20 000A $\underbrace{\text{LDA}}_{00}$ $\underbrace{\text{LENGTH}}_{0033}$ → format 3

→ PC relative, disp = TA − PC

P=1 = 0033 − 000D = 026

→ not immediate, not indirect, not indexed so

n=1, i=1, x=0

			n	i	x	b	p	e	
0	0	0	1	1	0	0	1	0	026

0 3 2 026

LDA LENGTH ⇒ 032026

25 000D $\underbrace{\text{COMP}}_{28}$ #0

→ immediate not PC relative because operand is direct value but not memory address.

∴ displacement = operand = 000

→ immediate addressing n=0, i=1, b=0, p=0

			n	i	x	b	p	e	
2	1	0	0	1	0	0	0	0	000

2 9 0 000

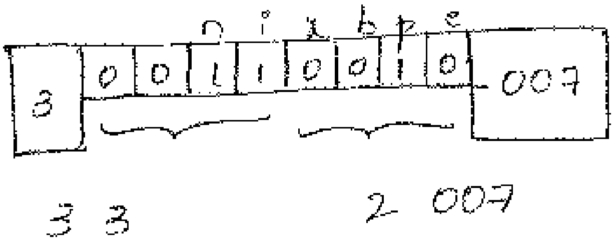
COMP #0 ⇒ 290000

30 0010 $\underbrace{\text{JEQ}}_{30}$ $\underbrace{\text{ENDFIL}}_{001A}$ ⇒ Format 3

- PC relative ∴ disp = TA − PC

$= 001A - 0013 = \underbrace{007}$ within range

- not immediate, not indirect n=1, i=1



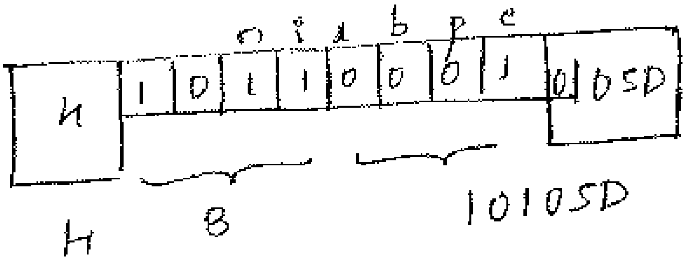
∴ JEQ ENDFIL ⇒ 332007

$\frac{35}{65}$ 0013 +$\underbrace{\text{JSUB}}_{48}$ $\underbrace{\text{WRREC}}_{105D}$ ⇒ Format 4

- displacement = address of operand

$= \underbrace{0105D}$

~~within~~



+JSUB WRREC ⇒ 4B10105D

40 0017 $\underbrace{\text{J}}_{3C}$ $\underbrace{\text{CLOOP}}_{0006}$ ⇒ Format 3

disp = TA − PC = 0006 − 001A

= −14 (it take 2's complement)

= FEC

- it is PC relative p=1

- not immediate, not indirect n=ɸ, i=1

			n	i	x	b	p	e	
3	1	1	ɸ	1	0	0	1	0	FEC

object code : 3 F 2FEC

45 001A ENDFIL $\underbrace{\text{LDA}}_{00}$ $\underbrace{\text{EOF}}_{002D}$ ⇒ Format 3 (PC relative)

disp = TA - PC = 002D - 001D = ɸ010

			n	i	x	b	p	e	
0	0	0	1	1	0	0	1	0	010

0 3 2 010

LDA EOF ⇒ 032010

50 001D $\underbrace{\text{STA}}_{0C}$ $\underbrace{\text{BUFFER}}_{0036}$ ⇒ Format 3 (PC relative)

disp = TA - PC = 0036 - 0020 = ɸ016

			n	i	x	b	p	e	
0	1	1	1	1	0	0	1	0	016

0 F 2016

∴ STA BUFFER ⇒ 0F2016

55 0020 LDA #3 ⇒ immediate address

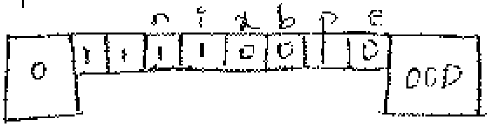
disp = 003

			n	i	x	b	p	e	
0	0	0	0	1	0	0	0	0	003

LDA #3 ⇒ 010003

60 0023 $\underbrace{\text{STA}}_{0C}$ $\underbrace{\text{LENGTH}}_{0033}$ ⇒ Format 3 (pc relative)

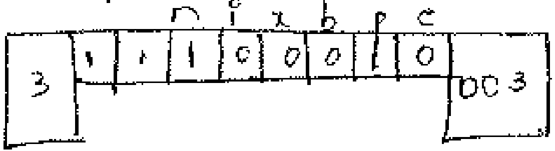
disp = TA - PC = 0033 - 0026 = ~~0~~00D



0 $\underbrace{\quad}_{F}$ $\underbrace{\quad}_{2}$ 00D ⇒ 0F200D

70 002A $\underbrace{\text{J}}_{3C}$ $\underbrace{\text{@RETADR}}_{0030}$ ⇒ Format - 3 - Indirect

disp = TA - PC = 0030 - 002D = ~~0~~003



3 $\underbrace{\quad}_{E}$ $\underbrace{\quad}_{2}$ 003 ⇒ 3E2003

80 002D EOF BYTE C 'EOF'

⇒ Convert EOF to hexadecimal ascii value

E — 45

O - 4F

F — 46

125 1036 RDREC $\underbrace{\text{CLEAR}}_{B4}$ X

A	X	L	B	S	T	F	PC	SW
0	1	2	3	4	5	6	8	9

⇒ B410 ↳ this is not accumulator

⇒ only 2 bytes since it is register-to-register mode

130 1038 $\underbrace{\text{CLEAR}}_{B4}$ A ⇒ B400

132 103A $\underbrace{\text{CLEAR}}_{B4}$ S ⇒ B440

130 1047 $\underbrace{\text{COMPR}}_{\text{AO}}$ A, S ⇒ A00H

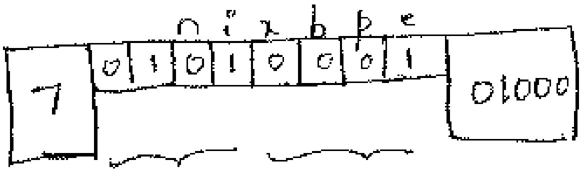
140 1057 $\underbrace{\text{TIXR}}_{\text{B8}}$ T ⇒ B850

035 1066 $\underbrace{\text{TIXR}}_{\text{B8}}$ T ⇒ B850

210 105D CORPLC $\underbrace{\text{CLEAR}}_{\text{B4}}$ X ⇒ B410

133 103C +LDT #4096 ⇒ Format 4 4 Immediate addressing

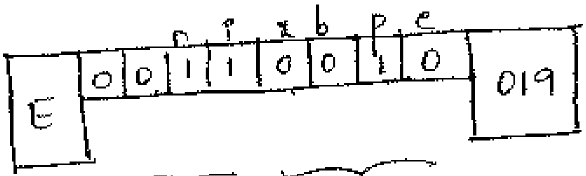
disp = $(4096)_{16}$ =01000



7 5 101000 ⇒ 75101000

135 1040 RLOOP $\underbrace{\text{TD}}_{\text{E0}}$ $\underbrace{\text{INPUT}}_{\text{105C}}$ ⇒ Formal 3 + PC relative

disp = TA − PC = 105C − 1043 = ∅019



E 3 2019 ⇒ E32019

140 1043 $\underbrace{\text{JEQ}}_{30}$ $\underbrace{\text{RLOOP}}_{1040}$ ⇒ Format 3 + PC relative

disp = TA − PC = 1040 − 1046 = −6 ⇓ 2's complement ⇓ FFA

			n	i	x	b	~~p~~	p	e	
3	0	0	1	1	0	0	~~0~~	1	0	FFA

3 $\underbrace{\quad}_{3}$ $\underbrace{\quad}_{2}$ FFA ⇒ 332FFA

155 104B $\underbrace{\text{JEQ}}_{30}$ $\underbrace{\text{EXIT}}_{1056}$ ⇒ Format 3 + PC relative

disp = TA − PC = 1056 − 104E = 008

			n	i	x	b	p	e	
3	0	0	1	1	0	0	1	0	008

3 $\underbrace{\quad}_{3}$ $\underbrace{\quad}_{2}$ 008 ⇒ 332008

145 1046 $\underbrace{\text{RD}}_{D8}$ $\underbrace{\text{INPUT}}_{105C}$ ⇒ Format 3 + PC relative

disp = TA − PC = 105C − 1049 = 013

			n	i	x	b	p	e	
D	1	0	1	1	0	0	1	0	013

D $\underbrace{\quad}_{B}$ $\underbrace{\quad}_{2}$ 013 ⇒ DB2013

160 104E $\underbrace{\text{STCH}}_{54}$ $\underbrace{\text{BUFFER,X}}_{0036}$ ⇒ Indexed + pc relative

**

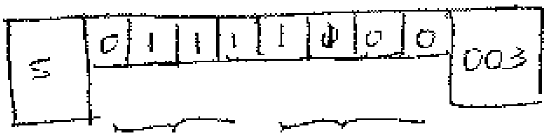
$$dsp = TA - PC = 0036 - 1051 = -101B$$

$$= \underbrace{EFE5}_{(4123)_{10}} > 2047$$

∴ it is not pc relative, go for base relative

$$dsp = \overset{TA}{BUFFER} - \overset{B}{B}$$ (length is stored in base register at 0033)

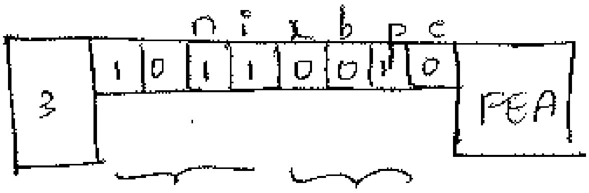
$$= 0036 - 0033 = \cancel{0}003$$



5 7 C 003 ⇒ 57C003

170 1053 $\underbrace{\text{JLT}}_{38}$ $\underbrace{\text{RLOOP}}_{1040}$ ⇒ Format 3 + pc relative

$$dsp = TA - PC = 1040 - 1056 = FEA$$



3 B 2 FEA ⇒ 3B2FEA

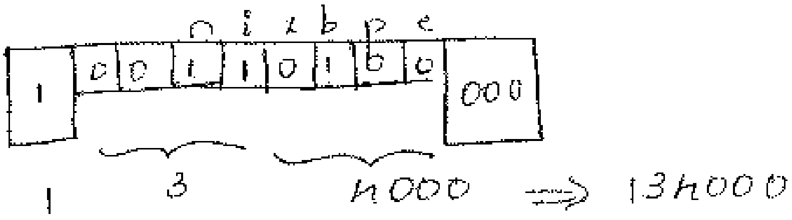
175 1056 EXIT $\underbrace{\text{STX}}_{10}$ $\underbrace{\text{LENGTH}}_{0033}$ ⇒ Format 3 + PC relative

*9

disp = TA − PC = 0033 − 1059 = EFDA > 2047

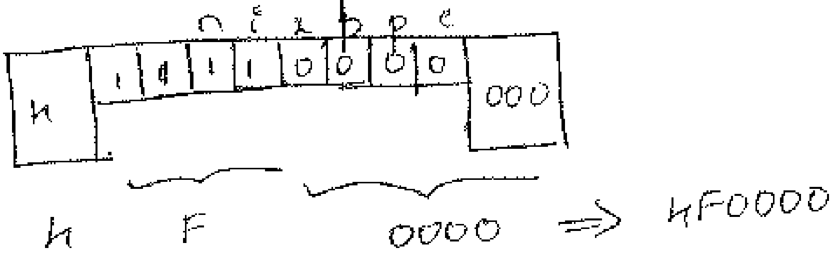
∴ go for base relative mode

disp = TA − (B) = 0033 − 0033 = ∅000



180 1059 $\underbrace{\text{RSUB}}_{4C}$ ⇒ Format 3

no displacement ∵ no operand.



185 105C INPUT BYTE X'F1'

⇒ character string ∴ store as it is

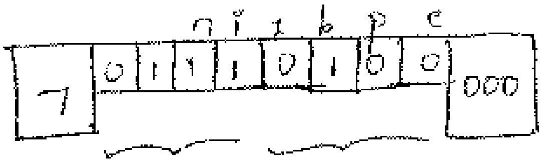
212 105F $\underbrace{\text{LDT}}_{74}$ $\underbrace{\text{LENGTH}}_{0033}$ ⇒ Format 3

**

disp = TA − PC = 0033 − 1062 = $\underbrace{\text{EFD1}}_{4443}$ > 2047

∴ go for base relative

disp = TA − (B) = 0033 − 0033 = 0000

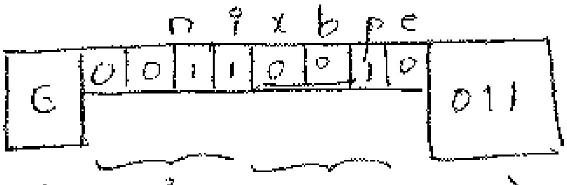


7 7 4000 ⇒ 774000

915 1062 WLOOP TD OUTPUT ⇒ Format 3 + PC relativ

TD → E0, OUTPUT → 1076

disp = TA − PC = 1076 − 1065 = 011

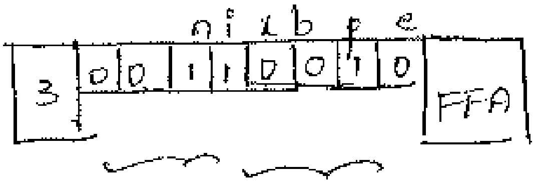


E 3 2011 ⇒ E32011

920 1065 JEQ WLOOP ⇒ Format 3 + PC relativ

JEQ → 30, WLOOP → 1062

disp = TA − PC = 1062 − 1068 = FFA



3 3 2FFA ⇒ 332FFA

925 1068 LDCH BUFFER, X ⇒ indexed

LDCH → 50, BUFFER → 0036

disp = TA − PC = 0036 − 106B = −1035 > 2047

∴ go for base relative mode

disp = TA − B = 0036 − 0033 = 0003

			n	i	x	b	p	e	
5	0	0	1	1	1	1	0	0	003

5 3 C 003 ⇒ 53C003

230 1069 WD OUTPUT ⇒ F3 + PC relative

$\underbrace{\text{WD}}_{DC}$ $\underbrace{\text{OUTPUT}}_{1076}$

disp = TA - PC = 1076 - 106E = ~~0~~008

			n	i	x	b	p	e	
D	1	1	1	1	0	0	1	0	008

D F 2 008 ⇒ DF2008

240 1070 JLT LOOP ⇒ Format 3 + PC relative

$\underbrace{\text{JLT}}_{38}$ $\underbrace{\text{LOOP}}_{1062}$

disp = TA - PC = 1062 - 1073 = FEF

			n	i	x	b	p	e	
3	1	0	1	1	0	0	1	0	FEF

3 B 2 FEF ⇒ 3B2FEF

245 1073 RSUB ⇒ Format 3

$\underbrace{\text{RSUB}}_{4C}$

			n	i	x	b	p	e	
4	1	1	1	1	0	0	0	0	000

4 F 0000 ⇒ 4F0000

250 1076 OUTPUT BYTE X '05'

⇒ Character string ∴ store as it is ⇒ 05

4) object program

H^COPY ^000000^001077

T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^
3F2FEC^032010

T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46

T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^
332FFA^DB2013^A004^332008^57C003^B850

T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^
53C003^DF2008^B850

T^001070^07^3B2FEF^4F0000^05

E^000000

Loading into memory

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000	17	20	2D	69	20	2D	4B	10	10	36	03	20	26	29	00	00	①
0010	33	20	07	4B	10	10	5D	3F	2F	EC	03	20	10	0F	20	16	
0020	01	00	03	0F	20	0D	4B	10	10	5D	3E	20	03	45	4F	46	②
0030	RETADR			LENGTH													
⋮							BUFFER										
⋮																	
1030							B4	10	B4	00	B4	40	75	10	10	00	
1040	E3	20	19	33	2F	FA	DB	20	13	A0	04	33	20	08	57	C0	③
1050	03	B8	50	3B	2F	EA	13	40	00	4F	00	00	F1	B4	10	77	
1060	40	00	E3	20	11	33	2F	FA	53	C0	03	DF	20	08	B8	50	④
1070	3B	2F	EF	4F	00	00	05	⑤									

1. Generate the complete object program for the following assembly level program
CLEAR - B4, LDA - 00, LDB - 68, ADD - 18, TIX - 2C,
JLT - 38 STA - 0C

PASS - I	LENGTH	LABEL	OPCODE	OPERAND	PASS - II
0000		SUM	START	0	
0000	2		CLEAR	X	B410
0002	3		LDA	#0	010000
0005	4		+LDB	#TOTAL	69101789
			BASE	TOTAL	
0009	3	LOOP	ADD	TABLE,X	1BA00D
000C	3		TIX	COUNT	2F2007
000F	3		JLT	LOOP	3F2FF7
0012	4		+STA	TOTAL	0F101789
0016	3	COUNT	RESW	1	
0019	1770	TABLE	RESW	2000 (1770)H	
1789	3	TOTAL	RESW	1	
178C			END	FIRST	

RESW 2000 ⇒ 2000x3 = $(6000)_d$ bytes = $(1770)_H$

∴ 0019 + 1770 = 1789

Program Length = 178C − 0000 = 178C

1) 0000 CLEAR X (Register-to-Register)

⇒ directly opcode with register numbers

⇒ B410

2) 0002 LDA #0 ⇒ Immediate Addressing

disp = 000

			n	i	x	b	p	e	
0	0	0	0	1	0	0	0	0	000

0 01 0 000 ⇒ 010000

3) 0005 +LDB #TOTAL ⇒ Extended & immediate - Format 4 + immediate with PC relative

disp = operandaddress
= 1789

			n	i	x	b	p	e	
6	1	0	0	1	0	0	0	1	1789

6 9 1 01789 ⇒ 69101789

4) 0009 LOOP ADD TABLE,X ⇒ indexed with PC relative

TA = PC+disp

disp = TA - PC

= 0019 - 000C = 00D

			n	i	x	b	p	e	
1	1	0	1	1	1	0	1	0	00D

1 B A 00D ⇒ 1BA00D

5) 000C TIX COUNT ⇒ Format-3

disp = 0016 - 000F = 007

			n	i	x	b	p	e	
2	1	1	1	1	0	0	1	0	007

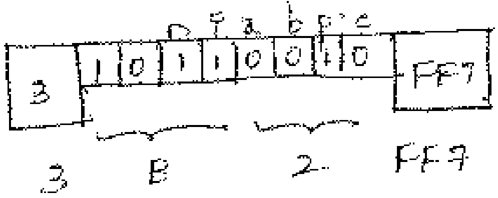
2 F 2 007 ⇒ 2F2007

6) 000F JLT LOOP ⟹ Format-3 PC relative

disp = TA - PC

= 0009 - 0012 = FF7

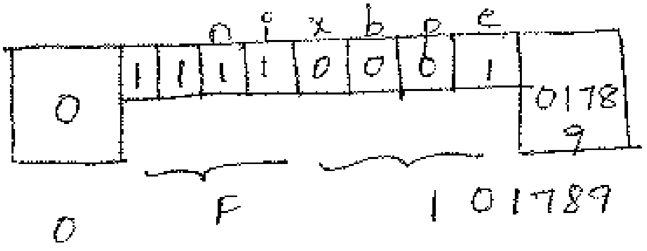
$\xrightarrow[\text{range}]{\text{within}}$ -2048 ≤ FF7 ≤ 2047



⟹ 3F2FF7

7) 0012 +STA TOTAL ⟹ Format 4

disp = operand value = 1789



⟹ 0F101789

<u>object program</u> (Pass-2)

H^ sum ^0000. ^00178C

T^ 000000^16^ BH10^010000^69101789^1BA00D^2F2007^3F2FF7^0F101789

E^ 000000

SYMTAB → (pass-1)

SYMBOL NAME	VALUE
LOOP	0009
COUNT	0016
TABLE	0019
TOTAL	1789

8. Generate the complete object program for the following assembly level program. Also indicate the contents of symbol table at the end. Assume standard SIC model and assume the following mle codes in HEX

LDA = 00 STA = 0C TIX = 2C JLT = 38

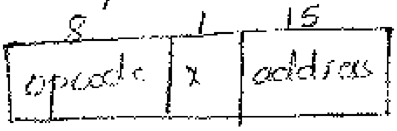
LDX = 04 ADD = 18 RSUB = 4C

LOCCTR (PASS: 1)	LENGTH	LABEL	OPCODE	OPERAND	OBJECT CODE
		SUM	START	4000	
4000	3	FIRST	LDX	ZERO	045788
4003	3		LDA	ZERO	005788
4006	3	LOOP	ADD	TABLE,X	18C015
4009	3		TIX	COUNT	2C5785
400C	3		JLT	LOOP	384006
400F	3		STA	TOTAL	0C578B
4012	3		RSUB		4C0000
4015	1770	TABLE	RESW	2000 (1770)H	
5785	3	COUNT	RESW	1	
5788	3	ZERO	WORD	0.	000000
578B	3	TOTAL	RESW	1	
578E			END	FIRST	

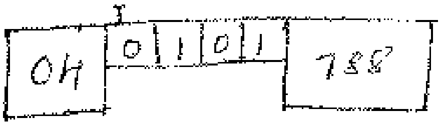
Program length = END address - starting address
= 578E - 4000 = 178E

→ Since it is SIC program, we have two addressing mode < direct addressing (x=0) / indexed addressing (x=1)

- SIC directly put opcode with operand address



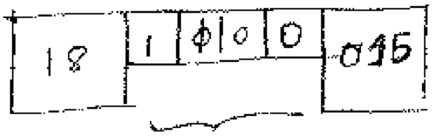
1) 4000 FIRST LDX ZERO



⇒ 045788

2) 4006 ADD TABLE, X ⇒ indexed addressing

$\underbrace{\text{ADD}}_{18}$ $\underbrace{\text{TABLE}}_{4015}$



18 C 015 ⇒ 18C015

SYMTAB

NAME	VALUE
FIRST	4000
LOOP	4006
TABLE	4015
COUNT	5785
ZERO	5788
TOTAL	578B

object program

H^ SUM ^00 4000 ^00178E

T^ 004000^ 15^ 045788^ 005788^ 18C015^ 2C5785^ 384006^ 0C578B^ 4C0000

T^ 005788^ 03 ^000000

E^ 004000

LOADING INTO MAIN MEMORY

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000																
0010																
⋮						.	. .	. .								
4000	04	57	88	00	57	88	18	C0	15	2C	57	85	38	40	06	0C
4010	57	8B	4C	00	0C											
4020																
⋮						TABLE .	.	.	.	.						
5780						COUNT			00	00	00	TOTAL				
5790																

3. Generate the object code for each statement in the following SIC/XE program and generate the object program for the same.

LOCCTR	LENGTH	LABEL	OPCODE	OPERAND	OBJECT-CODE
		SUM	START	0	
0000	3	FIRST	LDX	#0	050000
0003	3		LDA	#0	010000
0006	4		+LDB	#TABLE2	69101790
			BASE	TABLE2	
000A	3	LOOP	ADD	TABLE, X	1BA013
000D	3		ADD	TABLE2, X	1BC000
0010	3		TIX	COUNT	2F200A
0013	3		JLT	LOOP	3B2FF4
0016	4		+STA	TOTAL	0F102F00
001A	3		RSUB		4F0000
001D	3	COUNT	RESW	1	
0020	1770	TABLE	RESW	2000 (1770)H	
1790	1770	TABLE2	RESW	2000 (1770)H	
2F00	3	TOTAL	RESW	1	
2F03		END	FIRST		

LDX = 04	LDB = 68	TIX = 2C	STA = 0C
LDA = 00	ADD = 18	JLT = 38	RSUB = 4C

→ Keep assigning the length for each instruction based on

(i) 1st operand is memory address — 3 byte

(ii) 1st operand is register — 2 byte

(iii) + (Extended format — 4 byte

→ Find the LOCCTR value ; Program length = 2F03 − 0000
= 2F03

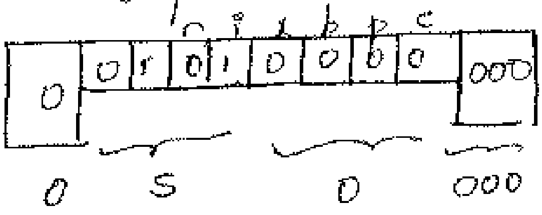
→ Create SYMTAB

Name	Value
FIRST	0000
LOOP	000A
COUNT	001D
TABLE	0020
TABLE2	1790
TOTAL	2F00

→ object code for each instruction

1) 0000 FIRST $\underset{04}{\underline{\text{LDX}}}$ #0 → immediate addressing

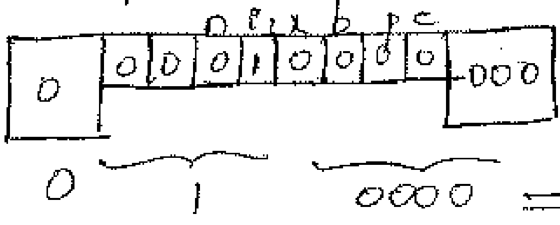
disp = 000



⇒ 050000

2) 0003 $\underset{00}{\underline{\text{LDA}}}$ #0 → immediate addressing

disp = 000



⇒ 010000

3) 0006 +LDB #TABLE2 ⟶ Extended + immediate
$\underset{68}{}$

~~TA = PC + disp~~

~~disp = TA = pc = 1790 = 000A~~

disp = $\overset{\text{Target}}{\text{address}}$ = $\underbrace{01790}_{\text{20 bits}}$

			n	i	x	b	p	e	
6	1	0	0	1	0	0	0	1	01790

6 9 101790 ⟹ 69101790

4) 000A LOOP ADD TABLE, X ⟶ Indexed + PC relative
18

TA = PC + disp

disp = TA − PC = 0020 − 000D = 0013

			n	i	x	b	p	e	
1	1	0	1	1	1	0	1	0	013

1 B A 013 ⟹ 1BA013

5) 000D ADD TABLE2, X ⟶ Indexed + base relative
18

(∵ TABLE2 is stored in base register)

Initially we can try for PC - relative & checkout whether displacement is within the range.

disp = TA − PC

= 1790 − 0010 = $(1780)_H$ ⇒ $(6016)_d > (2047)_d$

∴ go for base relative

disp = TA − B (look for address of TABLE2 in SYMTAB)

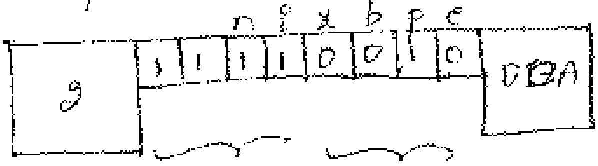
= 1790 − 1770 = 0000

			n	i	x	b	p	c	
1	1	0	1	1	1	1	0	0	000

1 B C 000 ⟹ 1BC000

6) 0010 $\underset{2C}{\text{TIX}}$ COUNT → PC relative

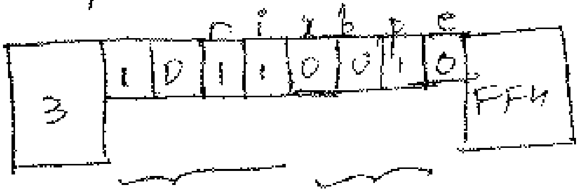
disp : TA − PC = 001D − 0013 = ~~0~~00A



2 F 2 00A ⇒ 2F200A

7) 0013 $\underset{38}{\text{JLT}}$ LOOP ⇒ PC relative

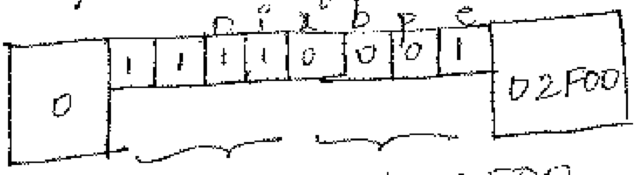
disp = TA − PC = 000A − 0016 = FFh



3 B 2 FFh ⇒ 3B2FFh

8) 0016 +$\underset{0C}{\text{STA}}$ TOTAL ⇒ Extended (Format 4)

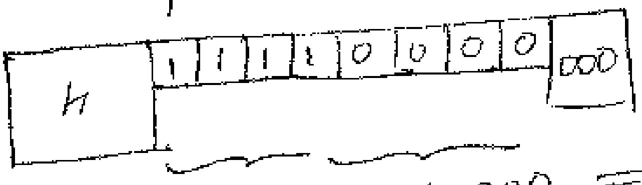
disp : address of TOTAL = 2F00



0 F 102F00

9) RSUB 001A RSUB

⇒ no operand ∴ no displacement



4 F 0000 ⇒ 4F0000

→ Object program

H^SUM ^000000^002F03

T^000000^1D^050000^010000^69101790^1BA013^1BC000^2F200A^3B2FF4^0F102F00^4F0000

E^000000

→ loader loads into memory

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	05	00	00	01	00	00	69	10	17	90	1B	A0	13	1B	C0	00
0010	2F	20	0A	3B	2F	F4	0F	10	2F	00	4F	00	00	COUNT		
0020						TABLE										
:																
1790																
						TABLE2										
2F00	TOTAL															

4 Generate the machine code for the following SIC/XE program

Given JSUB = A0, LDA = 80, LDX = 60, STA = 50, COMP = 90,

RSUB = 4C JEQ = BD, J = B8

LOCCTR	LENGTH	LABEL	OPCODE	OPERAND	OBJECT CODE
		COPY	START	1000	
1000	4	CLOOP	+JSUB	RDREC	
	3		LDA	LENGTH	
	3		COMP	ZERO	
	3		JEQ	EXIT	
	3		J	CLOOP	
	3	EXIT	STA	BUFFER	
	3		LDA	THREE	
	3		STA	TOTAL LENGTH	
	3		RSUB		
		BUFFER	RESW	100	
	3	EOF	BYTE	C'EOF'	
	3	ZERO	WORD	0	
	9	THREE	WORD	3	
	3	LENGTH	RESW	1	
	3	TOTAL_LENGTH	RESW	1	
	3	RDREC	LDX	ZERO	

2.2.2. Program Relocation

Absolute Assembly program is one which executes properly, only if program is loaded from specified location.

Ex: All SIC programs are absolute assembly program

Consider the SIC program

```
5    1000   COPY    START   1000
10   1000   FIRST   STL     RETADDR   141033
15   1003   CLOOP   JSUB    RDREC     482039
      :
55   101B           LDA     THREE     00102D
      :
85   102D   THREE   WORD    3         000003
```

→ Here program is loaded at address 1000.

→ Line NO. 55 specifies that the register A is to be loaded from memory address 102D (object codes).

→ suppose we attempt to load and execute the program at address 2000 instead of address 1000, the address 102D will not contain the value that we expected, as it might be part of some other users program.

→ obviously we need to make some change in the address portion of this instruction so we can load and execute the program at address 2000.

→ At the same time, there are statements like line no. 05. which generate a constant 3, that should remain the same regardless of where the program is loaded.

→ From the object code, we can't it is not possible to tell which values represent addresses and which represent constant data items.

→ This is all because the assembler doesnot know the actual location where the program will be loaded till load time. ∴ it cannot make the necessary changes required.

→ only parts of the program that require modification at load time are those that specify direct addresses.

This is achieved through $\underbrace{\text{relocatable program}}_{\text{program relocation}}$ (for SIC/XE machine)

2.2.2. Program Relocation

Program relocation is a process of modifying the addresses used in address sensitive instructions of a program such that program can execute correctly from allocated memory area. It is often needed to have more than one program at same time, sharing the memory and other resources of the machine. Because of this, it is necessary to load a program into memory wherever it is available. Hence relocation of the addresses in the program is required and this will be done during loading time. Assembler only indicates those instructions which need modification and this information is passed to loader.

The Assembler solves the relocation problem as follows.

→ keeping track of operand address relative to start of a program

→ Generating commands for loader which add the beginning address to operand relative address

~~The~~ An object program that contains the information necessary to perform this kind of modification is called a "relocatable program". we can accomplish this with a modification record as follows modification Record

col. 1 m

col 2-7 starting location of the address field to be modified, relative to the beginning of the program

col 8-9 Length of the address field to be modified, in half-bytes (hexadecimal).

→ The length is stored in half-bytes (rather than bytes) because the address field to be modified may not occupy an integral number of bytes.

Ex: 20 bits = 5 half-bytes

→ The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If this field occupies an odd number of half-bytes, it is assumed to begin in the middle of the first byte at the starting location.

Ex: SIC/XE program

```
5     0000   COPY     START
10    0000   FIRST    STL      RETADR    17202D
12    0003            LDB      #LENGTH   69202D
13                    BASE     LENGTH
15    0006   CLOOP    +JSUB    RDREC     4B101036
                 .       .
                 .       .
35    0013            +JSUB    WRREC     4B10105D
40    0017            J        CLOOP     3F2FEC
                 .       .
                 .       .
65    0026            +JSUB    WRREC     4B10105D
 .               .
 .               .
100   0036   BUFFER   RESB     4096
                 .
                 .
125   1036   RDREC    CLEAR    X         B410
```

program is loaded at address 0000

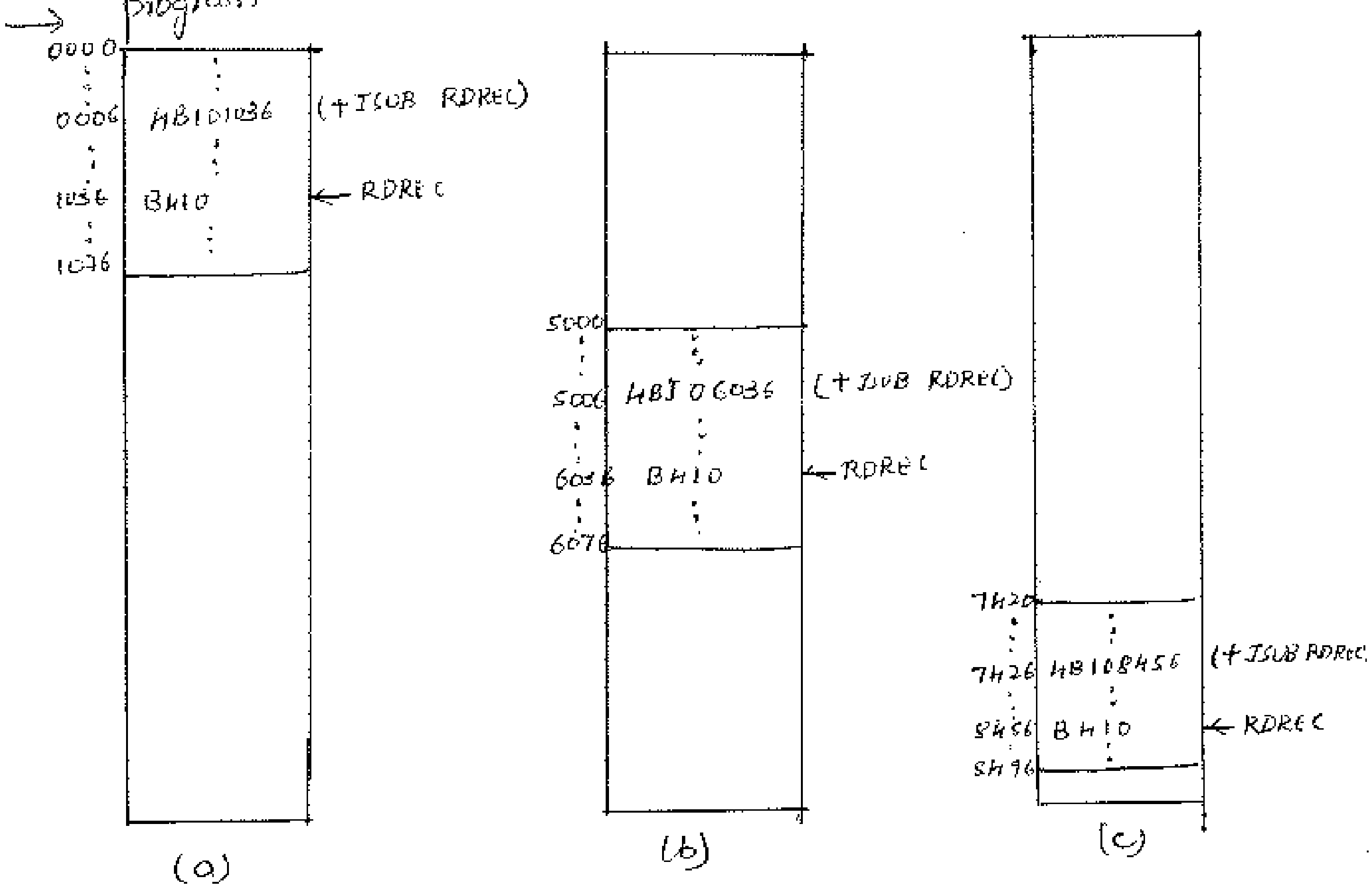


Fig: Example for program relocation

→ JSUB instruction at line 15 is loaded at address 0006

→ The address field contains 01036 (address of RDREC)

15 0006 +JSUB RDREC 4B101036

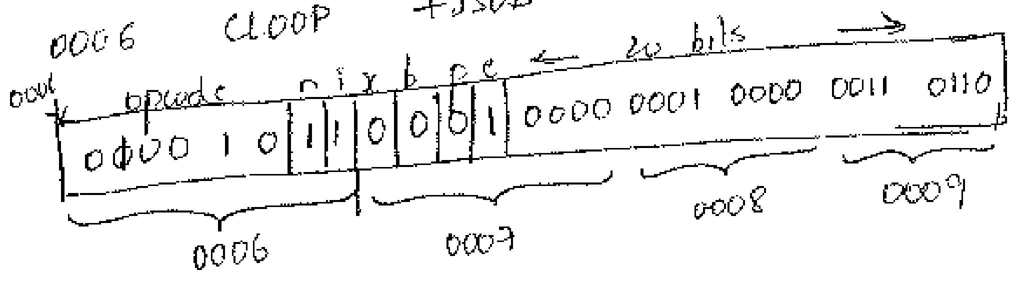
→ Suppose we want to load this program beginning at address 5000, as shown in fig (b), the address of instruction labeled RDREC will be 6036.

→ Likewise if we load at 7420 as in fig (c), then address of RDREC will be 4B108456.

→ It means, irrespective of the starting address loaded, RDREC is always 1036 bytes past the starting address of the program. This is the reason we initialized the location counter to 0. (ie relative to the starting address)

→ The modification record looks like

0006 CLOOP +JSUB RDREC 4B101036



$\boxed{M^000007^05}$

note: 05 because it is 20 bits address ⇒ 05 half byte
⇓
05×4 = 20 bits

Actualy at location 0007, first half byte is part of flag bits x, b, p, e. But length 05 tells loader to modify only last 5 half bytes. Hence instruction 4B1 remains unchanged.

Relocation for instruction of line 35 and 65

35	0013	+JSUB	CORREC	4B10105D
65	0026	+JSUB	WORREC	4B10105D

M^000014^05 & M^000027^05

→ If we add 5000 then address should be

```
 4B1/01036            4B1/01036
      5000          +     7420  ← relocatable address
 ---------          -----------
 4B1/06036            4BI08456
      7420
+
```

→ Some instructions like
CLEAR S
LDA #3
} doesnot need modification ∵ operand is not a memory address.

→ 10 STL RETADR ∴ doesnot need modification ∵ operand is specified using program-counter relative or base-relative addressing. Here the displacement is always 02D. Irrespective of location of program loaded, it is always 2D bytes away from the STL instruction

→ The ~~ten~~ distance between LENGTH and BUFFER will always be 3 byte.

The object program is rewritten as (Fig 2.6)

```
H^COPY   ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^' - ^4B10105D^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^ . .  . . .      ^B850
T^001053^1D^3B2FEA^134000^4F0000^ - - -        ^B850
T^001070^ 07^ 3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

2.3 Machine Independent Assembler features

→ machine independent means some assembler features that are not closely related to machine architecture.

This section includes

2.3.1 → The implementation of literals within an assembler

2.3.2 → Two assembler directives EQU and ORG used to define the symbols

2.3.3 → Use of expression in assembler language statements

2.3.4 → Implementation of program blocks

2.3.5 → Implementation of control sections

2.3.1 Literals

→ Constant operand can be specified as a part of the instruction that uses it, instead of using a label which is defined as constant else where. Such an operand is called a literal because the value is stated '<u>literally</u>' in the instruction

Ex:-

```
45   001A   ENDFIL   LDA   EOF      032010
                           Label
      ⋮
80   002D   EOF      BYTE  C'EOF'   454F46
```

⇓ can be written an

```
45   001A   ENDFIL   LDA   =C'EOF'  032010
                ⋮
            LTORG
*           =C'EOF'                 454F46
```

The object code generated for line 45, 215 and 230 in fig 2.6 and fig. 2.10 are identical.

(i) 45 001A ENDFIL LDA =C'EOF' 032010

opcode	n	i	x	b	p	e	disp
0000 00	1	1	0	0	1	0	0000 0001 0000

disp = opaddr - pc   TA = (pc)+disp

= 002D - 001D = 01D

⟹ 032010

(ii) 215 1062 WLOOP TD =X'05' E32011

TD = E0

disp : opaddress - pc

: 1076 - 1065 = 011

1110 ~~0000~~ 00	1	1	0	0	1	0	0000 0001 0001

E 3 2 0 1 1

(iii) 230 106B WD =X'05' DF2008

:

1076 * =X'05'

WD = DC

disp= opaddress - pc = 1076 - 106E = 008

		n	i	x	b	p	e	
1101	11	1	1	0	0	1	0	0000 0000 1000

D C

D F 2 0 0 8

```
Line      Source statement

   5      COPY      START    0            COPY FILE FROM INPUT TO OUTPUT
  10      FIRST     STL      RETADR       SAVE RETURN ADDRESS
  13                LDB      #LENGTH      ESTABLISH BASE REGISTER
  14                BASE     LENGTH
  15      CLOOP    +JSUB     RDREC        READ INPUT RECORD
  20                LDA      LENGTH       TEST FOR EOF (LENGTH = 0)
  25                COMP     #0
  30                JEQ      ENDFIL       EXIT IF EOF FOUND
  35               +JSUB     WRREC        WRITE OUTPUT RECORD
  40                J        CLOOP        LOOP
  45      ENDFIL    LDA      =C'EOF'      INSERT END OF FILE MARKER
  50                STA      BUFFER
  55                LDA      #3           SET LENGTH = 3
  60                STA      LENGTH
  65               +JSUB     WRREC        WRITE EOF
  70                J        @RETADR      RETURN TO CALLER
  93                LTORG    → Origin of literals
  95      RETADR    RESW     1
 100      LENGTH    RESW     1            LENGTH OF RECORD
 105      BUFFER    RESB     4096         4096-BYTE BUFFER AREA
 106      BUFEND    EQU      *
 107      MAXLEN    EQU      BUFEND-BUFFER    MAXIMUM RECORD LENGTH
 110      .
 115      .         SUBROUTINE TO READ RECORD INTO BUFFER
 120      .
 125      RDREC     CLEAR    X            CLEAR LOOP COUNTER
 130                CLEAR    A            CLEAR A TO ZERO
 132                CLEAR    S            CLEAR S TO ZERO
 133               +LDT      #MAXLEN
 135      RLOOP     TD       INPUT        TEST INPUT DEVICE
 140                JEQ      RLOOP        LOOP UNTIL READY
 145                RD       INPUT        READ CHARACTER INTO REGISTER A
 150                COMPR    A,S          TEST FOR END OF RECORD (X'00')
 155                JEQ      EXIT         EXIT LOOP IF EOR
 160                STCH     BUFFER,X     STORE CHARACTER IN BUFFER
 165                TIXR     T            LOOP UNLESS MAX LENGTH
 170                JLT      RLOOP          HAS BEEN REACHED
 175      EXIT      STX      LENGTH       SAVE RECORD LENGTH
 180                RSUB                  RETURN TO CALLER
 185      INPUT     BYTE     X'F1'        CODE FOR INPUT DEVICE
 195      .
 200      .         SUBROUTINE TO WRITE RECORD FROM BUFFER
 205      .
 210      WRREC     CLEAR    X            CLEAR LOOP COUNTER
 212                LDT      LENGTH
 215      WLOOP     TD       =X'05'       TEST OUTPUT DEVICE
 220                JEQ      WLOOP        LOOP UNTIL READY
 225                LDCH     BUFFER,X     GET CHARACTER FROM BUFFER
 230                WD       =X'05'       WRITE CHARACTER
 235                TIXR     T            LOOP UNTIL ALL CHARACTERS
 240                JLT      WLOOP          HAVE BEEN WRITTEN
 245                RSUB                  RETURN TO CALLER
 255                END      FIRST
```

Figure 2.9 Program demonstrating additional assembler features.

```
Line   Loc        Source statement                      Object code

  5    0000   COPY     START   0
 10    0000   FIRST    STL     RETADR                   17202D
 13    0003            LDB     #LENGTH                  69202D
 14                    BASE    LENGTH
 15    0006   CLOOP   +JSUB    RDREC                    4B101036
 20    000A            LDA     LENGTH                   032026
 25    000D            COMP    #0                       290000
 30    0010            JEQ     ENDFIL                   332007
 35    0013           +JSUB    WRREC                    4B10105D
 40    0017            J       CLOOP                    3F2FEC
 45    001A   ENDFIL   LDA     =C'EOF'                  032010
 50    001D            STA     BUFFER                   0F2016
 55    0020            LDA     #3                       010003
 60    0023            STA     LENGTH                   0F200D
 65    0026           +JSUB    WRREC                    4B10105D
 70    002A            J       @RETADR                  3E2003
 93                    LTORG
       002D   *        =C'EOF'                          454F46
 95    0030   RETADR   RESW    1
100    0033   LENGTH   RESW    1
105    0036   BUFFER   RESB    4096
106    1036   BUFEND   EQU     *
107    1000   MAXLEN   EQU     BUFEND-BUFFER
110           .
115           .        SUBROUTINE TO READ RECORD INTO BUFFER
120           .
125    1036   RDREC    CLEAR   X                        B410
130    1038            CLEAR   A                        B400
132    103A            CLEAR   S                        B440
133    103C           +LDT     #MAXLEN                  75101000
135    1040   RLOOP    TD      INPUT                    E32019
140    1043            JEQ     RLOOP                    332FFA
145    1046            RD      INPUT                    DB2013
150    1049            COMPR   A,S                      A004
155    104B            JEQ     EXIT                     332008
160    104E            STCH    BUFFER,X                 57C003
165    1051            TIXR    T                        B850
170    1053            JLT     RLOOP                    3B2FEA
175    1056   EXIT     STX     LENGTH                   134000
180    1059            RSUB                             4F0000
185    105C   INPUT    BYTE    X'F1'                    F1
195           .
200           .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205           .
210    105D   WRREC    CLEAR   X                        B410
212    105F            LDT     LENGTH                   774000
215    1062   WLOOP    TD      =X'05'                   E32011
220    1065            JEQ     WLOOP                    332FFA
225    1068            LDCH    BUFFER,X                 53C003
230    106B            WD      =X'05'                   DF2008
235    106E            TIXR    T                        B850
240    1070            JLT     WLOOP                    3B2FEF
245    1073            RSUB                             4F0000
255                    END     FIRST
       1076   *        =X'05'                           05
```

Figure 2.10 Program from Fig. 2.9 with object code.

Literal Pools:-

→ All the literal operands used in a program are gathered together into one or more literal pools.

→ normally literals are placed into a pool at the end of the program, which shows the assigned addresses and the generated data values.

→ The drawback of keeping literal pool at the end of the program is ~~use~~ the literal operand is too far away from the instruction referring it and requires a large amount of storage reservation for the buffer too.

→ To avoid this we use an assembler directive LTORG (ORIGIN OF LITERALS) which instructs the assembler to assemble the current literals pool immediately

→ when the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the program). ⇒ ie keep the literal operand close to the instruction.

→ Same literal may be used more than once in the program ie duplicate literals. but it stores only one copy of the specified data value.

Ex: 215 1062 WLOOP TD =X'05'

230 106B WD =X'05

→ Apart from one copy of data value, it stores only one data area with this value generated. Both instructions refer to the same address in the literal pool for their operand

→ There are two ways of recognizing the duplicate literals

(a) compare the character strings defining them. Same literal name with different value

Ex: X '05'

(b) compare the generated data value. This is better but increases the complexity of the assembler.

Ex: =C'EOF' and =X'454F46'

→ The problem of using character strings to recognize duplicate literals is, as we see '*' denotes a literal refers to the current value of program counter after line no. 93. There may be same literals that have the same name but different values,

for example the statements

```
BASE  *     —①
LDB   =*    —②
```

① — loads the beginning address of the program into register B. This value will be available later for base relative addressing.

x* → It causes a problem if we use of line No. 13

i.e 13 0003 LDB =*. 692003

it specifies an operand with value 0003.

55 0020 LDA =* 010020

i.e literal operands have identical names but they have different values and both must appear in the literal pool.

x* → The same problem arises if a literal refers to any other item whose value changes between one point in the program and another.

→ The datastructure used to store literal operands is literal table "LITTAB"

→ literal Table (LITTAB) : It is a hashtable using literal name or value as the key.

contains
↳ literal name
↳ operand value
↳ operand length
↳ address assigned

NAME	OPERAND VALUE	LENGTH	ADDRESS
=C'EOF'	EOF	03	002D
=X'05'	05	01	1076

pass-1

→ Builds LITTAB with literal name, operand value and length, leaving the address unassigned

→ when LTORG statement is encountered, assign an address to each literal not yet assigned an address. Along with this, location counter is updated to reflect the number of bytes occupied by each literal

Pass - 2

→ Search LITTAB for each literal operand encountered to generate respective object code

→ Generate data values using BYTE or WORD statements

→ Generate modification record for literals that represent an address in the program.

Difference between literal and an immediate operand

	Literal (=)	Immediate operand (#)
1.	Literal is an assembler directive	Immediate is a machine recognizable data
2.	The assembler generates the specified value as a constant at some other memory location. The address of this generated constant is used as target address for machine instructions	2. Here value is assembled as part of machine instruction.
3	Architectural support is required	Architectural support not required
4.	very slow since values are obtained from data memory	faster than literal ∵ data is within the instruction
5.	capable of storing large data	can't store larger data ie Fullword is opcode, registers
6.	[illegible] LDA =C'EOF'	Eg 55 0020 LDA #3 010$\boxed{003}$

3.3.2 Symbol defining statements

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.

The assembler directives are a) EQU b) ORG

a) EQU (Equate)

→ allows the programmer to define symbols (i.e enters it into SYMTAB) and assigns to it the specified value.

syntax: Symbol EQU value

constant
Expression
previously defined symbol

→ The value may be

(i) constant

(ii) An expression involving constant

(iii) previously defined symbols.

→ Uses of EQU

i) To establish symbolic names that can be used for improved readability in place of numeric values.

Ex: +LDT #4096 ; load the value 4096 into reg. T

⇓ replace with

MAXLEN EQU 4096

+LDT MAXLEN

→ when assembler encounters the EQU statement, it enters MAXLEN to SYMTAB with value 4096.

→ During assembly of LDT instruction, the assembler searches the SYMTAB for the MAXLEN symbols and using its value as the operand in the instruction.

→ The advantage of doing so is if we want to change the value 4096 to some other value, we need to change in only one place MAXLEN instead of searching or scanning through the program for #4096 for the replacement (required change) ⇒ #define in C

2) To define mnemonic names for registers.

Ex:
```
A   EQU  0
X   EQU  1
L   EQU  2
```

→ The symbols A, X, L has to be entered into SYMTAB with their corresponding values 0,1,2.

→ instruction RMO A, X searches the SYMTAB for A and X and their values to assemble the instruction

3) To reflect the logical function of the registers

Ex:
```
BASE    EQU  R1
COUNT   EQU  R2
INDEX   EQU  R3
```

→ implies Register R1 is used as base register, R3 as index registers and R2 as program counter,

→ Forward reference is not allowed in EQU ie all terms in the value field must have been defined previously during pass-1

Ex:
```
ALPHA  RESW  1     }             BETA   EQU   ALPHA }  not
BETA   EQU   ALPHA } Allowed     ALPHA  RESW  1     }  allowed
```

b) ORG (origin)

→ Assembler directive used to indirectly assign value to symbols.

→ syntax :

ORG value

→ value can be

(i) constant

(ii) expression involving constant

(iii) Previously defined symbols

→ when ORG is encountered, the assembler resets its LOCCTR to the specified value.

→ Location counter is used to control assignment of storage in the object program. Hence altering its value would result in an incorrect assembly.

∴ the directive should be minimum used.

→ The ORG statement will affect the values of all labels defined until the next ORG.

→ If the previous value of LOCCTR is automatically remembered, then we can return to the normal use of LOCCTR just by writing

ORG

→ Example : To define a symbol table with the following structure.

→ symbol table with the given structure

	SYMBOL	VALUE	FLAGS
STAB (100 entries)			
	6 bytes	3 bytes (1 word)	2 byte

↳ SYMBOL field contains user defined symbols.

↳ VALUE field represents the value assigned to the symbol

↳ FLAG field specifies symbol type and other information

↳ The space for this table is reserved as

```
STAB:   RESB    1100    ;
```

$\underbrace{100}_{\text{entries}} \times \underbrace{11}_{\text{each entry}} = 1100$

↳ we can access the label entries in two ways (usage of EQU and ORG)

↳ using <u>EQU</u> ⟹

```
SYMBOL    EQU    STAB
VALUE     EQU    STAB+6     offset from
FLAGS     EQU    STAB+9     STAB
```

(i) To fetch the value field,

```
LDA    VALUE,X    ; where X=0,11,22,.. for
                           each entry
```

↳ index register

*** (ii) This method of definition simply defines the labels, it doesnot make the structure of the table as clear as it might be.

(iii) Therefore we make use of <u>ORG</u>

↳ Using ORG ⇒

```
STAB     RESB   1100
         ORG    STAB         ← set LOCCTR to STAB
SYMBOL   RESB   6  |
VALUE    RESW   1  |   ← Size of each field
FLAGS    RESB   2  |
         ORG    STAB+1100    ← Restore LOCCTR
```

(i) The first ORG sets the location counter to the value of STAB

(ii) RESB statement defines SYMBOL to have the current value in LOCCTR

(iii) LOCCTR is then advanced, so the label on RESW statement assigns to VALUE to address (STAB +6) and then advanced to assign to FLAGS to address (STAB +9).

(iv) The last ORG statement sets LOCCTR back to its previous value. ~~which~~ That is the address of the next unassigned byte of memory after the table STAB.

→ Forward reference is not allowed in ORG ie all symbols used to specify the new location counter value have to be previously defined.

→ Example :

```
         ORG    ALPHA
BYTE1    RESB   1
BYTE2    RESB   1
```

```
BYTE3     RESB   1
          ORG
ALPHA     RESB   1
```

↳ cannot processed ∵ the assemble doesnot know what value has to be assigned to the location counter in response to the first ORG statement The symbols BYTE1, BYTE2, BYTE3 are not assigned addresses during pass 1.

↳ It has to be written as

```
ALPHA     RESB    1
          ORG     ALPHA
BYTE1     RESB    1
BYTE2     RESB    1
BYTE3     RESB    1
          ORG .
```

2.3.3. Expressions

→ The assembler allows the use of expressions as operand.

→ It calculates the expressions and produces a single operand address or value.

→ The expression consists of

(i) operators : + - * / (division is usually defined to produce an integer value)

(ii) Individual terms : Constants, user-defined symbols, special terms like * (current value of the location counter).

```
Ex:   MAXLEN    EQU    BUFEND - BUFFER
      STAB      RESB   (6+3+1) * MAX
      BUFEND    EQU    *
```

→ The values of terms can be absolute (independent of program location) such as constants or relative (to the beginning of the program) such as Address labels, data areas, references to the location counter value.

→ Expressions are classified as

(i) Absolute Expressions } based on type of value
(ii) Relative Expressions } produced.

(i) Absolute Expressions :

↳ Absolute means independent of program location and contains absolute terms like constants

↳ It may also contain relative terms provided the relative terms occur in pair and the terms in each such pair have opposite signs.

↳ It is not necessary that the paired terms be adjacent to each other in the expression however, all relative terms must be capable of being paired in this way.

↳ None of the relative terms may enter into a multiplication or division operation.

(ii) Relative Expressions

↳ Relative means relative to the beginning of the program. such as labels on the instruction, data areas, references to the location counter value.

↳ Here, all of the relative terms accept one can be paired as in absolute expressions and the remaining unpaired relative term must have a positive sign.

↳ no relative terms may enter into a multiplication or division operation.

<u>note</u> : If Either of absolute expression or relative expression do not meet the conditions, they are flagged as errors

→ A relative term or expression represents some value which is written as (S+r)

- S = starting address of the program
- r = value of term or expression relative to the starting address.

Ex: (1) MAXLEN EQU BUFEND - BUFFER

↳ both BUFEND and BUFFER are relative terms representing an address within the program. The expression represents an absolute value.

(2) Illegal Expressions

BUFEND + BUFFER ; no opposite sign

100 - BUFFER ; both are not relative terms

3 * BUFFER ; * can't be used

→ Type of expression is determined by keeping track of symbol types in the program. This is done by adding a flag (R or A) in the SYMTAB for each symbol defined

Ex:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

} few symbols of fig 2.10

2.3.4 Program Blocks

→ Till now, we have seen that the program being assembled was treated as a single unit, eventhough it had subroutine, dataareas etc resulting in a single block of object code.

→ within this object code (program) the generated machine instructions and data appeared in the same order as they were written in the source program.

→ But sometime it is required to logically rearrange the statements of the source program so that the 1) large buffer area can be moved to the end of object program, 2) no need of using extended instruction format, 3) the base register usage is not required, 4) the problem of placing literals in program has to be more flexible etc.

** → All these are achieved through some of the assembler features such as program blocks and control sections.

** → Program blocks: Allows the generated machine instructions and data to appear in the object program in a different order from the corresponding source statement.

or

Program blocks are segment of code that are rearranged within a single object program unit.

Assembler directive : USE

syntax : USE BLOCKNAME

Fig. 2.11 shows the source program with program blocks

→ There are three blocks in the program

(i) Unnamed program block contains the executable instructions of the program

(ii) CDATA program block contains all 'data areas that consists of ~~larger~~ few block of memory ie few words or less in length

(iii) CBLKS program block contains all data areas that consists of larger blocks of memory.

→ At the begining, statements are assumed to be part of the Unnamed (default) block, If no USE statements are included, the entire program belongs to this single block.

→ USE on line 92 indicates the begining of CDATA block

→ USE on line 103 indicates the begining of CBLKS block

→ USE on line 123 resumes the default block

* → Each program block may contain several separate segments of the source program but assembler will logically rearrange these segments to gather together the pieces of each block and assign address.

→ Program readability is better if data areas are placed in the source program close to the statements that reference them.

The assembler accomplishes this logical rearrangement of code by maintaing during pass-1 and pass 2

(i) Pass-1

Fig 2.12(b) shows the pass-1 of program blocks

→ A seperate location counter for each program block is assigned and is assigned to ZERO when a program block begins

↳ Saving and Restoring the current value of LOCCTR when occurs whebe switching between blocks

↳ Each label is assigned an address relative to the start of the block.

↳ Stores the block name and number in the SYMTAB along with the assigned relative address of the label.

↳ At the end of pass-1, indicate the block length as the latest value of LOCCTR for each block.

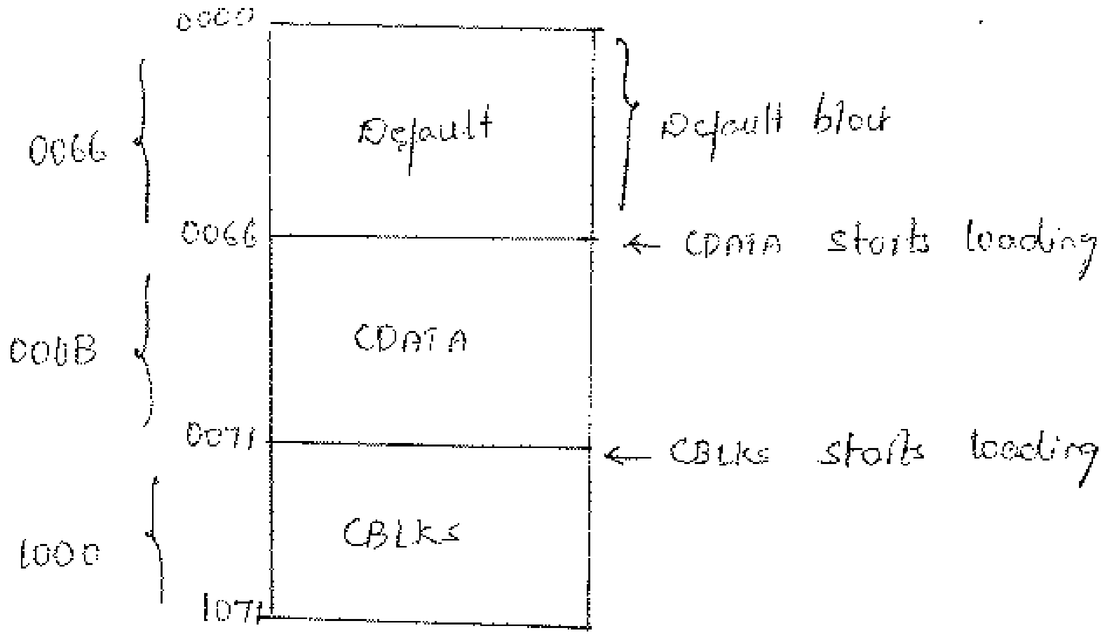
↳ constructs a table which contains the starting address and length for all blocks.

↳ Assembler assigns to each block a starting address in the object program (beginning with relative location 0).

latest value of LC ↓

Block name	Block Number	Address	Length	
default	0	0000	0066	0063 + 0003 = 0066
CDATA	1	0066	000B	0066 + 000B = 0071
CBLKS	2	0071	1000	1071 - 0071 = 1000

↳ Flag is also added in this table



ii) Pass 2:

→ calculates the address for each symbol relative to the start of the object program (not the start of the individual program block) by adding

(i) The location of the symbol relative to the start of its block (from SYMTAB)

(ii) The starting address of this block.

Example

```
20     0006    0      LDA      LENGTH
 :
                      USE      CDATA
92     0000    1
100    0003    1   LENGTH   RESW     1
```

→ The value of the operand LENGTH is 0003 relative to block 1 (CDATA)

∴ address (TA) = 0003 + 0066 = 0069 relative to program when this instruction is executed

```
Line    Loc/Block         Source statement                  Object code

  5     0000  0     COPY      START     0
 10     0000  0     FIRST     STL       RETADR              172063
 15     0003  0     CLOOP     JSUB      RDREC               4B2021
 20     0006  0               LDA       LENGTH              032060
 25     0009  0               COMP      #0                  290000
 30     000C  0               JEQ       ENDFIL              332006
 35     000F  0               JSUB      WRREC               4B203B      T1
 40     0012  0               J         CLOOP               3F2FEE
 45     0015  0     ENDFIL    LDA       =C'EOF'             032055
 50     0018  0               STA       BUFFER              0F2056
 55     001B  0               LDA       #3                  010003
 60     001E  0               STA       LENGTH              0F2048
 65     0021  0               JSUB      WRREC               4B2029      T2
 70     0024  0               J         @RETADR             3E203F
 92     0000  1               USE       CDATA
 95     0000  1     RETADR    RESW      1           →   0066+0000 = 0066  (block 1)
100     0003  1     LENGTH    RESW      1           →   0066+0003 = 0069
103     0000  2               USE       CBLKS
105     0000  2     BUFFER    RESB      4096        →   0071+0000+0000 = 0071
106     1000  2     BUFEND    EQU       *
107     1000        MAXLEN    EQU       BUFEND-BUFFER     1000-0000 = 1000
110                 .
115                 .         SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
123     0027  0               USE
125     0027  0     RDREC     CLEAR     X                   B410
130     0029  0               CLEAR     A                   B400
132     002B  0               CLEAR     S                   B440
133     002D  0              +LDT       #MAXLEN             75101000
135     0031  0     RLOOP     TD        INPUT               E32038      T3
140     0034  0               JEQ       RLOOP               332FFA
145     0037  0               RD        INPUT               DB2032
150     003A  0               COMPR     A,S                 A004
155     003C  0               JEQ       EXIT                332008
160     003F  0               STCH      BUFFER,X            57A02F
165     0042  0               TIXR      T                   B850
170     0044  0               JLT       RLOOP               3B2FEA
175     0047  0     EXIT      STX       LENGTH              13201F      T4
180     004A  0               RSUB                          4F0000
183     0006  1               USE       CDATA
185     0006  1     INPUT     BYTE      X'F1'               F1          T5 → 0066+0006
                                                                              = 006C
                                                                              (block 1)
195                 .
200                 .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
208     004D  0               USE
210     004D  0     WRREC     CLEAR     X                   B410
212     004F  0               LDT       LENGTH              772017
215     0052  0     WLOOP     TD        =X'05'              E3201B
220     0055  0               JEQ       WLOOP               332FFA      T6
225     0058  0               LDCH      BUFFER,X            53A016
230     005B  0               WD        =X'05'              DF2012
235     005E  0               TIXR      T                   B850
240     0060  0               JLT       WLOOP               3B2FEF
245     0063  0               RSUB                          4F0000
252     0007  1               USE       CDATA
253                           LTORG                                     T7 → 0066+7
        0007  1     *         =C'EOF'                       454F46            = 006D
        000A  1     *         =X'05'                        05
255                           END       FIRST                     → 0066+000A = 0070
```

Default(0)

CDATA(1)

CBLKS(2)

Default(0)

CDATA(1)

Default(0)

CDATA(1)

Figure 2.12(a) Program from Fig. 2.11 with object code.

```
Line          Source statement

  5    COPY     START   0              COPY FILE FROM INPUT TO OUTPUT
 10    FIRST    STL     RETADR         SAVE RETURN ADDRESS
 15    CLOOP    JSUB    RDREC          READ INPUT RECORD
 20             LDA     LENGTH         TEST FOR EOF (LENGTH = 0)
 25             COMP    #0
 30             JEQ     ENDFIL         EXIT IF EOF FOUND
 35             JSUB    WRREC          WRITE OUTPUT RECORD
 40             J       CLOOP          LOOP
 45    ENDFIL   LDA     =C'EOF'        INSERT END OF FILE MARKER
 50             STA     BUFFER
 55             LDA     #3             SET LENGTH = 3
 60             STA     LENGTH
 65             JSUB    WRREC          WRITE EOF
 70             J       @RETADR        RETURN TO CALLER
 92             USE     CDATA
 95    RETADR   RESW    1
100    LENGTH   RESW    1              LENGTH OF RECORD
103             USE     CBLKS
105    BUFFER   RESB    4096           4096-BYTE BUFFER AREA
106    BUFEND   EQU     *              FIRST LOCATION AFTER BUFFER
107    MAXLEN   EQU     BUFEND-BUFFER  MAXIMUM RECORD LENGTH
110    .
115    .        SUBROUTINE TO READ RECORD INTO BUFFER
120    .
123             USE
125    RDREC    CLEAR   X              CLEAR LOOP COUNTER
130             CLEAR   A              CLEAR A TO ZERO
132             CLEAR   S              CLEAR S TO ZERO
133            +LDT     #MAXLEN
135    RLOOP    TD      INPUT          TEST INPUT DEVICE
140             JEQ     RLOOP          LOOP UNTIL READY
145             RD      INPUT          READ CHARACTER INTO REGISTER A
150             COMPR   A,S            TEST FOR END OF RECORD (X'00')
155             JEQ     EXIT           EXIT LOOP IF EOR
160             STCH    BUFFER,X       STORE CHARACTER IN BUFFER
165             TIXR    T              LOOP UNLESS MAX LENGTH
170             JLT     RLOOP            HAS BEEN REACHED
175    EXIT     STX     LENGTH         SAVE RECORD LENGTH
180             RSUB                   RETURN TO CALLER
183             USE     CDATA
185    INPUT    BYTE    X'F1'          CODE FOR INPUT DEVICE
195    .
200    .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
208             USE
210    WRREC    CLEAR   X              CLEAR LOOP COUNTER
212             LDT     LENGTH
215    WLOOP    TD      =X'05'         TEST OUTPUT DEVICE
220             JEQ     WLOOP          LOOP UNTIL READY
225             LDCH    BUFFER,X       GET CHARACTER FROM BUFFER
230             WD      =X'05'         WRITE CHARACTER
235             TIXR    T              LOOP UNTIL ALL CHARACTERS
240             JLT     WLOOP            HAVE BEEN WRITTEN
245             RSUB                   RETURN TO CALLER
252             USE     CDATA
253             LTORG
255             END     FIRST
```

Figure 2.11 Example of a program with multiple program blocks.

disp : TA - (PC)

= 0069 - 0009

= 0060

PC = 0000 + 0009 = 0009

{ starting address of def block) + 0009 = 0009

	n i x b p e	
0000 00	1 1 0 0 1 0	060

→ 032060

SYMTAB

Label name	Block number	Address	Flag
Length	1	0003	

note: line 107

1000 MAXLEN EQU BUFEND-BUFFER

→ shown without a block number indicates that MAXLEN is an absolute symbol, whose value is not relative to the start of any program block.

object Program :

→ It is not necessary to physically rearrange the generated code in the object program. The assembler just simply inserts the proper load address in each text record. The loader will load these code into correct place **

→ Header record as before

→ Text records : the first 2 text records generated from line 5 through 70.

→ when USE statement on line 92 is encountered, the assembler writes the new Text record eventhough there is room (space) in the previous text record.

→ The process continues till the end of the program

```
H^COPY ^000000^001071
T^000000^1E^172063^4B2021^ . . . -       . . .  . ^010003
T^00001E^09^0F2048^4B2029^3E203F
T^000027^1D^B410^B400^ . .    . . . .  . ^B850
T^000044^09^3B9FEA^13201F^4F0000
T^00006C^01^F1           . . . .    ^4F0000
T^00004D^19^B410^772017^ . . . . .
T^00006D^04^454F46^05
E^000000
```

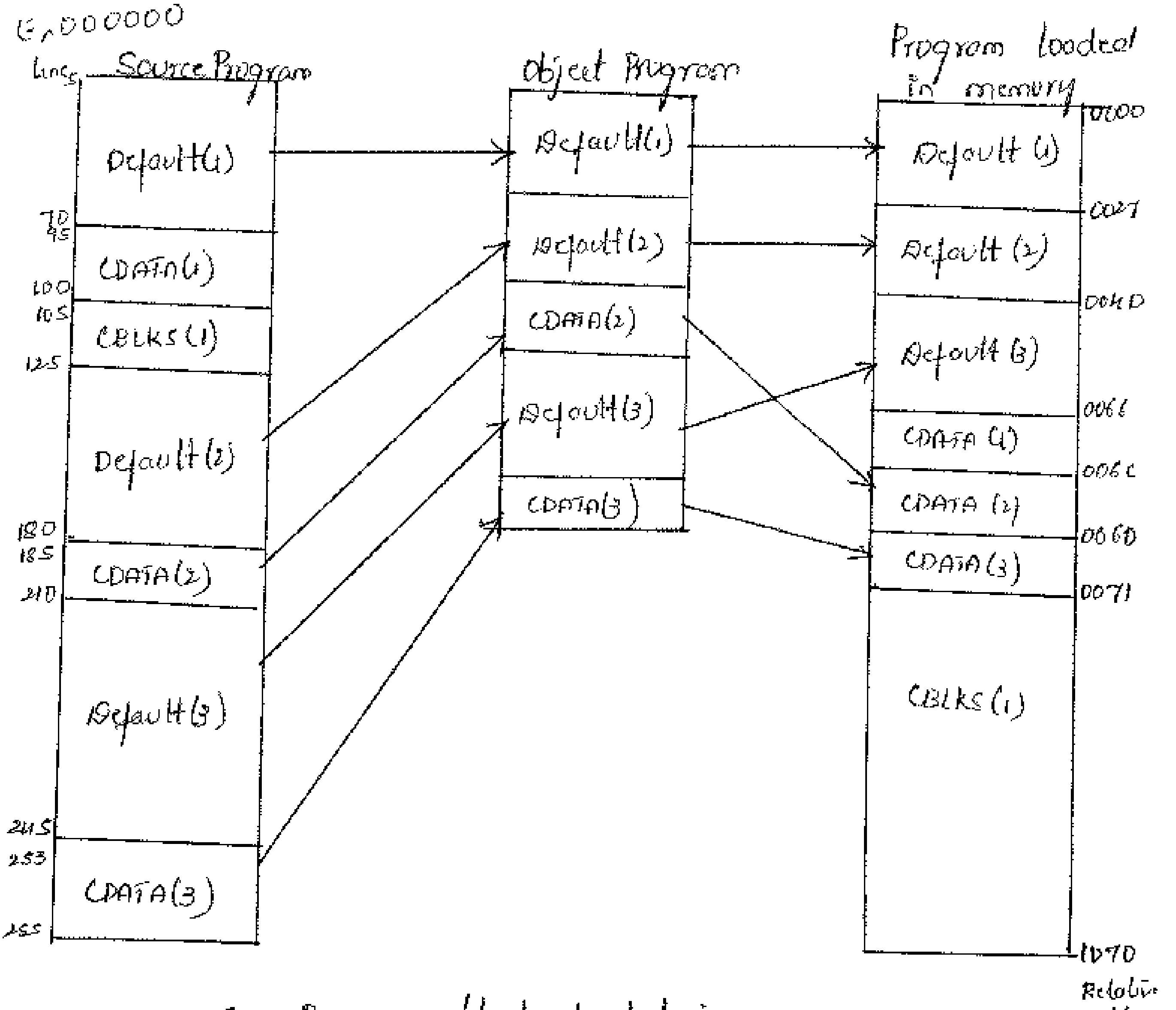


Fig: Program blocks loaded in memory

```
begin
 block number = 0 LOCCTR[i] = 0 for all i
 read the first input line
 if OPCODE = 'START' then
 begin
   write line to intermediate file ;
   read next input line
 end {if START}
 while OPCODE ≠'END' do
 if OPCODE = 'USE'
 begin
   if there is no OPEREND name then
     set block name as default
   else block name as OPERAND name
   if there is no entry for block name then
     insert (block name, block number ++) in block table
   i = block number for block name
   if this is not a comment line then
     begin
     if there is a symbol in the LABEL field then
       begin
       search SYMTAB for LABEL
       if found then
         set error flag (duplicate symbol)
       else
         insert (LABEL, LOCCTR[i]) into SYMTAB
       end {if symbol}
     Search OPTAB for OPCODE
     if found then
       add 3 instruction length to LOCCTR[i]
     else if OPCODE = 'WORD' then
       add 3 to LOCCTR[i]
     else if OPCODE = 'RESW' then
       add 3 * #[OPERAND] to LOCCTR[i]
     else if OPCODE = 'RESB' then
       add #[OPERAND] to LOCCTR[i]
     else if OPCODE = 'BYTE' then
     begin
       find length of constant in bytes
       add length to LOCCTR[i]
     end {if byte}
 else
```

Figure 2.12(b) Pass 1 of program blocks.

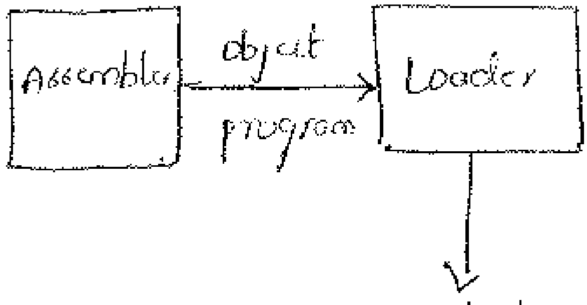
```
    Set error flag
    end {if not a comment}
  write line to intermediate file
  read Text input line
  end {while not END}
write last line to intermediate file
save Length[i] as LOCCTR[i] for all i
Address[0] = starting address
Address[i] = address(i - 1) + Length(i - 1)
             {for i = 1 to max(block number)}
insert(address[i], Length[i]) in block table for all i
end {Pass 1}
```

Figure 2.12(b) *(cont'd)*

```
If OPCODE = 'USE' then
  set block number for block name with OPERAND field
  search SYMTAB for OPERAND
  store symbol value + address [block number] as operand address
end {Pass 2}
```

Figure 2.12(c) Pass 2 of program blocks.

Loading



contacts OS & OS says
start loading at particular address

Assembler generate the object program(Header Record, Text record, modify record and End record). Assembler interacts with loader through object program. loader contacts operating system to load at particular address. Then OS checks if that particular ~~address~~ space is free if so it starts loading. If not it will tell the loader to either wait or remove unnecessary space

∴ "loader loads the object program residing in hard disk to main memory and start executing".

→ when there is no enough space, somebody has to instruct the loader to change its address and try loading. It is done by assembler not by OS.

∴ Assembler instructs the loader to change the address.

ie line 15/35/65 .   + JSUB   RDREC

line 15 → address is 4B101036 at 0006.

01036 starts from 0007 (middle). ie we can access 1 byte but not 1 nibble.

0006 - 4B

0007 → 10 → go here and modify the record. This is done by assembler ∴ we have modification record M^000007^05

→ loader should listen to both assemble and os

→ Assembler says goto 0007 and modify but os says it is loaded at 5000 ∴ modify at 5007.

** → +JSUB #4096 → doesnot need modification record ∵ it is immediate addressing. Irrespective of relocation it remains same.

→ Execution is part of microprocessor

Relocation → All instructions works except F4 instructions.

E1   +JSUB   RDREC   4B101036

4	10110001	01036

TA = 01036 → loaded at 0000

** if it is loaded at 5000, it will not work properly. loader stops functioning ∵ TA = 01036.

+JSUB 06036 ✗ ∴ go for modification record

→ Start adding length as
- memory address - 3 bytes
- Register - to - register - 2 bytes
- Extended - 4 bytes

→ note: All literals should be placed where LTORG appears in the program if LTORG is not present all literals will be inserted at the end of the program. (line no 253)

→ At line ~~no~~ 95, block 1 (CDATA) starts ∴ it stores the LOCCTR value ie 0027 in LOCCTR-0. Then starts assigning 0000 to block 1.

→ At line 105, block 2 (CBLCK) starts ∴ it saves the LOCCTR value - 0006 in LOCCTR-1 column.

→ line 125, block 0 starts again. It restores the LOCCTR value 0027 and starts over till line no. 185 (LOCCTR = 004D)

→ line 185, block-1 (CDATA) restarts by restoring the saved LOCCTR value & saves LOCCTR = 0007 and starts over till line

→ line 210, restores 004D and starts over till line no 245 having LOCCTR = 0066 which is stored in LOCCTR-0 column

THREE LOCATION COUNTERS

LOCCTR-0 (Default block)	LOCCTR-1 (CDATA Block)	LOCCTR-2 (CBLKS)
0000	0000	0000
0027	0006	1000
004D	0007	
0066	000B	

LITERAL TABLE

Literal name	value of Literal	Length of Literal	Address of Literal
=C'EOF'	454F46	03	0007
=X'05'	05	01	000A
			000B

BLOCK TABLE

Block name	Block number	Address	Length
Default	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

} program Length = 66 + 0B + 1000 = 1071

SYMBOL TABLE (with block number)

Symbol name	value	Block no
FIRST	0000	0
CLOOP	0003	0
ENDFIL	0015	0
RETADR	0000	1
LENGTH	0003	1
BUFFER	0000	2
BUFEND	1000	2

Symbol name	value	Block NO
MAXLEN	1000	
RDREC	0027	0
RLOOP	0031	0
EXIT	0047	0
INPUT	0006	1
WRREC	004D	0
WLOOP	0052	0

→ line 252, we have USE CDATA (Block 1) and 253 → LTORG ∴ store LOCCTR = 0007 at line 253

ie 253 0007 LTORG

** ↓

all literals should be placed where LTORG appears in the program. we have two literals here ie $= C\text{'}\underbrace{\text{EOF}}_{3\text{ Byte}}\text{'}$ and $= X\text{'}\underbrace{\text{05}}_{1\text{ Byte}}\text{'}$ ⇒ 4 bytes

starts at 0007, 0008, 0009, 000A
(E) (O) (F) (05)

So it stores 000B in LOCCTR-1 column

→ line 105, block 2 starts and it reserve 1000 bytes of memory ∴ it saves 1000 in LOCCTR-2 column

note → BUFEND EQU 100 ⇒ value of bufend is 100

BUFEND EQU * ⇒ value will be current location

value = 0000 + 1000 = 1000

Stores these values in literal table.

10 0000 0 STL RETADR

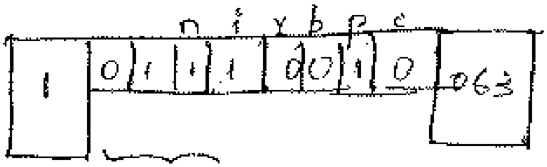
↳ present in block 1 ∴ add use of block 0 (default block)

∴ displacement = sizeof (previous block) + TA - PC

= sizeof (B0) + RETADR - PC

= 0066 + 0000 - 0003

= 0063



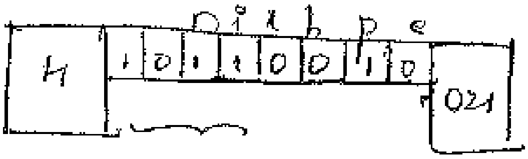
1 7 2 063 ⟹ 172063

15 0003 0 JSUB RDREC

Block (0)

disp = sizeof previous block + TA - PC

= 0 + 0027 - 006 = 021



4 B 2 021 ⟹ 4B2021

20 0006 0 LDA $\underbrace{\text{LENGTH}}_{\text{block 1}}$

disp: size q previous block + TA - PC

= 0066 + 0003 - 0009 = 0060

			n	i	x	b	p	e	
0	0	0	1	1	0	0	1	0	060

$\underbrace{0}$ $\underbrace{3}$ $\underbrace{2060}$ = 032060

35 000F $\underbrace{\text{JSUB}}_{48}$ $\underbrace{\text{WRREC}}_{\text{belongs}}$ to block 0

⟹ As before

** 45 0015 LDA = C'EOF'

↓

belongs to CDATA not to default block since it is literal which is placed after LTORG.

disp: size of previous block + TA - PC

= size q BO + C'EOF' - PC

= 0066 + 0007 - 0018 = 0055

			n	i	x	b	p	e	
0	0	0	1	1	0	0	1	0	055

0 $\underbrace{3}$ $\underbrace{2055}$ ⟹ 032055

5D 001B STA BUFFER

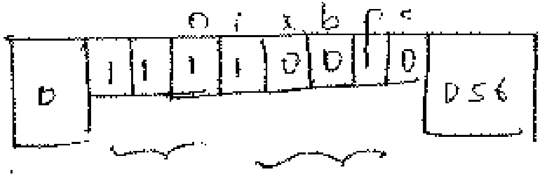
01 belongs to block 2 (CBLKS)

disp = size of (B0 + B1) + BUFFER - PC

= size of (Default block + CDATA) + BUFFER - PC

= 0066 + 000B + 0000 - 001B

= 0071 - 001B = ~~0~~056



0 F 2 056 ⇒ 0F2056

object Program

H^COPY ^ 000000 ^ 001071

1 T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE ^032055^0F2056^010003

2 T^00001E^09^0F2048^4B2029^3E203F

3 T^000027^1D^B410^B410^B400^B440^75101000^E32038^332FFA^ ^DB2032^A004^332008^57A02F^B850

4 T^000044^09^3B2FEA^13201F^4F0000

5 T^00006C^01^F1

6 T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850 ^3F2FEE^4F0000

7 T^00006D^04^454F46^05

E^000000

Loading the object program into memory

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000	17	20	63	4B	20	21	03	20	60	29	00	00	33	20	06	4B	cxpull(1)
0010	20	3B	3F	2F	EE	03	20	55	0F	20	56	01	00	03	0F	20	T1+T2
0020	48	4B	20	29	3E	20	3F	B4	10	B4	00	B4	40	75	10	10	cxpull(2)
0030	00	E3	20	38	33	2F	FA	DB	20	32	A0	04	33	20	08	57	T3+T4
0040	A0	2F	B8	50	3B	2F	EA	13	20	1F	4F	00	00	B4	10	17	Bxpull(3)
0050	40	17	E3	20	1B	33	2F	FA	53	A0	16	0F	20	12	B8	50	T5
0060	3B	2F	EE	4F	00	00	RETADR			LENGTH			F1	45	4F	46	
									(DATA(1)				CLOOP				
0070	05																
	[illegible]																
0080																	
0090							BUFFER										
.																	
.																	
.																	
1050																	
1060																	
1070																	

How does microprocessor execute an instruction

L = 666600

Ex: 10 ^{0000} STL RETADR 172063

⟹ store the contents of linkage register into RETADR location

⟹ opcode for STL = 14 (known by microprocessor)

			n	i	x	b	p	e	
1	0	1	1	1	0	0	1	0	063

1 7 2063

TA = PC + disp

= 0003 + 063 = $\underline{0066}$

RETADR

ie STL 0066 ⟹ copy the contents of linkageregister (666600) into address 0066 note :- If starting address (location) is changed from 0000 to 5000, it works as usual ∵ we don't have format to address. ∴ no need of modification record as before. ⟶ advantage of program block.

2.3.5 Control sections and Program linking

→ Control section is a part of the program that maintains its identity after assembly.

→ Each control section can be loaded and relocated independently of the other

→ Control sections are usually used for subroutines or other logical subdivisions of a program.

→ The programmer can assemble, load and manipulate each of these control sections separately

* → It uses a assembler directive ; CSECT which indicates the beginning of the control section. where each control section starts its location ~~loc~~ counter separately

** → when control sections form logically related parts of a program, it is necessary to provide some means for linking them together. This is because instructions in one control section may need to refer to instruction or data located in another control section. and assembler has no idea where exactly the control sections will be located at execution time.

→ such reference between control sections are called external references.

→ The assembler generate information for each external reference that will allow the loader to perform the required linking.

→ There are two type of ~~exte~~ reference external symbols

(1) External Definition (EXTDEF)

- Symbols that are defined in one section and are used by other sections

  syntax: EXTDEF name [,name]

  Ex: EXTDET BUFFER, BUFEND

(2) External Reference (EXTREF)

- symbols that are used in this control sections but are defined in someother control sections.

  syntax: EXTREF name [,name]

  Ex : EXTREF RDREC, WRREC

note: To reference a external symbols, extended format instruction (Format 4) is needed

```
Line   Loc          Source statement                       Object code

  5    0000   COPY      START    0
  6                     EXTDEF   BUFFER,BUFEND,LENGTH
  7                     EXTREF   RDREC,WRREC
 10    0000   FIRST     STL      RETADR                    172027
 15    0003   CLOOP    +JSUB     RDREC                     4B100000
 20    0007             LDA      LENGTH                    032023
 25    000A             COMP    #0                         290000
 30    000D             JEQ      ENDFIL                    332007        T1
 35    0010            +JSUB     WRREC                     4B100000      (1D)
 40    0014             J        CLOOP                     3F2FEC
 45    0017   ENDFIL    LDA     =C'EOF'                    032016
 50    001A             STA      BUFFER                    0F2016
 55    001D             LDA     #3                         010003
 60    0020             STA      LENGTH                    0F200A        T2
 65    0023            +JSUB     WRREC                     4B100000      (0D)
 70    0027             J       @RETADR                    3E2000
 95    002A   RETADR    RESW     1
100    002D   LENGTH    RESW     1
103                     LTORG
       0030   *        =C'EOF'                             454F46        T3 (03)
105    0033   BUFFER    RESB     4096
106    1033   BUFEND    EQU      *
107    1000   MAXLEN    EQU      BUFEND-BUFFER

109    0000   RDREC     CSECT
110           .
115           .         SUBROUTINE TO READ RECORD INTO BUFFER
120           .
122                     EXTREF   BUFFER,LENGTH,BUFEND
125    0000             CLEAR    X                         B410
130    0002             CLEAR    A                         B400
132    0004             CLEAR    S                         B440
133    0006             LDT      MAXLEN                    77201F
135    0009   RLOOP     TD       INPUT                     E3201B
140    000C             JEQ      RLOOP                     332FFA
145    000F             RD       INPUT                     DB2015        T1
150    0012             COMPR    A,S                       A004          (1D)
155    0014             JEQ      EXIT                      332009
160    0017            +STCH     BUFFER,X                  57900000
165    001B             TIXR     T                         B850
170    001D             JLT      RLOOP                     3B2FE9
175    0020   EXIT     +STX      LENGTH                    13100000
180    0024             RSUB                               4F0000        T2
185    0027   INPUT     BYTE     X'F1'                     F1            (0E)
190    0028   MAXLEN    WORD     BUFEND-BUFFER             000000

193    0000   WRREC     CSECT
195           .
200           .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205           .
207                     EXTREF   LENGTH,BUFFER
210    0000             CLEAR    X                         B410
212    0002            +LDT      LENGTH                    77100000
215    0006   WLOOP     TD      =X'05'                     E32012
220    0009             JEQ      WLOOP                     332FFA
225    000C            +LDCH     BUFFER,X                  53900000      T1
230    0010             WD      =X'05'                     DF2008        (1C)
235    0013             TIXR     T                         B850
240    0015             JLT      WLOOP                     3B2FEE
245    0018             RSUB                               4F0000
255                     END      FIRST
       001B   *        =X'05'                              05
```

Figure 2.16 Program from Fig. 2.15 with object code.

```
Line        Source statement

  5    COPY      START     0              COPY FILE FROM INPUT TO OUTPUT
  6              EXTDEF    BUFFER,BUFEND,LENGTH
  7              EXTREF    RDREC,WRREC
 10    FIRST     STL       RETADR         SAVE RETURN ADDRESS
 15    CLOOP    +JSUB      RDREC          READ INPUT RECORD
 20              LDA       LENGTH         TEST FOR EOF (LENGTH = 0)
 25              COMP     #0
 30              JEQ       ENDFIL         EXIT IF EOF FOUND
 35             +JSUB      WRREC          WRITE OUTPUT RECORD
 40              J         CLOOP          LOOP
 45    ENDFIL    LDA      =C'EOF'         INSERT END OF FILE MARKER
 50              STA       BUFFER
 55              LDA      #3              SET LENGTH = 3
 60              STA       LENGTH
 65             +JSUB      WRREC          WRITE EOF
 70              J        @RETADR         RETURN TO CALLER
 95    RETADR    RESW      1
100    LENGTH    RESW      1              LENGTH OF RECORD
103              LTORG
105    BUFFER    RESB      4096           4096-BYTE BUFFER AREA
106    BUFEND    EQU       *
107    MAXLEN    EQU       BUFEND-BUFFER

109    RDREC     CSECT
110    .
115    .         SUBROUTINE TO READ RECORD INTO BUFFER
120    .
122              EXTREF    BUFFER,LENGTH,BUFEND
125              CLEAR     X              CLEAR LOOP COUNTER
130              CLEAR     A              CLEAR A TO ZERO
132              CLEAR     S              CLEAR S TO ZERO
133              LDT       MAXLEN
135    RLOOP     TD        INPUT          TEST INPUT DEVICE
140              JEQ       RLOOP          LOOP UNTIL READY
145              RD        INPUT          READ CHARACTER INTO REGISTER A
150              COMPR     A,S            TEST FOR END OF RECORD (X'00')
155              JEQ       EXIT           EXIT LOOP IF EOR
160             +STCH      BUFFER,X       STORE CHARACTER IN BUFFER
165              TIXR      T              LOOP UNLESS MAX LENGTH
170              JLT       RLOOP            HAS BEEN REACHED
175    EXIT     +STX       LENGTH         SAVE RECORD LENGTH
180              RSUB                     RETURN TO CALLER
185    INPUT     BYTE      X'F1'          CODE FOR INPUT DEVICE
190    MAXLEN    WORD      BUFEND-BUFFER

193    WRREC     CSECT
195    .
200    .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
207              EXTREF    LENGTH,BUFFER
210              CLEAR     X              CLEAR LOOP COUNTER
212             +LDT       LENGTH
215    WLOOP     TD       =X'05'          TEST OUTPUT DEVICE
220              JEQ       WLOOP          LOOP UNTIL READY
225             +LDCH      BUFFER,X       GET CHARACTER FROM BUFFER
230              WD       =X'05'          WRITE CHARACTER
235              TIXR      T              LOOP UNTIL ALL CHARACTERS
240              JLT       WLOOP            HAVE BEEN WRITTEN
245              RSUB                     RETURN TO CALLER
255              END       FIRST
```

Figure 2.15 Illustration of control sections and program linking.

In Fig 2.16, there are three control sections.

1) main program → COPY from line 5 to line 107

2) read subroutine → RDREC from line 109 to line 190

3) write subroutine → WRREC from line 193 to 255.

→ Assembler establishes a separate location counter (beginning at 0) for each control section

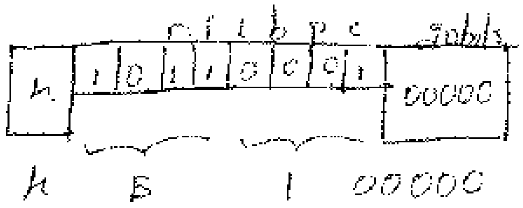
→ Control section names COPY, RDREC, WRREC are not named in EXTDEF because they are automatically considered to be external symbols.

→ Assembler handles the external reference as follows

a) 15 0003 CLOOP $\underbrace{+JSUB}_{48}$ $\underbrace{RDREC}_{EXTREF}$

- Assembler is unaware of RDREC address, so it inserts an address of zero and pass this to loader, which is taken ~~take~~ care during loading.

- The address of RDREC will have no predictable relationship to anything in the control section by name COPY, therefore relative addressing is not possible. Thus an extended format instruction must be ~~p~~ used to provide room for the actual address to be inserted.



b) 160 0017 +STCH BUFFER, X 57900000

→ BUFFER is used in RDREL control section but defined in COPY control section.

c) 190 0028 MAXLEN WORD BUFEND - BUFFER 000000

→ two external references in the expression BUFEND and BUFFER in WDREC section.

→ The assembler inserts an address of 300, it passes information to the loader to add to this data area the address of BUFEND and subtract from this data area the address of BUFFER, which results in the desired value.

d) 107 1000 MAXLEN EQU BUFEND - BUFFER

→ Both expressions (107 & 190) looks same but the difference is here BUFEND and BUFFER are defined and used in the same control section. so value can be calculated immediately.

MAXLEN EQU 1033 - 0033

= 1000

→ A reference to MAXLEN in the COPY control section will use the definition on line 107, whereas a reference to MAXLEN in RDREC control section will use the definition on line 190.

→ object program

- Along with header record, text record, modification record, two more records are added

a) Define Record → information of symbols defined in this control section

col. 1	D
col. 2-7	name of external symbol defined in this control section
col. 8-13	Relative address within this control section (Hexadecimal)
col. 14-73	Repeat information in col. 2-13 for other external symbol

b) Refer Record → symbols that are used as external references in this control section

col. 1	R
col 2-7	name of external symbol refered in this control section
col. 8-73	name of other external reference symbol

c) modification Record

col. 1	M
col 2-7	starting address of the field modified, in half-bytes (hexadecimal
col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field

COPY

```
H^COPY    ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032025^290000^332007^4B100000^3F2FEC^
                                          032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000
```

RDREC

```
H^RDREC   ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^
                                                       57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER
E
```

WRREC

```
H^WRREC   ^000000^00001C
R^LENGTH^ BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^
                                            ^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

2.4 Assembler Design options

. we will learn two alternatives of the standard two-pass assembler.

a) One-pass Assemblers

b) Multi-pass Assemblers

a) <u>One-pass Assemblers</u>

→ As we know already, assembling the forward reference is very difficult ∵ we don't know the address. This can be eliminated very easily for data items. That is data items are defined in the source program before they are referenced. (Ex. ~~Assign~~ declaration of data variable in c).

→ It is not the same for labels on instructions.

Ex: If the program has a forward jump ie escaping from a loop after taking some condition → here we can't define before itself. Therefore the assembler has to provide the way to handle forward reference.

→ Two type of one-pass assemblers.

(i) load-go assembler : Assembler produces object program code directly in memory for immediate execution.

→ Here object program is not written out and no loader is needed.

→ Application : program development and testing.

→ Ex :- A university computing system for student. Here a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

→ load-and-go assembler avoids the overhead of writing the object program out (secondary storage) and reading it back in. ∴ forward references can be handled easily.

→ Avoids usage of forward references

→ Used on systems where external working-storing devices are either slow or not available.

```
Line   Loc    Source statement                         Object code

  0    1000   COPY     START   1000
  1    1000   EOF      BYTE    C'EOF'                  454F46
  2    1003   THREE    WORD    3                       000003
  3    1006   ZERO     WORD    0                       000000
  4    1009   RETADR   RESW    1
  5    100C   LENGTH   RESW    1
  6    100F   BUFFER   RESB    4096
  9           .
 10    200F   FIRST    STL     RETADR                  141009
 15    2012   CLOOP    JSUB    RDREC                   48203D
 20    2015            LDA     LENGTH                  00100C
 25    2018            COMP    ZERO                    281006
 30    201B            JEQ     ENDFIL                  302024
 35    201E            JSUB    WRREC                   482062
 40    2021            J       CLOOP                   302012
 45    2024   ENDFIL   LDA     EOF                     001000
 50    2027            STA     BUFFER                  0C100F
 55    202A            LDA     THREE                   001003
 60    202D            STA     LENGTH                  0C100C
 65    2030            JSUB    WRREC                   482062
 70    2033            LDL     RETADR                  081009
 75    2036            RSUB                            4C0000
110           .
115           .        SUBROUTINE TO READ RECORD INTO BUFFER
120           .
121    2039   INPUT    BYTE    X'F1'                   F1
122    203A   MAXLEN   WORD    4096                    001000
124           .
125    203D   RDREC    LDX     ZERO                    041006
130    2040            LDA     ZERO                    001006
135    2043   RLOOP    TD      INPUT                   E02039
140    2046            JEQ     RLOOP                   302043
145    2049            RD      INPUT                   D82039
150    204C            COMP    ZERO                    281006
155    204F            JEQ     EXIT                    30205B
160    2052            STCH    BUFFER,X                54900F
165    2055            TIX     MAXLEN                  2C203A
170    2058            JLT     RLOOP                   382043
175    205B   EXIT     STX     LENGTH                  10100C
180    205E            RSUB                            4C0000
195           .
200           .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205           .
206    2061   OUTPUT   BYTE    X'05'                   05
207           .
210    2062   WRREC    LDX     ZERO                    041006
215    2065   WLOOP    TD      OUTPUT                  E02061
220    2068            JEQ     WLOOP                   302065
225    206B            LDCH    BUFFER,X                50900F
230    206E            WD      OUTPUT                  DC2061
235    2071            TIX     LENGTH                  2C100C
240    2074            JLT     WLOOP                   382065
245    2077            RSUB                            4C0000
255                    END     FIRST
```

Figure 2.18 Sample program for a one-pass assembler.

ii) Assembler generating object codes.
Working process of load go assembler

↳ Fig 2.18 shows an example for one-pass assembler

↳ Here all data item definitions are placed ahead of the code that references them.

```
Ex:   01   1000   EOF      BYTE   C'EOF'
      02   1003   THREE    WORD   3
      03   1006   ZERO     WORD   0
                    :
      06   100F   BUFFER   RESB   4096.
```

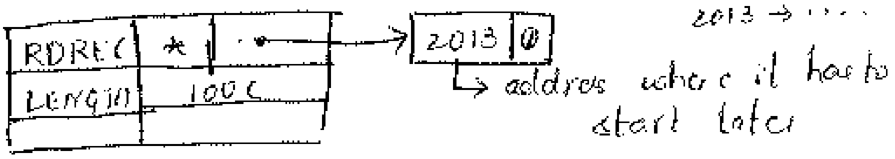
↳ The assembler generates object code as it scans the source program

↳ If an instruction operand is a symbol that has not yet been defined (forward reference), the operand address is ommitted during assembly.

↳ The symbol is entered into symbol table if not exists along with a flag indicating undefined.

** ↳ The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.

```
Ex:   15   2012   CLOOP   JSUB   RDREC   480000
                                          2012 → 48
                                          2013 → ....
```



↳ when the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if exists) and the proper address is inserted into any instructions previously generated.

↳ Fig 2.19 shows the object code and symbol table entries as they were scanned. till line no.40

↳ 15 2012 CLOOP JSUB RDREC.
~~undefined~~
↳ undefined

∴ symbol table entry is RDREC | * | → 2013 | ∅

2013 is the address location where it has to load once found further.

↳ Same for line noe 30, 35.

memory Address	Contents			
1000	H5HFH600	00030000	00XXXXXX	XXXXXXXX
1010	XXXXXXXX	XXXXXXX	XXXXXXXX	XXXXXXXX
⋮				
2000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXXh
2010	1009H8--	--00100C	28100630	----H8--
2020	--302012			
⋮				

SYMBOL TABLE

LENGTH	100C	
RDREC	*	→ 2013 | ∅
THREE	1003	
ZERO	1006	
WRREC	*	→ 201F | ∅
EOF	1000	
ENDFIL	*	→ 201C | ∅
RETADR	1009	
BUFFER	100F	
CLOOP	2012	
FIRST	200F	

Fig 2.19: Object code in memory and symbol table entries after scanning line 40

```
Memory
Address      Contents
1000     454F4600   00030000   00XXXXXX   XXXXXXXX
1010     XXXXXXXX   XXXXXXXX   XXXXXXXX   XXXXXXXX
 .
 .
 .
2000     XXXXXXXX   XXXXXXXX   XXXXXXXX   XXXXXX14
2010     10094820   3D00100C   28100630   202448--
2020     --3C2012   0010000C   100F0010   030C100C
2030     48----08   10094C00   00F10010   0004100C
2040     00100660   20393020   43D82039   28100630
2050     ----5490   0F
```

SYMBOL TABLE

LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	* → 201F → 2031 ∅
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	* → 2050 ∅
RLOOP	2043

Fig: object code in memory and symbol table entries after scanning line 160 of fig 9.18

```
1000     454F4600   00030000   00XXXXXX   XXXXXXXX
1010     XXXXXXXX   XXXXXXXX   XXXXXXXX   XXXXXXXX
 .
 .
 .
2000     XXXXXXXX   XXXXXXXX   XXXXXXXX   XXXXXX14
2010     10094820   3D00100C   28100630   20244820
2020     633C2012   0010000C   100F0010   030C100C
2030     48206208   10094C00   00F10010   00041006
2040     001006E0   20393020   43D82039   28100630
2050     205B5490   0F2C203A   38204310   100C4100
2060     00050410   06E02061   30206550   900FDC20
2070     612C100C   3820654C   0000
2080
```

LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	2062
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	205B
RLOOP	2043
OUTPUT	2061
WLOOP	2065

Fig: complete object program in memory and symbol table entries note : If any symbols in the SYMTAB are still marked with *. should be flagged by the assembler as errors. (Undefined symbol error in C long after compiling completely)

↳ The assembler searches SYMTAB for the value of the symbol named END statement and jumps to this location to begin execution of the assembled program.

↳ In load-and-go assembler, the actual address must be known at assembly time.

54626

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND} as starting address
      initialize LOCCTR as starting address
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
      if there is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin
              search SYMTAB for LABEL
                if found then
              begin
                  if symbol value as null
                  set symbol value as LOCCTR and search
                    the linked list with the corresponding
                    operand
                  PTR addresses and generate operand
                    addresses as corresponding symbol
                    values
                  set symbol value as LOCCTR in symbol
                    table and delete the linked list
              end
              else
                insert (LABEL, LOCCTR) into SYMTAB
            end
              search OPTAB for OPCODE
                if found then
                  begin
                    search SYMTAB for OPERAND address
                if found then
                  if symbol value not equal to null then
                    store symbol value as OPERAND address
                  else
                    insert at the end of the linked list
                      with a node with address as LOCCTR
                  else
                    insert (symbol name, null)
```

Figure 2.19(c) Algorithm for One pass assembler.

```
              add 3 to LOCCTR
            end
            else if OPCODE = 'WORD' then
              add 3 to LOCCTR & convert comment to
                object code
            else if OPCODE = 'RESW' then
              add 3 #[OPERAND] to LOCCTR
            else if OPCODE = 'RESB' then
              add #[OPERAND] to LOCCTR
            else if OPCODE = 'BYTE' then
              begin
                find length of constant in bytes
                add length to LOCCTR
                convert constant to object code
              end
          if object code will not fit into current
            text record then
            begin
              write text record to object program
              initialize new text record
            end
          add object code to Text record
        end
      write listing line
      read next input line
    end
  write last Text record to object program
  write End record to object program
  write last listing line
end {Pass 1}
```

Figure 2.19(c) *(cont'd)*

references that could not be handled by the assembler. Of course, the objec

ii) Assembler generating object code.

↳ This type of one-pass assembler also works in the same manner as before (load-and-go) except where the definition of symbol is encountered

↳ when a symbol definition is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. ⟹ means they have already been written out as part of a Text Record in the object program.

↳ In such situations, assembler generates another Text record with the correct operand address.

↳ when the program is loaded, this address will be inserted into the instructions by the loader.

↳ The object program for fig 2.18 is shown below during one pass-one.

```
  H^COPY  ^001000^00107A
1 T^001000^09^454F46^000003^000000
  T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
2 T^00201C^02^2024
3 T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^
4                                              E^001000

5 T^002013^02^203D
```

6 T^00203D^1E^041006^001006^ E02039^302043^D82039^281006^
300000^ 549000F^2C203A^382043

7 T^00205D^02^205B

8 T^00205B^07^101000^4C0000^05

9 T^00201F^02^2062

10 T^002031^02^2062

11 T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^
38 2065^ 4C0000

E^00200F

- The second text record contains the object code generated from lines 10 through 40 in fig 2.18.
- The operand addresses for instructions on line 15, 30 and 45 has been generated as 0000.
- when definition of ENDFIL on line 45 is encountered, the assembler generates the third Text Record. It indicate that the value 2024 (address of ENDFIL) has to be loaded at location 201C.
- This continues for all the forward references encountered.

b) multi-pass Assemblers

↳ we know that whenever we use 'EQU' assembly directive, any symbol used on the right-hand side should be defined previously in the source program. This is not true always.

↳ Ex: ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1

↳ As we see above, we have multiple forward references ie Alpha depends on value of Beta, Beta depends on value of delta.

↳ Any assembler that make only two sequential passes over the source program cannot resolve such a sequential sequence of definitions.

↳ To overcome this we go for multi-pass assembler which makes as many passes as needed to process the definitions of symbols.

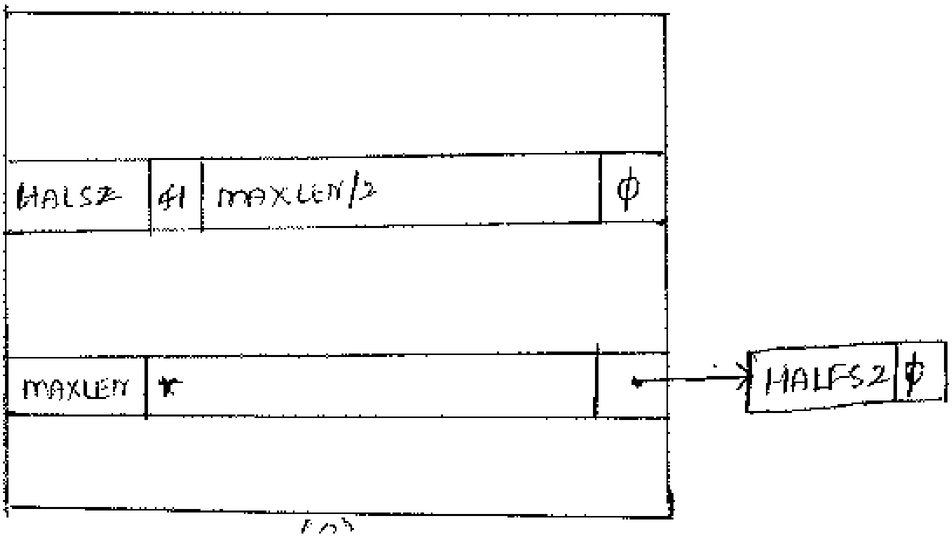
↳ It is not necessary for multi-pass assembler to make more than two passes over the entire program.

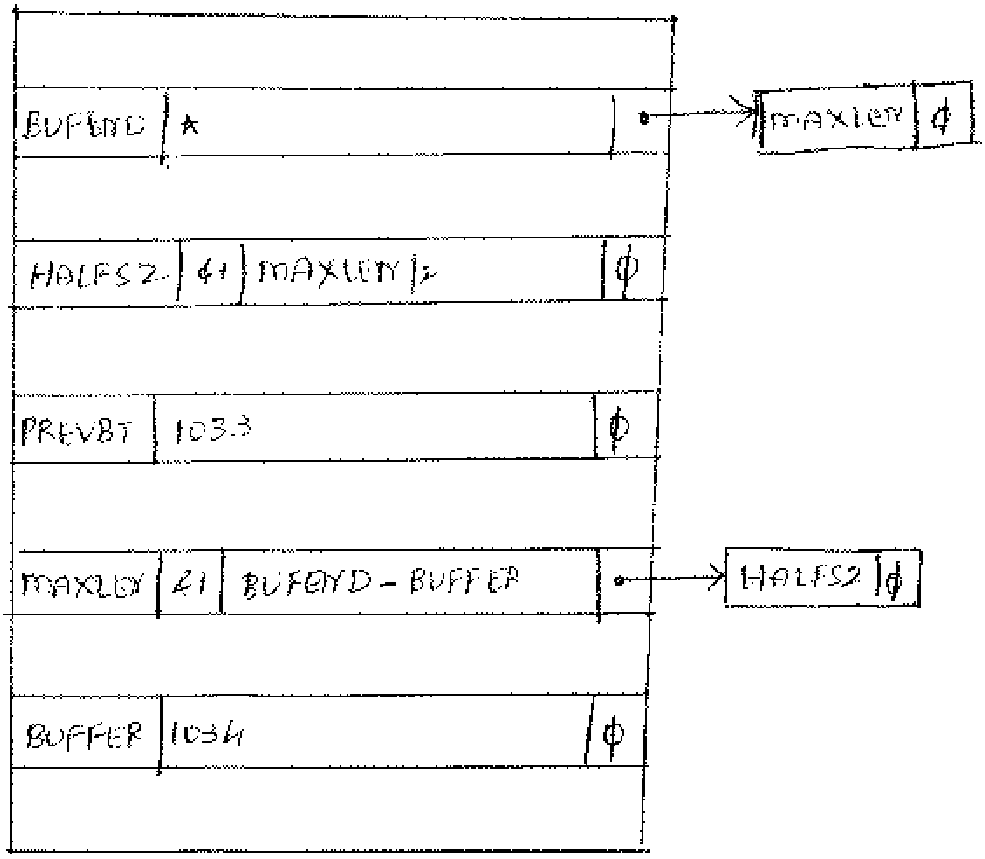
** ↳ Instead, the portions of the program that involve forward reference in symbol definitions are saved during pass-1. Additional passes through these stored definitions are made as the assembly progress. This process is followed by a normal Pass-2

↳ SYMTAB stores the symbol definition, symbols which are dependent on this, no of symbols dependent on this symbol

```
            100
Ex:-  1          HALFSZ     EQU     MAXLEN/2

      2          MAXLEN     EQU     BUFEND - BUFFER

      3          PREVBT     EQU     BUFFER -1

      4   1034   BUFFER     RESB    4096 ⇒ (1000)16

      5   2034   BUFEND     EQU     * - indicate the current
                                        location counter value
```

→ below fig shows the symbol table entry when it reads line no. 1 indicating that Halfsz depends on maxlen value





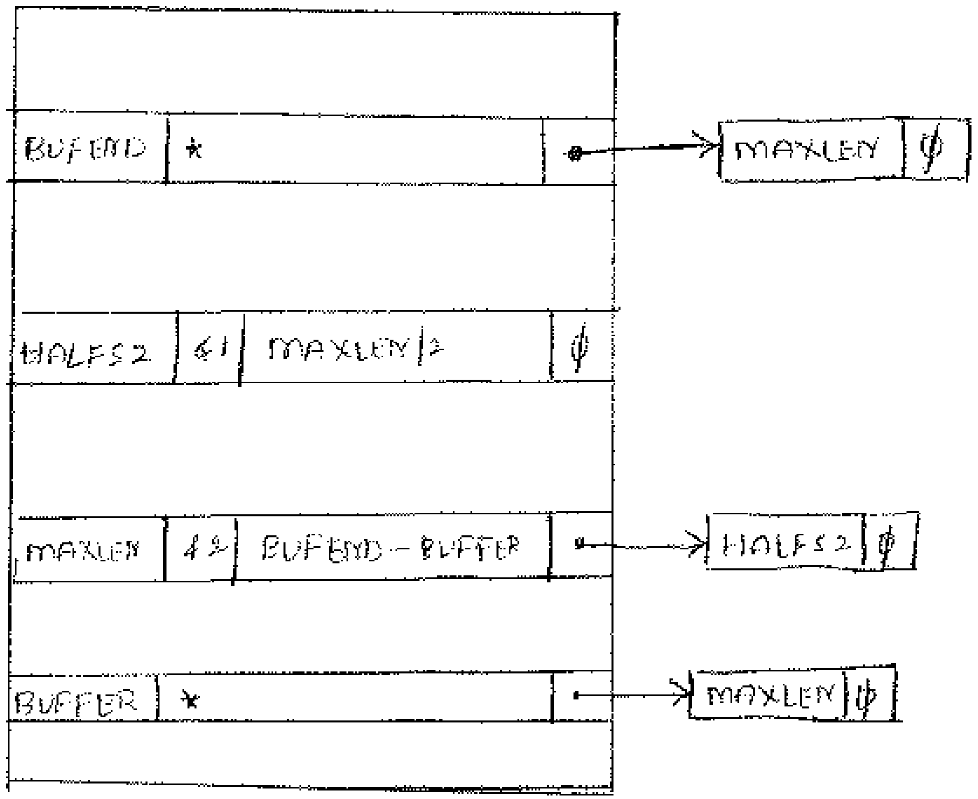
(d)

↳ fig (d) shows the symbol table entry after scanning line no. (h). whose location counter value is 1034. MAXLEN dependency value 42 is reduced to 41.

BUFEND	2034	ϕ
HALFSZ	800	ϕ
PREVBT	1033	ϕ
MAXLEN	1000	ϕ
BUFFER	1034	ϕ

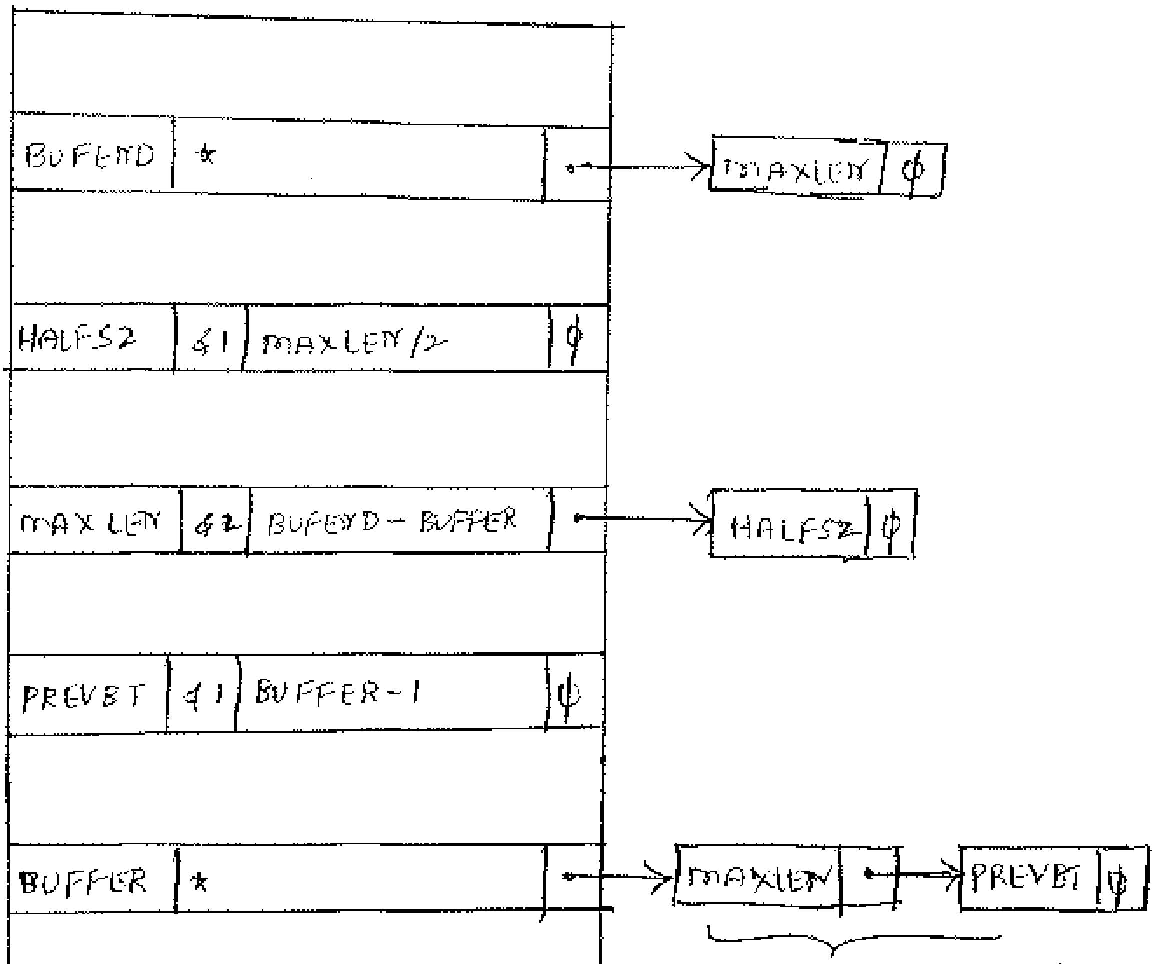
(e)

↳ fig (e) indicates the complete symbol table entry procu



(b)

→ fig (b) shows symbol table entry after reading line no 3
As we see, MAXLEN depends on 2 symbols BUFEND & BUFFUR . ∴ MAXLEN &2.


BUFEND | * | •
MAXLEN | φ
HALFS2 | &1 | MAXLEN/2 | φ
MAX LEN | &2 | BUFEND - BUFFER | •
HALFS2 | φ
PREVBT | &1 | BUFFER-1 | φ
BUFFER | * | •
MAXLEN | •
PREVBT | φ

(c)

These two are dependent on value of Buffer.

CHAPTER 4

Macro Processors

Chapter 4 : Macroprocessor

→ we are going to study definition of macro

→ what is the need for macro

→ Datastructure used in macro invocation and expansion

Macro : It is a single instruction that expands automatically into a set of instructions to perform a particular task. Thus macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macroprocessor.

Ex: In SIC/XE, 7 instructions (STA, STB, etc) is required to save the contents of all registers before calling a subprogram, but by using a macro instruction, the programmer can write a single start instruction like SAVEREGS. The SAVEREGS macro instruction would be expanded into seven instruction required to save the contents of all registers

↳ LOADREGS macro instruction would be used to reload the register contents after returning from the subprogram

* → Macro processor performs no analysis of the text it handles and is not concerned about the meaning of the involved statements during macro expansion. ∴ The design of a macro processor is machine independent.

2.1 Basic macro processor Functions.

→ Macros refers to a set of statements which will replace every invocation to it. The two concepts associated with macros are :

(i) Macro Definition
(ii) Macro Expansion

(i) Macro definition :

→ Consists of macro prototype, one or more module and macro preprocessor.

→ macro definition is a set of statements present inbetween a macro header statement (MACRO) and a macro end statement (MEND). MACRO and MEND are two assembly directives used in macro definition

* Syntax of macro prototype.

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

where <macroname> : The mnemonic field of a statement

<formal parameter>: < parmeter name>[< parameter kind>]

each parameter begins with '&'

Syntax: macro call

<macro name> [<actual parameter specification> [,..]]

→ In general, macro definition is given as

~~NAME MACROPARAMETERS~~

NAME MACRO PARAMETERS

:

body; the statements which are generated as the expansion of the macros

:

MEND

// macro invocation is as

:

NAME PARAMETERS

:

→ As in fig 4.1, macro definition is at line 10

10 RDBUFF MACRO &INDEV, &BUFADR, &RECLTH

:

95 MEND

→ macro invocation in fig 4.1

190 RDBUFF F1, BUFFER, LENGTH

(i) Macro Expansion

→ Macro ~~expansion~~ invocation statements are the statements of the macro body that are expanded each time the macro is invoked.

→ The program in fig 4.1 is supplied as input to a macro processor.

→ Fig 4.2 shows the output that would be generated

→ In expanding the macro invocation on line 190, argument F1 is substituted for the parameter &INDEV, BUFFER is substituted for &BUFADR, LENGTH is substituted for &RECLTH.

→ line 190a through 190m show the complete expansion of the macro invocation on line 190.

→ Same in line 210a through 210h for WRBUF MACRO

→ As we see the macro body does have any labels

∵ for example, line 140 → JEQ *-3 and line 155 JLT *+14. If we put label, it would be generated twice on line 210d and 220d resulting in an error (duplicate label definition) when the program is assembled.

→ *-3 , *+9 , ... indicate PC-relative addressing

```
Line           Source statement
       LABEL     OPCODE    OPERAND

  5    COPY      START     0                   COPY FILE FROM INPUT TO OUTPUT
 10    RDBUFF    MACRO     &INDEV,&BUFADR,&RECLTH
 15    .
 20    .         MACRO TO READ RECORD INTO BUFFER
 25    .
 30              CLEAR     X                   CLEAR LOOP COUNTER
 35              CLEAR     A
 40              CLEAR     S
 45              +LDT      #4096               SET MAXIMUM RECORD LENGTH
 50              TD        =X'&INDEV'          TEST INPUT DEVICE
 55              JEQ       *-3                 LOOP UNTIL READY
 60              RD        =X'&INDEV'          READ CHARACTER INTO REG A
 65              COMPR     A,S                 TEST FOR END OF RECORD
 70              JEQ       *+11                EXIT LOOP IF EOR
 75              STCH      &BUFADR,X           STORE CHARACTER IN BUFFER
 80              TIXR      T                   LOOP UNLESS MAXIMUM LENGTH
 85              JLT       *-19                  HAS BEEN REACHED
 90              STX       &RECLTH             SAVE RECORD LENGTH
 95              MEND
100    WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
105    .
110    .         MACRO TO WRITE RECORD FROM BUFFER
115    .
120              CLEAR     X                   CLEAR LOOP COUNTER
125              LDT       &RECLTH
130              LDCH      &BUFADR,X           GET CHARACTER FROM BUFFER
135              TD        =X'&OUTDEV'         TEST OUTPUT DEVICE
140              JEQ       *-3                 LOOP UNTIL READY
145              WD        =X'&OUTDEV'         WRITE CHARACTER
150              TIXR      T                   LOOP UNTIL ALL CHARACTERS
155              JLT       *-14                  HAVE BEEN WRITTEN
160              MEND
165    .
170    .         MAIN PROGRAM
175    .
180    FIRST     STL       RETADR              SAVE RETURN ADDRESS
190    CLOOP     RDBUFF    F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
195              LDA       LENGTH              TEST FOR END OF FILE
200              COMP      #0
205              JEQ       ENDFIL              EXIT IF EOF FOUND
210              WRBUFF    05,BUFFER,LENGTH    WRITE OUTPUT RECORD
215              J         CLOOP               LOOP
220    ENDFIL    WRBUFF    05,EOF,THREE        INSERT EOF MARKER
225              J         @RETADR
230    EOF       BYTE      C'EOF'
235    THREE     WORD      3
240    RETADR    RESW      1
245    LENGTH    RESW      1                   LENGTH OF RECORD
250    BUFFER    RESB      4096                4096-BYTE BUFFER AREA
255              END       FIRST
```

Figure 4.1 Use of macros in a SIC/XE program.

```
Line           Source statement

  5    COPY      START     0                   COPY FILE FROM INPUT TO OUTPUT
180    FIRST     STL       RETADR              SAVE RETURN ADDRESS
190    .CLOOP    RDBUFF    F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
190a   CLOOP     CLEAR     X                   CLEAR LOOP COUNTER
190b             CLEAR     A
190c             CLEAR     S
190d             +LDT      #4096               SET MAXIMUM RECORD LENGTH
190e             TD        =X'F1'              TEST INPUT DEVICE
190f             JEQ       *-3                 LOOP UNTIL READY
190g             RD        =X'F1'              READ CHARACTER INTO REG A
190h             COMPR     A,S                 TEST FOR END OF RECORD
190i             JEQ       *+11                EXIT LOOP IF EOR
190j             STCH      BUFFER,X            STORE CHARACTER IN BUFFER
190k             TIXR      T                   LOOP UNLESS MAXIMUM LENGTH
190l             JLT       *-19                  HAS BEEN REACHED
190m             STX       LENGTH              SAVE RECORD LENGTH
195              LDA       LENGTH              TEST FOR END OF FILE
200              COMP      #0
205              JEQ       ENDFIL              EXIT IF EOF FOUND
210              WRBUFF    05,BUFFER,LENGTH    WRITE OUTPUT RECORD
210a             CLEAR     X                   CLEAR LOOP COUNTER
210b             LDT       LENGTH
210c             LDCH      BUFFER,X            GET CHARACTER FROM BUFFER
210d             TD        =X'05'              TEST OUTPUT DEVICE
210e             JEQ       *-3                 LOOP UNTIL READY
210f             WD        =X'05'              WRITE CHARACTER
210g             TIXR      T                   LOOP UNTIL ALL CHARACTERS
210h             JLT       *-14                  HAVE BEEN WRITTEN
215              J         CLOOP               LOOP
220    .ENDFIL   WRBUFF    05,EOF,THREE        INSERT EOF MARKER
220a   ENDFIL    CLEAR     X                   CLEAR LOOP COUNTER
220b             LDT       THREE
220c             LDCH      EOF,X               GET CHARACTER FROM BUFFER
220d             TD        =X'05'              TEST OUTPUT DEVICE
220e             JEQ       *-3                 LOOP UNTIL READY
220f             WD        =X'05'              WRITE CHARACTER
220g             TIXR      T                   LOOP UNTIL ALL CHARACTERS
220h             JLT       *-14                  HAVE BEEN WRITTEN
225              J         @RETADR
230    EOF       BYTE      C'EOF'
235    THREE     WORD      3
240    RETADR    RESW      1
245    LENGTH    RESW      1                   LENGTH OF RECORD
250    BUFFER    RESB      4096                4096-BYTE BUFFER AREA
255              END       FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

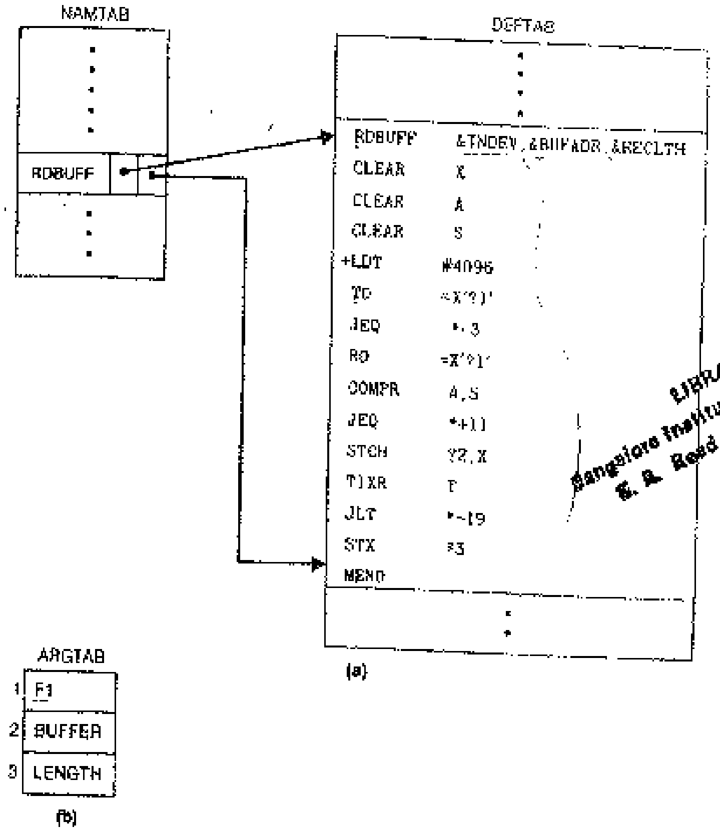
```
1  MACROS   MACRO    {Defines SIC standard version macros}
2  RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                .
                .    {SIC  standard version}
                .
3           MEND     {End of RDBUFF}
4  WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                .
                .    {SIC standard version}
                .
5           MEND     {End of WRBUFF}
                .
                .
                .
6           MEND     {End of MACROS}
```

(a)

```
1  MACROX   MACRO    {Defines  SIC/XE macros}
2  RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                .
                .    {SIC/XE version}
                .
3           MEND     {End of RDBUFF}
4  WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                .
                .    {SIC/XE version}
                .
5           MEND     {End of WRBUFF}
                .
                .
                .
6           MEND     {End of MACROX}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.



LIBRARY
Bangalore Institute of Technology
K. R. Road, Bangalore-4.

Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

4.4.2 Macro Processor Algorithm and data structures

→ For designing two pass macro processor, all macro definitions are processed during pass-1 and all macro invocation statements are expanded during pass-2.

→ The two pass macro processor would not allow the body of one macro instruction to contain definition of other macros, ∵ all macros defined during the pass before any macro invocation were expanded.

→ Example of recursive macro definition is shown in Fig 4.3 (a) for SIC machine and Fig 4.3 (b) for SIC/XE machine

→ The same program can be run on either a SIC machine or SIC/XE machine. Invocation of MACROS or MACROX is only changed for use.

→ A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition provided that a main definition of a macro should appear before the invocation.

Introduction to one pass macro processor (Fig 4.4)

→ There are three main data structures involved

(i) Definition table (DEFTAB)

↳ It stores the macro definitions which contains the macro prototype and the statements that make up the macro body

↳ Comment lines are omitted ∵ they are not part of the macro expansion

↳ References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

(ii) Name Table (NAMTAB)

↳ it stores the macro names, which serves as index to DEFTAB.

↳ For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in ~~DEFTAB~~ DEFTAB.

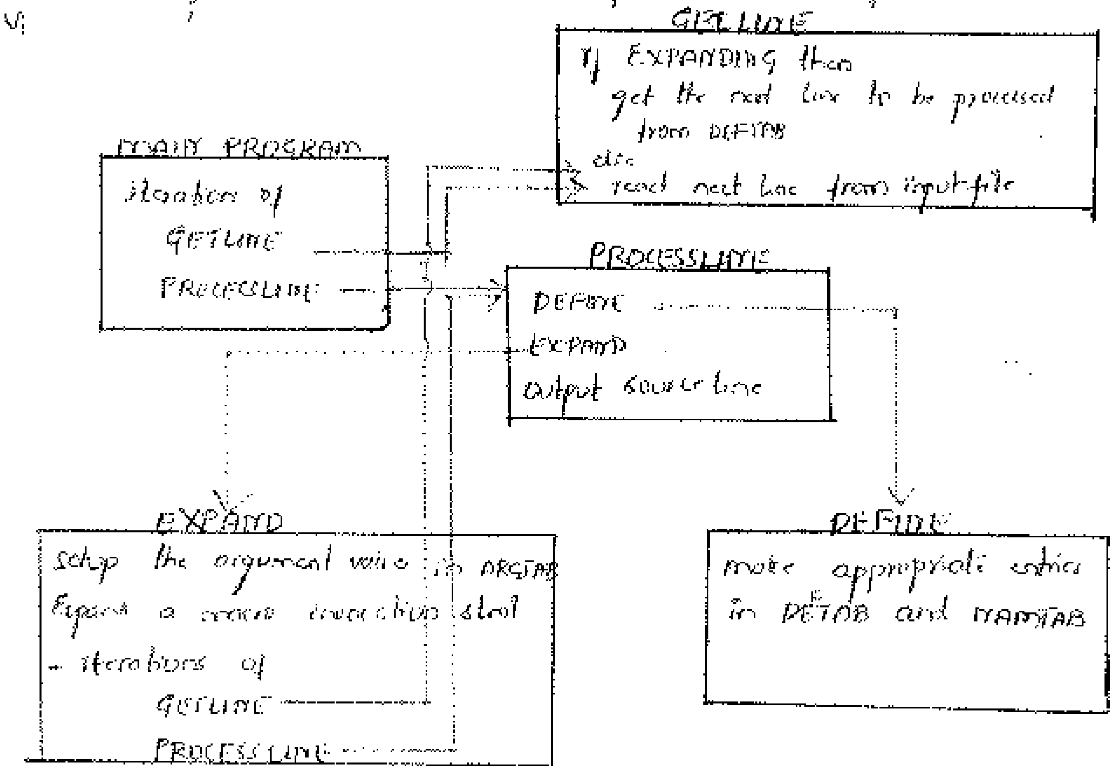
(iii) Argument Table (ARGTAB)

↳ used during the expansion of macro invocations.

↳ when a macro invocation stmt is recognized, the arguments are stored in ARGTAB according to their position in the argument list.

↳ when it is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.

Algorithm for a one-pass macro processor



```
begin (macro processor)
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}
```

Figure 4.5 Algorithm for a one-pass macro processor.

```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL := 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}


procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}


procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

Figure 4.5 (cont'd)

Handling nested macro definition within macros

→ In DEFTAB (define procedure), when a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached. This will not work for nested macro defn : the first MEND encountered in the inner macro will terminate the whole macro definition process

→ To solve this problem, a DEFINE procedure is used which maintains a counter named LEVEL. The LEVEL value ie incremented by 1 when MACRO directive is read. The value is determined by 1 when MEND directive is read. when LEVEL value becomes 0, the MEND that corresponds to the original MACRO directive has been found. This process is very much like matching left and right parenthesis when scanning an arithmetic expression.

LOADERS AND LINKERS.

- 3 processes a system program performs →

1. Loading – bringing the object program into memory for execution

2. Relocation – modify the object program so that it can be located at a different location from the original one

3. Linking – combing 2 or more seperate object ~~and~~ programs and supplying information needed to allow reference b/w them.

Loaders – system program that perform loading function.
– Can also support linking & relocating.

Linker – seperate system program for linking operation.

LOADERS 3.1

→ Basic Loader Function or fundamental

- The most basic loader function is → bringing object program into memory and starting execution.

- Absolute Loader

  It is the most basic loader that just performs loading function.

  It performs all its functions in a single pass.

  - It checks Header record to verify that correct program is being loaded and that it ~~is~~ will fit in the memory space available
  - It reads each Text record and the object code is moved to its corresponding memory location
  - The End record ~~poo~~ indicates end of object code and gives address of location from where execution starts.

Scanned by CamScanner

ALGORITHM :: Absolute loader

```
begin
    read Header record
    verify program name and length.
    read first Text record
    While record type ≠ `E' do
        begin
            {if object code is in character form, convert i
             internal representation}
            move object code to specified location in memoy
            read next object program record.
        end
    jump to address specified in End record
end.
```

<u>Note</u> →.

In the object program, each byte of assembled code is given usig hexadecimal representation in character form.

eg - OPcode for STL → 14

It is represented usig pair of characte '1' & '4'

So, when loader reads this, they occupy 2 bytes of memory. But, in the instruction loaded for execution ~~th~~ it is to be stored as 1 byte represented by hexadecimal 14.

⇒ Each pair of byte from object prog record must be packed together into 1 byte dury loadiy.

This method of representation is in<u>sufficient</u>

So, object program can be stored in binary form → each byte of object code stored in 1 byte of memory but they aren't easy to read for humans.

- Simple Bootstrap Loader

– Special absolute loader that is ~~load~~ executed when computer is first started or restarted.

– It loads the 1st program to be run on the computer, ie OS.

```
BOOT    START   0           BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND
. ENTERS IT INTO MEMORY LOCATION STARTING FROM
. ADDRESS 80h. AFTER LOADING IS COMPLETE ~~CONTROL~~ JUMP
. TO 80h IS EXECUTED TO BEGIN EXECUTION OF PROGRAM.
. REGISTER X CONTAINS NEXT ADDRESS TO BE LOADED
        CLEAR   A           CLEAR REG A TO 0
        LDX     #128        INITIALIZE REG X TO 80h
LOOP    JSUB    GETC        READ HEX DIGIT FROM PROG
        RMO     A,S         SAVE IN REG S.
        SHIFTL  S,4         MOVE TO HIGH-ORDER 4 BITS
        JSUB    GETC        GET NEXT HEX DIGIT
        ADDR    S,A         COMBINE DIGITS TO 1 BYTE
        STCH    0,X         STORE AT ADDR. IN X.
        TIXR    X,X         ADD 1 TO MEMORY ADDRESS
        J       LOOP        LOOP TILL EOF REACHED.
.
. SUBROUTINE TO READ FROM DEVICE AND CONVERT IT
. FROM ASCII TO HEXA DIGIT VALUE AND RETURN IT
. TO REG A. IF EOF ENCOUNTERED, CONTROL TRANSFERRED
. TO 80h
.
GETC    TD      INPUT       TEST INPUT DEVICE
        JEQ     GETC        LOOP UNTIL READY
        RD      INPUT       READ CHARACTER
        COMP    #4          IF CHAR IS 04h (EOF)
        JEQ     80          JUMP TO START OF PROG LOADED
        COMP    #48         COMP TO 30h ('0')
        JLT     GETC        SKIP CHAR <'0'
        SUB     #48         SUBTRACT 30h FROM ASCII
        COMP    #10         FOR 'A' TO 'F', RESULT <10 THEN
        JLT     RETURN      CONVERSION COMPLETE ELSE.
        SUB     #7          SUBTRACT 7 MORE.
RETURN  RSUB                RETURN TO CALLER
INPUT   BYTE    X'F1'       INPUT DEVICE
        END     LOOP
```

Scanned by CamScanner

- The bootstrap begins at address 0. [Line 0]
- It loads the OS startig at address 80h
  by initializig register X (the pointer) to 80 h
  [LINE 8]
- As this is the 1st prog to be loaded, its
  loadig is simple.

  The object program from device F1 is
  - represented as 2 hexadecimal digit for 1byte
  - has no Header or End record or any
    Other control information.

  Here, the object code is loaded into consecutive
  bytes of memory starty at 80h.

- Subroutine GETC →
  It reads 1 char. from device F1 and
  converts it from ASCII to the hexadecimal
  digit it represented.
  When it encounters EOF, the control moves
  to 80h (i.e start of loaded program)

So in the program,
the main loop keeps track of the next memory
location for loadiy and reads ~~the~~ 2 characters &
stores it as 1byte.
The subroutine reads the character and converts
it from ASCII to represented hex value.

3.2
→ Disadvantage of absolute loader.
- The program ~~has~~ needs absolute memory location for
  loadiy to be specified by the programmer.
  But in larger & advanced machine, multiple
  independent program run together sharing memory.
  Here predefiny memory for loady is impossible.
- The subroutines of libraries aren't used efficiently.
  For efficient use only required subroutines should
  be loaded but this isn't possible with
  absolute addresses

MACHINE DEPENDENT LOADER FEATURES

Scanned by CamScanner

→ In most modern computer, the loaders also perform the relocation and linking function, in addition to the basic loading function.

- Relocation.
  - Loaders that allow relocation are called relocating loader or relative loader.
  - Methods for specifying relocation as part of object program

(i). Modification record is used to describe each part of object code that must be changed when program relocates

And the instructions whose value is affected by relocation are ones that use extended format.

The modification record specify the starting address & length of fields to be altered. It then describes the modification to be performed

But, this method isn't suited for all machines

eg- In a SIC machine, there is no relative addressing & so, almost all instructions need to be modified during relocation. This leads to a lot of Modification record that dramatically increases object code size.

(ii) There is a relocation bit associated with ~~Text Record~~ each word of object code in Text ~~Record~~

It is primarily used in machines that use direct addressing & fixed instruction format

In SIC machine, each instruction occupies 1 word, i.e one relocation bit per instruction

The relocation bits gathered together to a bit mask which is present in the Text Record following the ~~record~~ length indicator

eg- T^001057,0A^800^100036^4C0000^F1^001000

If relocation bit correspondiy to a word is
– 1 → modification required
prog's starty addr is to be added to this word duiy relocation

– 0 → no modification required.

If Text record has fewa than 12 words, then ~~correspondiy~~ for the unused words corresponding ~~word~~ relocation bit = 0.

eg – FFC (1111 1111 1100)
First 10 words need to be modified.

~~iii) Some~~
iii) Some computa have hardwae relocation capability that eliminats need of loada to relocato program.

The SIC/XE machine usually use the Modification record scheme for relocation.

ALGORITHM: SIC/XE relocation loader

```
begin.
    get PROGADDR from operaty system
    while not end of input do
        begin
            read next record
            while record type ≠ 'E' do
                begin
                    read next input record
                    while record type = 'T' then
                        begin.
                            more object code from a
                            record to location ADDR
                            + specified address
                        end
                    while record type = 'M'
                        add PROGADDR at location
                        PROGADDR + specified address
                end.
        end
end.
```

Scanned by CamScanner

The SIC machines usually use the modification bit scheme. ~~as~~

ALGORITHM: SIC reloaction loader algorithm.

```
begin
    get PROGADDR from operating system
        while not end of input do
            begin
                read next record
                while ~~end~~ record type ≠ 'E' do
                while record type = 'T'
                    begin
                        get length = second data.
                        mask bits(M) as third data.

                        for ( i=0, i<length, i++)
                            if $M_i$ = 1 then
                                add PROGADDR at the
                                location PROGADDR + specified
                                address.
                            else
                                move object code from record
                                to location PROGADDR +
                                specified address
                        read next record.
                    end
            end
end.
```

- Program Linking

  - ~~In~~ programs made up of multiple control sections can be assembled in 2 ways
    - all control sections together i.e in same invocation of assembly
    - each indepently.

  In both case they will appear as seperate segments of object code after assembly

  Assembler sees code only as control sections that are to be loaded, relocated & linked. It doesn't need to know which control sections

Scanned by CamScanner

were assembled at same time

Consider 3 programs each containing single control section :

```
Loc              Source statement                           Object code

0000    PROGA    START    0
                 EXTDEF   LISTA.ENDA
                 EXTREF   LISTB,ENDB,LISTC.ENDC
                  .
                  .
                  .
0020    REF1     LDA      LISTA                             03201D
0023    REF2    +LDT      LISTB+4                           77100004
0027    REF3     LDX     #ENDA-LISTA                        050014
                  .
                  .
                  .
0040    LISTA    EQU      *
                  .
                  .
0054    ENDA     EQU      *
0054    REF4     WORD     ENDA-LISTA+LISTC                  000014
0057    REF5     WORD     ENDC-LISTC-10                     FFFFF6
005A    REF6     WORD     ENDC-LISTC+LISTA-1                00003F
005D    REF7     WORD     ENDA-LISTA-(ENDB-LISTB)           000014
0060    REF8     WORD     LISTB-LISTA                       FFFFC0
                 END      REF1
```

```
Loc              Source statement                           Object code

0000    PROGB    START    0
                 EXTDEF   LISTB,ENDB
                 EXTREF   LISTA,ENDA,LISTC,ENDC
                  .
                  .
                  .
0036    REF1    +LDA      LISTA                             03100000
003A    REF2     LDT      LISTB+4                           772027
003D    REF3    +LDX     #ENDA-LISTA                        05100000
                  .
                  .
                  .
0060    LISTB    EQU      *
                  .
                  .
0070    ENDB     EQU      *
0070    REF4     WORD     ENDA-LISTA+LISTC                  000000
0073    REF5     WORD     ENDC-LISTC-10                     FFFFF6
0076    REF6     WORD     ENDC-LISTC+LISTA-1                FFFFFF
0079    REF7     WORD     ENDA-LISTA-(ENDB-LISTB)           FFFFF0
007C    REF8     WORD     LISTB-LISTA                       000060
                 END
```

```
Loc              Source statement                           Object code

0000    PROGC    START    0
                 EXTDEF   LISTC,ENDC
                 EXTREF   LISTA,ENDA,LISTB,ENDB
                  .
                  .
                  .
0018    REF1    +LDA      LISTA                             03100000
001C    REF2    +LDT      LISTB+4                           77100004
0020    REF3    +LDX     #ENDA-LISTA                        05100000
                  .
                  .
                  .
0030    LISTC    EQU      *
                  .
                  .
0042    ENDC     EQU      *
0042    REF4     WORD     ENDA-LISTA+LISTC                  000030
0045    REF5     WORD     ENDC-LISTC-10                     000008
0048    REF6     WORD     ENDC-LISTC+LISTA-1                000011
004B    REF7     WORD     ENDA-LISTA-(ENDB-LISTB)           000000
004E    REF8     WORD     LISTB-LISTA                       000000
                 END
```

Scanned by CamScanner

- LIST A , LISTB, LISTC → list of items of each prog.

ENDA , ENDB , ENDC → marks end of lists

Reference to external symbol in
REF1 to REF3 → as instruction operands
REF4 to REF8 → values of data word.

REF1

In PROGA , REF1 is a reference to label within the program so, no modification for relocation or linking needed

In PROGB & PROGC → REF1 is reference to an external symbol so, assembler uses extended format instruction with address field → 0000 & Modification record required to tell loader that add value of LISTA is to be added after linking

REF2

Similiar to REF1 but here PROGB has local
reference L PROGA & PROG C have external symbol.

REF3

It is an immediate operand whose value is ENA-LISTA
It is an immediate operand whose value is ENA-LISTA
In PROGA it can be directly computed but in the other 2 ~~prog~~, the value is unknown
The expression is assembled as extend reference & final result is an absolute value independent of location of where program is loaded.

~~Gener~~ General approach → assembler evaluate as much of the expression as it can & remain is passed onto loader via Modification record.

eg- REF4.

In PROGA assembler can evaluate all expression except for LISTC
The result is an initial value of 000014h and 1 Modification record
↳(0054-0040)

In PROGB no terms can be evaluated by the assembler
The result is an initial value of 000000h & 3 Modification record -

In PROGC assembler can supply value of LISTC but rest is unknown.

Initial value is relative address of LISTC and Modification record telly to add value of ENDA 2 subtract value of LISTA.

- Consider the 3 progs have been loaded into memory with PROGA starty at address 4000, with PROGB & PROGC immediately following

# REF4 to REF8 will end up with same value in each of the 3 program after relocation and linking.

eg - value of reference REF4 in PROGA. located at 4054 (4000 + relative address of REF4(0054))

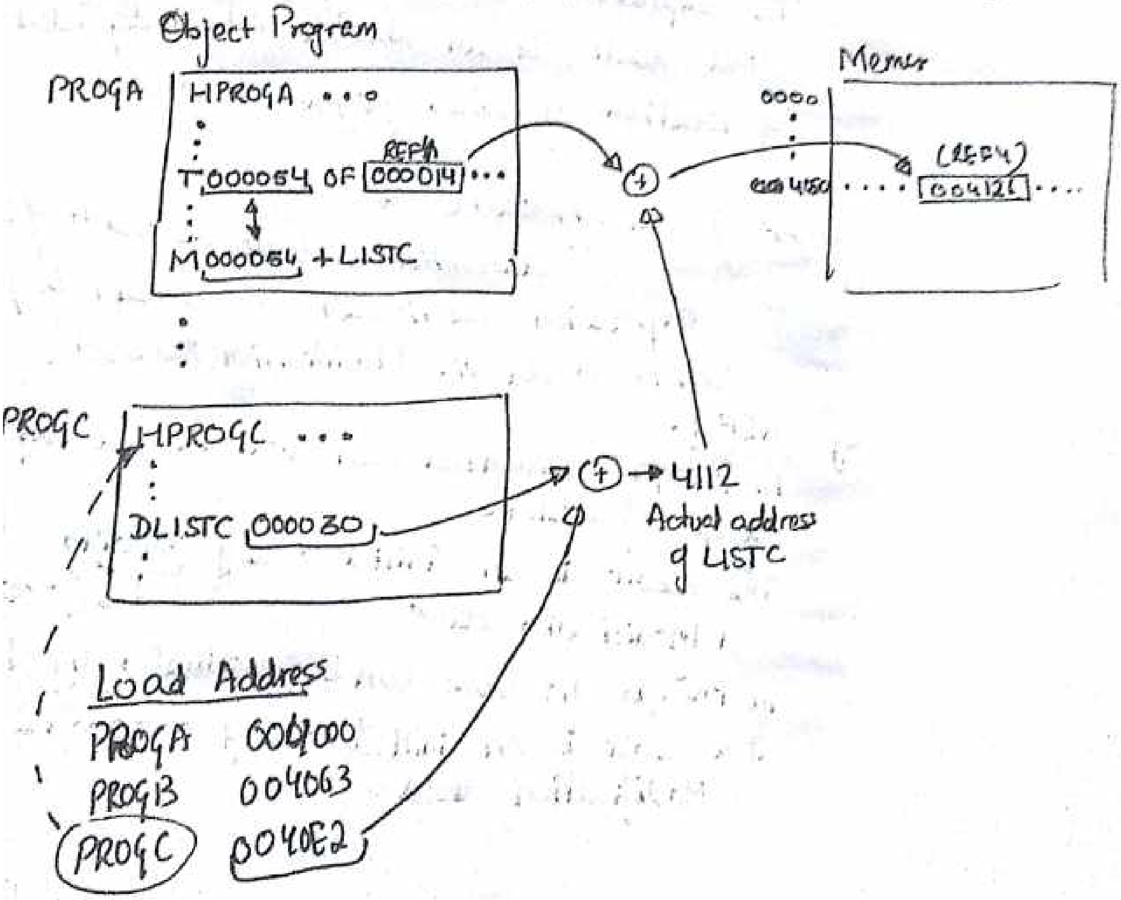
Initial value of REF4 → 000014h (from the Text record)

To this we add address assigned to LISTC (4112) [beginning of PROGC + 30]

⇒ value in memory 4054

→ 000014 + ~~00~~4112

= 004126.



In PROG B for REF4

~~ini~~ located at relative address 70

∴ so, memory location (4063+70 → 40D3)

∴ initial value → 000000

+ENDA → 4054 (4000+54)

+LISTC → 4112 (40E2+30)

−LISTA → 4040 (4000+40)

= 004126

→ same as in PROGA

Similarly for PROGC, REF4 also results in 004126

* REF1−REF3 → which are reference that are instruction operand, calculated values after loading aren't always equal as additional address calculation step involved in case of base or PC relative instruction

eg − REF1 →

For PROGA → target address 4040.
displacement 01D + PC (4023)

For PROGA → REF1 is extended format instruction with direct address which is 4040
(LISTA location → 4000+40→4040)

→ Algorithm & Data Structure for Linking Loader.

− Algorithm for linking & relocating loader that uses Modification record for relocation so that linking & relocation function are performed using same mechanism.

− i/p to loader is set of object programs that are to be linked together

Programs may contain external reference to symbol whose definition come later & so linking operation can't be performed till the external symbol is assigned and address.

∴ Linking loader makes 2 passes over its i/p.

Pass1 → assigns address to all external symbols

Pass2 → performs actual loading, relocation & linking.

Data structure needed,

- ESTAB → external symbol table
  → it stores name & address of each external symbol in the set of control sections (programs) that are loaded
  → Hashed organization is used for this table.

°CPROGA

Important variable needed,

- PROGADDR → program load address
- CSADDR → control section address

PROGADDR is the beginning address where linking program is to be loaded. Its value is supplied by the OS

CSADDR – starting address of st control section currently being scanned by loader

Pass1

→ loader only concerned with Heade & Define record types.

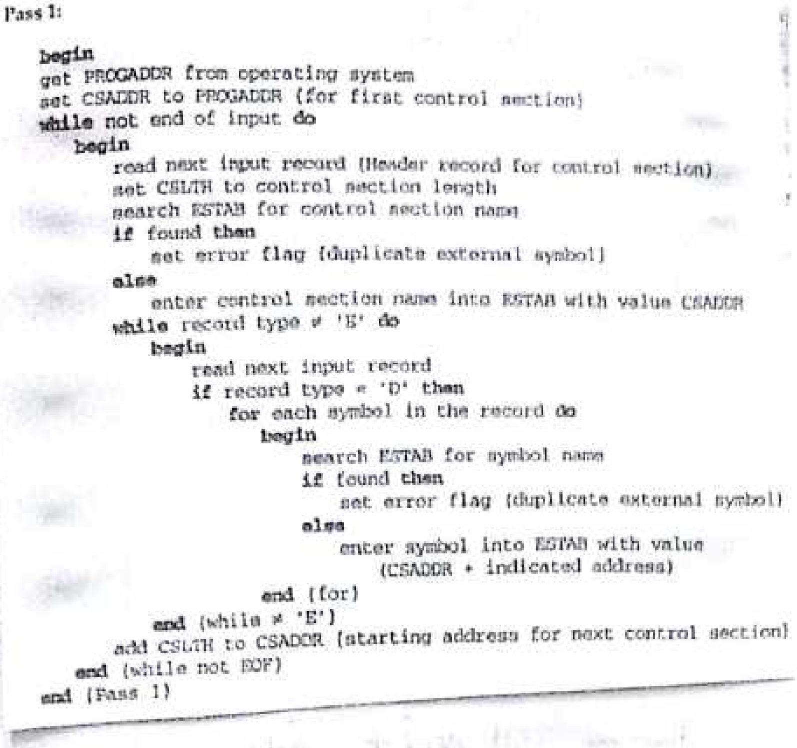
→ Value for PROGADDR is obtained from OS, which is the CSADDR for the 1st control section.

→ Control section name is obtained from Header record and is entered in ESTAB with its correspondg value given by CSADDR.

→ All external symbols that appear in Define record also entered in ESTAB. Their address is relative address + CSADDR.

→ When end record reached,
control section length (SDH) added to CSADDR → this is CSADDR for next section.

Scanned by CamScanner



Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR (for first control section)
while not end of input do
    begin
        read next input record (Header record for control section)
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag (duplicate external symbol)
        else
            enter control section name into ESTAB with value CSADDR
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end (for)
            end (while ≠ 'E')
        add CSLTH to CSADDR (starting address for next control section)
    end (while not EOF)
end (Pass 1)
```

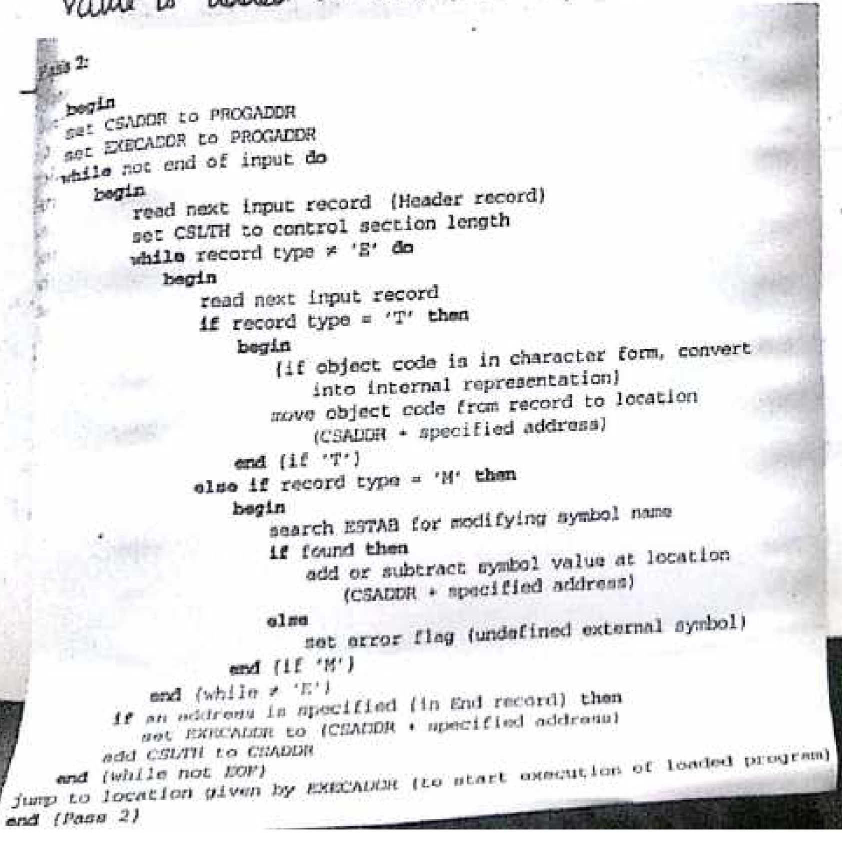
Pass 2.

- Here actual loading, relocation & linking is done

- As Each Text Record is read, object code is moved to its specified address which is,

  relative address + CSADDR.

- When Modification Record is encountered, symbol required for modification is looked up in ESTAB & its value is added or subtracted from intended location.



Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record (Header record)
        set CSLTH to control section length
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        (if object code is in character form, convert
                            into internal representation)
                        move object code from record to location
                            (CSADDR + specified address)
                    end (if 'T')
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end (if 'M')
            end (while ≠ 'E')
        if an address is specified (in End record) then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end (while not EOF)
jump to location given by EXECADDR (to start execution of loaded program)
end (Pass 2)
```

Scanned by CamScanner

Last step performed by loader,
transfer of control to loaded program to Begin execution. The End record for each control section may contain address of 1st instruction in that control section to be executed.
If more than 1 control section specifies transfer address, loader uses the last one encountered
If no control section specifies transfer address, loader uses beginning of linked program (i.e PROGADDR).

→ Algorithm can be made more efficient if we use reference no for external symbol in Modification record, instead of the symbol name
Then we will need to add a Refer record that specifies the symbol & its reference no
eg. R∧02 LISTB 03 ENDB 04 LISTC 05 ENDC.
→ Reference record in PROGA.
So the modification record will be of the form,
M∧000024∧05∧+02

Advantage of this method →
- avoids multiple searches of ESTAB for same symbol while loading of control section
Now, only 1 lookup in ESTAB required for each external reference symbol.

MACHINE INDEPENDENT LOADER FUNCTION

- Automatic Library Search →

– Many linking loaders can automatically incorporate subroutines from program libraries into the program being loaded.

– Some std. libraries are used in such a way, other libraries may be specified by control statement or by parameters to loaders.

- Subroutines called by program being loaded are automatically fetched from the library and linked to the program while loading. This is known as.

→ Automatic library call (or) library Search.

How is it done?

The linking loader that supports this must be able to keep track of external symbols used that arent part of the input.

To do this the loaders enters all external symbol it encounters into the ESTAB, ~~When it~~ if the symbol isnt already not present. When it encounters the a external symbol's definition it complete its entry (if present) by filling in its address.

If at the end of Pass1, some unresolved symbols present in ESTAB then loader searches for them in the libraries.

It is possible that subroutines featched from libraries may also contain external symbols so library search needs to be repeated till all external references have been resolved.

This process allows programmers to override the standard library's subroutines by providing our own subroutine as of input to loader so when loader goes to search library for unresolved symbol reference, the overrided subroutine reference is already defined & resolved.

How libraries are searched?

The libraries themselves have ~~are~~ assembled or compiled version of subroutines. It is possible to search them by their Define records, but it is inefficient.

Special structure called directory used to search libraries. It contains name of each routine & a pointer to its address within the file.
If a subroutine refered to by multiple names; there is an entry for each name and all point to same location.

- This same technique applies to resolution of external reference to data items.

- Loader Options
  - Loaders allow options that modify standard processing of the loader. Many loaders have a special command language that is used to specify options. Sometimes there is a separate i/p file that contains such control statement, sometimes the statements are embedded in the primary input stream b/w object programs or can be included in the same program.

  - On some systems options are specified as part of job control language that is processed by the OS. Here, OS incorporates the options specified into a control block that is made available to loader when its invoked

  - Some options -
    - to select alternative sources of i/p
      eg - INCLUDE program-name (lib-name)
      This directs loader to read object program from a library & treat it as primary loader i/p's part.

Scanned by CamScanner

- to allow users to delete external symbols or entire sections

  RDELETE csect-name

  deletes control section(s) from set of progs being loaded

- to change external reference within prog being loaded & linked

  CHANGE name1, name2

  name1 is changed to name2 wherever it appears in the object prog.

eg- Consider a main program say COPY that has 2 subprograms- RDREC : to read records
WRREC : to write records

Each has its own control section

Suppose utility subroutine available such that it contains subroutines READ & WRITE and it is more favorable for COPY to use them

As a temp measure, first we use some loader commands to make these changes without ~~resembly~~ reassembling the program, to test the new routines

```
INCLUDE   READ (UTLIB)    } tells loader to include
INCLUDE   WRITE (UTLIB)   } control section READ
                            & WRITE from UTLIB
                            library
DELETE    RDREC, WRREC  → tells not to load
                          RDREC & WRREC
CHANGE    RDREC, READ   }
CHANGE    WRREC, WRITE  } → changes all
                            external reference to
                            RDREC to refer to READ
                            & reference to WRREC
                            to refer to WRITE.
```

- LIBRARY MYLIB
  → it automatically includes library routines to satisfy external references

- NOCALL SYMBOLS
  → tells loader ~~not~~ that these external references are to remain unresolved.

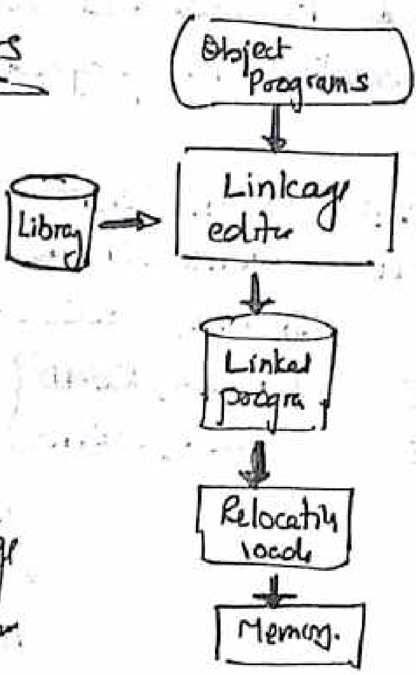
- option to specify that no external reference is to be resolved
  Usefull when programs are to be linked but not immediately executed
- option to specify where execution should begin
- option to control whether or not loader should execute program if error is detected during load

LOADER DESIGN OPTIONS

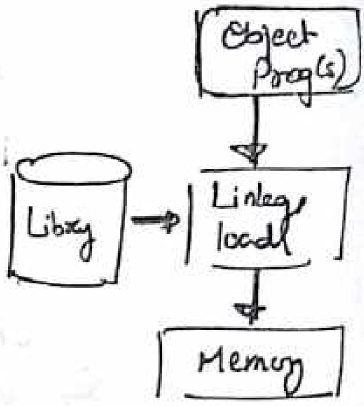
Organisation of loading function

- linking & relocation take place at load time (used by linking loader)
- linkage editors — linking is performed prior to load time
- dynamic linking — linking is performed at execution time

→ Linkage Editors



a) Linkage editor



In ~~linkage editor, the source program is first~~ assembled ~~or compiled~~

• Linkage Editor vs Linking Loader.

→ Linking Loader performs all linking & relocation function & loads linked program directly into memory for execution

– Linkage editor produces a linked version of program called load module or executable image, which is written into a file or library for later execution.

→ Linkage ~~loa~~ editor is useful for programs that need to be executed multiple times without reassembling everytime.

For execution, relocation loader loads program into memory. Only the object code modification required is getting the actual address for loading, rest is done during linking. So, now loading can be done in 1 Pass.

→ Linking Loader is better when program needs to be reassembled for nearly every execution.

• The linked program produced by linkage editor is in a form that is suitable for processing by relocating loader.

- All external references are resolved
- relocation is indicated by some mechanism like Modification record or bit mask.

Information about external references are often retained in the linked program as it allows subsequent relinking of program to replace control sections, modify external references, etc.

• If actual address for loading is known, then linkage editor can perform the relocation, i.e result is linked program that is exact image of way program will appear in memory.

But,

flexibility of loading program at any location is preferred over the reduction of overhead for performing relocation at run time

• Other useful functions →

→ modification of a linked program without having to process the entire program.

eg– consider a program PLANNER that has multiple subroutines. One of its subroine PROJECT had to be changed due to error a to improve efficieny. After new version of PROJECT is assembled or compiled, linkage editor can replace this subroutine in the linked version of PLANNER. using some linkage editor commands.

```
INCLUDE   PLANNER (PROGLIB)
DELETE    PROJECT
INCLUDE   PROJECT (NEWLIB)
REPLACE   PLANNER (PROGLIB)
```

→ linkage editor can be used to build packages of subroutines or other control sections that are generally used together.

This is useful while dealing with subroutine libraries that support high level programming lang.

eg- In a typical implementation of FORTAN, there are large number of subroutines that are used to handle formatted input & output. There are large no. of cross-reference b/w these subprograms because they are closely related

But, it is desirable to keep them as seperate modules for program modularity & maintability.

But, same set of cross-reference will be processed for almost every ~~each~~ FORTAN program linked. This represents a substantial overhead.

We can use the linkage editor to combine the subroutines into a package using commands like,

```
INCLUDE   READR (FTNLIB)
INCLUDE   WRITER (FTNLIB)
INCLUDE   ENCODE (FTNLIB)
   :
   :
SAVE      FTNIO (SUBLIB)
```

The linked module FTNIO can be indexed in directory of SUBLIB under same name as original subroutine. Thus, search of SUBLIB before FTNLIB would retreive FTNIO instead of seperate routines.

Scanned by CamScanner

And as FTN10 could already have all cross-reference b/w subroutines resolved, these linkage wouldnt need to be reprocessed when user's program is linked.

→ linkage editor allow user to specify that external references are not to be resolved by automatic library search/

eg- If 100 FORTAN program using I/O routines, in FTN10 are to be stored in a library, the library will store 100 copies of FTN10 if all external reference were resolved

This wastes a lot of library space

We can use commands to specify that no library seach is to be performed during linkage editing and so they can only be resolved during execution.

This will ~~r~~ require slightly more overhead due to 2 linkage operation but it results in large saving of library space.

- Linkage ~~to~~editors are in general more flexible than linking loader & also offer more control.
  But they also are more complex and have greater overhead.

→ Dynamic Linking (or dynamic loading or load on call)

- Here the linking is performed during execution time.
  i.e a subroutine is loaded & linked to rest of program when it is first called.

- It is used to allow several executing programs to share 1 copy of a subroutine or library.

eg- run-time support routines ~~for~~ for a high level lang like C could be stored in a dynamic link library.

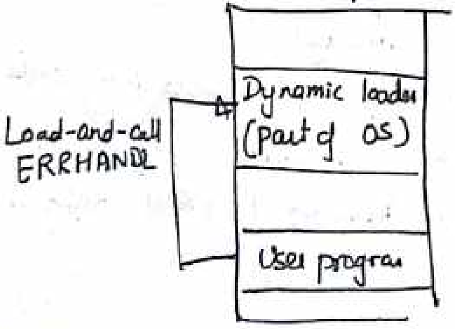
A single copy of the routines could be loaded into memory & all executing C programs could be linked to this copy instead of having seperate copy for each.

- In object-oriented system, dynamic linkiy is used for references to software objects.

This allows implementation of object & its methods to be deimined duriy run-time. The implementation can be changed anytime without affecty program that uses the object

- Advantage of dynamic linking —
  - *it provides ability to load routine only when they are needed
    This results in saviy of time & memory space
  - eg – consider program contains subroutines that correct or diagnose error in i/p data duriy execution. If no error occurs (which can be common) then these subroutines will not be used and so will not be loaded & linked.
    - If program has many subroutines but uses only a few dependiy on its input, then only the subroutines required can be loaded & linked duriy execution.

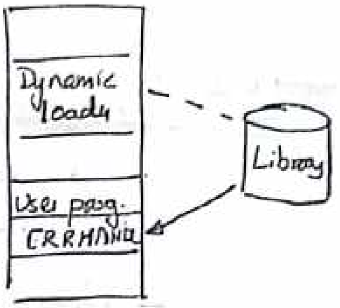
- How to accomplish loady & linkiy of called subroutine?
  - The routine that must be dynamically loaded must be called via OS service request, ie the request is to the part of the loader that is kept in memory. ~~So a JSUB instruction that refers to an external symbol~~
  - So instead of executy a JSUB instruction that refers to an ~~ext~~ external symbol, program makes a load & call request to OS with symbolic name of routine as the parameter.



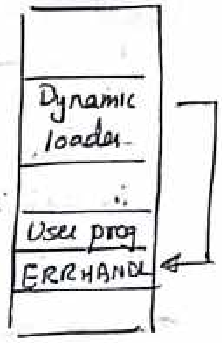
Here, the user program sends a load-and-call request for ERRHANDL subroutine.

- The OS examines internal table to determine whether or not routine is already loaded

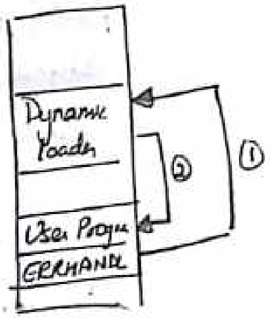
If not, routine is loaded from specified user of or system library
[Load]



and then control is passed to the routine being called.
[Call]

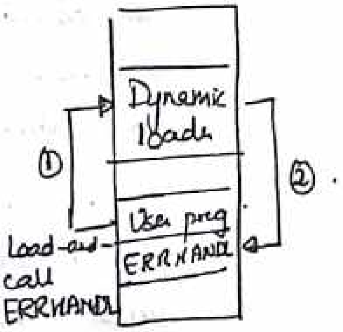


When subroutine completes its processing, it returns to its caller (i.e OS routine that handles load-and-call request).
The OS then returns the control to the user program



After subroutine is completed, the memory that was allocated for loading may be released & used for other purpose
But, this isn't done immediately as if a 2nd call to it occurs, another load operation won't be required.
So, it is desirable to keep the subroutine till memory isn't required by an.

If subroutine called is still in memory, control is directly passed to it from the dynamic loader.



- In dynamic loading, binding of symbolic name to actual address is delayed from load time until execution time which results in greater flexibility

Scanned by CamScanner

- But, this also requires more overhead as OS intervenes in the calling process

→ Bootstrap Loaders

In a idle computer with no program in memory, how do thing start?

- When machine is empty and idle there is no need for relocation, only absolute address for program being 1st loaded is needed. (this program is usually the OS). For this we need an absolute loader loaded.
  - Early computers required operator to enter into memory the object code of absolute loader using switches on computer console. But, this is too inconvenient & error-prone.
- In some computer, absolute loader program is perminantly present in a ROM. When some hardware signal occurs indicating start up of the system, the machine begins executing this ROM program.
  On some computer, program is executed in the ROM on others, program is copied ~~to~~ to main memory & executed.
  But, it is inconvenient to change the ROM program if modification necessary.
- Intermediate solution,
  have a built-in hardware function (or small ROM program) that reads fixed length records from some device into memory at fixed location
  After reading operation is complete, control is transferred to address in memory where records is stored. These records contain ~~address in~~ machine instructions that ~~absolute loader~~ loads the absolute program that follows.
  If the instructions can't be fit in 1 record, then record causes reading of other records & they in turn cause reading of more records.
  → hence the term bootstrap.
  1st record(s) → bootstrap loader.

This loader added to begining of all object programs that are to be loaded into empty & idle system

IMPLEMENTATION EXAMPLE →

→ MS-DOS Linker for Pentium & other x86 system.

- Most MS-DOS compiler & assembler produce object modules, not executable machine language programs.
- These object modules have extension .OBJ and they contain binary image of translated instructions & data of program. It also describes structure of program.
- MS-DOS LINK – linkage editor that combines one or more object modules to produce a complete executable program.

The executable program have extension .EXE. LINK can also combine the translated program with other module from object code libraries.

- A typical MS-DOS object module,

Record Type	Description
THEADR	Translator Header.
TYPDEF PUBDEF EXTDEF	External symbol & references
LNAMES SEGDEF GRPDEF	Segment defination and grouping
LEDATA LIDATA	Translated instructions & data.
FIXUPP	Relocation & linking information
MODEND	End of object module.

similar to Header & End record of SIC/XE {
- THEADR record – specifies name of object module
- MODEND record – marks end of module & contains reference to entry point of program

Scanned by CamScanner

- PUBDEF record — contains list of external symbol called public names that are defined in the object module.

- EXTDEF record — contains list of external symbols that are referred to in the object module.

Similiar to Define & Refer record of SIC/XE
Both PUBDEF & EXTDEF contain info abt data type designated by an external name.

- TYPEDEF record — defines the types

- SEGDEF record → describes segment in object module including their name, length & alignment

GRPDEF record — specify how these segments are combined into groups

LNAMES record — contains list of all segment & class names used in program.

SEGDEF & GRPDEF refer to segment by giving the position of its name in the LNAMES records.

~~LE DATA~~

- LEDATA record — contains translated informations & data from source program
It is similiar to Text record of SIC/XE

LIDATA record — specify translated instructions & data that occur in repeating pattern.

- FIXUPP record — used to resolve external references & carry out address modifications that are associated with relocation & grouping of segment within the program.
It's similiar to Modification record of SIC/XE
But FIXUPP records are more complicated.
A FIXUPP record must immediately follow the LEDATA or LIDATA record to which it applies.

Scanned by CamScanner

- LINK performs its functions in two Passes.

Pass1 – computes starting address for each segment in the program

It constructs a symbol table that associates an address with each segment (using LNAMES, ~~SEGMEN~~ SEGDEF & GRPDEF records) and each external symbol (using EXTDEF & PUBDEF records).

If unresolved external symbols remain after all object modules are processed, LINK searches the specified libraries.

Pass 2 – LINK extracts translated instruction & data from object modules & build an image of executable program in memory.

This is because,
executable program is organised by segment & not by order of object modules.

Building a memory image, most efficient way to handle rearrangements caused by combining & concatenating segment.

If enough memory isn't available, LINK uses temp disk file in addition.

Here LINK process each LEDATA & LIDATA record along with corresponding FIXUPP records & places binary data from LEDATA & LIDATA record into memory image at locations reflecting segment address computed during Pass1.

Relocation & resolving of external reference is done here. A table of segment fixups is maintained that is used to perform relocation that reflects actual segment address when program is executed.

Once memory image is complete LINK writes it to .EXE file, which also contains a header that contains table of segment fixups & information about memory requirement & entry points & also initial contents of CS & SP registers

Scanned by CamScanner

→ Sunos Linkers for SPARC system.

- Sunos provides 2 different linkers
  - run-time linker
  - link-editor

- Link-editor is most commonly used in process of compiling a program.
  It takes 1 or more object module produced by assemblers & compilers & combines them to produce a single o/p module.

- Types of output module →
  1. Relocatable object module
     It is suitable for further link-editing
  2. Static executable
     It has all symbolic references bound & ready to run
  3. Dynamic executable
     It has some symbolic references that are to be bound at run-time
  4. Shared object
     It provides services that can be bound at runtime to 1 or more dynamic executables

- Object module contain multiple sections which represent instructions & data areas from source program.
  These sections have a set of attributes such as "executable", "writable".
  Object modules also include list of relocation & linking operations that need to be performed & a symbol table that describes the symbols used.

- Sun-OS link-editor reads the object modules that are given to it to process. Sections that have same attributes are concatenated to from new section in o/p file.

Scanned by CamScanner

- Symbol table from i/p files are processed to match symbol definations & references, and relocation & linking per operations are performed within o/p file.
  Linking generates new symbol table & new set of relocation instruction in output file. They represent symbols that need to be bound at run-time & relocations that need to be performed during loading.

- Relocation & linking operation are specified using set of processor-specific code
  The codes reflect instruction format & addressing modes that are found in the machine as they describe the size of the field to be modified & calculations that need to be performed.

- Symbolic references from i/p file that aren't resolved are processed by reffering to archives & shared objects

  ↓ collection of relocatable object modules.

  Directory within archives associate symbol name with object module that contains its definition & selected module from archive is included to resolve the reference

  ~~Share~~
  Shared object is an indivisible unit that was generated by some previous link-edit operation.
  If reference symbol is defined in a shared object, entire content of shared object becomes logical part of o/p file.
  Link-editor records dependency to shared object, actual inclusion of the shared object happens at run time

- SunOS run-time linker,
  used to build dynamic executable & shared objects at execution time
  It determines what shared objects are required by dynamic executable & ensures that they are included.
  It also resolves any additional depencies on other shared objects.

Scanned by CamScanner

After locating & including necessary objects, linker performs relocation & linking to prepare program for execution.

They bind symbol to actual memory address to which segment is loaded &. Then control is passed to executable program after binding data reference.

Binding of procedure call is done during execution. During link-editing, calls to globally defined procedure is converted to reference to a procedure linking table. When procedure is called for the 1st time, control is passed to run-time linker via the table. The linker looks up the actual address of the procedure & includes it to linkage table.

So, subsequent call will directly go to called procedure → lazy binding.

- Run-time linker provides ~~ex~~ flexibility.

  During execution, prog can dynamically bind to new shared objects, ~~by~~ this allows prog to choose b/w no. of shared objects.

  If a shared object isn't needed it isn't ~~binded~~. binded.

→ Cray MPP Linker for Cray T3E system.

- T3E system contain large no. of processing elements (PEs).

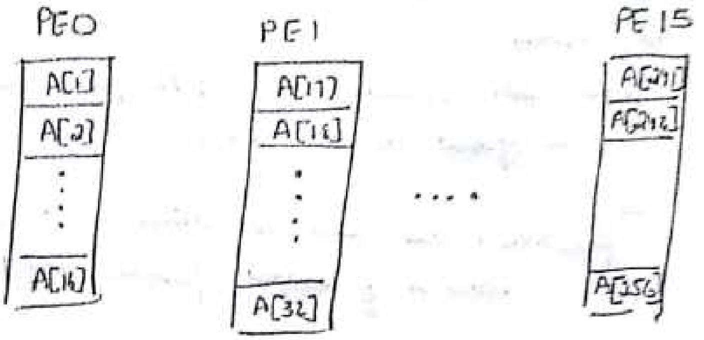
  Each PE has its own local memory & can access memory of all other PEs.

- An application program on a T3E system is allocated a partition that consist of several PEs. (to take advantage of ||el architecture of machine)

Scanned by CamScanner

- Work to be done is divided ~~into~~ b/w the PEs

eg - partition contain consists of 16 PEs, 2 elements of a 1D array is distributed



If prog contains loop that process all 256 elements, PE0 can execute loop for A[1] to A[16]
PE1 can execute loops for A[17] to A[32] & so on.

- Shared data → data that is divided among no. of PEs.

Private data → data that isn't shared by dividing it, each PE contains a copy of the data.
Or PE has private data that exists only in its own local memory

- When program is loaded,
each PE gets a copy of executable code, its private data & its portion of shared data.

- MPP linker organizes blocks of code or data from object program into lists.
The blocks on a given list all share some same property.
The blocks on each list is collected, address is assigned to each ~~b~~ block & relocation and linking operations are performed.
The linker then writes a executable file that contains relocated & linked blocks. It also specifies no. of PEs required & other control information.

- Distribution of shared data depends on no. of PEs.

  If no. of PEs is specified at compile time,
  it can't be overridden later.

  If not, either
  linker can create executable file that targets for a fixed no. of PEs

  or
  partition size can be chosen at run time
  This is ~~p~~ called plastic executable

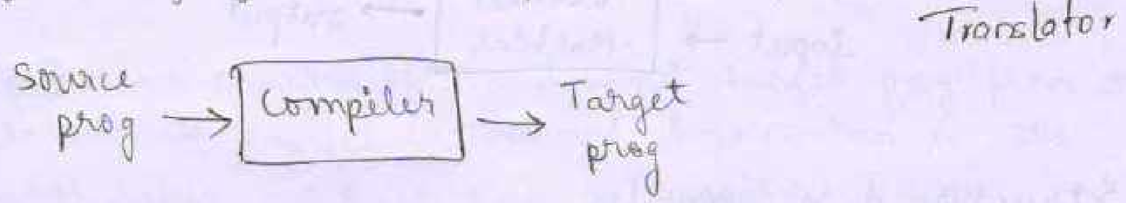
Plastic executable is often larger than one targeted for fixed no. of PEs as,
it must contain copy of all relocatable object module & all linker directives that are needed to produce final executable.

Compiler Design - 10CS63

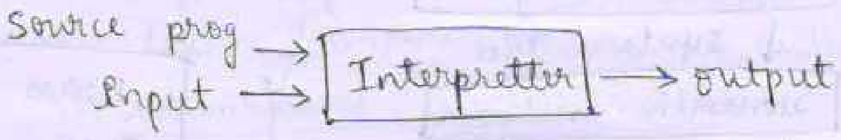
UNIT 1: Introduction

Translator - Any program that converts a high level language program to Machine (Low Language) code.

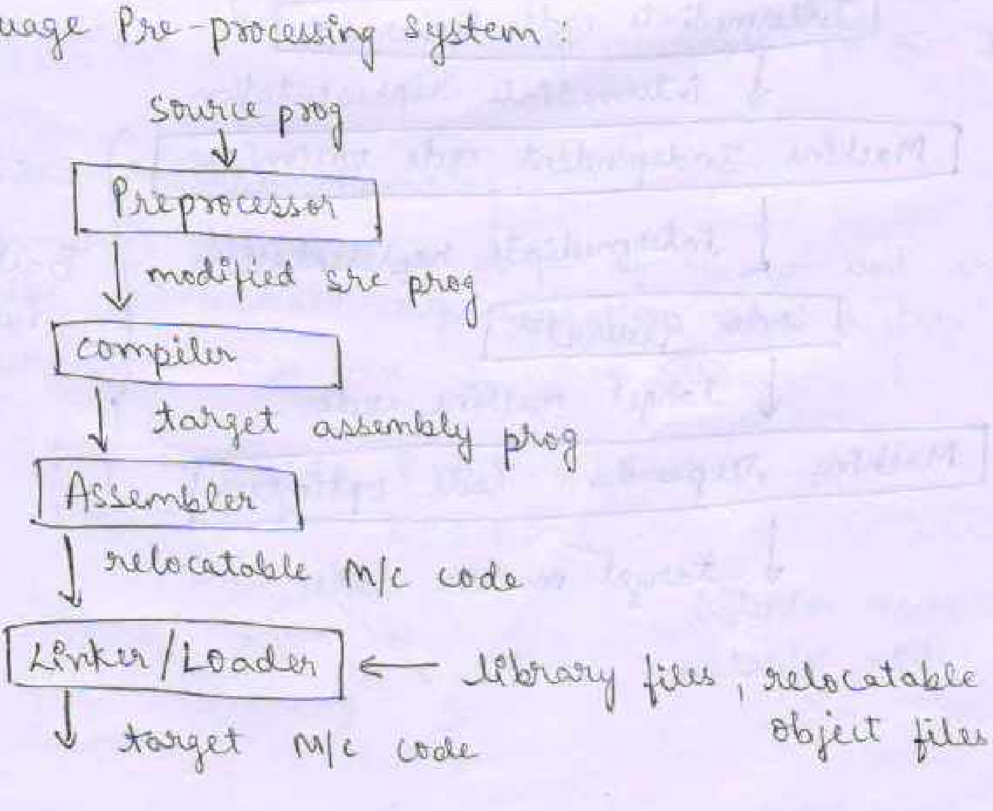
compiler - Program that reads code in one language i,e source code and translates it into another language i,e target language is a compiler.



Interpretter - A kind of language processor which does not produce target program as a translation, but directly execute the operations specified in source program, on inputs supplied by the user

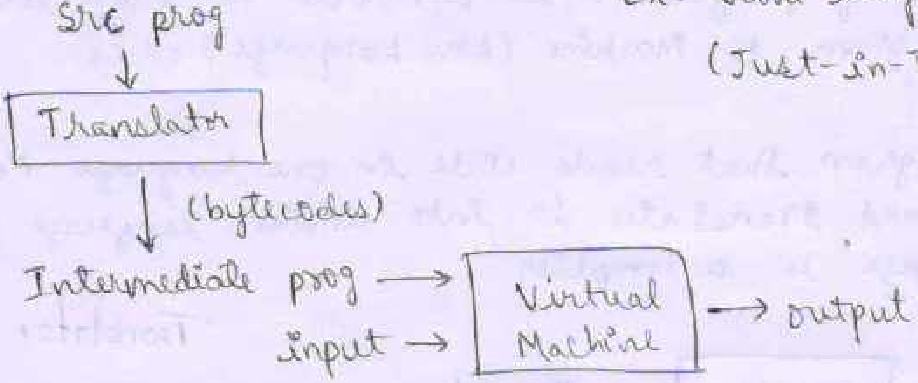


Language Pre-processing system :

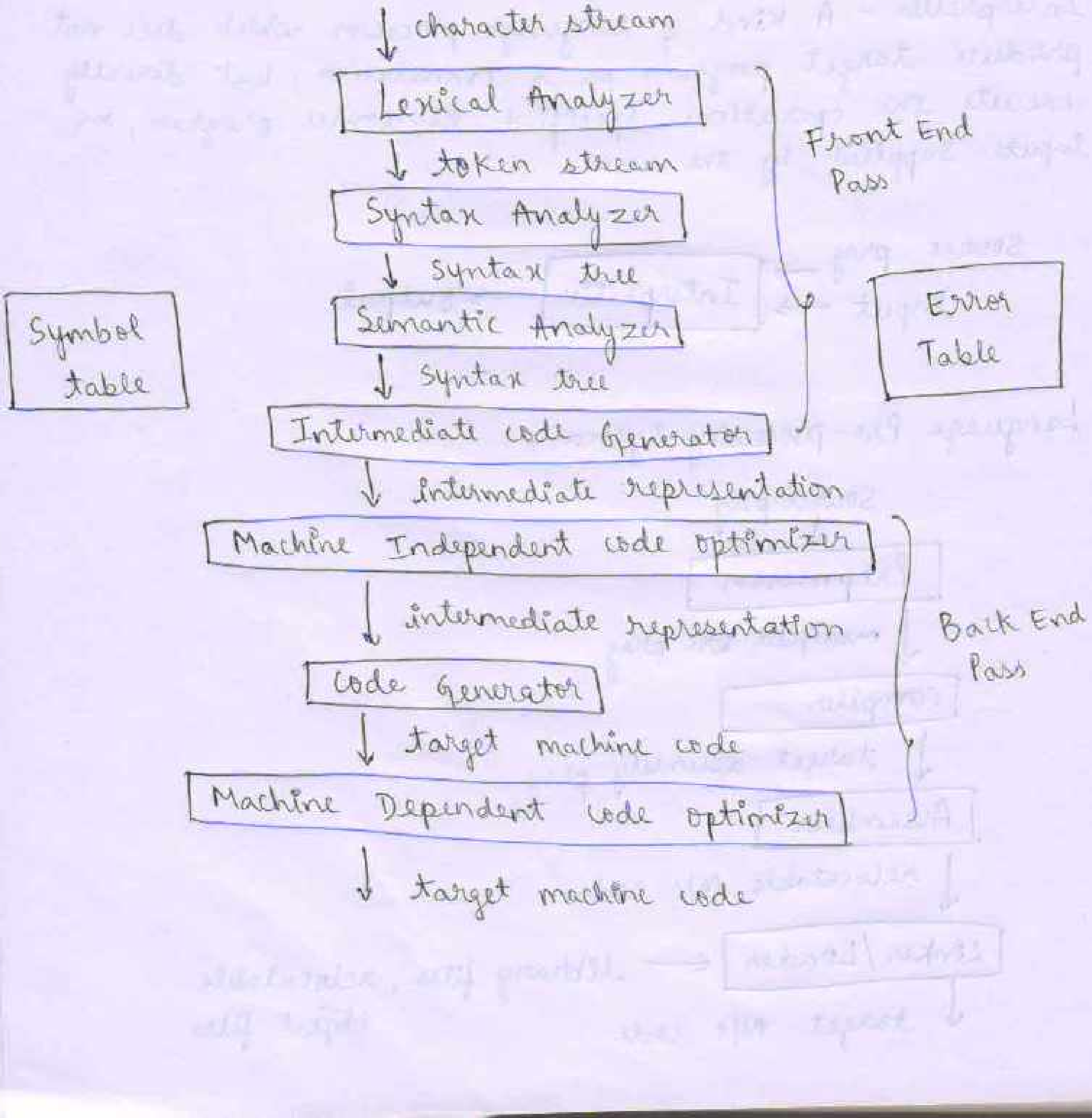


- A Hybrid compiler :

Ex: Java lang processor
(Just-in-Time)



Structure of a compiler :



→ 2 main parts:

① Analysis - breaks up source prog into constituent pieces & imposes grammatical structure on them. Based on the structure it creates intermediate representation of source prog. Collects information about prog, stores it in the "Symbol Table". (Front End of compiler)

② Synthesis - constructs the desired target prog from the intermediate representation and information in the symbol table. (Back End of compiler)

→ 7 phases:

① Lexical Analysis - Scanning

- On reading character stream of src prog, it groups them into meaningful sequences called "Lexemes".
- for each lexeme, analyser produces as output a token of form:

< token-name , attribute value >

abstract symbol used in parser

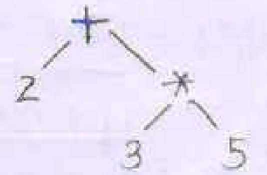
points to entry in the symbol table for this token

② Syntax Analysis - Parsing

- parser uses tokens i,e output of Scanner and creates a tree like intermediate representation that depicts the "Grammatical Structure" of token stream

Ex: For Grammar $E \to E+E \mid E*E \mid num$

For Input $2+3*5$



interior node: operators
exterior node: arguments

③ Semantic Analysis

- uses syntax tree and information in symbol table to check source prog for semantic (meaning) consistency with lang definition.
- It gathers type information and saves it in either syntax tree or symbol table for use in ICG.
- Type checking - compiler checks whether each operator has the matching operands
- coercions - Lang specification may permit some type conversion

④ Intermediate code Generation (ICG)

- The intermediary code during processing may be in the form of syntax tree or reduced form of source code.
- properties :

→ should be easy to produce
→ should be easy to translate into target M/C.

⑤ code optimization (M/C independent)

- to improve intermediate code to get better target code

→ Better in terms of : faster, shorter, less power consuming code

- Instead of using int to float operation,
replace integer by its floating-point value directly

⑥ code Generation

— input from intermediate representation maps to target lang.

- If target lang is M/C code - the instructions are translated into sequences of M/C instruction to perform same task.
- Judicious assignment of registers to hold variables is done

→ Compiler construction Tools :

① Parser Generators - automatically produce syntax analyzers from a grammatical description of a prog lang

② Scanner Generators - produce lexical analyzers from a regular expression description of tokens of lang

③ Syntax directed translation engines - produce collections of routines for walking a parse tree and generate ICG

④ code generator generators - produce CG from collection of rules for translating each operation of Intermediate lang into M/C lang for a target M/C.

⑤ Data flow analysis engines - facilitate gathering of data about how values are transmitted from one part of prog to every other part.

⑥ compiler construction toolkits - provide integrated set of routines for constructing various compiler phases.

Application of compiler Technology :

① Implementation of high level prog lang using modern OOPS concept like,
→ Data Abstraction
→ Inheritance properties

② optimization for computer architectures
→ Parallelism
(i) at instruction level - multiple operations executed together
(ii) at preprocessor level - different threads run seperately

→ Memory Hierarchy
Building very <u>Large</u> or <u>Fast</u> storage, but not both

③ Design New computer Architectures

→ RISC - reduces complex memory addressing, support data structure access, procedure invocation ...

→ Specialized Architectures -

Data flow M/C, vector M/C, VLIW & SIMD M/C.

④ Program Translations

(i) Binary Translation - Increases S/w availability

(ii) Hardware Synthesis - Verilog, VHDL - reduces time & effort

(iii) Database Query Interpretter - SQL queries effective retrieval

(iv) compiled simulation - model run, to validate design.

(v) Reduce redundancy in code

⑤ Software Productivity Tools

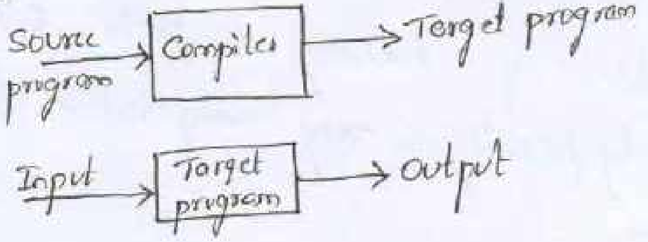
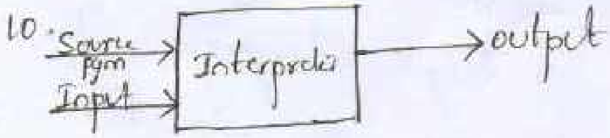
(i) Type checking - to catch program inconsistency

(ii) Bounds checking - Lang. provides range checking like for the buffer overflow, security, optimize range check, sophisticated analyses, error detection tools.

(iii) Memory Management Tools - (Garbage collection)

- Automatic memory management tracks all memory related errors - leaks...

1. Write the difference between compiler and Interpreter

Compiler	Interpreter
1. Compiler translates the entire program in one go and then executes it	1. Interpreter first converts high level language into an intermediate code and then executes it line by line. The intermediate code is executed by another program
2. It produces efficient object code therefore programs runs faster	2. No intermediate object code is generated
3 Error reporting is time consuming (displayed after entire pgm is checked)	3. Errors are displayed for every instruction interpreted if any. (Error reporting is immediate)
4 Conditional control statements are executed faster	4. conditional control statements executed slower
5 Memory requirement is more ∵ single object code is generated	5. memory requirement is less
6. Program neednot be compiled every time	6. Everytime high level program is converted into lower level pgm
7. Difficult to use	7. Easy to use for beginners
8. Translate once and then run the result (stand-alone code, faster exn	8. read- check- execute loop -> slower, not stand-alone
9. Eg:- c, c++	9. Eg:- python, prolog
10  Source program Compiler Target program Input Target program output	10. 

→ Examples showing detail phases of compiler :

① position = initial + rate * 60

↓

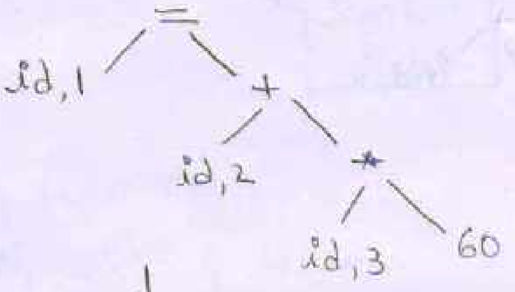
Lexical Analysis

<id,1> <=> <id,2> <+> <id,3> <*> <60>

↓

Syntax Analysis

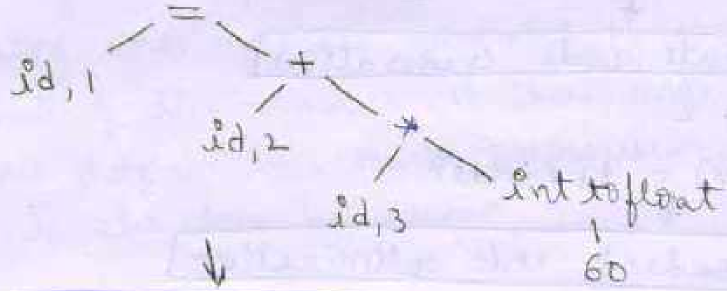
1	position	
2	initial	
3	rate	



Syntax Tree

↓

Semantic Analysis



↓

Intermediate code Generation

t_1 = int tofloat(60)
t_2 = $id_3 * t_1$
t_3 = $id_2 + t_2$
id_1 = t_3

↓

M/c independent code optimization

t_1 = $id_3 * 60.0$
id_1 = $id_2 + t_1$

⟶ Code Generation

```
LDF  R1, id3
MULF R1, R1, #60.0
LDF  R2, id2
ADDF R2, R2, R1
STR  id1, R2
```

② a[index] = 4+2+index

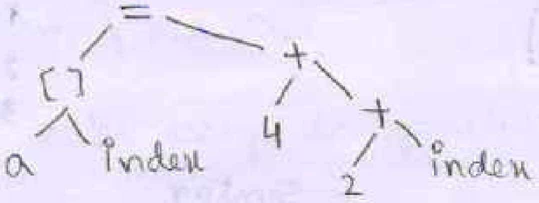
↓

Lexical Analysis

<id,1> <[> <id,2> <]> <=> <4> <+> <2> <+> <id,2>

↓

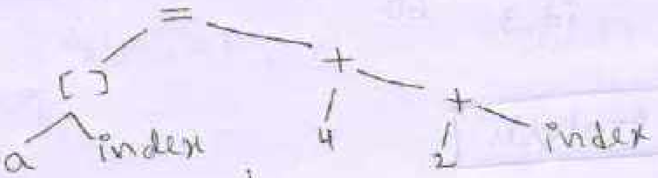
Syntax Analysis



1	a	
2	index	

↓

Semantic Analysis



↓

Intermediate code Generation

$t_1 = 4+2$

a[index] = t_1 + index

↓

M/C independent code optimization

a[index] = 6 + index

↓

Code Generation

```
mov index, R0      // R0 = index
mov &a , R1        // R1 = Starting address of array a
add R0 , R1        // R1 = R0+R1
mov #6 , R2        // R2 = 6
add R1, R2         // R2 = R1+R2 = R1 + 6
mov R2, &R1        // store R2 in &R1 i.e &a's value
```

→ Environments and States :

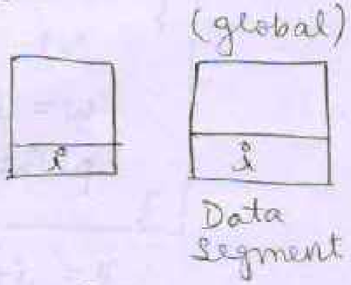
$$\begin{tikzcd} \text{names} \arrow[r, bend left, "\text{environment}"] & \text{locations (variables)} \arrow[r, bend left, "\text{state}"] & \text{values} \end{tikzcd}$$

- Environment is mapping from names to location in the store
- State is mapping from location in store to their values

Dynamic Mapping Exceptions :

(i) Static Binding of Names to Locations - global variable declaration - location in store once for all.

Ex:
```
int i ;                 // global i
void fun (...) {
     int i ;            // local i
}
```



(ii) Static Binding of Locations to values - declared constants

Ex: # define ARRAYSIZE 1000 //static bind

Static Scope and Block Structure :

① main () {
```
    int a=1;                          B1
    int b=1;
    {
        int b=2;                      B2
        {
            int a=3;                  B3
            cout <<a <<b ;
        }
        {
            int b=4;                  B4
            cout <<a<<b ;
        }
        cout <<a<<b ;
    }
    cout <<a<<b;
}
```

→

Declaration	Scope
int a=1 ;	B1 - B3
int b = 1 ;	B1 - B2
int b = 2 ;	B2 - B4
int a = 3 ;	B3
int b=4 ;	B4

②

```
main ()
{
   int  w,x,y,z ;
   int  i = 4 ; int  j = 5 ;
   {
      int  j = 7 ;  i = 6 ;
      w = i + j ;
      printf (w) ;                 → 6+7 = 13
   }
   x = i + j ;
   printf (x) ;            ——————→ 6 +5 = 11
   {  int  i = 8 ;
      y = i + j ;
      printf ( y ) ;            → 8+5 = 13
   }
   z = i+j ;
   printf (z) ;            ——————→ 6+5 = 11
}
```

③

```
#define   a   (x+1)
 int  x = 2 ;
 void  b( )    {  x = a ;  printf ("%d", a) ; }
 void  c( )   { int x = 1 ;  printf ("%d", a) ; }
void  main ()
 {  b() ;   c() ; }
```

o/p

3

2

(9)

```
int  w, x, y, z ;
int  i = 3 ;
int  j = 4 ;
{
   int  i = 5 ;
   w = i + j ;                5+4=9
}
   x = i + j ;                3+4=7
{  int  j = 6 ;
   i = 7 ;
   y = i + j ;                7+6=13
}

   z = i + j ;                7+4=11
```

10. what is printed by the following c cod.

a)
```
# define a (x+1)
int x=2;
void b() { x=a; printf("%d\n", x); }  → 3
void c() { int x=1; printf("%d\n", a); }  → 2
void main() { b(); c(); }
```

b)
```
# define a (x+1)
int x=2;
void b() { x=a; printf("%d\n", x); }  → 3
void c() { printf("%d\n", a); }  → 4      ∵ redesignment for thi
                                             same variable x
void main() { b(); c(); }                    x=3 , a=3+1=4
```

c)
```
# define a (x+1)
int x=2;
void b() { int x=1; printf("%d\n", a); }  → 2
void c() { printf("%d\n", a); }  → 3
void main() { b(); c(); }
```

d)
```
# define a (x+1)
int x=2;
void b() { int x=a; printf("%d\n", a); }  → 4 ∵ ( x=2+1=3
                                                 again a=3+1=4)
void c() { printf("%d\n", a); }  → 4
void main() { b(); c(); }
```

→ Parameter Passing Mechanisms :

(1) Actual parameters - parameters used in call of procedure
(2) Formal parameters - parameters used in procedure definition

① Call by value

- The actual parameter is evaluated (if an expression) or copied (if a variable), The value is placed in the location belonging to corresponding formal parameter of called procedure.
- It has all computations involving formal parameter done by called procedure is local to that procedure.

② Call by reference

- The address of actual parameter is passed to the callee as the value of corresponding formal parameter
- Uses of formal parameter in code of callee are implemented by following this pointer to location indicated by caller.
- changes to formal parameter ⇒ Appear as changes in actual param

- If actual parameter is expression, it is evaluated before the call and its value stored in a location of its own.
- changes to formal parameter change value in this location, But - No effect on data of caller.

③ Call by name

- used in early prog - Algol 60.
- it requires callee execute as if actual parameter were Substituted literally for formal parameter in the code of the callee as if formal parameter were macro standing for the actual parameter.

→ Examples :

① call by value

```
int add ( int a, int b )
  {
     return (a+b);
  }
main ()
  {
    :
      c = add ( 10,20)
    :
  }
```

② call by reference

```
int add ( int *a, int *b)
  {
    return (a+b);
  }
main ()
  {
    :
       int p=10;
       int q=20;
       c= add (&p, &q);
    :
  }
```

③ call by Name- Aliasing

```
int add (int a , int b)
  {
     return (a+b);
  }
main ()
  {
       int p=10;
       int q=20;
       c= add ( &p, &q);
  }  :
```

- Aliasing:

→ Interesting consequence of call by reference parameter passing where references to objects are passed by value.

- It is possible that two formal parameters refer to the Same location – Such variables are ALIAS to one another.
- Though they may be <u>distinct</u> formal parameters, they may be Alias of one other.

Ex: Let a be array in procedure p

```
p
  {
    // q(x,y)  call
    q(a, a) ;
  }
```

array names are references to location ⇒ Alias

x[i] = y[i]

Chapter-1 Introduction

Questions

1. Define Compilers?
2. Differentiate b/w compilers & Interpreter?
3. Explain The long processor System?
4. Describe the analysis-Synthesis model of the Compiler or Explain in detail the Various phases of Compiler with an example?
5. Explain in detail the Various phases of Compilation for The i/p string

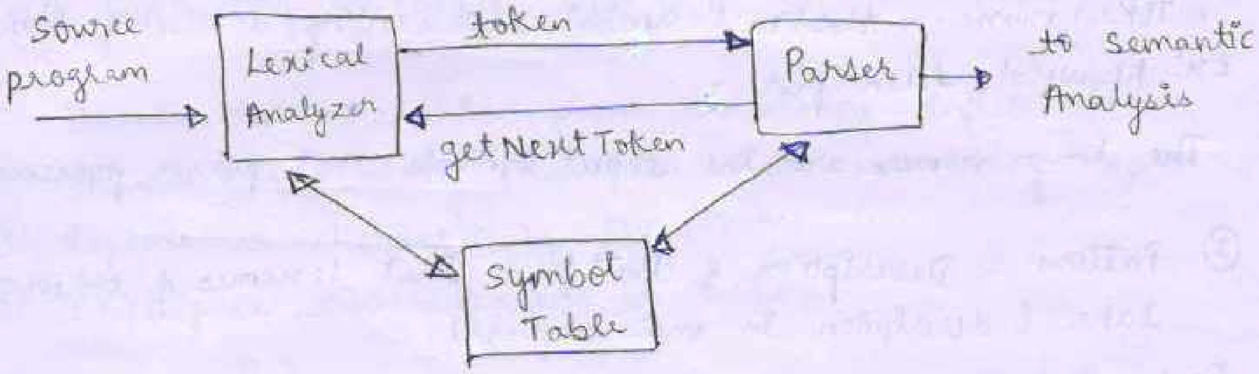
   a. $p = i + r * 60$ c. $a = (b+c) * (b+c) * 2$

   b. $x = a * b + a * b$ d. $a[index] = 4 + 2 + index$
6. why is it necessary to group phases of Compiler
7. what Is The purpose of Compiler Constⁿ tool. Describe the different Compiler Construction tool we used?
8. Analyse The s/w productivity tool and explain
9. Explain The different parameter passing technique with an example?

chapter-3 - Lexical Analysis

→ Lexical Analysis :

Interaction between Lexer and Parser :



→ Task of Lexer :

① Identification of Lexemes
② Stripping out comments
③ Removing whitespace (blank, \n, \t)
④ corelating error messages generated by compiler
⑤ Keep track of line numbers to show error
⑥ If source program uses Macro-preprocessor, the expansion of macros is also done by scanner.

→ Lexer - cascade of 2 processes :

① Scanning consists of simple processes that do not require tokenization of input, such as deletion of comments & compaction of consecutive whitespace characters into one.

② Lexical Analysis proper in more complex portion, where scanner produces sequence of tokens as output.

Lexer versus Parser : Seperate phases because :

① simplicity of compiler design - important consideration
② compiler efficiency improved - use specialized technique for lexical Analysis (Input Buffering)
③ compiler portability is enhanced.

→ Tokens, Patterns, Lexemes :

① Token : A pair consisting of a token name and an optional attribute value.

- Token name – Abstract symbol representing a kind of lexical unit

Ex: Keyword, identifier ....

The token names are the input symbols that parser processes.

② Pattern : Description of the form that lexemes of token may take (description in metalanguage).

Ex: token name : identifier
pattern : $[\_a{-}zA{-}Z]^{+}[a{-}zA{-}Z0{-}9]^{*}$

③ Lexeme : Sequence of characters in source program that Matches the pattern of a token and is identified by the lexer as an instance of that token.

Ex: token name : Keyword
pattern : [i][f]
lexeme : if

Token	Informal Description	Sample Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparision	<, >, <=, >=, ==, !=	<=, <>
id	letter followed by letter and digits	pi, score
number	numeric constants	3.14, 0, 6.9e8.
literal	enclosed within " "	"core dumped"

→ Lexical Errors : Recovery options

① Panic mode recovery - delete successive characters from remaining input until lexer finds well known token at beginning of input left out.

② Delete one character from remaining input

③ Insert one missing character into remaining input

④ Replace a character by the other

⑤ Transpose two adjacent characters.

Examples :

fi (a<b) ⇒ if (a<b)

int a, ; ⇒ int a ; or int a, b ;

→ Input Buffering : To speed up reading of src prog.

① Single Buffer | 1-Buffer Technique

We use only one single buffer to store processed character from large no. of characters from source prog.

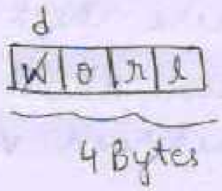
Main overhead is that if,

lexeme size > Buffer size

we loose the lexeme

World

5 Bytes



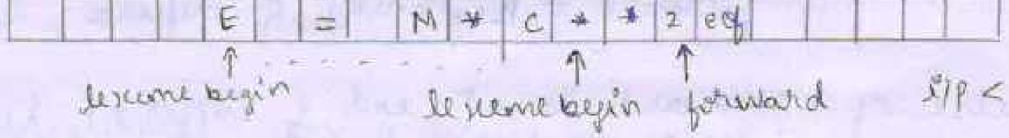
· It reloads data, removes old data.

② 2-Buffer Technique < without sentinel / with sentinel

We use two buffers that are alternately reloaded,

Each buffer of same size N, N = size of a disk block. (4096 Byte)

· Using read system call, N characters are read.



i/p < Buffsize

→ special char eof marks end of src file and this char is different from any other char of src prog.

→ Two pointers maintained :

① Lexeme Begin - marks beginning of current lexeme whose extent we are attempting to determine.

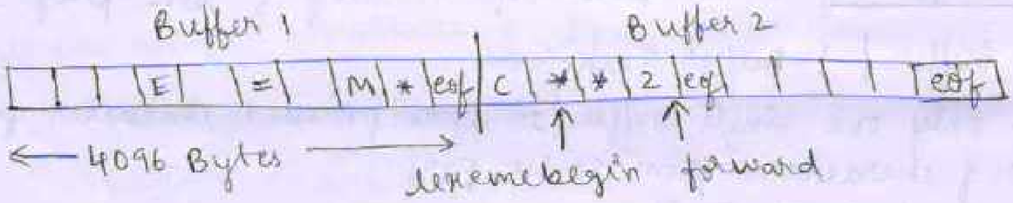
② Forward - Scans ahead until a pattern match is found. When forward reaches end of next lexeme, ** we retract one position back and return token

- We need 2 checks in 2 Buffer without sentinels :

1) Advancing forward requires whether we reached the end of one of the buffer, if Yes Reload other buffer and make forward point to newely loaded buffer beginning.

2) Before returning token check whether valid or not.

→ Sentinels (2 Buffer technique with Sentinels)
Using sentinel character at the end which is a special char. That is not part of src prog (usually eof)



Here check if reached end of Buffer or not.
Look Ahead is atmost 1 char, make previous char as returned valid token.

→ Look Ahead code with sentinel :

```
Switch ( * forward ++ )
{
   case eof : if ( forward is at end of Buffer 1) {
                  reload Buffer2 ;
                  forward = Beginning of Buffer2 ; }

              else if ( forward is at end of Buffer 2) {
                  reload Buffer 1 ;
                  forward = Beginning of Buffer 1 ; }
else   /* eof within a Buffer marks end of input */
        terminate lexical Analysis
   break;
   cases for other char
}
```

① Alphabet - finite set of symbols   Ex: $\Sigma = \{0,1\}$
string - finite sequence of symbols from Σ Ex: 0101
Language - countable set of strings over Σ.

② Prefix of string - string obtained by removing zero or more symbols from end of string.
Ex: ban, banana, ϵ are prefixes of banana.

③ Suffix of string - string obtained by removing zero or more symbols from beginning of string.
Ex: nana, banana, ϵ are suffixes of banana

④ Substring - string obtained by deleting any prefix and any suffix from string.
Ex: banana, nan, ϵ are substrings of banana

⑤ Proper prefix - prefixes, which is not ϵ or equal to string
Ex: ban, banan

⑥ Proper suffix - suffix which is not ϵ or equal to string itself
Ex: anana, na

⑦ Proper substring - substring from string which is not ϵ or the string itself
Ex: anan, banan, anana

Subsequence - string formed by deleting zero or more not necessarily consecutive positions of string.
Ex: baan, anaa -- for banana

→ Operations on Languages:

operation	definition & notation
Union of L & M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L & M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

→ Regular Definition:

For some alphabet set Σ, sequence of regular definition;

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
⋮
$d_n \rightarrow r_n$ where

1) each d_i is new symbol (not in Σ & other d_i)
2) r_i is regular expression over $\Sigma \cup \{d_1, d_2 \text{ -- } d_{i-1}\}$

Ex ① C identifiers:

letter → A|B|---|Z|a|b|--|z|_
digit → 0|1|---|9|
id → letter (letter|digit)*

Ex ② unsigned numbers:

digit → 0|1|---|9|
digits → digit digit*
optional fraction → . digits | ε
optional exponent → (E(+|-|ε) digits) | ε
number → digits optional fraction optional exponent

Algebraic Laws for Regular Expressions

LAW	DESCRIPTION
1 $r \mid s = s \mid r$	$\mid$ is commutative
2 $r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
3 $r(st) = (rs)t$	Concatenation is associative
4 $r(s \mid t) = rs \mid rt$; $(s \mid t)r = sr \mid tr$	concatenation distribute over $\mid$
5 $\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
6 $r* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
7 $r** = r*$	* is idempotent

→ Recognition of Tokens:

stm → if expr then stmt | if expr then stmt else stmt | ϵ

expr → term relop term | term

term → id | number

where,

number → $[0-9]^+ (.[0-9]^+)? (E[+-]?[0-9]^+)?$

id → $[\_a\text{-}zA\text{-}Z][\_a\text{-}zA\text{-}Z0\text{-}9]^+$

if → if

then → then

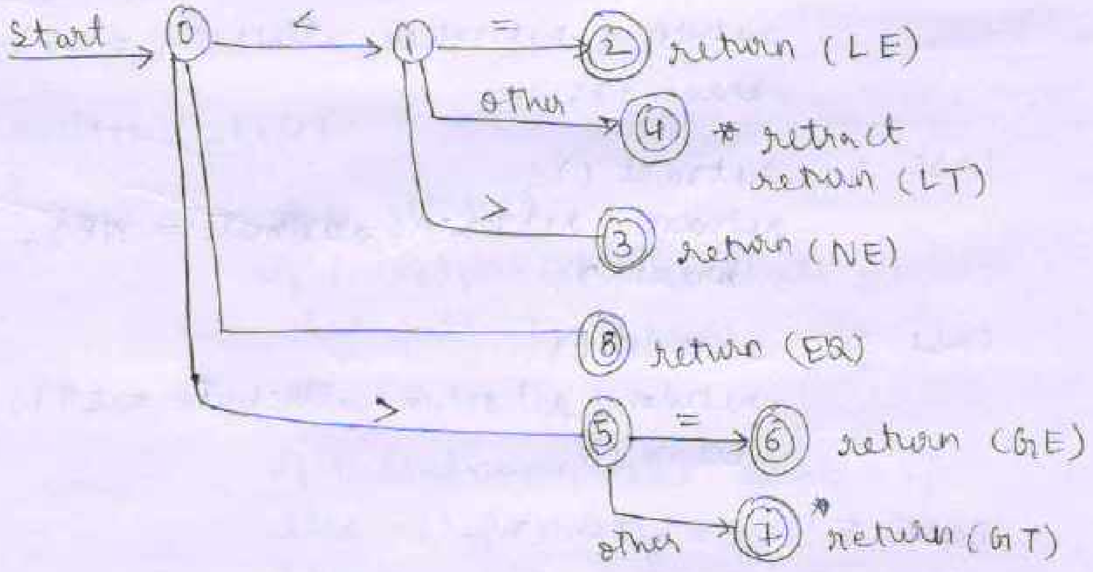
else → else

relop → < | > | <= | >= | = | <>

white space : ws → (blank | tab | newline)$^+$

→ Transition diagram:

(1) For relational operator, regular definition is
relop → < | > | <= | >= | = | <>



code:

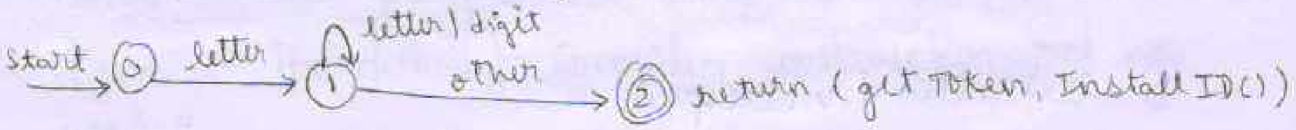
```
state = 0 ;
TOKEN getrelop ()
{
  Token retTOKen = new (relop);
  while (1)
  {
   switch (state)
   {
     case 0 : c = newchar () ;   or   c = getch ();
     if ( c = = '<') state = 1;
     else if ( c = = '>'; state = 5 ;
     else if ( c = = '=') state = 8;
      else fail (); break;

   case 1 : c = getch () ;
    if ( c = = '=') state = 2;
    else if ( c = = '>') state = 3;
    else if ( c = = '...') state = 4; // other
     else fail (); break;

    case 2 : retract () ;
             return ( retTOKen. attribute = LE);
              break ();

     case 3 : retract ();
              return (retTOKen. attribute = NE);
                break();

     case 4 : retract ();
               return ( retToken. attribute = LT);
                break ();

      case 5 : c = getch ();
       if ( c == '=') state = 6;
       else if ( c = = '...') state = 7 ;  // other
        else fail (); break();
case 6 : retract ();
         return ( retToken . attribute = GE);
          break ;
case 7 : retract ();
         return ( retToken . attribute = GT);
         break ;
case 8 : retract ();
         return ( retToken . attribute = EQ) ;
          break ;
       }
   }
}
```

② for identifier

letter → [a-zA-Z_]

digit → [0-9]

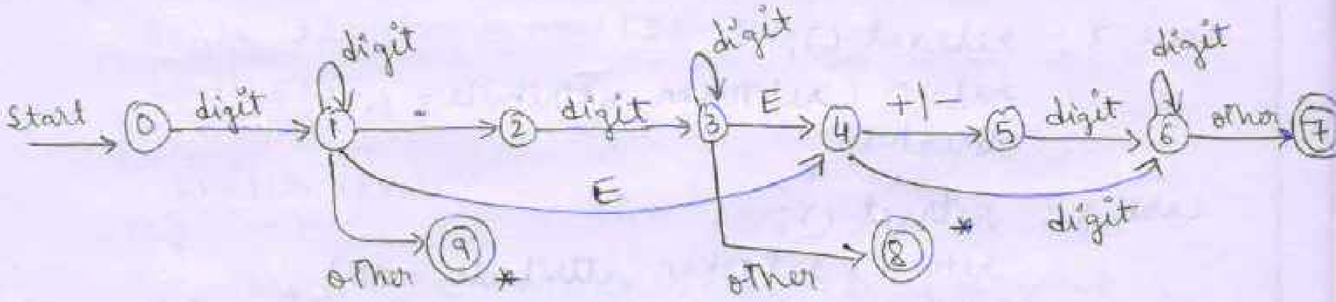
id → letter (letter | digit)*



```
state = 0 ;
 for ( ; ; )
{
  switch ( state)
  {
   case 0 : ch = getch();
              if ( isalpha (ch)) state = 1 ;
              else fail (); break ;

   case 1 : ch = getch () ;
              if ( isalnum (ch)) state = 1 ;
              else state = 2 ;
              break ;

   case 2 : retract () ;
              InstallID () ;
              return ( retToken ) ;
              break ;
      }
}
```

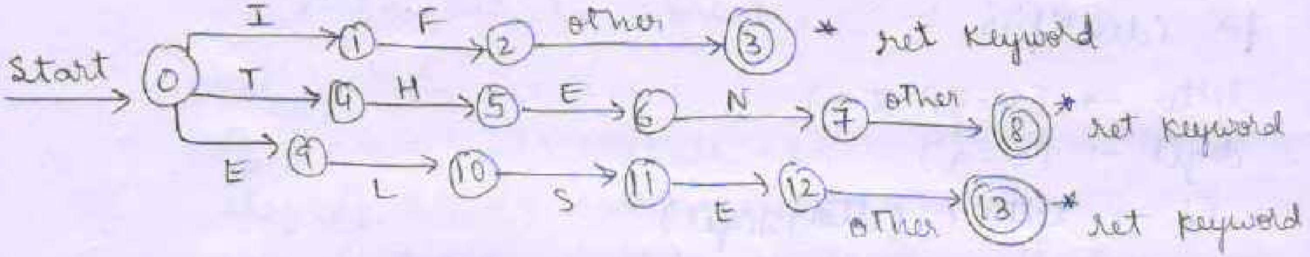
③ unsigned number

digit → [0-9]



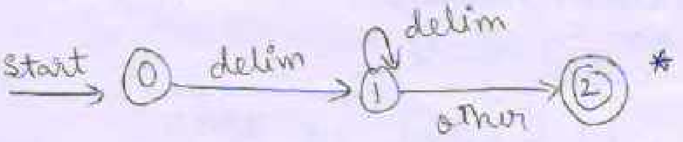
④ Keywords

Ex: keyword → IF | THEN | ELSE



⑤ Delimiter | Whitespace

delim → space | tab | newline



chapter-2 Lexical Analysis

Questions

1. Explain Lexical Analysis in detail with block diagra
2. Explain the reason for separating analysis phase of compiler for lexical Analysis and syntax Analysis
3. what do you mean by lexical errors? How do we recover them
4. Define the terms token, pattern, lexeme with an example
5. why 2-buffer technique is used in LA? write an algorithm for lookahead code with sentinel.
6. Give the formal definitions for operations on languages with notations
7. list the algaibraic laws for Regular Expression.
8. Define the term prefix, suffix, substring, proper prefix, proper suffix, proper substring, subsequence with an example.
9. write regular definition for Identifiers, unsigned numbers, keywords, relational operators and whitespace
10. Draw the transition diagram for relop, identifiers, unsigned number, keywords and white spaces.

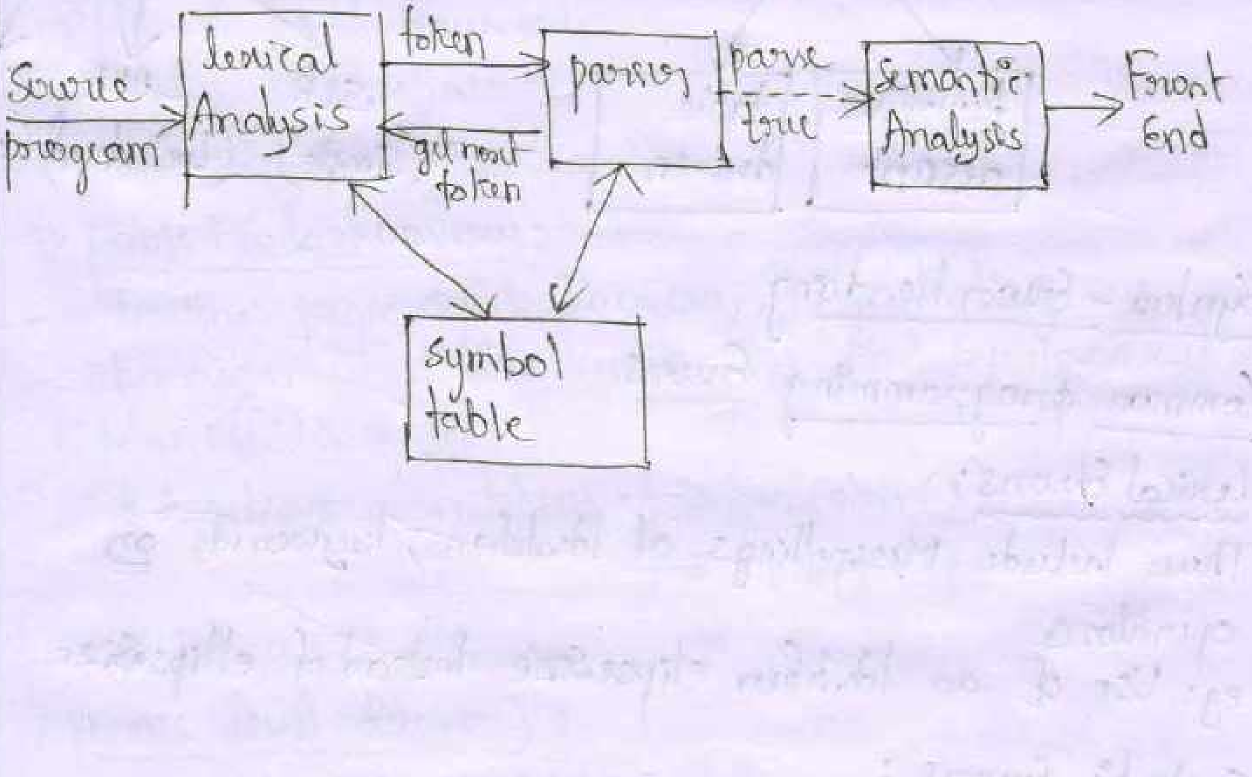
UNIT-2 Syntax Analysis -I

Topics

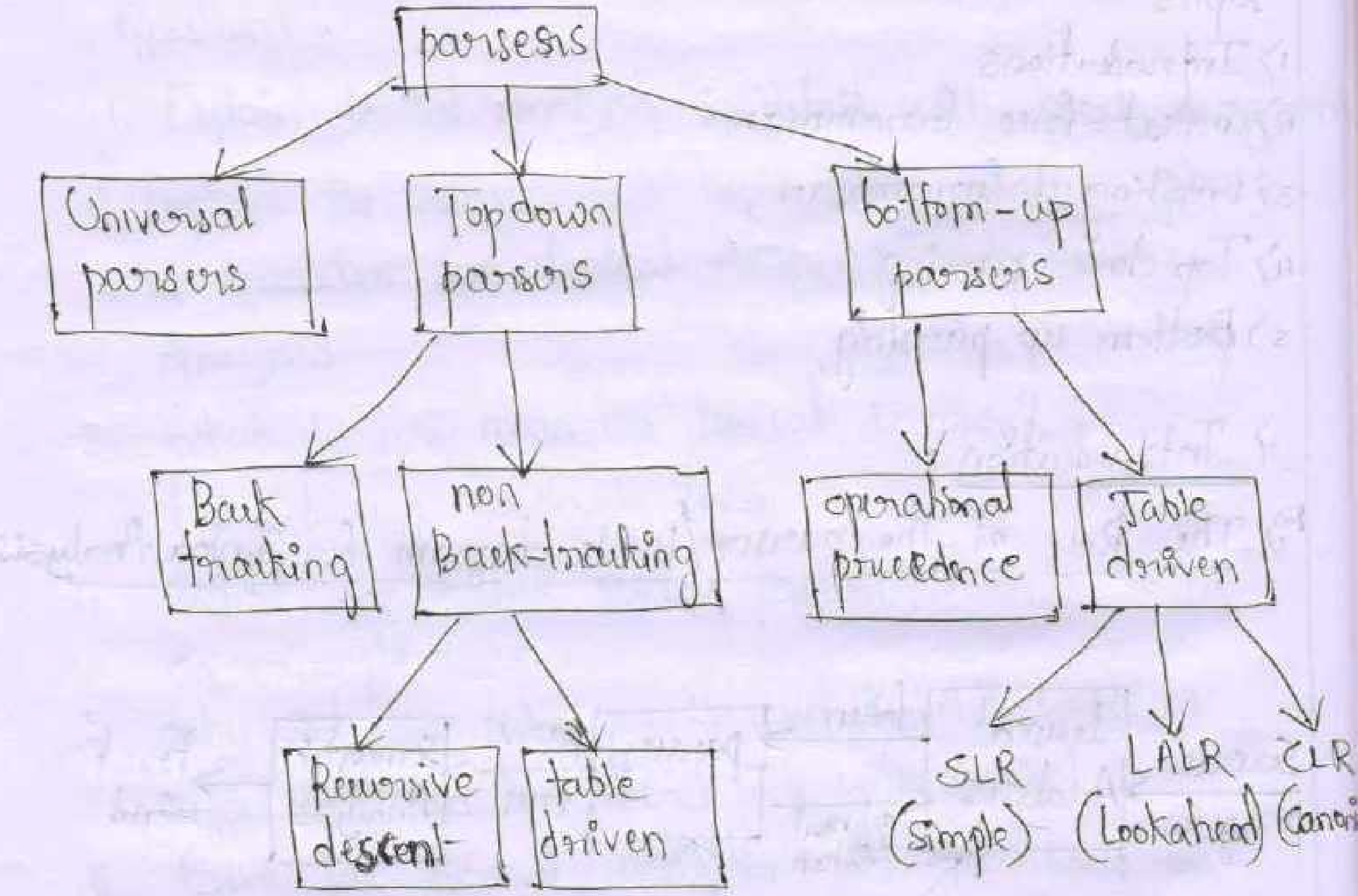
1) Introductions
2) Context-free Grammams
3) writing a Grammar
4) Top down parsing
5) Bottom up parising

1) Introdution :

i) The Role of the parser/Block diagram for Syntax Analysis.



The General types of parses for grammers



Syntax - Error Handling

Common programming Errors

i) Lexical Errors:

These include Misspellings of Identifiers, keywords or operators

eg: Use of an Identifier elipsesize instead of ellipseSize

ii) Syntactic Errors:

These errors include misplaced Semicolon/Extra or missing braces;

iii) Semantic Errors

There Include type mismatches b/n operators and operands.

An example: return statement in a Java method with result type Void

iv) Logical Errors:

Can be Anything from Incorrect reasoning on the part of the programmer

eg: Using '=' instead of '==' in C programming.

Error Recovery Techniques:

i) panic Mode Recovery

ii) phrase level Recovery

iii) Error productions

iv) Global Corrections

i) panic Mode Recovery:

→ In the panic Mode Recovery, keep deleting one character at a time untill we find Synchronization tokens((;) and(}))

* Synchronization tokens → Semicolon (;)
→ Epelog (})

eg: int a, ; //Error

ii) phrase level Recovery:

→ It Include Insert, delete, update

→ On discovering an error, a parser may perform local Correction on the remaining i/p : that is, It may replace a prefix of the remaining i/p by some string that allows the parser to Continue.

→ It Includes → Replacing a Comma, by a Semicolon
→ delete an extra Semicolon
→ Inserting a Missing Semicolon

eg: int a, ;
Replace , by ; and delete extra ;

iii) Error productions :

→ By anticipating Common errors that might be encountered, we Can augment the grammer for the language at hand with productions that generate Incorrect Constructs

→ A parser Constructed from a grammer augmented by these error productions detects the anticipated errors when an error productions is used during parsing

iv) Global Corrections :

→ Ideally, we would like a Compiler to make as few changes as possible in processing an incorrect input string. There are algorithm for choosing a minimal Sequence of changes to obtain a globally least Cost Correction.

→ Given an Incorrect input-string x and Grammar G, these algorithms will find a parse tree for a related string y, Such that the number of Insertions, deletions, and changes of tokens required to transform x into y is as small as possible

drawback of Global Corrections :

→ These generally too Costly to Implement in terms of time and space, So these are Currently only a theoretical Interest.

CONTEXT FREE GRAMMARS

defn: Context free grammer is a 4-tuple difined as (V,T,P,S), where

V: Set of Variable

T: set of Terminals

P: set of production

S is the start Symbols

Differentiate b/n CFG and RE

CFG	RE
1. It is the part of the Syntax Analysis	1. It is the part of the lexical Analysis
2. Usefull for describing nested gramatical structure Such as balanced paranthesis and So, on	2. Usefull for describing the structure of construct / lexical construct such as Identifiers, keywords etc
3. CFG's are Combined using pushdown automata	3. Regular Expressions are Combined using Finite Automata
4. CFG Can keep track of no. of Symbols Seen so far	4. RE Cannot keep track of no. of symbols Seen So far
5. Every CFG need not be RE	5. Every RE is a CFG
6. CFG are more powerfull	6. RE are less powerfull as Compared to CFG
7. Eg: letter → [A-Z a-z _] digit → [0-9] id → letter(letter/digit)	7. Eg: [a-z A-Z 0-9][0-9]*

Q) For the fallowing CFG

a. Give the LMD for the string

b. Give the RMD for the string

c. Give the parse tree for the string

d. Is the grammar ambigous/Unambigous? Justify

1. $S \rightarrow SS+ \mid SS* \mid a \Rightarrow aa+a*$
2. $S \rightarrow 0S1 \mid 01 \Rightarrow 000111$
3. $S \rightarrow +SS \mid *SS \mid a \Rightarrow +*aaa$
4. $S \rightarrow S(S)S \mid \epsilon \Rightarrow (()())$
5. $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a \Rightarrow (a+a)*a$
6. $S \rightarrow (L) \mid a \quad L \rightarrow L,S \mid S \Rightarrow (a,a)$
7. $S \rightarrow aSbS \mid bSaS \mid \varepsilon \Rightarrow aabbab$
8. bexpr → bexpr <u>or</u> bterm | bterm
   bterm → bterm and bfactor | bfactor
   bfactor → not bfactor | (bexpr) | true | false
   ⇒ not (true | false)
9. $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id \Rightarrow id+id+id$
10. $S \rightarrow iEtS \mid iEtSeS \mid a \Rightarrow$ If E_1 then If E_2 then S_1 else S_2
11. $R \rightarrow R'|'R \mid RR \mid R* \mid (R) \mid a \mid b \mid c \Rightarrow a|b*c$

1⟩ $S \rightarrow SS+ \mid SS* \mid a \Rightarrow aa+a*$

$S \xrightarrow{lm} SS*$ $\quad\quad$ $S \xrightarrow{rm} SS*$

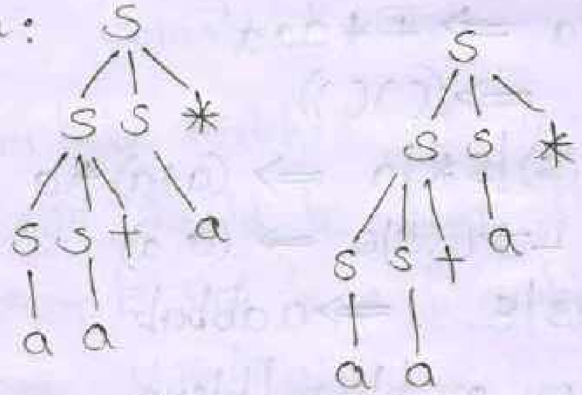
$\Rightarrow SS+S*$ $\quad\quad$ $\Rightarrow Sa*$

$\Rightarrow aS+S*$ $\quad\quad$ $\Rightarrow SS+a*$

$\Rightarrow aa+S*$ $\quad\quad$ $\Rightarrow Sa+a*$

$\Rightarrow aa+a*$ $\quad\quad$ $\Rightarrow aa+a*$

parse tree:



The Grammar Is Unambigous because It has only one LMD and One RMD

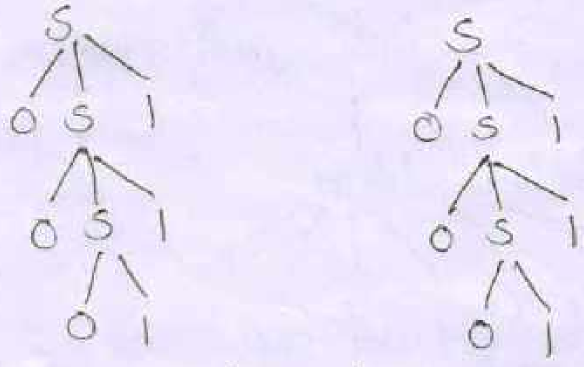
2⟩ $S \rightarrow 0S1 \mid 01 \Rightarrow 000111$

$S \xrightarrow{lm} 0\underline{S}1$ $\quad\quad$ $S \xrightarrow{rm} 0S1$

$\Rightarrow 00\underline{S}11$ $\quad\quad$ $\Rightarrow 00S11$

$\Rightarrow 00S111$ $\quad\quad$ $\Rightarrow 000111$

parse tree:



The Grammar Is Unambigous because It has only one LMD and Only one RMD

3>

3> $S \rightarrow +SS \mid *SS / a \implies +*aaa$

$S \xrightarrow{lm} +SS$

$\xrightarrow{lm} +*SSS$

$\implies +*aSS$

$\implies +*aaS$

$\implies +*aaa$

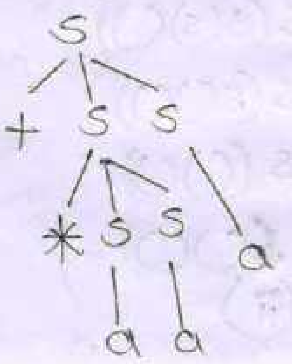
$S \xrightarrow{rm} +SS$

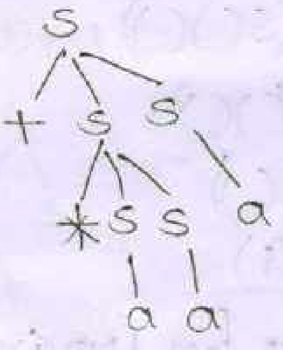
$\xrightarrow{rm} +*SSS$

$\implies +*SSa$

$\implies +*Saa$

$\implies +*aaa$





The Grammar Is Unambigous
because It has only one LMD and one RMD

4> $S \rightarrow S(S)S \mid \varepsilon \implies (()())$

$S \xrightarrow{lm} S(S)S$

$\implies (\underline{S})S$

$\implies (S(S)S)S$

$\implies ((S)\underline{S})S$

$\implies (()S(S)S)S$

$\implies (()(S)S)S$

$\implies (()()S)S$

$\implies (()())S$

$\implies (()())$

(i)

$S \xrightarrow{rm} \underline{S}(S)S$

$\xrightarrow{lm} (S)S$

$\xrightarrow{lm} (\underline{S}(S)S)S$

$\xrightarrow{lm} (S(S)S(S)S)S$

$\implies ((S)S(S)S)S$

$\implies (()S(S)S)S$

$\implies (()(S)S)S$

$\implies (()()S)S$

$\implies (()())S$

$\implies (()())$

(ii)

RMD:

$S \xrightarrow{rm} S(S)S$

$\Rightarrow S(S)$

$\Rightarrow S(S(S)S)$

$\Rightarrow S(S(S)S(S))$

$\Rightarrow S(S(S)S())$

$\Rightarrow S(S(S)())$

$\Rightarrow S(S()())$

$\Rightarrow S(()())$

$\Rightarrow (()())$

(i)

$S \xrightarrow{rm} S(S)\underline{S}$

$\Rightarrow S(S)$

$\Rightarrow S(S(S)S)$

$\Rightarrow S(S(S))$

$\Rightarrow S(\underline{S}())$

$\Rightarrow S(S(S)S())$

$\Rightarrow S(S(S)())$

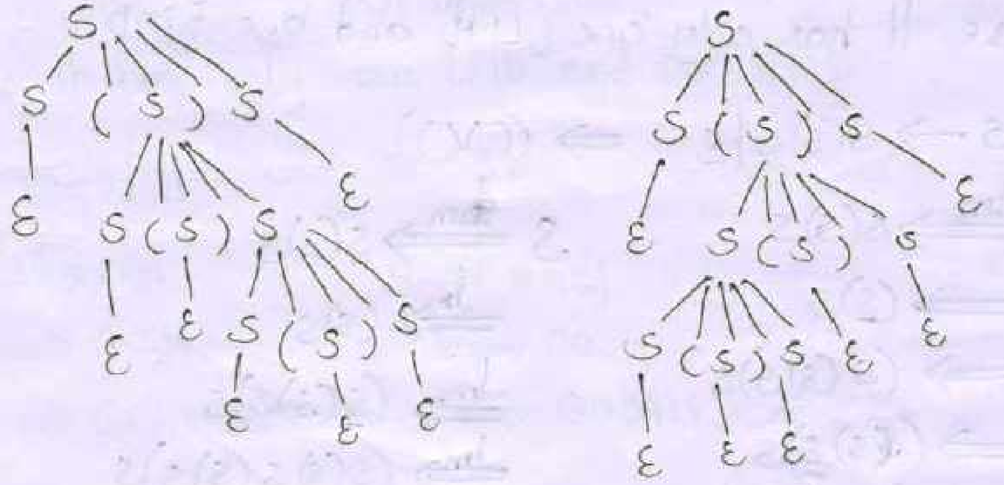
$\Rightarrow S(S()())$

$\Rightarrow S(()())$

$\Rightarrow$ ~~S~~$(()())$

(ii)

parse tree for LMD (i) and (ii)



The Grammar Is ambigous Since It has 2 LMD and 2 RMD

5) $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a \Rightarrow (a+a)*a$

$S \xrightarrow{lm} SS$ $\qquad\qquad$ $S \xrightarrow{rm} SS$

$\Rightarrow S*S$ $\qquad\qquad$ $\Rightarrow S*S$

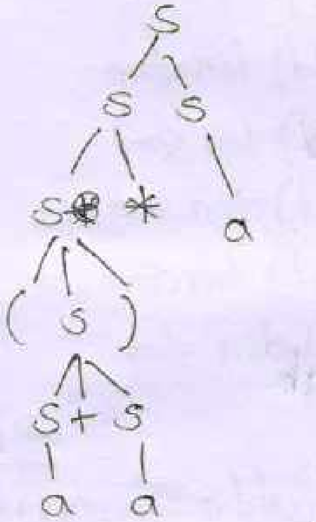
$\Rightarrow (S)*S$ $\qquad\qquad$ $\Rightarrow S*a$

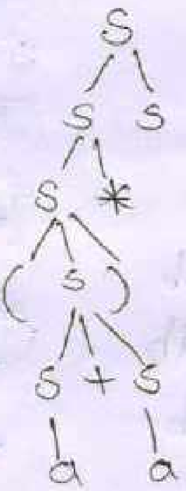
$\Rightarrow (S+S)*S$ $\qquad\qquad$ $\Rightarrow (S)*a$

$\Rightarrow (a+S)*S$ $\qquad\qquad$ $\Rightarrow (S+S)*a$

$\Rightarrow (a+a)*S$ $\qquad\qquad$ $\Rightarrow (S+a)*a$

$\Rightarrow (a+a)*a$ $\qquad\qquad$ $\Rightarrow (a+a)*a$





The Grammar Is Unambigous
because It has only one RMD and LMD

6) $S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S \Rightarrow (a.a)$

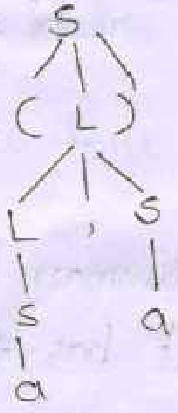
$S \xrightarrow{lm} (L)$ $\qquad\qquad$ $S \xrightarrow{rm} (L)$

$\Rightarrow (L,S)$ $\qquad\qquad$ $\Rightarrow (L,S)$

$\Rightarrow (S,S)$ $\qquad\qquad$ $\Rightarrow (L,a)$

$\Rightarrow (a.S)$ $\qquad\qquad$ $\Rightarrow (S,a)$

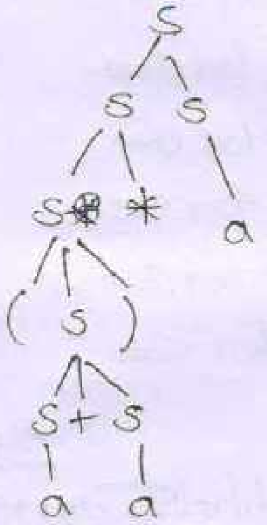
$\Rightarrow (a.a)$ $\qquad\qquad$ $\Rightarrow (a.a)$

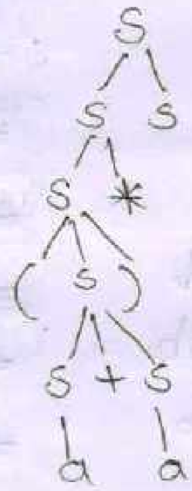


The Grammar Is Unambigous
because It has only one LMD and RMD

5) $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a \Rightarrow (a+a)*a$

$S \xrightarrow{lm} SS$	$S \xrightarrow{rm} SS$
$\Rightarrow S*S$	$\Rightarrow S*S$
$\Rightarrow (S)*S$	$\Rightarrow S*a$
$\Rightarrow (S+S)*S$	$\Rightarrow (S)*a$
$\Rightarrow (a+S)*S$	$\Rightarrow (S+S)*a$
$\Rightarrow (a+a)*S$	$\Rightarrow (S+a)*a$
$\Rightarrow (a+a)*a$	$\Rightarrow (a+a)*a$

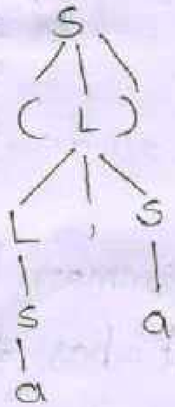




The Grammar Is Unambigous because It has only one RMD and LMD

6) $S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S \Rightarrow (a,a)$

$S \xrightarrow{lm} (L)$	$S \xrightarrow{rm} (L)$
$\Rightarrow (L,S)$	$\Rightarrow (L,S)$
$\Rightarrow (S,S)$	$\Rightarrow (L,a)$
$\Rightarrow (a.S)$	$\Rightarrow (S,a)$
$\Rightarrow (a,a)$	$\Rightarrow (a,a)$



The Grammar Is Unambigous because It has only one LMD and RMD

7) $S \rightarrow aSbS \mid bSaS \mid \varepsilon \Rightarrow aabbab$

LMD:

$S \xrightarrow{lm} a\underline{S}bS$
$\Rightarrow aa\underline{S}bSbS$
$\Rightarrow aab\underline{S}bS$
$\Rightarrow aabbSaSbS$
$\Rightarrow aabbaSbS$
$\Rightarrow aabbabS$
$\Rightarrow aabbab$

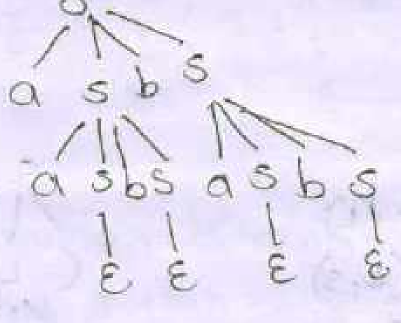
$S \xrightarrow{lm} aSbS$
$\Rightarrow aa\cancel{S}bSbS$
$\Rightarrow aabSbS$
$\Rightarrow aabSbS$
$\Rightarrow aabbaSbS$
$\Rightarrow aabbabS$
$\Rightarrow aabbab$

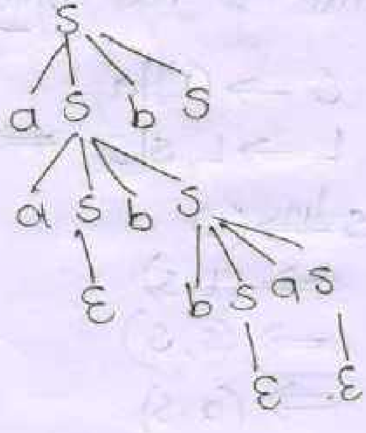
RMD:

$S \xrightarrow{rm} aSbS$
$S \Rightarrow aSb$
$\Rightarrow aaSbSb$
$\Rightarrow aaSbbSaSb$
$\Rightarrow aaSbbSa\cancel{S}b$
$\Rightarrow aaSbbSab$
$\Rightarrow aaSbbab$
$\Rightarrow aabbab$

$S \xrightarrow{rm} aSbS$
$\xrightarrow{rm} aSbaSbS$
$\xrightarrow{rm} aSbaSb$
$\Rightarrow aSbab$
$\Rightarrow aaSbSbab$
$\Rightarrow aaSbbab$
$\Rightarrow aabbab$

parse tree:





The grammar is ambigous
Since It has 2 LMD and 2 RMD

8) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

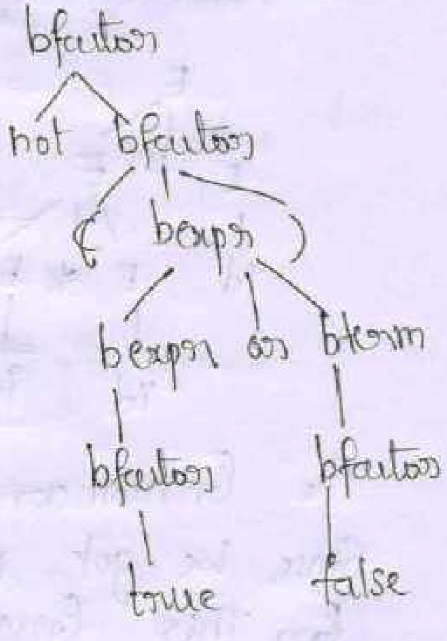
not (true or false)

LMD:

bexpr $\xrightarrow{lm}$ bterm
⟹ bfactor
⟹ not bfactor
⟹ not (bexpr)
⟹ not (bexpr or bterm)
⟹ not (bterm or bterm)
⟹ not (bfactor or bterm)
⟹ not (true or bterm)
⟹ not (true or bfactor)
⟹ not (true or false)

RMD:

bexpr $\xrightarrow{rm}$ bterm ~~$~~
⟹ bfactor
⟹ not bfactor
⟹ not (bexpr)
⟹ not (bexpr or bterm)
⟹ not (bexpr or bfactor)
⟹ not (bexpr or false)
⟹ not (bterm or false)
⟹ not (bfactor or false)
⟹ not (true or false)



9> $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$

$\Rightarrow id + id * id$

<u>LMD</u>

$E \xrightarrow{lm} E+E \; (E \rightarrow E*E)$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

$E \xrightarrow{lm} E*E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

<u>RMD</u>

$E \xrightarrow[rm]{} E+E$

$\Rightarrow E+E*E$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

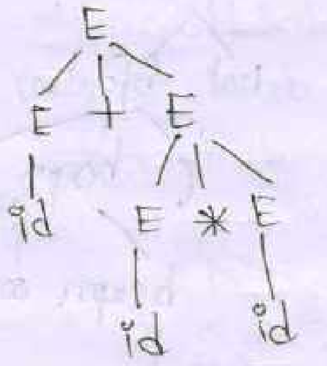
$E \xrightarrow{rm} E*E$

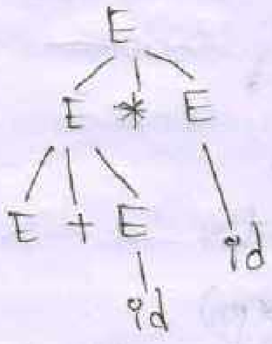
$\Rightarrow E*id$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$



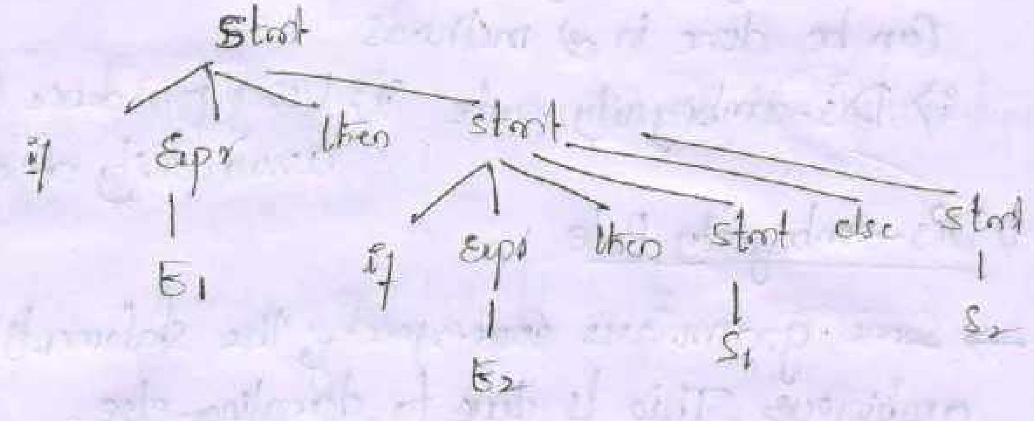


The Grammar Is Ambigous
Since we got more than 1 LMD and more than 1 RMD for This Grammar

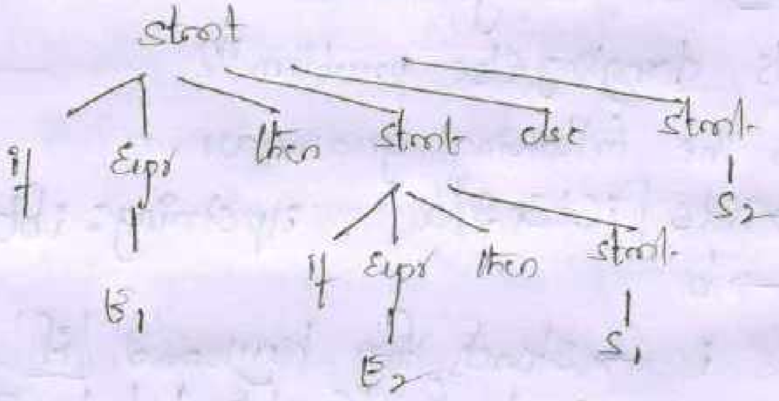
10. $S \rightarrow iEtS / iEtSeS / a \quad E \rightarrow b$

$\Rightarrow$ If E_1 Then If E_2 then S_1 Else S_2

1)



2)



The given grammar is ambiguous ∵ it has two different parse trees

ii) $R \rightarrow R|R \mid RR \mid R* \mid (R) \mid a \mid b \mid c \quad \Rightarrow a|b*c$

LMD 1

$R \rightarrow R|R$
$\stackrel{lm}{\Rightarrow} a|R$
$\stackrel{lm}{\Rightarrow} a|RR$
$\stackrel{lm}{\Rightarrow} a|R*R$
$\stackrel{lm}{\Rightarrow} a|b*R$
$\Rightarrow a|b*c$

LMD 2

$R \rightarrow RR$
$\stackrel{lm}{\Rightarrow} R|RR$
$\Rightarrow a|RR$
$\Rightarrow a|R*R$
$\Rightarrow a|b*R$
$\Rightarrow a|b*c$

The given grammar is ambiguous ∵ it has LMDs.

Eliminating ambiguity

Can be done in 2 methods

i) Dis-ambiguity rule ii) Using precedence & associativity of operators

i) Dis-ambigouty Rule

→ Some grammars ~~coorresponding~~ the statements are ambigous, This Is due to dangling-else.

The dangling else problem can be eliminated and thus ambiguity of the grammar can also be eliminated

Q) what Is dangling else problem ??

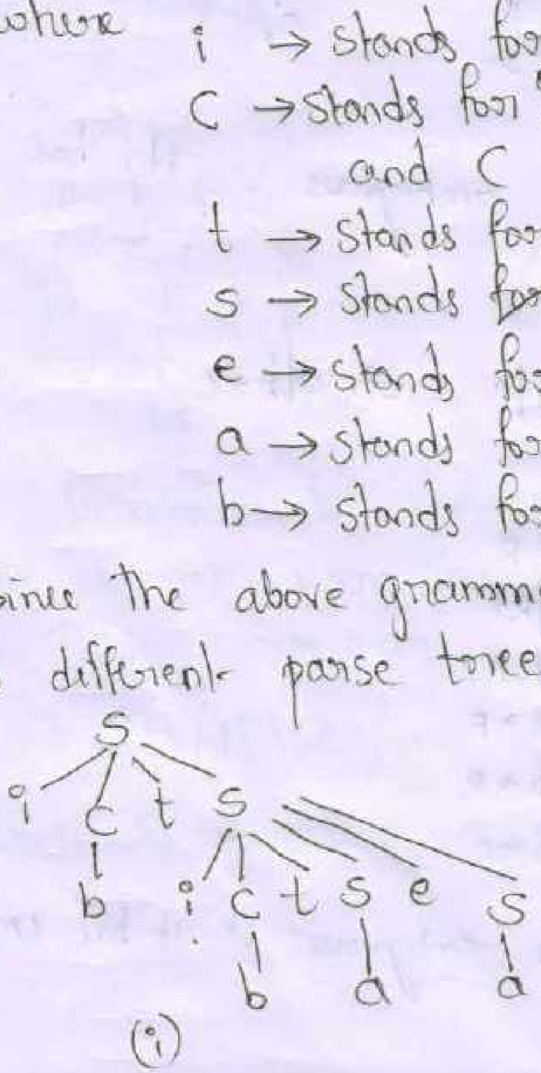
→ Consider the fallowing grammar

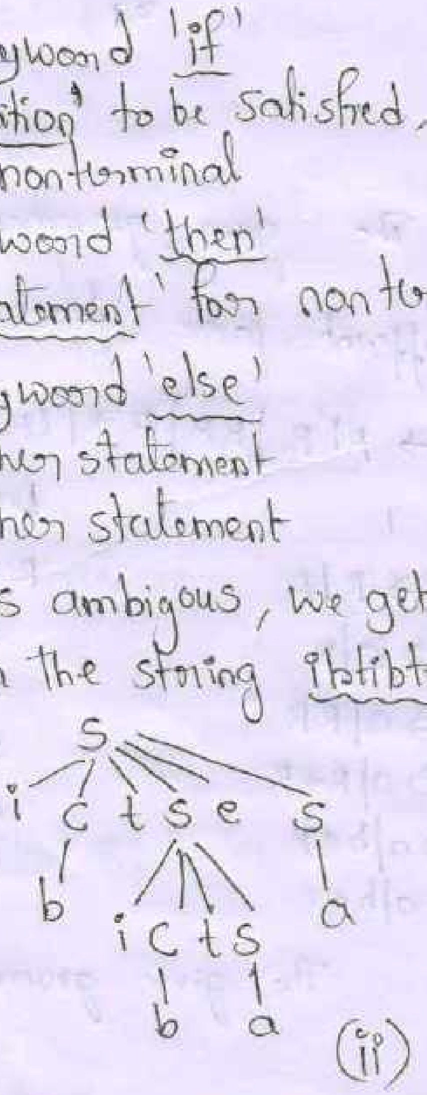
S→iCtS | iCtSeS | a   i/p string: ibtibtaea

C→b

where
- i → stands for Keyword 'if'
- C → stands for 'Condition' to be satisfied, and C is nonterminal
- t → stands for keyword 'then'
- S → stands ~~for~~ 'Statement' for nonterminal
- e → stands for keyword 'else'
- a → stands for other statement
- b → stands for other statement

Since the above grammar Is ambigous, we get two different parse tree for the string ibtibtaea





Since there are 2 parse tree for the Same storing ibtibtaea the given grammar is ambigous.
obsowe the fallowing points

→The first parse tree associates else with 2nd statement

→The Second parsetree associates else with first If stmt

The ambiguity wheather to associate else with first If statement / Second If-statements Is called dangling-else problem.

Eg> Q> Eliminate ambiguity from the fallowing ambigous grammar:

$S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$

→ In all programming languages when If-statements are nested, the first parse tree is preferred. So, the general rule is "Match each else with closest unmatched then". This rule Can be directly incorpporated into grammar and ambiguity Can be eliminated as shown below:

step1> The matched stmt M is an If-else statement where the Statement S before else and after else keyword is matched. This can be expressed as:

$M \rightarrow iCtMeM$

step 2: An Unmatched statement U is the one Consisting of:

a) Simple If-statement where the statement S is matched statement / Unmatched statement. ∴ The equivalent production is → $U \rightarrow iCtS$

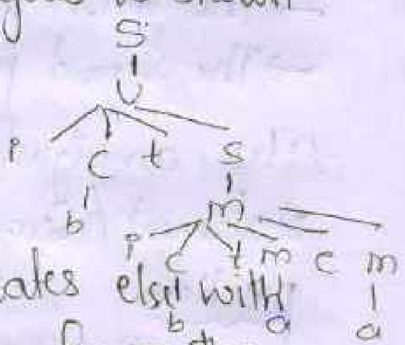
b> If-else statement where the statement before else is matched and statement after else is Unmatched. ∴ The equivalent production is: $U \rightarrow iCtMeU$

step 3: The matched statement M and Un-matched statement U can obtained using the Statement S as shown below:

$S \rightarrow M|U$

So, the final grammar which is un-ambigous is shown below:

$S \rightarrow M|U$
$M \rightarrow iCtMeM/a$
$U \rightarrow iCtS$
$U \rightarrow iCtMeU \quad c \rightarrow b$



observe that the above grammar associates else with closest then and eliminates ambiguity from the grammar

Eliminating ambiguity using precedence and Associativity

This method is explained using the fallowing example:

eg. Q) Convert the ambigous grammar into Unambigous grammar:

$E \rightarrow E*E|E-E$
$E \rightarrow E^E|E/E$
$E \rightarrow E+E$
$E \rightarrow (E)|id$

The grammar can be converted into unambigous grammar using the precednce of operators as ~~shown~~ well as associativity operators as shown below:

step 1: Arrange the operators in increasing order of The precedence along with the associativity as shown below:

operators	Associativity	non-terminal used
+, -	LEFT	E
*, /	LEFT	T
^	RIGHT	P

Since there are three levels of precedence, we associate three non-terminals : E, T and P. Also an extra non-terminal F, generating basic units in an arithmatic expression

Step 2: The basic units in expression are id (identifier) and paranthesized expressions. the production corresponding to this can be written as:

$$F \rightarrow (E) | id$$

Step3: The next highest priority operator is ^ and it is right associative. So, the production must start from the non-terminal P and it should have right recursion as shown below:

$$P \rightarrow F \wedge P | F$$

Step4: The next highest priority operators are * and / and they are left associative, So, the production must start from the non-terminal T and it should have left recursion as shown below:

$$T \rightarrow T * P | T / P | P$$

Step5: The next highest priority operators are + and - and they are left associativity. So, the production must start from the non-terminal E and it should have left recursion as shown below:

$$E \rightarrow E + T | E - T | T$$

Step6: The final grammar which is unambigous grammar can be written as shown below:

$$E \rightarrow E + T / E - T / T$$
$$T \rightarrow T * P | T / P | P$$
$$P \rightarrow F \wedge P | F$$
$$F \rightarrow (E) | id$$

Q) Convert the fallowing Ambigous grammar into Unambigous grammar by Considering * and - operators lowest priority and they are left associative, / and + operators have The highest priority and are right associatiative and ^ operator have the highest priority and are right associativity and ^ operator has precedence in between and It is left associativity.

$E \rightarrow E+E$
$E \rightarrow E-E$
$E \rightarrow E^{\wedge}E$
$E \rightarrow E*E$
$E \rightarrow E/E$
$E \rightarrow (E)|id$

→ The grammar Can be Converted Into unambigous grammar using the precedence of operators as well as associativity operators as shown below:

step1: Arrange the operators in Increasing order of the precedence along with associativity as shown below:

precedence	operators	Associativity	non-terminal used
(lowest)	*, -	LEFT	E
	^	LEFT	P
(highest)	/, +	RIGHT	T

Since there are three levels of precedence we associate three non-terminals: E, P and T. Also use an extra non-terminal F generating basic units in an arith-matic expression

step2: The basic units in expression are id (identifier) and parenthesized expressions. The production corresponding to this can be written as:

$F \rightarrow (E)|id$

step3: The next highest priority operators are + and / They are right associative, So, The production must start from the nonterminal T and it should be right recursive in RHS of the production as shown below:

$$T \rightarrow F+T \mid F/T \mid F$$

step4: The next highest priority operator is ^ and It is left associative. So, the production must start from the non-terminal P and It should be left recursive in RHS of the production as shown below:

$$P \rightarrow P \wedge T \mid T$$

step5: The next highest priority operators are * and - and They are left associative. So, the production must start from the non-terminal E and it should be left recursive in RHS of the production as shown below;

~~$P \rightarrow P \wedge T \mid T$~~ $E \rightarrow E+P \mid E-P \mid P$

~~step6: The next highest priority operators are * and - and they~~

step6: The final grammar which is unambigous can be written as shown below:

$$E \rightarrow E+P \mid E-P \mid P$$
$$P \rightarrow P \wedge T \mid T$$
$$T \rightarrow F+T / F/T \mid F$$
$$F \rightarrow (E)/id$$

Q> Define Ambiguity ? show that the fallowing grammar is ambigous
$R \rightarrow R'|'R \,|\, RR \,|\, R* \,|\, (R) \,|\, a|b|c$ for input storing
$a/b*c$

Give an unambigous grammar for the above grammar such that precedence order from lowest to highest are Concatenation, $*$, $|$, $( )$, identifier and all are left to right associativity

Ans: The grammar Is said to be ambigous if It has more than one LMD | more than one RMP

or

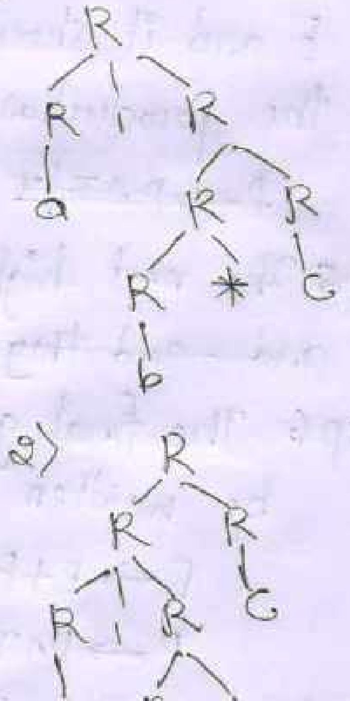
If there are two different parse trees for the input storing by applying LMD | by applying RMD

→ i/p storing: $a|b*c$

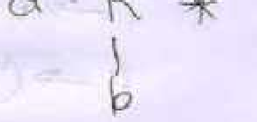
LMD 1	LMD 2
$R \rightarrow R'|'R$	$R \rightarrow RR$
$\overset{lm}{\Longrightarrow} a'|'R$	$\overset{lm}{\Longrightarrow} R'|'RR$
$\Rightarrow a'|'RR$	$\overset{lm}{\Longrightarrow} a'|'RR$
$\Rightarrow a'|'R*R$	$\overset{lm}{\Longrightarrow} a'|'R*R$
$\Rightarrow a'|'b*R$	$\Rightarrow a|b*R$
$\Rightarrow a'|'b*c$	$\Rightarrow a|b*c$

It has two LMD'S ∴ the given grammar is Unambigous

parse tree:



2)



→ Unambigous grammar

1. Arrange the operators in the ascending order with the precedence and associativity

operators	Associativity	non-terminal used
•	LEFT	R
*	LEFT	S
|	LEFT	T

2. The basic units in expression are (R) and a, b, c we use additional non-terminal U for generating those U→(R)|a|b|c

3. The next highest priority operator | and it is left associative. So the production must start from the non-terminal T and it must be left recursive is RHS of the production

   T→T'|'U|U

4. The next highest priority operator is * and is left associative. So the production must start from non-terminal S and it is a Unary operator

   S→T*|T

5. The next highest priority operator is Concatenation and is left associative. So the production must start from the non-terminal R and it should have left recursion as

   R→RS|S

Step6: The final grammar which is unambigous can be written as

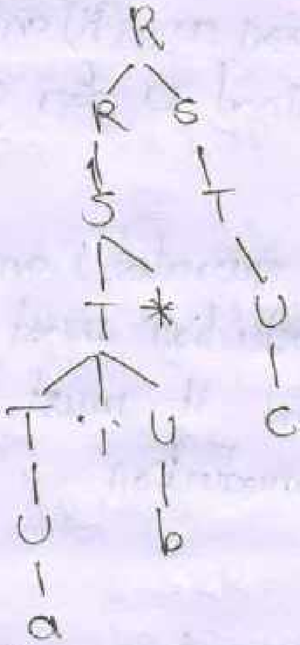
$R \rightarrow RS|S$

$S \rightarrow T*|T$

$T \rightarrow T'|'U|U$

$U \rightarrow (R)|a|b|c$

The parse tree for the i/p a|b*c is



Left Recursion :

defn:

If Non-terminal Symbol and the 1st Symbol of the production are Same, then It Is left recursion

General form: $A \rightarrow A\alpha_1 \,|\, A\alpha_2 \,|\, A\alpha_3 \,|\, \beta$

$$\Downarrow$$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha_1 A' \,|\, \alpha_2 A' \,|\, \varepsilon$$

Algorithm for left Recursion Elimination

Algorithm Left_Recursion

i/p: Grammar G with no cycles or ε-production

o/p: An equivalent grammar with no left-recursion

Method: Apply the algorithm to G. Note that the resulting non-left-recursive grammar may have ε-productions.

```
1. Arrange the non-terminals in Some order $A_1, A_2 \ldots A_n$
2. For(each i from 1 to n){
3.     for (each j from 1 to i-1){
4.     replace each production of the form $A_i \rightarrow A_j\gamma$ by
       the productions $A_i \rightarrow \delta_1\gamma \,|\, \delta_2\gamma \,|\, \ldots \,|\, \delta_k\gamma$
       where $A_j \rightarrow \delta_1 \,|\, \delta_2 \,|\, \ldots \,|\, \delta_k$ are all current
       $A_j$-productions
5   }
6. Eliminate the Immediate left recursion among the
   $A_i$-production
7 }
```

Example on Removing/Eliminating left Recursion

1. $E \rightarrow \underbrace{E}_{A}\underbrace{+T}_{\alpha} \,|\, \underbrace{T}_{\beta}$

$\Downarrow$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$ → Epsilon

[here $E \rightarrow E+T|T$ (both are Same) ∴ The grammar Contains left recursion]

2. $T \rightarrow \underbrace{T}_{A}\underbrace{*F}_{\alpha} \,|\, \underbrace{F}_{\beta}$ $\Rightarrow \boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' / \epsilon \end{array}}$

$\Downarrow$

$T \rightarrow FT'$

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

3. $S \rightarrow \underbrace{S}_{A}\underbrace{(S)S}_{\alpha} \,|\, \underbrace{\epsilon}_{\beta}$

$\Downarrow$

$S \rightarrow S'$

$S' \rightarrow (S)SS' / \epsilon$

4. $S \rightarrow SS+ \,|\, SS* \,|\, a$

$\Downarrow$

$S \rightarrow aS'$

$S' \rightarrow S+S' \,|\, S*S' \,|\, \epsilon$

5. $E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)/id$

$\Downarrow$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Left factoring

General form: $A \to \alpha\beta_1 \,|\, \alpha\beta_2 \,|\, \alpha\beta_3 \,---\, |\, \alpha\beta_n \,|\, \gamma$

$$\Downarrow$$

$$A \to \alpha A' \,|\, \gamma$$

$$A' \to \beta_1 \,|\, \beta_2 \,---\, |\, \beta_n$$

Algorithm for left factoring

Algorithm left_factoring

i/p: Grammar G

o/p: An equivalent left-factored grammar

method: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$ – i,e there is a nontrivial common prefix – replace all of the A-productions

$A \to \alpha\beta_1 \,|\, \alpha\beta_2 \,|\, ... \,|\, \alpha\beta_n \,|\, \gamma$,

where γ represents all alternatives that do not begin with α, by

$$A \to \alpha A' \,|\, \gamma$$

$$A' \to \beta_1 \,|\, \beta_2 \,|\, --- \,|\, \beta_n$$

Here A' is new non-terminal. Repeatedly apply this transformation untill no two alternatives for a non-terminal have a common prefix

Examples on left factoring

1. $s \rightarrow \underbrace{ss}_{\alpha}\underbrace{+}_{\beta_1} \mid \underbrace{ss}_{\alpha}\underbrace{*}_{\beta_2} / \underbrace{a}_{\gamma}$

$\Downarrow$

$s \rightarrow sss' / a$

$s' \rightarrow + / *$

The common terminals we have to take as α
and the remaining term we have to take as β_1, β_2 -- so on in each production

2. ~~$S \rightarrow 0S' \mid$~~ ~~$S' \rightarrow$~~ $S \rightarrow \underbrace{0}_{\alpha}\underbrace{S1}_{\beta_1} \mid \underbrace{0}_{\alpha}\underbrace{1}_{\beta_2}$

$\Downarrow$

$S \rightarrow 0S' / \varepsilon$

$S' \rightarrow S1 \mid 1$

3. $s \rightarrow iEts \mid iEtses \mid a$

$E \rightarrow b$

$\Downarrow$

$s \rightarrow \underbrace{iEts}_{\alpha}\underbrace{\ }_{\beta_1} \mid \underbrace{iEts}_{\alpha}\underbrace{es}_{\beta_2} / \underbrace{a}_{\gamma}$

$\Downarrow$

$s \rightarrow iEtss' / a$

$s' \rightarrow \varepsilon / es$

$E \rightarrow b$

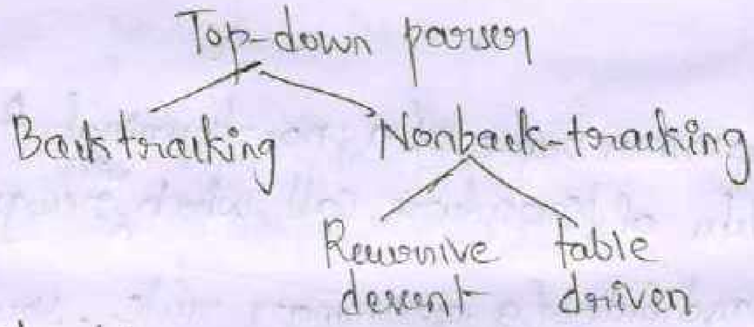
Since β_1 is empty we'll take β_1 as ε

Top down parsers :

→ dfn: is a parser of an i/p string of token by tracing out the steps in a left most derivation, it denies the string from the start symbol

→ It is termed as topdown because the parse tree is traversed in a preorder way that is from the root node to the leaf node

→ It has various types.



i) Backtracking

→ Backtracking tries different possibilities for parsing an i/p string by backing up an arbitrary amount in the i/p if any possibilities fails

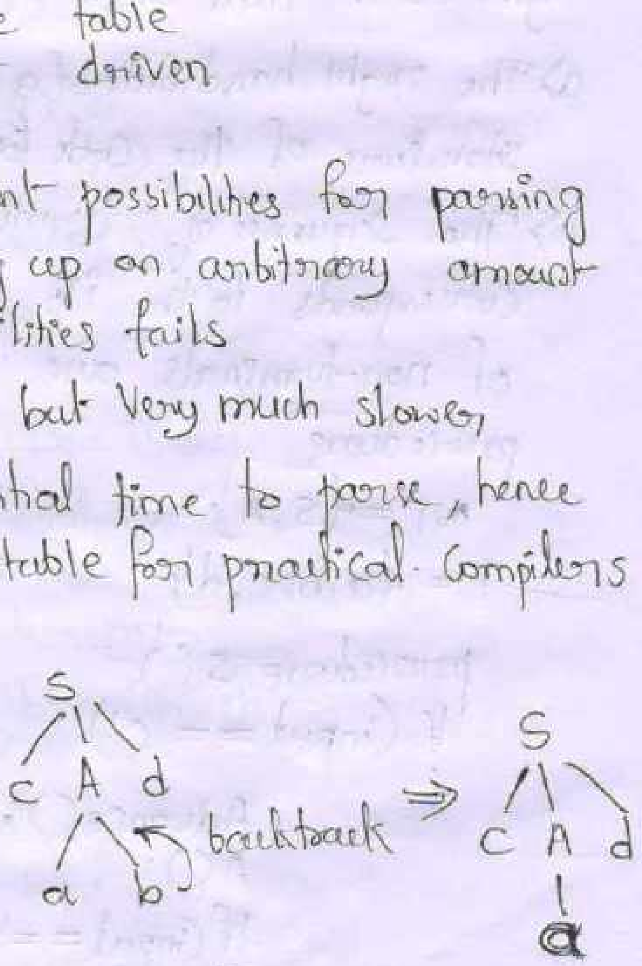
→ These are more powerfull but very much slower, as they require exponential time to parse, hence they're not ~~avaitable~~ suitable for practical compilers

eg: S → CAd

A → ab|a

i/p string → cabd

i/p string → cad



ii> Non-backtracking parsers:

i) Recursive descent   ii) Table driven

i) <u>Recursive descent parser:</u>

→ Recursive descent parsers are more Versatile & Suitable for handwritten parsers

→ It helps to study the method for parsing and serves as basis for topdown parses

$S \rightarrow cAd$

$A \rightarrow ab|a$

here, the grammar rule of a non-terminal A is give as a defn of procedure call which will recognize A

a> The right hand Side of a grammar rule, specifies the structure of the Code for the procedure.

b> The sequence of terminals on the right hand side corresponds to the i/p matches while the sequence of non-terminals are calls with the corresponding procedure

$NT = \{S, A\}$

$T = \{a, b, c, d\}$

```
procedure S()
if (input == 'c')
{
        Advance();
        A();
        if (input == 'd')
        {
```

```
        Advance();
        return(true);
    }
    else
    {
        return(false); }
    }
    else {
        return(false); }
    }

procedure A()
{
    isave=in-ptr;
    If (input=='a')
    {
        Advance();
        If (input=='b')
        {
            Advance();
            return (true);
        }
        else {
            in-ptr=isave
            If (input=='a')
            {
                Advance();
                return (true);
            }
        }
    return(false);
}
```

isave:

saves the i/p pointer position before each alternate production to facilitate backtracking whenever a terminal is encountered the i/p pointer

Advances the next position If alternate phase the in pointer retoraces to the previous position to trace the next alternate.

Advance():

advance is a procedure that is written to advance the i/p pointer to the next position on a successfull completion of the parsing action the parser returns a true value

drawbacks of Recursive descent parsings

1. left recursion → It has the ~~procedure~~ production of the form $A \rightarrow A\alpha$

the parsers goes into Infinite loop

eg: $A \rightarrow Ab|a$

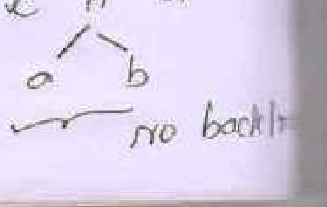
i/p → abb

The device string abb there is an ambiguity as to how many times the nonterminals has to be expanded

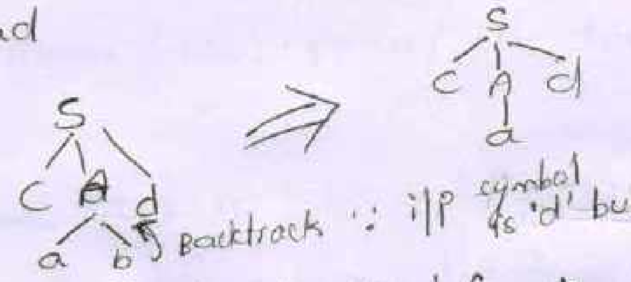
2. Backtracking: It occurs when there is more than one alternate In the production to be tried while parsing the i/p string

$S \rightarrow cAd$

$A \rightarrow ab|a$ i/p: cabd



i/p string : cad



Backtrack ∵ i/p symbol is 'd' but ptr is pointing to b

3.3. It is Very difficult to Identify the posn of the errors

Example : Recursive descent

Q) write a recursive descent parser for the following grammar

$E \to TE'$

$E' \to +TE' | \varepsilon$

$T \to FT'$ i/p: id+id*id

$T' \to *FT' / \varepsilon$

$F \to (E) | id$

→

```
→ procedure E()
  { if (input=='T')
      T():
  } Eprime():
  procedure T()
  { F();
  } Tprime();
  procedure F()·
  {
      if(input=="C")
      { Advance();
        E():
        If (input==')")
            return(True);
        else
            return(False);
      }
```

```
elseif(input ==id")
{
    Advance();
    return(True);
}
else
    return(False);
}
procedure Eprime()
{
    If(input =="+")
    {
        Advance();
        T();
        Eprime();
        return(True);
    }
    return(False);
}
procedure Tprime()
{
    If(input =="*")
    {
        Advance();
        F();
        Tprime();
        return(True);
    }
    else
        return(False);
}
```

ii) <u>Table driven</u>:

-> Table driven is also called as predictive parsing

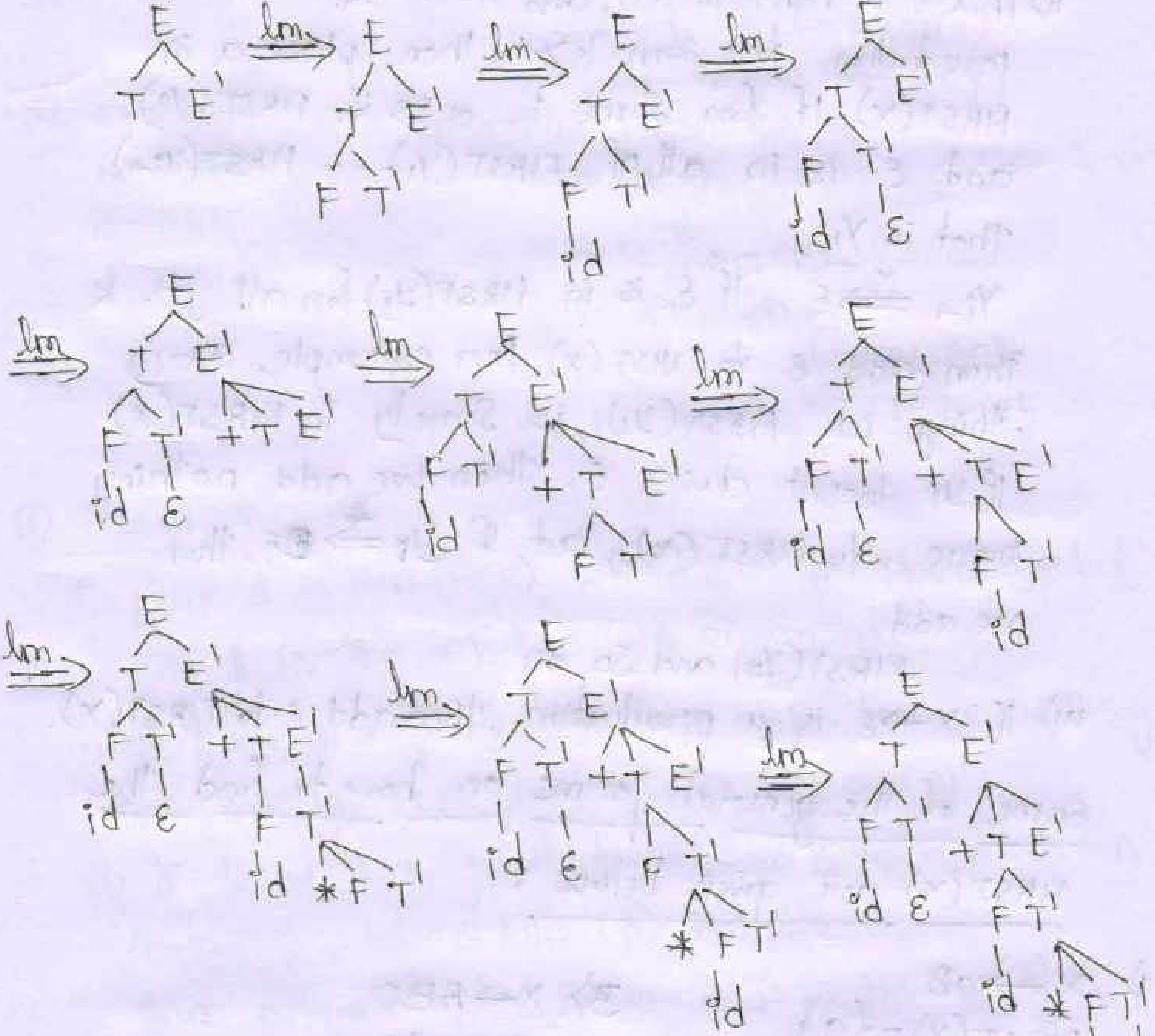
-> predictive parser is a recursive decent parser, which ~~production~~ has the capability to predict which production is to be used to replace the i/p String

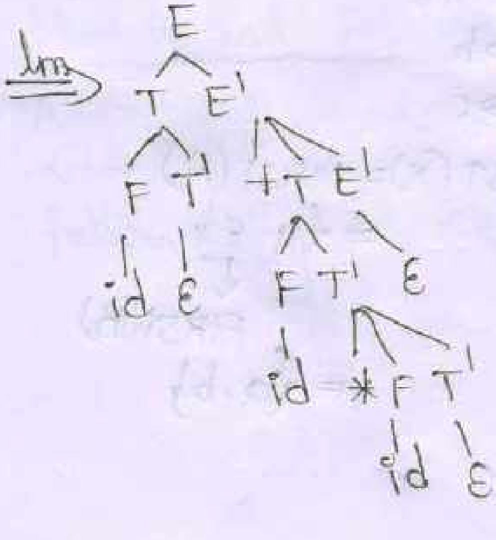
-> The predictive parser does not suffer from backtracking

Predictive Parsers / LL(1) parsers / Table driven parsers.

Q) why do we need a FIRST and FOLLOW set

Consider The below given top down approach for the example:





here, T' has two options i.e T'→*FT' and T'→ε we are in ambiguity Situation to replace with which RHS of the production.

If we have n alternate productions for the same Non-terminal then we have to backtrack (n-1) times So, It is Very tedious process,

So, we find FIRST & FOLLOW set.

FIRST AND FOLLOW SETS:

i) FIRST:

i> If X is a terminal, than FIRST(X) = {X}

ii> If X is a nonterminal, and $X \rightarrow Y_1Y_2 -- Y_k$ is a production for som $k \geq 1$, then place a in FIRST(X) If for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1) --- FIRST(Y_{i-1}) That is Y_1

$Y_{i-1} \overset{x}{\Longrightarrow} \epsilon$, If ϵ is in FIRST(y_j) for all $j=1 .. k$ then add ϵ to FIRST(X), For example, every-thing in FIRST(y_1) is surely in FIRST(X) If y_1 doesnt drive ϵ, then we add nothing more to FIRST(X), but If $y_1 \overset{*}{\Longrightarrow} \epsilon$, than we add

FIRST(Y_2) and so on

iii> If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Some of the general forms on how to find the FIRST(X) are given below:

1> $X \rightarrow aB$

FIRST(X) = {a}

$X \rightarrow \epsilon$

FIRST(X) = ϵ

2> $X \rightarrow ABC$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow c$

FIRST(X) = FIRST(A)

= {a}

3> $X \rightarrow ABC$

$A \rightarrow a | \epsilon$

$B \rightarrow b$

$C \rightarrow c$

FIRST(X) = FIRST(A)

= {a, ϵ}

↓

FIRST(B)

= {a, b}

4> $X \rightarrow ABC$
$A \rightarrow a|\varepsilon$
$B \rightarrow b|\varepsilon$
$C \rightarrow c$

FIRST(X) = FIRST(A)
= {a, ε}
↓
FIRST(B)
{b, ε}
↳ FIRST(C)
{c}
= {a, b, c}

5> $X \rightarrow ABC$
$A \rightarrow a|\varepsilon$
$B \rightarrow b|\varepsilon$
$C \rightarrow c|\varepsilon$

FIRST (X) = FIRST (A)
= {a, ε}
↓
FIRST (B)
{b, ε}
↳ FIRST (C)
↳ {c, ε}

∴ = {a, b, c, ε}

ii> FOLLOW :

i> place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker

ii> If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B)

iii> If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST(β) Contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

How to find FOLLOW :

1> $A \rightarrow \underbrace{X}_{\alpha}\underbrace{B}_{B}\underbrace{CD}_{\beta}$
$C \rightarrow c$

fallow(B) = {FIRST (CD)}
= {c}

2> $A \rightarrow \underbrace{X}_{\alpha}\underbrace{B}_{B}\underbrace{CD}_{\beta}$

Since β = CD
∴ FOLLOW(B) = {FIRST(β)}
= FIRST(CD}
= {c, d}

3> $A \rightarrow \overbrace{X}^{\alpha}\overbrace{B}^{B}\overbrace{CD}^{\beta}$

$C \rightarrow c|\epsilon$

$D \rightarrow d|\epsilon$

FOLLOW(B)= first (β)

= first (CD)

= {c, d} + FOLLOW (A)

4> $A \rightarrow \underbrace{X}_{\alpha}\underbrace{B}_{B}\underbrace{\;}_{\beta}$ (Since β = ε, here)

fallow (B) = FALLOW (left most nonterminal)

= FALLOW(A)

5> $A \rightarrow \overbrace{X}^{\alpha}\overbrace{B}^{B}\overbrace{\underbrace{CD}_{\alpha}\underbrace{B}_{B}\underbrace{e}_{\beta}}^{\beta}$ — ① — ②

$C \rightarrow c|\epsilon$

$D \rightarrow d$

FALLOW(B)= FIRST (CDBe)

= (C, D) ——— ①

+

FALLOW (B) = FIRST (e)

= {e}

∴ FAL(B) = ① + ②

FAL(B)={c, d, e}

Find the FIRST and FOLLOW set for the following grammars

1. $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)|id$

2. $S \rightarrow iEtS|iEtSeS|a$

$E \rightarrow b$

3) $S \rightarrow , G, H ;$
$G \rightarrow aF$
$F \rightarrow bF \mid \varepsilon$
$H \rightarrow KL$
$K \rightarrow m \mid \varepsilon$
$L \rightarrow n \mid \varepsilon$

4) $S \rightarrow aB \mid ac \mid sd \mid se$
$B \rightarrow bBc \mid f$
$C \rightarrow g$

5) $S \rightarrow aBDh$
$B \rightarrow eC$
$C \rightarrow bC \mid \varepsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \varepsilon$
$F \rightarrow f \mid \varepsilon$

~~6) D → EF~~
~~E → g | ε~~ → epsilon
~~F → f | ε~~

6. $S \rightarrow (L) \mid a$
$L \rightarrow L , S \mid S$

7) $S \rightarrow L = R \mid R$
$L \rightarrow * R \mid id$
$R \rightarrow L$

8. $S \rightarrow AaAb \mid BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

9) $S \rightarrow aABb$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow d \mid \varepsilon$

10. stmt-Sequence → stmt stmt-sequence'
stmt_sequence' → ; stmt_sequence | ε
stmt → s

11) $S \rightarrow asbs \mid bsas \mid \varepsilon$

12) $S \rightarrow a \mid \uparrow \mid (T)$
$T \rightarrow T , S \mid S$

13) $S \rightarrow AS \mid b$
$A \rightarrow SA \mid a$

Answer.

i) $E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) / id$

	E	E'	T	T'	F
FIRST	(id	+ ε	(id	* ε	(id
FOL	$)	$)	+ $)	+ $)	*

here FOLLOW(E) = $F \rightarrow \underset{\alpha}{(}\underset{B}{E}\underset{\beta}{)}$

FOLLOW(E) = FIRST( ))

= {) }

FOLLOW(E') = $E \rightarrow \underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{\ }$ and $E' \rightarrow \underset{\alpha}{+T}\underset{B}{E'}\underset{\beta}{\ }$

FOL(E') = FIRST(β) ↓ ε FOL(E') = FIRST(β) ↓ ε

∴ FOL(E') = FOL(E) ∴ FOL(E') = FOL(E

So, on FOLLOW(T) = $E' \rightarrow \underset{\alpha}{+}\underset{B}{T}\underset{\beta}{E'}$ $E \rightarrow \underset{\alpha}{\ }\underset{B}{T}\underset{\beta}{E'}$

= FIRST(β) = FIRST(β)

= FIRST(E') = FIRST(E')

= {+} = {+}

<u>Note</u>: we should not write <u>ε</u> in the <u>follow</u> set

2) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

	S	E
FIRST	i a	b
FOLLOW	$ e	t

$\text{FOLLOW}(S) = S \rightarrow iEt\underset{B}{S}\,\overset{\sim}{\beta}$

$\text{FOLLOW}(S) = \text{FIRST}(\beta)$ {ε

$= \text{FOLLOW}(S)$

$S \rightarrow \underbrace{i}_{\alpha}\underbrace{E}_{B}\underbrace{tses}_{\beta}$

$\text{FOLLOW}(S) = \text{FIRST}(\beta)$

$= \text{FIRST}(es)$

$= \{e\}$

3) $S \rightarrow ,GH;$

$G \rightarrow aF$

$F \rightarrow bF \mid \varepsilon$ — Epsilon

$H \rightarrow KL$

$K \rightarrow m \mid \varepsilon$

$L \rightarrow n \mid \varepsilon$

	S	G	F	H	K	L
FIRST	,	a	b ε	m n ε	m ε	n ε
FOLLOW	$	m n ;	m n ;	;	n ;	;

4) S → aB|aC|Sd|Se
B → bBc|f
C → g

	S	B	C
FIRST	a	b f	g
FOLLOW	$ d e	$ d e c	$ d e

5) S → aBDh
B → eC
C → bC|ε
D → EF
E → g|ε
F → f|ε

	S	B	C	D	E	F
FIRST	a	e	b ε	g f ε	g ε	f ε
FOLLOW	$	g f h	g f h	h	h f	h

6) S → (L)|a
L → L,S|S

	S	L
FIRST	(a	(a
FOLLOW	$) ,	) ,

8) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

	S	A	B
FIRST	a b	ε	ε
FOLLOW	$	a b	b a

9) $S \rightarrow aABb$

$A \rightarrow (\mid \varepsilon$

$B \rightarrow d \mid \varepsilon$

	S	A	B
FIRST	a	(	d
		ε	ε
FOLLOW	$	d b	b

10) $S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

	S	T
FIRST	a ↑ (	a ↑ (
FOLLOW	$) ,	))

8) S → As|b
A → SA|a

	S	A
FIRST	b a	a
FOLLOW	$ a b	a b

9) S → L =R|R
L → *R|id
R → L

	S	L	R
FIRST	* id	* id	* id
FOLLOW	$	= $	$ =

10) stmt_sequence → stmt stmt_seqence'
stmt_seqence' → ; stmt_seqence|ε
stmt → s

	stmt_segence	stmt_squnce'	stmt
FIRST	s	; ε	s
FOLLOW	$	$	; $

1R) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

	S
FIRST	a b ε
Follow	\$ a b

$\text{Follow}(S) \Longrightarrow S \rightarrow \underbrace{a}_{\alpha}\underbrace{S}_{B}\underbrace{bS}_{\beta}$ $\qquad S \rightarrow \underbrace{b}_{\alpha}\underbrace{S}_{B}\underbrace{aS}_{\beta}$ $\qquad \cancel{S \rightarrow \varepsilon}$

$= \text{FIRST}(\beta) \qquad = \text{FIRST}(\beta)$

$= \text{FIRST}(b\$) \qquad = \text{FIRST}(aS)$

$\text{Follow}(S) = \{b\} \qquad = \{a\}$

$\text{FOL}(S) \rightarrow \underbrace{asb}_{\alpha}\underbrace{S}_{B}\underbrace{\ }_{\beta}$ $\qquad \text{Fol}(S) \rightarrow \underbrace{bsa}_{\alpha}\underbrace{S}_{B}\underbrace{\ }_{\beta}$

$= \text{FIRST}(\beta) \rightarrow \varepsilon \qquad = \text{FIRST}(\beta) \rightarrow \varepsilon$

$\text{FOL}(S) = \text{FOL}(S) \qquad \text{FOL}(S) = \text{FOL}(S)$

Top-down Parsers

preductive parsing table / LL(1) grammar / Table Driven predictive parser

LL(1) grammar

- L → scan the i/p from left to right
- L → left most derivation
- (1) → lookahead symbol

Steps:

1> Eliminate left recursion from the grammar
2> perform left factoring
3> find the FIRST and FOLLOW set
4> Construct the preductive parsing table
5> check wheather the given i/p string is accepted/not

Algorithm for Constructing preductive parsing table

INPUT: Grammar G

OUTPUT: parsing table M

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following

1. For each terminal α in FIRST(A), add $A \rightarrow \alpha$ to $M[A, \alpha]$
2. If ε is in FIRST(α), Then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ε is in FIRST(α) & \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well

Preductive parsing Algorithm

INPUT: A string w and a parsing table M for a grammar G.

OUTPUT: If w is in L(G) and LMD of w; otherwise an error condition

Input:

			a	+	b	$

METHOD: Initially, the parser is in a configuration with w$ in the i/p buffer and the start symbol S on top of the stack, above $. The pgm in fig. uses the predictive parsing table M to procedure a predictive parse for the i/p

```
set 'ip' to point to the first symbol of w;
set 'x' to the top stack symbol;
while (x≠$) { /* stack is not empty */)
If (x is a) pop the stack & a advance ip;
elseif (x is a terminal) error();
elseif (M[x, a] is an error entry) error();
else if (M[x, a] = x→y1, y2 --- yk) {
output the production x→y1, y2 --- yk;
pop the stack;
push yk, yk-1 --- y1 onto the stack, with y1 on-top
}
Set x to the top stack symbol;
}
```

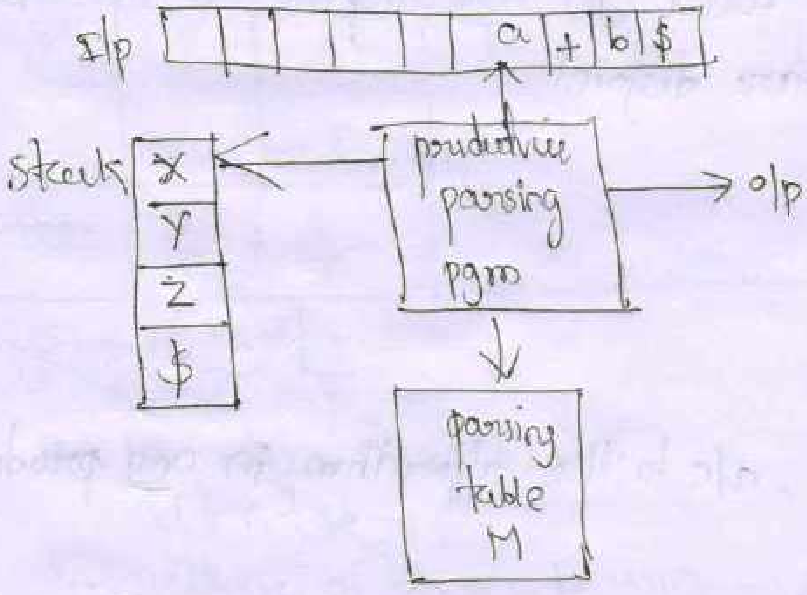


fig: model of a table driven predictive parser

checking wheather the given grammar is LL(1) ~~or~~ not without using parsing table

A grammar is LL(1) iff whenever, $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following Conditions hold

i) For no terminal 'a' do with α and β derive strings beginning with $a \Rightarrow$ FIRST(α) and FIRST(β) are disjoint.

ii) Atmost one of α and β Can derive the empty string $\Rightarrow$ either FIRST(α) $\rightarrow \varepsilon$ or FIRST(β) $\rightarrow \varepsilon$ but not both.

iii) If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW (A). Likewise, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β doesnot derive any string beginning with a terminal in FOLLOW (B)

$\Rightarrow$ FIRST(α) and FOLLOW(A) are disjoint. or FIRST(β) and FOLLOW(A) are disjoint

General forms:

① $A \rightarrow \overset{\alpha}{aB} | \overset{\beta}{aC}$

FIRST(α) = {a}

FIRST(β) = {a}

are not disjoint. a/c to the algorithm in any production

eg: $A \rightarrow aB | bC$

FIRST(α) = {a}

FIRST(β) = {b} are disjoints

2) $A \to Bc \mid CD$ either $FIRST(\beta) \Rightarrow \varepsilon$ or
$B \to b \mid \varepsilon$ $FIRST(\alpha) \Rightarrow \varepsilon$
$C \to c \mid \varepsilon$ but not both

3) i) $A \to a \mid B$
$B \to cAa \mid \varepsilon$
$FIRST(\alpha) = \{a\}$
$FOLLOW(A) = \{\$, a\}$ are not disjoint

ii) $A \to B \mid a$
$B \to cAa \mid \varepsilon$
$FIRST(\beta) = \{a\}$ and $FOLLOW(A) = \{\$, a\}$
are not disjoint

Examples:

1. $S \to iEtSS' \mid a$
$S' \to eS \mid \varepsilon$
$E \to b$

$$S \to \overset{\alpha}{iEtSS'} \mid \overset{\beta}{a}$$

a. $FIRST(\alpha) \cap FIRST(\beta) = \phi$
$\{i\} \cap \{a\} = \phi$

b. neither of α or β are ε

	S	S'	E
FIRST	i	e	b
	a	ε	
FOLLOW	\$ e	\$ e	t

~~1. $S \to \underbrace{iEtSS'}_{\alpha} \mid \underbrace{a}_{\beta}$~~

~~a. $FIRST(\alpha) \cap FIRST(\beta) = \phi$~~
~~$\{i\} \cap \{a\} = \phi$~~

~~b. neither of α or β are $\Rightarrow \varepsilon$~~

~~c. $\beta \Rightarrow \varepsilon$, then $FIRST(\alpha) \cap FOLLOW(A) = \phi$~~
~~$FIRST(eS) \cap FOLLOW(S') = \phi$~~
~~$\{e\} \cap \{\$e\} \neq 0$~~

~~∴ The given grammar is not LL(1).~~ Condition fails

$S' \rightarrow \overset{\alpha}{eS} \mid \overset{\beta}{\varepsilon}$

a) $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$

$\{e\} \cap \{\varepsilon\} = \phi$

b) The given grammer

$\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c) $\beta \Rightarrow \varepsilon$, then $\text{FIRST}(\alpha) \cap \text{FOL}(A) = \phi$

$\text{FIRST}(eS) \cap \text{FOL}(S') = \phi$

$\{e\} \cap \{\$ e\} \neq \phi$

Condition fails

∴ The given grammar is not LL(1)

2) $S \rightarrow S(S)S \mid \varepsilon \Rightarrow S \rightarrow S'$

$S' \rightarrow (S)\, SS' \mid \varepsilon$

	S	S'
FIRST	(ε	(ε
FOLLOW	$) (	$) (

1. $S \rightarrow S'$

not required because we dont have β production

2. $S' \rightarrow \underset{\alpha}{(S)\,SS'} \mid \underset{\beta}{\varepsilon}$

a. $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) \neq \phi$

$\{(\} \cap \{\varepsilon\} = \phi$

b. only $\beta \Rightarrow \varepsilon$ and $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$, then

$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi$

$\{(\} \cap \{\$ \; (\;)\} \neq \phi$

∴ The given grammar is not LL(1)

3. $S \rightarrow SS+ \mid SS* \mid a \Rightarrow S \rightarrow aS'$

$S' \rightarrow S+S' \mid S*S' \mid \varepsilon$

$\Downarrow$ left factoring

final production $\left\{ \begin{array}{l} S \rightarrow aS' \\ S' \rightarrow SS'' \mid \varepsilon \\ S \rightarrow +S' \mid *S' \end{array} \right.$

	S	S'	S''
FIRST	a	a ε	+ *
FOLLOW	\$ + *	\$ + *	\$ + *

1. $S \rightarrow aS'$

not required

2. $S'' \rightarrow +S' \mid *S'$

a. $\text{FIRST}(+S') \cap \text{FIRST}(*S') = \phi$

$\{+\} \cap \{*\} = \phi$

b. neither α or $\beta \Rightarrow \varepsilon$

all Condition are satisfied

3. $S' \rightarrow SS'' \mid \varepsilon$

a. $\text{FIRST}(SS'') \cap \text{FIRST}(\varepsilon)$

$\{a\} \cap \{\varepsilon\} = \phi$

b. $\beta \Rightarrow \varepsilon$ but not α

c $\beta \Rightarrow \varepsilon$ then

$\text{FIRST}(SS'') \cap \text{FOLLOW}(S') = \phi$

$\{a\} \cap \{\$ + *\} = \phi$

all Conditions are satisfied

∴ The grammar is LL(1).

4) $E \rightarrow E+T \mid T$ $\qquad$ $E \rightarrow TE'$

$T \rightarrow T*F \mid F \quad \Rightarrow \quad E' \rightarrow +TE' / \varepsilon$

$F \rightarrow (E) \mid id$ $\qquad$ $T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) \mid id$

	E	E'	T	T'	F
FIRST	(id	+ ε	(id	* ε	(id
FOLLOW	\$)	\$)	+ \$)	+ \$)	+ * \$)

i) $E \rightarrow TE'$

ii) $E' \rightarrow +E' / \varepsilon$

a. $\text{FIRST}(+TE) \cap \text{FIRST}(\varepsilon) = \phi$

$\{+\} \cap \{\varepsilon\} = \phi$

b. $\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$ then $\text{FIRST}(+TE) \cap \text{FOL}(E') = \phi$

$\{+\} \cap \{\$)\} = \phi$

iii) $T \rightarrow FT'$

iv) $T' \rightarrow *FT' / \varepsilon$

a. $\text{FIRST}(*FT') \cap \text{FIRST}(\varepsilon) = \phi$

$\{*\} \cap \{\varepsilon\} = \phi$

b. $\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$ then $\text{FIRST}(*FT') \cap \text{FOL}(T') = \phi$

$\{*\} \cap \{+\$)\} = \phi$

v) $F \rightarrow (E) \mid id$

a. $\text{FIRST}((E)) \cap \text{FIRST}(id) = \phi$

$\{(\} \cap \{id\} = \phi$

b. neither of them are not $\Rightarrow \varepsilon$

checking wheather the given grammar is LL(1) or not with Constructing the predictive parsing table

1. E → E+T|T
   T → T*F|F   i/p: id+id*id
   F → (E)|id

i) Remove left Recursion

E → TE'
E' → +TE'|ε
T → FT'
T' → *FT'|ε
F → (E)|id

ii) Remove left factoring
→ here, not required

iii) find FIRST and FOLLOW set

	E	E'	T	T'	F
FIRST	(id	+ ε	(id	* ε	(id
Follow	$)	$)	+ $)	+ $)	* + $)

iv) Construct the parsing table

	(	id	)	+	*	$
E	E→TE'	E→TE'				
E'			E'→ε	E'→+TE'		E'→ε
T	T→FT'	T→FT'				
T'			T'→ε	T'→ε	T'→*FT'	T'→ε
F	F→(E)	F→id				

v)

Stack	Input	Action
E$	id+id*id$	
TE'$	id+id*id$	push E→TE'
FT'E'$	id+id*id$	push T→FT'
idT'E'$	id+id*id$	push F→id
T'E'$	+id*id$	matched 'id'
E'$	+id*id$	T'→ε
+TE'$	+id*id$	push E'→+TE'
TE'$	id*id$	matched '+'
FT'E'$	id*id$	push T→FT'
idT'E'$	id*id$	push F→id
T'E'$	*id$	matched 'id'
*FT'E'$	*id$	push T'→*FT'
FT'E'$	id$	matched '*'
idT'E'$	id$	push F→id
T'E'$	$	matched 'id'
E'$	$	T'→ε
$	$	E'→ε

∴ The grammar is accepted the i/p storing i.e, The i/p is parsed Succeesfully

The grammar is LL(1) ∵ no multiple entories

2) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

i/p: If E_1 then if E_2 then S_1 else S_2

If b then if b then a else a

i) No left Recursion

ii) Remove left factoring

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

iii) FIRST and FOLLOW Set

	S	S'	E
FIRST	i a	e ε	b
FOLLOW	\$ e	\$ e	t

iv) parsing table

	\$	i	e	b	a	t
S		$S \rightarrow iEtSS'$				
S'	$S' \rightarrow \varepsilon$		$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			
E				$E \rightarrow b$		

The grammar is not LL(1) because it has multiple entries for the same terminal in a table

v)

Stack	Input	Action
S \$	~~iEtEtS_1eS_2~~ ibtibtaea \$	
iEtSS' \$	ibtibtaea \$	$S \rightarrow iEtSS'$
EtSS' \$	btibtaea \$	matched 'i'
btSS' \$	btibtaea \$	$E \rightarrow b$

Stack	input	Action
tss'$	tibtaea$	matched 'b'
ss'$	ibtaea$	matched 't'
iEtss's'$	ibtaea$	S→iEtss'
Etss's'$	btaea$	matched 'i'
btss's'$	btaea$	E→b
tss's'$	taea$	matched 'b'
ss's'$	aea$	matched t
as's'$	aea$	S→a
s's'$	ea$	matched 'a'

↳ ambiguity wheather to push s'→es or s'→ε

⇒ i/p : If E then S else S
ibtaea

stack	input	Action
S$	ibtaea$	S→iEtss'
iEtss'$	ibtaea$	matched 'i'
Etss'$	btaea$	push E→b
btss'$	btaea$	match 'b'
tss'$	taea$	match t
ss'$	aea$	S→a
as'$	aea$	match a
s'$	ea$	

↳ ambiguity. whether to push s'→es or s'→ε

3> $S \to SS+|SS*|a$ i/p: aa+a*

i> Remove left recursion

$S \to aS'$

$S' \to S+S'|S*S'|\varepsilon$

ii> Remove left factoring

$S \to aS'$

$S' \to SS''|\varepsilon$

$S'' \to +S'|*S'$

iii> find FIRST and FOLLOW set

	S	S'	S''
FIRST	a	a ε	+ *
FOLLOW	\$ + *	\$ + *	\$ + *

iv> find the predictive parsing table

	\$	a	+	*
S		$S \to aS'$		
S'	$S' \to \varepsilon$	$S' \to SS''$	$S' \to \varepsilon$	$S' \to \varepsilon$
S''			$S'' \to +S'$	$S'' \to *S'$

The given grammar is LL(1)
becaus. There are no multiple production.

v> parse the i/p string

stack	input	action
S\$	aa+a*\$	
aS'\$	aa+a*\$	$S \to aS'$
S'\$	a+a*\$	match 'a'
SS''\$	a+a*\$	$S' \to aS'$
aS'S''\$	a+a*\$	$S \to aS'$
S'S''\$	+a*\$	match 'a'
S''\$	+a*\$	$S' \to \varepsilon$

Stack	input	Stack
$+s'\$$	$+a*\$$	$s'' \to +s'$
$s'\$$	$a*\$$	match '+'
$Ss''\$$	$a*\$$	push $s' \to ss''$
$as's''\$$	$a*\$$	push $s \to as'$
$s's''\$$	$*\$$	match 'a'
$s''\$$	$*\$$	push $s' \to \varepsilon$
$*s'\$$	$*\$$	push $s'' \to *s'$
$s'\$$	$\$$	match *
$\$$	$\$$	push $s' \to \varepsilon$

The i/p string is Succesfully parsed

4) $S \to 0S1/01$ i/p string: 000111

i) Remove left recursion

→ not needed

ii) Remove left factoring

$S \to 0S'$

$S' \to S1/1$

iii) FIRST and FOLLOW Set

	S	$
FIRST	0	0 1
FOLLOW	$ 1	$ 1

iv) Construct the predictive parsing table

	$	0	1
S		$S \to 0S'$	
S'		$S' \to S1$	$S' \to 1$

The grammar is LL(1), Since It does not have any multiple poroduction

v) parse the input storing

Stack	input	action
$S\$$	$000111\$$	
$0S'\$$	$000111\$$	push $S \rightarrow 0S'$
$S'\$$	$00111\$$	match '0'
$S1\$$	$00111\$$	push $S' \rightarrow S1$
$0S'1\$$	$00111\$$	$S \rightarrow 0S'$
$S'1\$$	$0111\$$	match '0'
$S11\$$	$0111\$$	$S' \rightarrow S1$
$0S'11\$$	$0111\$$	push $S \rightarrow 0S'$
$S'11\$$	$111\$$	match 0
$111\$$	$111\$$	$S' \rightarrow 1$
$11\$$	$11\$$	match 1
$1\$$	$1\$$	match 1
$\$$	$\$$	match 1

The input storing is Successfully parsed

5> S → +SS|*SS|a   i/p: +*aaa

i> Remove left recursion ⇒ not required

ii> Remove left factoring ⇒ Not required

iii> Construct FIRST and Follow set

	S
FIRST	+ * a
FOLLOW	\$ + * a

iv> Construct the predictive parsing table

	\$	+	*	a
S		+SS	*SS	a

The grammar is LL(1) since It Contains no more than 1 production

v> i/p string: +*aaa

Stack	input	Action
S\$	+*aaa\$	
+SS\$	+*aaa\$	S→+SS
SS\$	*aaa\$	match '+'
*SSS\$	*aaa\$	S→*SS
SSS\$	aaa\$	match *
aSS\$	aaa\$	push S→a
SS\$	aa\$	match a
aS\$	aa\$	push S→a
S\$	a\$	match a
a\$	a\$	push S→a
\$	\$	~~pa~~ match a

i/p is Successfully parsed

~~6) S → S(S)S | ε   i/p: (()())~~

~~i) Remove left Recursion~~

~~S → εS'~~

~~S' → (S)SS' | ε~~

7) S → S+S | SS | (S) | S* | a
i/p: (a+a)*a

i) Remove left Recursion

S → aS' | (S) S'

S' → +SS' | SS' | *S' | ε

ii) Remove left factoring

→ not needed

iii) Find FIRST and FOLLOW

	S	S'
FIRST	a (	+, ε a (*
FOLLOW	$ +) a (*	$) + a, (, *

iv) Construct the predictive parsing table

	$	(	a	)	+	*
S		S→(S)S'	S→aS'			
S'	S'→ε	S'→SS' S'→ε	S→SS' S'→ε	S'→ε	S'→+SS' S'→ε	S'→*S' S'→ε

The grammar is not LL(1)

v) input : (a+a)*a

stack	input	Action
S$	(a+a)*a	
(S)S'$	(a+a) *a	S → (S)S'
S)S'$	a+a)*a	match (
aS')S'$	a+a)*a	push S→aS'
S')S'$	+a)*a	match a

↳ Ambigous, whether to parse S' → +SS' or S' → ε

6) $S \to S(S)S \mid \varepsilon$ i/p: (()())

i) Remove left Recursion

$S \to \varepsilon S'$

$S' \to (S)SS' \mid \varepsilon$

ii) Remove left factoring

→ not needed

iii) find FIRST and Follow set

	S	S'
FIRST	(ε	(ε
FOLLOW	\$) (	\$) (

iv) parsing table

	\$	(	)
S	$S \to \varepsilon$	$S \to S'$ $S \to \varepsilon$	$S \to \varepsilon$
S'	$S' \to \varepsilon$	$S' \to (S)SS'$ $S' \to \varepsilon$	$S' \to \varepsilon$

→ multiple production

The grammar is not LL(1), Since it has a multiple (~~transaction~~ productions

v) parse the i/p string: (()())

stack	Input	Action
S\$	(()())\$	
S'\$	(()()) \$	$S \to S'$ → ambiguity
(S)SS'\$	(()())\$	$S' \to (S)SS'$
S)SS'\$	()())\$	match '('
S')SS'\$	()())\$	push $S \to S'$

4) $S \to (L) | a$
$L \to L,S | S$ $\Rightarrow (a,a)$

step i) Remove left Recursion

$L \to \underset{A}{L}, \underset{\alpha}{S} | \underset{\beta}{S}$

$L \to SL'$

$L' \to ,SL' | \varepsilon$

$S \to (L) | a$

ii) Remove left- ~~Recursion~~ factoring
not required

iii) write FIRST and FOLLOW Set

	S	L	L'
FIRST	(a	(a	, ε
FOLLOW	\$,)	)	)

iv) write the predictive parsing table

	(	a	,	$
S	S→(L)	S→a		
L	L→SL'	L→SL'		
L'		L'→ε	L'→ ,SL	

v) Stack	Input	action
S$	(a,a) $	S→(L)
(L)$	(a.a) $	match (
L)$	a.a) $	L→SL'
SL')$	a,a) $	S→a
aL')$	a, a)$	match a
L')$	,a) $	L'→ ,SL'
,SL')$	,a) $	match ,
SL')$	a) $	S→a
aL')$	a) $	match a
L')$	)$	L'→ε
)$	)$	match)
$	$	

The grammar i/p is Succesfully parsed

8) S→ S+S| SS| (S) | S* | a ⇒ (a+a) *a

i) Remove left recursion

S→ S+S | SS | (S) | S*| a

S→(S)S'/ aS'

S'→ ~~(S)S'| aS'~~ +SS' | SS' | *S' | ε

ii) remove left factoring
not required

iii) Find FIRST and FOLLOW Set

	S	S'
FIRST	(	+
	a	a
		*
		ε

	S	S'
FOLLOW	\$	\$
	)	)
	+	+
	(	(
	a	a
	*	*

iv) write predictive parse table

	(	a	)	+	*	\$
S	$S \to (S)S'$	$S \to aS'$				
S'	$S \to +SS'$ $S \to SS'$ $S \to *S'$ $S \to \varepsilon$			$S' \to +SS'$		

∴ The given grammar is not LL(1)

g) $S \to aSbS \mid bSaS \mid \varepsilon$ ⇒ aabbab

i) Remove left factoring – not required

ii) Remove left Recursion – not required

iii) write FIRST and FOLLOW set

	S
FIRST	a
	b
	ε

	S
FOLLOW	b
	a
	\$

iv) write a predictive parsing table

	a	b	\$
S	$S \to aSbS$ $S \to \varepsilon$	$S \to bSaS$ $S \to \varepsilon$	$S \to \varepsilon$

v.

Stack	Input	Action
S$	aabbab$	
asbs$	aabbab$	s→asbs
sbs$	abbab$	match a
asbsbs$	abbab $	s→asbs
sbsbs $	bbab$	match a
bsasbs bs$	bbab$	s→bsas
Sasbsbs$	bab $	match b
bSasasbsbs$	bab$	s→bsas
Sasasbsbs$		

↳ambigous

10) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false
i/p: not (true or false)

→ i) Remove left Recursion
bexpr ⟶ ~~bterm~~ bexpr' or bterm | bterm
bexpr' ⟶ ~~or~~ bterm bexpr' ~~ε~~ ⟶ bterm → bterm and bfactor | bfactor
bterm → ~~bfactor bterm'~~ bfactor bterm'
bterm' → and bfactor bterm' | ε
bfactor → not bfactor | (bexpr) | true | false

ii) No left ~~recur~~ factoring

iii) Find the FIRST and FOLLOW set

	bexpr	bexpr'	bterm	bterm'	bfactor
FIRST	not (true false	or ε	not (true false	and ε	not (true false
FOLLOW	$)	$)	or) $	or $)	and or) $

predictive parsing table :

~~(not) or~~

~~bexpr btermbexpr' btermbexpr'.~~

	(	not	)	or	and	true	false	$
bexpr	btermbexpr'	btermbexpr'				bterm bexpr'	bterm bexpr'	
bexpr'			ε	or btermbexpr				ε
bterm	bfactorbterm'	bfactorbterm'				bfactor bterm'	bfactor bterm'	
bterm'			ε	ε	and bfactor bterm			ε
bfactor	(bexpr)	not bfactor				true	false	

stack	Input	Action
bexpr $	not(true or false)$	bexpr → btermbexpr'
btermbexpr'$	not (true or false)$	bterm → bfactor bterm'
bfactorbterm'bexpr'$	not (true or false)$	bfactor → not bfactor
not bfactor bterm' bexpr'$	not (true or false)$	bfactor → (bexpr)
bfactor bterm' bexpr' $	(true or false)$	~~match (~~
bexpr) bterm' bexpr'$	true or false)$	match (
btermbexpr')bterm' bexpr'$	true or false)$	bexpr → btermbexpr'
bfactorbterm'bexpr') bterm'bexpr'$	true or false)$	bterm → bfactorbterm'
bterm'bexpr') bterm'bexpr'$	or false)$	bfactor → bterm'
bexpr') bterm' bexpr'$	or false)$	bterm' → ε
bterm bexpr) bterm'bexpr'$	false)$	bexpr' → btermbexpr
bfactorbterm" bexpr)bterm'bexpr'$	false)$	bterm → bfactorbterm'
		bfactor → ε
bterm'bexpr) bterm' bexpr'$	)$ $	bterm' → ε & bexpr' → ε (match)
bterm' bexpr'$	$	
$		

Error recovery in predictive parsers:

1. panic mode Recovery: In the blank entries of the follow set of all NT, place "syn

Stack	i/p	Table Entry	Action
NT	T	blank	skip the terminal from the i/p
NT	T	Synch	Remove NT from the st except start symbol
T	T	match	pop the terminal from stack & input

Example:

1. $E \rightarrow E+T/T$ $\quad\quad E \rightarrow TE'$
   $T \rightarrow T*F/F$ $\quad\quad E' \rightarrow +TE'/\epsilon$
   $F \rightarrow (E)/id$ $\quad\quad T \rightarrow FT'$
   i/p=)id*+id $\quad\quad T' \rightarrow *FT'/\epsilon$
   $\quad\quad F \rightarrow (E)/id$

	E	E'	T	T'	F
FIRST	(id	+ ε	(id	* ε	(id
Follow	$)	$)	+ $)	+ $)	* $) +

predictive parsing table

	(	id	)	+	*	$
E	E→TE'	E→TE'	Synch			
E'			E'→ε	E'→+TE'		E'→ε
T	T→FT'	T→FT'	Synch ~~T→ε~~	Synch ~~T→*FT~~		Synch
T'			T'→ε	T'→ε	T'→*FT'	T'→ε
F	F→(E)	F→id	Synch	synch	Synch	Synch

stack	input	Action
$E\$$	$)id*+id\$$	Since It Is the start symbol Skip The i/p $[E,)]$ - Synch
$E\$$	$id*+id\$$	$E\rightarrow TE'$
$TE'\$$	$id*+id\$$	$T\rightarrow FT'$
$FT'E'\$$	$id*+id\$$	$F\rightarrow id$
$idT'E'\$$	$id*+id\$$	match id
$T'E'\$$	$*+id\$$	$T'\rightarrow *FT'$
$*FT'E'\$$	$*+id\$$	match *
$FT'E'\$$	$+id\$$	$[F,+]$ = Synch, remove NT from Stack
$T'E'\$$	$+id\$$	$T'\rightarrow \varepsilon$
$E'\$$	$+id\$$	$E'\rightarrow +TE'$
$+TE'\$$	$+id\$$	match +
$TE'\$$	$id\$$	$T\rightarrow FT'$
$FT'E'\$$	$id\$$	$F\rightarrow id$
$idT'E'\$$	$id\$$	match id
$T'E'\$$	$\$$	$T'\rightarrow \varepsilon$
$E'\$$	$\$$	$E'\rightarrow \varepsilon$
$\$$	$\$$	match $\$$

Q) show that the fallowing grammar is ambigous.
$E\rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid id$ and i/p: $id+id*id$
Give an unambigous grammar such that precedence order from lowest or highest are +, -, *, /, (), id and all are left to right associative.

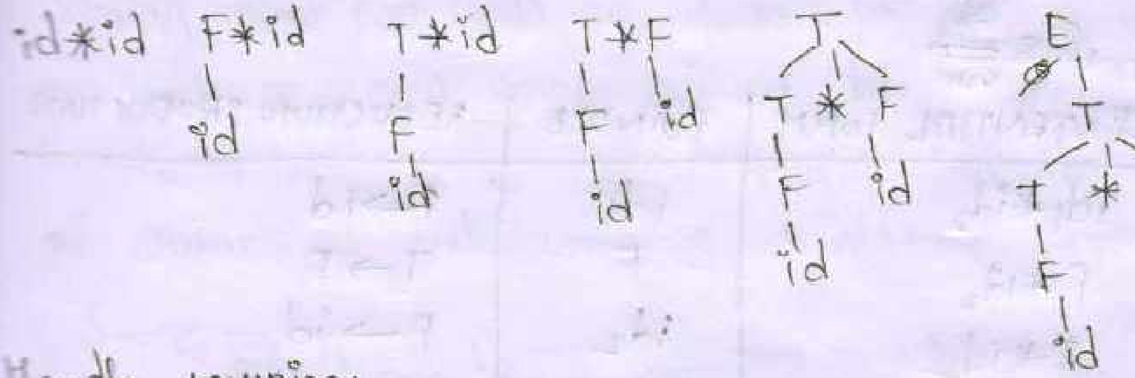
UNIT-3

Bottom up parsers :

Introduction :

→ A bottom-up parse Corresponds to the Construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)

→ eg: A bottom-up parses for id*id



Handle pruning:

Bottom up parsing during a left to right Scan of the input Constructs a right most derivation in reverse. Informally a 'handle' is a substring that matches the body of the production, and whose reduction represents one step along the reverse of the right-most derivation

Example:

adding subscripts to the tokens id for clarity, the handles during the parse of $id_1 * id_2$ a/c to the expressions grammar → { E → E+T/T, T → T*F/F, F → (E)|id } are shown in the figure.

Although T is the body of the production $E \rightarrow T$, the symbol T is not a handle in the sentential form $T*id_2$. If T were indeed replaced by E, we would get a string $E*id_2$, which cannot be derived from the start symbol E, Thus, the leftmost substring that matches the body of some production need not be a handle

~~formally, if $S \xRightarrow[rm]{*}$~~

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
id_1*id_2	id_1	$F \rightarrow id$
$F*id_2$	F	$T \rightarrow F$
$T*id_2$	id_2	$F \rightarrow id$
F	$T*F$	$E \rightarrow T*F$

Handles during a parse of id_1*id_2 (a)

Formally, if $S \xRightarrow[rm]{*} \alpha A w \xRightarrow[rm]{} \alpha \beta w$, as in figure (b), then production $A \rightarrow \beta$, in the position following α is a handle of $\alpha\beta w$, Alternatively, a handle of a right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right sentential form in a rightmost derivation of γ

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we prefer to the body β rather than $A \rightarrow \beta$ as a handle. Note we "a handle" rather than "the handle", because the grammar Could be ambigous, with moore than one rightmost derivation of $\alpha\beta w$. If a grammar is unambigous, then every right-Sentential form of The grammar has Exactly one handle.

A right-most derivation in reverse can be obtained by "handle pruning". That is, we start with a string of terminals w to be parsed. If w is a Sentence of a grammar at hand, then let $w = \gamma_n$, where γ_n is the n^{th} right Sentential form of Some as yet unknown rightmost derivation

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

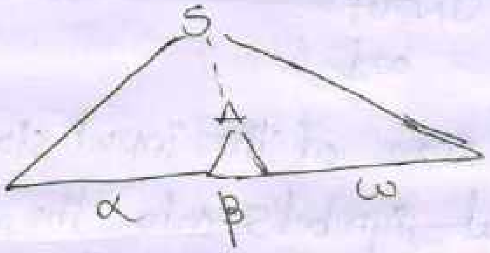


figure (b). A handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

To reconstruct this derivation in reverse order, we locate The handle β_n in γ_n and replace β_n by the head of relevant production $A_n \rightarrow \beta_n$ to obtain the previous right Sentential form γ_{n-1}. Note that we do not know how handles are to be found, but we shall see methods of doing so shortly

We the repeat this process. That is we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right Sentential form γ_{n-2}. If by Continuing this process we produce a right Sentential form Consisting only of the start symbol S. The we halt & that is Succesfull.

shift-Reduce parsing:

shift-reduce parsing is a form of bottom up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

→ we use $ to mark the bottom of the stack and also the right end of the input. conventionally, when discussing bottom-up parsing, we show the top of the stack on the right. rather on the left as we did for top-down parsing.

Initially, the stack is empty, and string ω is on the input as follows:

STACK	INPUT
$	ω$

During left to right scan of the input string, The parser. shifts zero/more input symbols onto the stack until it is ready to reduce a string β of the grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and input is empty.

Actions in shift-reduce parsing:

1. Shift: shift the next input symbol onto the top of stack

2. Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non terminal to replace the string

30. 3. Accept: Announce succesfull Completion of parsing

4. Error: Discover a syntax error and can an error recovery routine

Find the handles for the given RSF and Construct shift-reduce parser:

1. E → E+T/T
   T → T*F/F
   F → (E)/id

i/p: id+id
id+id*id

→ RMD

E → E+T
⇒ E+F
⇒ E+id
⇒ T+id
⇒ F+id
⇒ id+id

RSF	Handle	Action
id_1+id_2	id_1	F → id
F + id_2	F	T → F
T + id_2	T	E → T
E + id_2	id_2	F → id
E+F	F	T-F
E+T	E+T	E → E+T
E		

Stack	RSF	Action
\$	id_1+id_2 \$	shift id_1
\$ id_1	\$ id_2 \$	reduce F → id
\$ F	+id_2 \$	reduce T → F
\$ T	+id_2 \$	reduce E → T
\$ E	+id_2 \$	shift +
\$ E+	+id_2 \$	shift id_2
\$ E+id_2	\$	reduce F → id_2
\$ E + F	\$	reduce T → F
\$ E + T	\$	reduce E → T
\$ E		Success

i/p:- $id+id*id$

RMD:

$E \rightarrow E+T$

$\Rightarrow E+T*F$

$\Rightarrow E+T*id$

$\Rightarrow E+F*id$

$\Rightarrow E+id*id$

$\Rightarrow T+id*id$

$\Rightarrow F+id*id$

$\Rightarrow id+id*id$

RSF	Handle	Action
$id_1+id_2*id_3$	id_1	$F \rightarrow id$
$F+id_2*id_3$	F	$T \rightarrow F$
$T+id_2*id_3$	T	$E \rightarrow F$
$E+id_2*id_3$	id_2	$F \rightarrow id_2$
$E+F*id_3$	F	$T \rightarrow F$
$E+T*id_3$	id_3	$F \rightarrow id_3$
$E+T*F$	$T+F$	$T \rightarrow F*F$
$E+T$	$E+T$	$E \rightarrow E+T$
E		

Stack	RSF	Action
$\$$	$id_1+id_2*id_3\$$	shift id_1
$\$id_1$	$+id_2*id_3\$$	$F \rightarrow id_1$
$\$F$	$+id_2*id_3\$$	$T \rightarrow F$
$\$T$	$+id_2*id_3\$$	$E \rightarrow T$
$\$E$	$+id_2*id_3\$$	Shift $+$
$\$E+$	$id_2*id_3\$$	shift id_2
$\$E+id_2$	$*id_3\$$	reduce $F \rightarrow id$
		$T \rightarrow F$
		shift $*$
		shift id_3
$\$E+T*id_3$	$\$$	reduce $F \rightarrow id_3$
$\$E+T*F$	$\$$	reduce $F \rightarrow T*F$
		reduce $F \rightarrow E+T$
$\$E$	$\$$	Success

2) S → 0S1|01   i/p: 000111

$S \xRightarrow{rm} 0S1$

⟹ 00S11~~0~~

⟹ 000111

RSF	Handle	Action
000111	01	S→0S1
00S11	0S1	S→0S1
0S1	0S1	S→0S1

Stack	RSF	Action
$	000111$	Shift 0
$0	00111$	shift 0
$00	0111$	shift 0
$000	111$	shift 1
$0001	11$	reduce S→01
$00S	11$	shift 1
$00S1	1$	reduce S→0S1
$0S	1$	shift 1
$0S1	$	reduce S→0S1
$S	$	success

3) S → SS+|SS*|a   i/p: aaa*a++

S→SS+

⟹ SSS++

⟹ SSa++

⟹ SSS*a++

⟹ SSa*a++

⟹ Saa*a++

⟹ aaa*a++

RSF	Handle	Action
aaa*a++	a	S→a
Saa*a++	a	S→a
SSa*a++	a	S→a
SSS*a++	SS*	S→SS*
SSa++	a	S→a
SSS++	SS+	S→SS+
SS+*	SS+	S→SS+
S		

Stack	RSF	Action
$	aaa*a++$	shift a
$a	aa*a++$	reduce S→a
$S	aa*a++$	Shift a
$Sa	a*a++$	reduce S→a
$SS	a*a++$	Shift a
$SSa	*a++$	reduce S→a
$SSS	*a++$	Shift *
$SS*	a++$	reduce S→SS*
$SS	a++$	Shift a
$SSa	a++$	reduce S→a
$SSa	~~a~~++$	shift +
$SS+	+$	reduce S→SS+
$SS	+$	Shift +
$SS+	$	reduce S→SS+
$S	$	Success

Types of conflicts in shift-reduce parsers

Conflicts during shifts-Reduce parsing.

There are CFG's for which shift-reduce parsing Cannot be used. Every shift-reduce parser for such a grammar Can reach a Configuration in which the parser knowing the entire stack Contents & the next input symbol

Types:

i) shift/reduce conflict:

→ Cannot decide whether to shift or to reduce Called shift-reduce Conflict

ii) reduce/reduce conflict:

→ Cannot decide ~~whether~~ which of several reductions to make Called reduce-reduce Conflicts

eg: Consider the grammar

i) $E \rightarrow E+E$

$\quad | E-E$ i/p: 2+3*4

$\quad | NUM$

$\quad | ID$

No.	Stack	operation/grammar
1	2 NUM	shift 2
2	E	reduce E→NUM
3	E+	shift +
4	E+3	shift 3
5	E+E	reduce E→NUM
6	E	reduce E→E+E or shift * ie Shift reduce Conflict

~~ii) E → T~~ id+

~~T → id~~

~~F → id~~

eg 2) An ambigous grammar can never be LR. for eg. consider the dangling-else grammar

stmt → if expr then stmt
 | If expr then stmt else stmt
 | other

If we have a shift reduce parser in configuration

STACK	INPUT
...If expr then stmt	else ... $

→ we cannot tell whether If expr then stmt is the handle, no matter what happens below it on the stack. here there is a shift/reduce conflict. Depending on what follows the else on the input, it might be correct to reduce If expr then stmt to stmt, or it might be correct to shift else then to look for another stmt to complete the alternative If expr then stmt else stmt

eg 3) (1) stmt → id (parameter_list)
(2) stmt → expr := expr
(3) parameter_list → parameter_list , parameter
(4) parameter_list → parameter
(5) parameter → id
(6) expr → id (expr_list)
(7) expr → id
(8) expr_list → expr_list , expr
(9) expr_list → expr

STACK	INPUT
... id (id	, id) ...

In the problem id on the top of the stack must be reduced_ but by which production? The correct choice is production (5) if p is a procedure, but production (7) if p is an array. The stack does not tell which; information in the symbol table obtained. from the declaration of p must be used

So, In this case we have the Conflit that reduce id by paramitor or expr. It is called as reduce-reduce Conflict or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse. □

4.6 OPERATOR-PRECEDENCE PARSING

The largest class of grammars for which shift-reduce parsers can be built successfully – the LR grammars – will be discussed in Section 4.7. However, for a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property (among other essential requirements) that no production right side is ϵ or has two adjacent nonterminals. A grammar with the latter property is called an *operator grammar*.

Example 4.27. The following grammar for expressions

$$E \rightarrow EAE \mid (E) \mid -E \mid \textbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals. However, if we substitute for A each of its alternatives, we obtain the following operator grammar:

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \textbf{id} \qquad (4.17)$$

We now describe an easy-to-implement parsing technique called operator-precedence parsing. Historically, the technique was first described as a manipulation on tokens without any reference to an underlying grammar. In fact, once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using the nonterminals on the stack only as placeholders for attributes associated with the nonterminals.

As a general parsing technique, operator-precedence parsing has a number of disadvantages. For example, 1) it is hard to handle tokens like the minus sign, which has two different precedences (depending on whether it is unary or binary). Worse, 2) since the relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language. 3) Finally, only a small class of grammars can be parsed using operator-precedence techniques.

Nevertheless, because of its simplicity, numerous compilers using operator-precedence parsing techniques for expressions have been built successfully. Often these parsers use recursive descent, described in Section 4.4, for statements and higher-level constructs. Operator-precedence parsers have even been built for entire languages.

In operator-precedence parsing, we define three disjoint *precedence relations*, $<\cdot$, $\doteq$, and $\cdot>$, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

We should caution the reader that while these relations may appear similar to the arithmetic relations "less than," "equal to," and "greater than," the precedence relations have quite different properties. For example, we could have $a <\cdot b$ and $a \cdot> b$ for the same language, or we might have none of $a <\cdot b$, $a \doteq b$, and $a \cdot> b$ holding for some terminals a and b.

There are two common ways of determining what precedence relations should hold between a pair of terminals. The first method we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ <\cdot *$ and $* \cdot> +$. This approach will be seen to resolve the ambiguities of grammar (4.17), and it enables us to write an operator-precedence parser for it (although the unary minus sign causes problems).

2) The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees. This job is not difficult for expressions; the syntax of expressions in Section 2.2 provides the paradigm. For the other common source of ambiguity, the dangling **else**, grammar (4.9) is a useful model. Having obtained an unambiguous grammar, there is a mechanical method for constructing operator-precedence relations from it. These relations may not be disjoint, and they may parse a language other than that generated by the grammar, but with the standard sorts of arithmetic expressions, few problems are encountered in practice. We shall not discuss this construction here; see Aho and Ullman [1972b].

Using Operator-Precedence Relations

The intention of the precedence relations is to delimit the handle of a right-sentential form, with $<\cdot$ marking the left end, $\doteq$ appearing in the interior of the handle, and $\cdot>$ marking the right end. To be more precise, suppose we have a right-sentential form of an operator grammar. The fact that no adjacent nonterminals appear on the right sides of productions implies that no right-sentential form will have two adjacent nonterminals either. Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 \cdots a_n \beta_n$, where each β_i is either ϵ (the empty string) or a single nonterminal, and each a_i is a single terminal.

Suppose that between a_i and a_{i+1} exactly one of the relations $<\cdot$, $\doteq$, and $\cdot>$ holds. Further, let us use \$ to mark each end of the string, and define $\$ <\cdot b$ and $b \cdot> \$$ for all terminals b. Now suppose we remove the nonterminals from the string and place the correct relation $<\cdot$, $\doteq$, or $\cdot>$, between each pair of terminals and between the endmost terminals and the $'s marking the ends of the string. For example, suppose we initially have the right-sentential form **id** + **id** * **id** and the precedence relations are those given in Fig. 4.23. These relations are some of those that we would choose to parse according to grammar (4.17).

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Fig. 4.23. Operator-precedence relations.

Then the string with the precedence relations inserted is:

$$\$ \lessdot \mathbf{id} \gtrdot + \lessdot \mathbf{id} \gtrdot * \lessdot \mathbf{id} \gtrdot \$ \qquad (4.18)$$

For example, $\lessdot$ is inserted between the leftmost $ and **id** since $\lessdot$ is the entry in row $ and column **id**. The handle can be found by the following process.

1. Scan the string from the left end until the first $\gtrdot$ is encountered. In (4.18) above, this occurs between the first **id** and +.

2. Then scan backwards (to the left) over any $\doteq$'s until a $\lessdot$ is encountered. In (4.18), we scan backwards to $.

3. The handle contains everything to the left of the first $\gtrdot$ and to the right of the $\lessdot$ encountered in step (2), including any intervening or surrounding nonterminals. (The inclusion of surrounding nonterminals is necessary so that two adjacent nonterminals do not appear in a right-sentential form.) In (4.18), the handle is the first **id**.

If we are dealing with grammar (4.17), we then reduce **id** to E. At this point we have the right-sentential form E+**id***id**. After reducing the two remaining **id**'s to E by the same steps, we obtain the right-sentential form $E+E*E$. Consider now the string $+*$ obtained by deleting the nonterminals. Inserting the precedence relations, we get

$$\$ \lessdot + \lessdot * \gtrdot \$$$

indicating that the left end of the handle lies between + and * and the right end between * and $. These precedence relations indicate that, in the right-sentential form $E+E*E$, the handle is $E*E$. Note how the E's surrounding the * become part of the handle.

Since the nonterminals do not influence the parse, we need not worry about distinguishing among them. A single marker "nonterminal" can be kept on the stack of a shift-reduce parser to indicate placeholders for attribute values.

It may appear from the discussion above that the entire right-sentential form must be scanned at each step to find the handle. Such is not the case if we use a stack to store the input symbols already seen and if the precedence relations are used to guide the actions of a shift-reduce parser. If the precedence relation $<\cdot$ or $\doteq$ holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts; it has not yet found the right end of the handle. If the relation $\cdot>$ holds, a reduction is called for. At this point the parser has found the right end of the handle, and the precedence relations can be used to find the left end of the handle in the stack.

If no precedence relation holds between a pair of terminals (indicated by a blank entry in Fig. 4.23), then a syntactic error has been detected and an error recovery routine must be invoked, as discussed later in this section. The above ideas can be formalized by the following algorithm.

Algorithm 4.5. Operator-precedence parsing algorithm.

Input. An input string w and a table of precedence relations.

Output. If w is well formed, a *skeletal* parse tree, with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.

Method. Initially, the stack contains \$ and the input buffer the string $w\$$. To parse, we execute the program of Fig. 4.24. □

```
set ip to point to the first symbol of w$;
repeat forever
    if $ is on top of the stack and ip points to $ then
        return
    else begin
        let a be the topmost terminal symbol on the stack
            and let b be the symbol pointed to by ip;
        if a <· b or a ≐ b then begin
            push b onto the stack;
            advance ip to the next input symbol;
        end;
        else if a ·> b then          /* reduce */
            repeat
                pop the stack
            until the top stack terminal is related by <·
                to the terminal most recently popped
        else error()
    end
```

Fig. 4.24. Operator-precedence parsing algorithm.

Operator-Precedence Relations from Associativity and Precedence

We are always free to create operator-precedence relations any way we see fit and hope that the operator-precedence parsing algorithm will work correctly when guided by them. For a language of arithmetic expressions such as that generated by grammar (4.17) we can use the following heuristic to produce a proper set of precedence relations. Note that grammar (4.17) is ambiguous, and right-sentential forms could have many handles. Our rules are designed to select the "proper" handles to reflect a given set of associativity and precedence rules for binary operators.

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot\ \theta_1$. For example, if $*$ has higher precedence than $+$, make $* \cdot> +$ and $+ <\cdot *$. These relations ensure that, in an expression of the form $E+E*E+E$, the central $E*E$ is the handle that will be reduced first.

2. If θ_1 and θ_2 are operators of equal precedence (they may in fact be the same operator), then make $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if the operators are left-associative, or make $\theta_1 <\cdot\ \theta_2$ and $\theta_2 <\cdot\ \theta_1$ if they are right-associative. For example, if $+$ and $-$ are left-associative, then make $+ \cdot> +$, $+ \cdot> -$, $- \cdot> -$, and $- \cdot> +$. If $\uparrow$ is right associative, then make $\uparrow\ <\cdot \uparrow$. These relations ensure that $E-E+E$ will have handle $E-E$ selected and $E \uparrow E \uparrow E$ will have the last $E \uparrow E$ selected.

3. Make $\theta <\cdot$ **id**, **id** $\cdot> \theta$, $\theta <\cdot$ (, ($<\cdot\ \theta$,) $\cdot> \theta$, $\theta \cdot>$), $\theta \cdot> \$$, and $\$ <\cdot\ \theta$ for all operators θ. Also, let

($\doteq$)	$\$ <\cdot$ (	$\$ <\cdot$ **id**
($<\cdot$ (	**id** $\cdot> \$$	) $\cdot> \$$
($<\cdot$ **id**	**id** $\cdot>$)	) $\cdot>$)

These rules ensure that both **id** and (E) will be reduced to E. Also, \$ serves as both the left and right endmarker, causing handles to be found between \$'s wherever possible.

Example 4.28. Figure 4.25 contains the operator-precedence relations for grammar (4.17), assuming

1. $\uparrow$ is of highest precedence and right-associative,

2. $*$ and $/$ are of next highest precedence and left-associative, and

3. $+$ and $-$ are of lowest precedence and left-associative,

(Blanks denote error entries.) The reader should try out the table to see that it works correctly, ignoring problems with unary minus for the moment. Try the table on the input **id** $*$ (**id** $\uparrow$ **id**) $-$ **id** $/$ **id**, for example. □

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig. 4.25. Operator-precedence relations.

Handling Unary Operators

If we have a unary operator such as $\neg$ (logical negation), which is not also a binary operator, we can incorporate it into the above scheme for creating operator-precedence relations. Supposing $\neg$ to be a unary prefix operator, we make $\theta <\cdot \neg$ for any operator θ, whether unary or binary. We make $\neg \cdot> \theta$ if $\neg$ has higher precedence than θ and $\neg <\cdot \theta$ if not. For example, if $\neg$ has higher precedence than &, and & is left-associative, we would group $E\&\neg E\&E$ as $(E\&(\neg E))\&E$, by these rules. The rule for unary postfix operators is analogous.

The situation changes when we have an operator like the minus sign − that is both unary prefix and binary infix. Even if we give unary and binary minus the same precedence, the table of Fig. 4.25 will fail to parse strings like **id*****−id** correctly. The best approach in this case is to use the lexical analyzer to distinguish between unary and binary minus, by having it return a different token when it sees unary minus. Unfortunately, the lexical analyzer cannot use lookahead to distinguish the two; it must remember the previous token. In Fortran, for example, a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol.

Precedence Functions

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two *precedence functions* f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b,

1. $f(a) < g(b)$ whenever $a <\cdot b$,
2. $f(a) = g(b)$ whenever $a \doteq b$, and
3. $f(a) > g(b)$ whenever $a \cdot> b$.

Thus the precedence relation between a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$. Note, however, that error entries in the precedence matrix are obscured, since one of (1), (2), or (3) holds no matter what $f(a)$ and $g(b)$ are. The loss of error detection capability is generally not considered serious enough to prevent the using of precedence functions where possible; errors can still be caught when a reduction is called for and no handle can be found.

Not every table of precedence relations has precedence functions to encode it, but in practical cases the functions usually exist.

Example 4.29. The precedence table of Fig. 4.25 has the following pair of precedence functions,

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

For example, $* \lessdot$ **id**, and $f(*) < g(\textbf{id})$. Note that $f(\textbf{id}) > g(\textbf{id})$ suggests that **id** $\gtrdot$ **id**; but, in fact, no precedence relation holds between **id** and **id**. Other error entries in Fig. 4.25 are similarly replaced by one or another precedence relation. □

A simple method for finding precedence functions for a table, if such functions exist, is the following.

Algorithm 4.6. Constructing precedence functions.

Input. An operator precedence matrix.

Output. Precedence functions representing the input matrix, or an indication that none exist.

Method.

1. Create symbols f_a and g_a for each a that is a terminal or $.

2. Partition the created symbols into as many groups as possible, in such a way that if $a \doteq b$, then f_a and g_b are in the same group. Note that we may have to put symbols in the same group even if they are not related by $\doteq$. For example, if $a \doteq b$ and $c \doteq b$, then f_a and f_c must be in the same group, since they are both in the same group as g_b. If, in addition, $c \doteq d$, then f_a and g_d are in the same group even though $a \doteq d$ may not hold.

3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if $a \lessdot b$, place an edge from the group of g_b to the group of f_a. If $a \gtrdot b$, place an edge from the group of f_a to that of g_b. Note that an edge or path from f_a to g_b means that $f(a)$ must exceed $g(b)$; a path from g_b to f_a means that $g(b)$ must exceed $f(a)$.

4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path

beginning at the group of f_a; let $g(a)$ be the length of the longest path from the group of g_a. □

Example 4.30. Consider the matrix of Fig. 4.23. There are no $\doteq$ relationships, so each symbol is in a group by itself. Figure 4.26 shows the graph constructed using Algorithm 4.6.

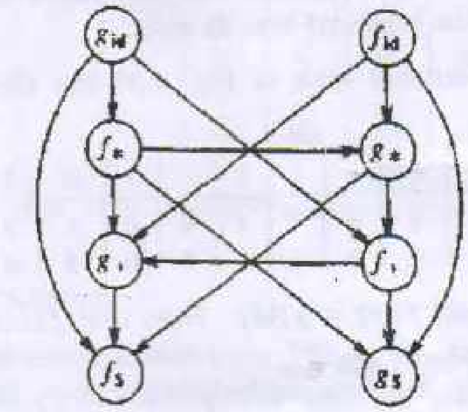


Fig. 4.26. Graph representing precedence functions.

There are no cycles, so precedence functions exist. As $f_\$$ and $g_\$$ have no out-edges, $f(\$) = g(\$) = 0$. The longest path from g_+ has length 1, so $g(+) = 1$. There is a path from g_{id} to f_* to g_* to f_+ to g_+ to $f_\$$, so $g(\text{id}) = 5$. The resulting precedence functions are:

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

□

Error Recovery in Operator-Precedence Parsing

There are two points in the parsing process at which an operator-precedence parser can discover syntactic errors:

1. If no precedence relation holds between the terminal on top of the stack and the current input.[1]
2. If a handle has been found, but there is no production with this handle as a right side.

Recall that the operator-precedence parsing algorithm (Algorithm 4.5) appears to reduce handles composed of terminals only. However, while nonterminals

[1] In compilers using precedence functions to represent the precedence tables, this source of error detection may be unavailable.

are treated anonymously, they still have places held for them on the parsing stack. Thus when we talk in (2) above about a handle matching a production's right side, we mean that the terminals are the same and the positions occupied by nonterminals are the same.

We should observe that, besides (1) and (2) above, there are no other points at which errors could be detected. When scanning down the stack to find the left end of the handle in steps (10-12) of Fig. 4.24, the operator-precedence parsing algorithm, we are sure to find a $<\cdot$ relation, since \$ marks the bottom of stack and is related by $<\cdot$ to any symbol that could appear immediately above it on the stack. Note also that we never allow adjacent symbols on the stack in Fig. 4.24 unless they are related by $<\cdot$ or $\doteq$. Thus steps (10-12) must succeed in making a reduction.

Just because we find a sequence of symbols $a <\cdot b_1 \doteq b_2 \doteq \cdots \doteq b_k$ on the stack, however, does not mean that $b_1 b_2 \cdots b_k$ is the string of terminal symbols on the right side of some production. We did not check for this condition in Fig. 4.24, but we clearly can do so, and in fact we must do so if we wish to associate semantic rules with reductions. Thus we have an opportunity to detect errors in Fig. 4.24, modified at steps (10-12) to determine what production is the handle in a reduction.

Handling Errors During Reductions

We may divide the error detection and recovery routine into several pieces. One piece handles errors of type (2). For example, this routine might pop symbols off the stack just as in steps (10-12) of Fig. 4.24. However, as there is no production to reduce by, no semantic actions are taken; a diagnostic message is printed instead. To determine what the diagnostic should say, the routine handling case (2) must decide what production the right side being popped "looks like." For example, suppose *abc* is popped, and there is no production right side consisting of *a*, *b* and *c* together with zero or more nonterminals. Then we might consider if deletion of one of *a*, *b*, and *c* yields a legal right side (nonterminals omitted). For example, if there were a right side *aEcE*, we might issue the diagnostic

`illegal` *b* `on line` (line containing *b*)

We might also consider changing or inserting a terminal. Thus if *abEdc* were a right side, we might issue a diagnostic

`missing` *d* `on line` (line containing *c*)

We may also find that there is a right side with the proper sequence of terminals, but the wrong pattern of nonterminals. For example, if *abc* is popped off the stack with no intervening or surrounding nonterminals, and *abc* is not a right side but *aEbc* is, we might issue a diagnostic

`missing` *E* `on line` (line containing *b*)

Here E stands for an appropriate syntactic category represented by nonterminal E. For example, if a, b, or c is an operator, we might say "expression;" if a is a keyword like **if**, we might say "conditional."

In general, the difficulty of determining appropriate diagnostics when no legal right side is found depends upon whether there are a finite or infinite number of possible strings that could be popped in lines (10-12) of Fig. 4.24. Any such string $b_1b_2 \cdots b_k$ must have $\doteq$ relations holding between adjacent symbols, so $b_1 \doteq b_2 \doteq \cdots \doteq b_k$. If an operator precedence table tells us that there are only a finite number of sequences of terminals related by $\doteq$, then we can handle these strings on a case-by-case basis. For each such string x we can determine in advance a minimum-distance legal right side y and issue a diagnostic implying that x was found when y was intended.

It is easy to determine all strings that could be popped from the stack in steps (10-12) of Fig. 4.24. These are evident in the directed graph whose nodes represent the terminals, with an edge from a to b if and only if $a \doteq b$. Then the possible strings are the labels of the nodes along paths in this graph. Paths consisting of a single node are possible. However, in order for a path $b_1b_2 \cdots b_k$ to be "poppable" on some input, there must be a symbol a (possibly \$) such that $a \lessdot b_1$. Call such a b_1 *initial*. Also, there must be a symbol c (possibly \$) such that $b_k \gtrdot c$. Call b_k *final*. Only then could a reduction be called for and $b_1b_2 \cdots b_k$ be the sequence of symbols popped. If the graph has a path from an initial to a final node containing a cycle, then there are an infinity of strings that might be popped; otherwise, there are only a finite number.

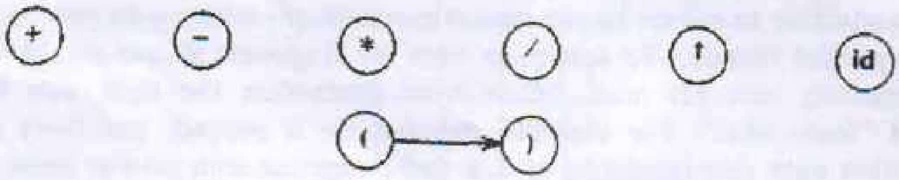


Fig. 4.27. Graph for precedence matrix of Fig. 4.25.

Example 4.31. Let us reconsider grammar (4.17):

$$E \to E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$$

The precedence matrix for this grammar was shown in Fig. 4.25, and its graph is given in Fig. 4.27. There is only one edge, because the only pair related by $\doteq$ is the left and right parenthesis. All but the right parenthesis are initial, and all but the left parenthesis are final. Thus the only paths from an initial to a final node are the paths +, −, *, /, **id**, and ↑ of length one, and the path from (to) of length two. There are but a finite number, and each corresponds to the terminals of some production's right side in the grammar. Thus the error checker for reductions need only check that the proper set of nonterminal markers appears among the terminal strings being reduced. Specifically, the checker does the following:

1. If +, −, *, /, or ↑ is reduced, it checks that nonterminals appear on both sides. If not, it issues the diagnostic

   `missing operand`

2. If **id** is reduced, it checks that there is no nonterminal to the right or left. If there is, it can warn

   `missing operator`

3. If ( ) is reduced, it checks that there is a nonterminal between the parentheses. If not, it can say

   `no expression between parentheses`

Also it must check that no nonterminal appears on either side of the parentheses. If one does, it issues the same diagnostic as in (2). □

If there are an infinity of strings that may be popped, error messages cannot be tabulated on a case-by-case basis. We might use a general routine to determine whether some production right side is close (say distance 1 or 2, where distance is measured in terms of tokens, rather than characters, inserted, deleted, or changed) to the popped string and if so, issue a specific diagnostic on the assumption that that production was intended. If no production is close to the popped string, we can issue a general diagnostic to the effect that "something is wrong in the current line."

Handling Shift/Reduce Errors

We must now discuss the other way in which the operator-precedence parser detects errors. When consulting the precedence matrix to decide whether to shift or reduce (lines (6) and (9) of Fig. 4.24), we may find that no relation holds between the top stack symbol and the first input symbol. For example, suppose a and b are the two top stack symbols (b is at the top), c and d are the next two input symbols, and there is no precedence relation between b and c. To recover, we must modify the stack, input or both. We may change symbols, insert symbols onto the input or stack, or delete symbols from the input or stack. If we insert or change, we must be careful that we do not get into an infinite loop, where, for example, we perpetually insert symbols at the beginning of the input without being able to reduce or to shift any of the inserted symbols.

One approach that will assure us no infinite loops is to guarantee that after recovery the current input symbol can be shifted (if the current input is \$, guarantee that no symbol is placed on the input, and the stack is eventually shortened). For example, given ab on the stack and cd on the input, if $a \leq\cdot c$[2]

[2] We use $\leq\cdot$ to mean $<\cdot$ or $\doteq$.

we might pop b from the stack. Another choice is to delete c from the input if $b \leq\cdot d$. A third choice is to find a symbol e such that $b \leq\cdot e \leq\cdot c$ and insert e in front of c on the input. More generally, we might insert a string of symbols such that

$$b \leq\cdot e_1 \leq\cdot e_2 \leq\cdot \cdots \leq\cdot e_n \leq\cdot c$$

if a single symbol for insertion could not be found. The exact action chosen should reflect the compiler designer's intuition regarding what error is likely in each case.

For each blank entry in the precedence matrix we must specify an error-recovery routine; the same routine could be used in several places. Then when the parser consults the entry for a and b in step (6) of Fig. 4.24, and no precedence relation holds between a and b, it finds a pointer to the error-recovery routine for this error.

Example 4.32. Consider the precedence matrix of Fig. 4.25 again. In Fig. 4.28, we show the rows and columns of this matrix that have one or more blank entries, and we have filled in these blanks with the names of error handling routines.

	id	(	)	$
id	e3	e3	·>	·>
(	<·	<·	≐	e4
)	e3	e3	·>	·>
$	<·	<·	e2	e1

Fig. 4.28. Operator-precedence matrix with error entries.

The substance of these error handling routines is as follows:

e1: /* called when whole expression is missing */
insert **id** onto the input
issue diagnostic: "missing operand"

e2: /* called when expression begins with a right parenthesis */
delete) from the input
issue diagnostic: "unbalanced right parenthesis"

e3: /* called when **id** or) is followed by **id** or (*/
insert + onto the input
issue diagnostic: "missing operator"

e4: /* called when expression ends with a left parenthesis */
pop (from the stack
issue diagnostic: "missing right parenthesis"

Let us consider how this error-handling mechanism would treat the erroneous input **id** +). The first actions taken by the parser are to shift **id**, reduce it to E (we again use E for anonymous nonterminals on the stack), and then to shift the +. We now have configuration

STACK	INPUT
$\$E+$	$)\$$

Since $+ \cdot\!>)$ a reduction is called for, and the handle is +. The error checker for reductions is required to inspect for E's to left and right. Finding one missing, it issues the diagnostic

```
missing operand
```

and does the reduction anyway.

Our configuration is now

$\$E$	$)\$$

There is no precedence relation between \$ and), and the entry in Fig. 4.28 for this pair of symbols is e2. Routine e2 causes diagnostic

```
unbalanced right parenthesis
```

to be printed and removes the right parenthesis from the input. We are now left with the final configuration for the parser.

$\$E$	$\$$

□

4.7 LR PARSERS

This section presents an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, k is assumed to be 1. LR parsing is attractive for a variety of reasons.

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

handle - substring that match the right side of the prod

classmate
Date
Page

Shift Reduce Process:

stack	RSF	Action
$	000111	shift 01
$01	0011	reduce S→01
$S	0011	

27/04/18: Conflict in shift reduce parsing:

1. Shift Reduce conflict — example:

Stack	RSF	Action
$	iEtSeS$	
$\underbrace{\$iEtS}_{S}$	eS$	→ shift else reduce they

2. Reduce-Reduce conflict
↓
2 production with same production on the right

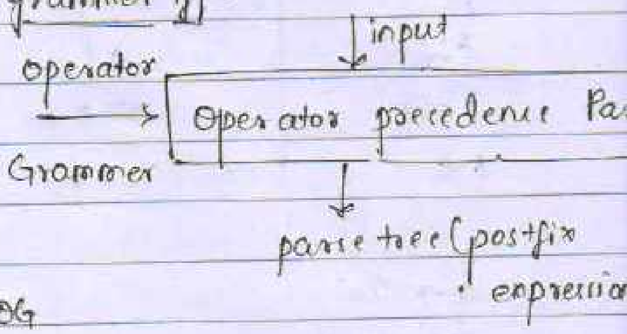
S→id
P→id
{ Parser doesn't know either id shou be reduced to S or P.

Operator precedence Parser:

Grammer G is operator grammer iff:

i) No Eproduction

ii) No two adjacent non-terminals.



Ex: E→EAE|id } not a OG
A→*|+

E→E+E|E*E|id ✓ OG.

Steps for operator precedence parsing:

~~Problem~~:

1. Check whether the given grammer is operator grammer or not. possible try to convert.
2. Generate operator relation table
3. Parse the inputstring
4. Construct the parse tree.

PROBLEM:

1. Construct the operator precedence parser for the given grammer and parse the given input string.

$E \rightarrow EAE \mid id$

$A \rightarrow + \mid *$

(i) Converting to operator grammer

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

(ii) Operator Relation Table:

Assumption: identifier - highest precedence   Right associative (<·)

* - left associative (·>)

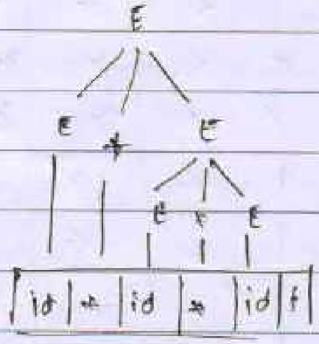
+ - left associative.

$ - least precedence

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-

Accept

(iv)



(iii) Parse the input string.

Stack	input	Relation	Action
$	id+id*id $	<·	push id
$id	+id*id $	·>	pop id
$	+id*id $	<·	push +
$+	id*id $	<·	push id
$+id	*id$	·>	pop id
$+	*id $	<·	push *
$+*	id $	<·	push id
$+*id	$	·>	pop id
$+*	$	·>	pop *
$+	$	·>	pop +
$	$	-	Accept

28|04|18

2. E → E+E | E−E | E*E | E/E | E↑E | (E) | id.

input: id *(id↑id) −id / id

i) It is an OG

ii) Generate relation table

id - highest precedence   + − → left associative

( ) - right associative   \$ - least precedence

↑ - right associative

*/ - left associative.

	id	+	−	*	/	↑	(	)	\$
id	–	·>	·>	·>	·>	·>	–	·>	·>
+	<·	·>	·>	<·	<·	<·	<·	·>	·>
−	<·	·>	·>	<·	<·	<·	<·	·>	·>
*	<·	·>	·>	·>	·>	<·	<·	·>	·>
/	<·	·>	·>	·>	·>	<·	<·	·>	·>
↑	<·	·>	·>	·>	·>	<·	<·	·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	–
)	–	·>	·>	·>	·>	·>	–	·>	·>
\$	<·	<·	<·	<·	<·	<·	<·	–	Accept

(iii) Parse input:

Stack	input	Relation	Action
\$	id *(id↑id)-id/id\$	<·	push id
\$id	* (id↑id)-id/id\$	·>	pop id
\$	* (id↑id)-id/id\$	<·	push *
\$*	(id↑id) - id/id\$	<·	push (
\$*(	id↑id) - id/id\$	<·	push id
\$*(id	↑id) - id/id\$	·>	pop id
\$*(	↑id) - id/id\$	<·	push↑
\$*(↑	id)-id/ id\$	<·	push id
\$*(↑id	)-id/id\$	·>	pop id
\$*(↑	) - id / id\$	·>	pop ↑
\$*(	) -id/id\$	≐	push)
\$*()	-id / id\$	·>	pop),(
\$*	-id /id\$	·>	pop *

$-	id/id$	<·	push id
$-id	/id $	·>	pop id
$ -	/id $	<·	push /
$-/	id $	<·	push id
$-/ id	$	·>	pop id
$ - /	$	·>	pop /
$-	$	·>	pop -
$	$	-	Accept

Algorithm: Operator precedence parsing algorithm:

Input: An input string w and a table of precedence relations

Output: if w is well formed, a skeletal parse tree, with a placeholder non-terminal E labelling all interior nodes otherwise, an error indication.

Method: Initially the stack contains $ and the input buffer the input buffer the string w$. To parse, we execute the program.

1. Set input to point to the first symbol of w$
2. repeat forever:
3. if $ is on top of stack and ip points to $ then
4. return
5. Let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by ip
6. if a <· b or a ≐ b then begin
7. push b onto the stack
8. advance ip to next input symbol
   end;
9. else if a ·> b then
10. repeat
11. pop the stack over any of ≐
12. until the top stack terminal is related by <· to the terminal most recently popped
13. else error()

classmate
Date
Page

30/04/18

3. S → (L) | a
L → L,S | S
i/p : (a, (a,a))

a - highest (- left
$ least , - left

i) It is an OG

ii) Relation table:

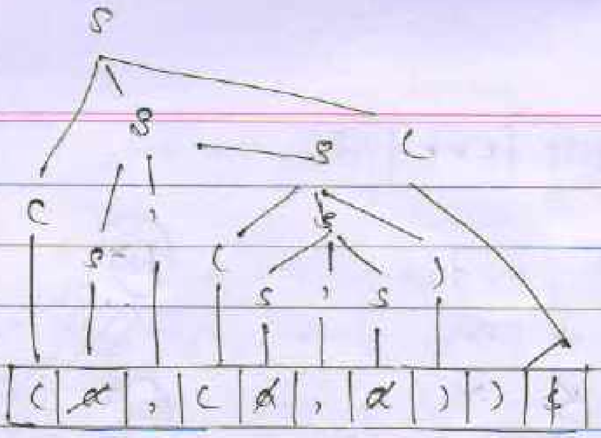
	id	,	(	)	$
id	-	⋗	- ~~⋗~~ -	⋗	⋗
,	⋖	-	⋖	⋗	⋗
(	⋖	⋖	⋖	≐	-
)	-	⋗	-	⋗	⋗
$	⋖	⋖	⋖	-	Acept

Note: Do remember to put the relation operator based on associativity i.e which one is evaluated first. After parsing we get the parse tree

iii) Input is (a, (a,a))$

Stack	Input	Relation	Action.
$	(a,(a,a))$	⋖	Push (
$(	a,(a,a))$	⋖	Push a
$(a	,(a,a))$	⋗	Pop a
$(	,(a,a))$	⋖	push ,
$(,	(a,a))$	⋖	push (
$(,(	a,a))$	⋖	push a
$(,(a	,a))$	⋗	pop a
$(,(	,a))$	⋖	~~pop~~ push ,
$(,(,	a))$	⋖	~~push~~ push
$(,(,a	))$	⋗	~~pop~~ pop
$(,(,	))$	⋗	~~pop,~~ pop
$(,(	))$	≐	~~push)~~ pus
$(,()	)$	⋗	~~pop~~ pop
$(,	)$	⋗	pop ,
$(	)$	≐	push)
$()	$	⋗	pop ()
$	$	Acept	

iv)



Drawbacks of operator relation table:

→ It is very difficult to handle tokens like '–' which has two precedence functions based on whether it is unary operator or binary operator.

→ Only small class of grammars can be parsed

→ If we ever have 4 operators, then the no. of entries in the table are 4×4 = 16 entries i.e in general, if the number of operators are n, we need $o(n^2)$ entries. To overcome this we go for operator precedence functions.

Operator precedence functions:

– The parsers doesnot store relation table instead they make use of precedence function's which map the terminal symbols to integers.

– It uses two functions i.e fa and gb for the symbols a and b.

(i) if a ⋗ b, then there is an arrow (edge) from function fa to function g_b

(ii) if a ⋖ b, then there is an edge from g_b to fa

(iii) If a ≐ b then fa = g_b are in the same group. Note that even if they are not related by ≐ directly we group them together for example if a ≐ b and c ≐ b then fa and fc are in the same group, since they are both in the same group as g_b

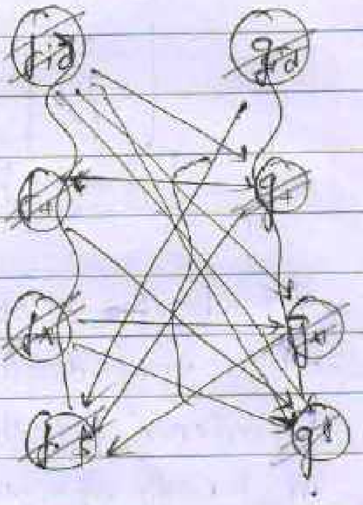
(iv) If the graph constructed has no cycle, then the precedence functions exists.

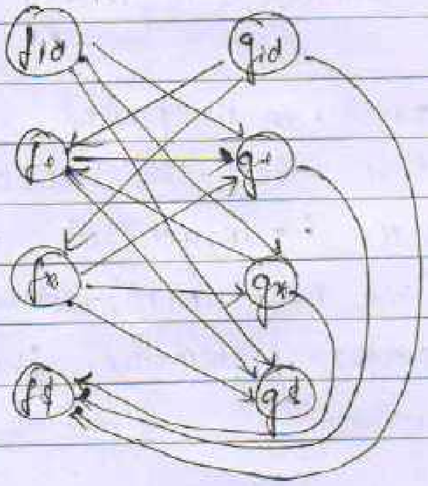
(v) Find the longest path in the function starting from terminal to $ i.e f(a) to $ and g(a) to $. using these

Drawbacks of operator relation table

Example: E → E+E | E*E | id

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	-





The longest path is

fid → g* → f+ → g+ → f$

gid → f* → g* → f+ → g+ →

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Advantages: lesser entries

disadvantages: For blank entries of relation tables we get non-blank entries in function table i.e we cant mak out the errors during parsing

<u>H.W</u> Construct an operator precedence parser for the given gram and parse an input string

✓ (i) E → E+E | E*E | (E) | id   ip: (id + id * id): it is an OG

(ii) E → E+T | T
T → T*V | V   input: a + b * c*d.
V → a | b | c | d

(ii) i) It is an OG

ii) Relation table

	id	+	*	$
id	-	·>	·>	·>
+	<·	·>	<·	·>

input a+b*c*d$

Stack	Input	Relation	Action.
$	a+b*c*d$	⋖	push a
$a	+b*c*d$	⋗	pop a
$	+b*c*d$	⋖	push +
$+	b*c*d$	⋖	push b
$+b	*c*d$	⋗	pop b
$+	*c*d$	⋖	push *
$+*	c*d$	⋖	push c
$+*c	*d$	⋗	pop c
$+*	*d$	⋗	pop *
$+	*d$	⋖	push *
$+*	d$	⋖	push d
$+*d	$	⋗	pop d
$+*	$	⋗	pop *
$+	$	⋗	pop +
$	$	Accept.	

(i)

Relation table

	id	+	*	(	)	$
id	–	⋗	⋗	–	⋗	⋗
+	⋖	⋗	⋖	⋖	⋗	⋗
*	⋖	⋗	⋗	⋖	⋗	⋗
(	⋖	⋖	⋖	⋖	≐	–
)	–	⋗	⋗	–	⋗	⋗
$	⋖	⋖	⋖	⋖	–	Accept

Stack	input	Relation	Action
$	(id+id*id)$	⋖	push (
$(	id+id*id)$	⋖	push id
$(id	+id*id)$	⋗	pop id
$(	+id*id)$	⋖	push +
$(+	id*id)$	⋖	push id
$(+id	*id)$	⋗	pop id
$(+	*id)$	⋖	push *
$(+*	id)$	⋖	push id
$(+*id	)$	⋗	pop id
$(+*	)$	⋗	pop *
$(+	)$	⋗	pop +
$(	)$	≐	push)
$()	$	⋗	pop
$	$	Accept	

UNIT - 5

SYNTAX - DIRECTED TRANSLATION

CONTENTS

→ Syntax directed difinations

→ Evaluat^n orders for SDD's

→ Applicat^n of SDT

→ SDT schemes

Syntax Directed Difinations:

A syntax directed difination is a context free grammers with attributs & rules. Attributs are associated with grammer symbols & rules with productions. If 'x' is a symbol, 'a' is one of attributes then we write X.a to denl value of a at a particular parse tree Node Labelled X

* Attributs may be of many kinds : numbers, types, table references, strings, etc...

* 2 types of attributes : 1) Synthesised attr

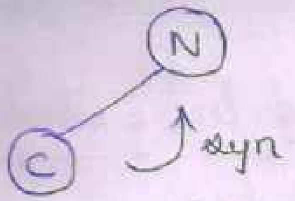
2) Inherited attr

1) Synthesised attr : A synth attr for a nonterminal A at a parse tree node N is difined by a semantic rules associated with the production at N. Note that the production must have A as its head. A synthesized attr at node N is difined only in terms of attribute values at the children of N & at N itself.

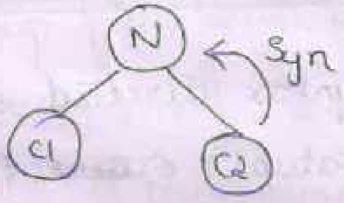
2) Inherited attr A inherited attr for a nonterminal B at a parse tree node N is defined by a semantic rule associated with the product^n at the parent of N. Note that the product^n must have B as a symbol in its body. An inherited attr at node N is defined only

in terms of attr values at N's parent, N itself & N's sibling

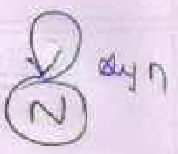
NOTE : ① Synthesized ⟨N → Node under considerat^n, C → child⟩



Case(i) Single child

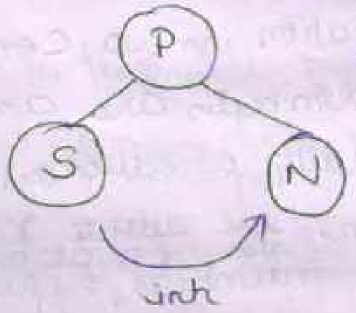


Case(ii) Rightmost child

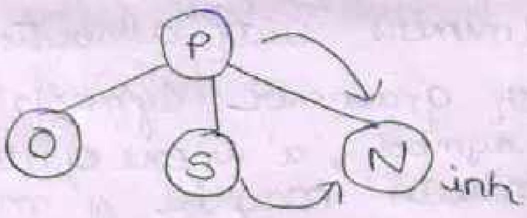


Case(iii) No child ⇒ itself.

② Inherited ⟨P → parent, N → node under considerat^n, S → Sibling, O → operator.⟩



Case(i) Sibling



Case(2) : inherited from both parent & sibling.

③ Terminals can have Synthesized attributes but not inherited attributes.

* Attr of terminals have lexical value that are supplied by lexical analyzer.

* Types of SDD : ① S Attributed SDD
② L Attributed SDD.

① S-Attributed SDD

* ~~A~~ SDD that involves only synthesized attr then it is called S-attributed SDD

* In S-attributed SDD, each rule computes

an attribute for the nonterminal at the head of a production from attribute taken from body of production.

* S-attributed SDD can be implemented naturally in conjunction with an LR parser / bottom up parser.

* <u>Annotated parse tree</u>

A parse tree showing the value(s) of attribute(s) is called an annotated parse tree.

* It is used in <u>bottom-up parser</u>
* Order of evaluation is <u>postorder traversal</u>.
* Example of S-attributed SDD.

Production	Semantic rules
1) $L \rightarrow En$	$L.val = E.val$
2) $E \rightarrow E1 + T$	$E.val = E1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow digit$	$F.val = digit.lexval$

② <u>L-Attributed SDD</u>

* Example of mixed attributes / L-attributed SDD

Production	Semantic rules
1) $T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow *FT'$	$T_1'.inh = T'.inh * F.val$ $T.syn = T_1'.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow digit$	$F.val = digit.lexval$

- A SDD which has both synthesised & inherited attributes is called as L-attributed SDD.
- It is used in top down parsers.
- Order of evaluation is topological sorting.

Evaluating Orders For SDD's

- A dependency graph is used to determine the order of computation of attributes.
- While an annotated parse trees shows the values of attributes, a dependency graph helps us to determine how those values can be computed.

DEPENDENCY GRAPHS

A dependency graph predicts the flow of information among the attribute instances in a particular parse tree: An edge from one attribute instance to an other means that the value of first is needed to compute the second.

1) For each parse tree node, say node X, the dependency graph has a node for each attribute associated with X.

2) If a semantic rule ~~(defines)~~ associated with a product^n 'p' defines the value of synthized attribute A.b in terms of value of X.c then the dependency graph has an edge from X.c to A.b

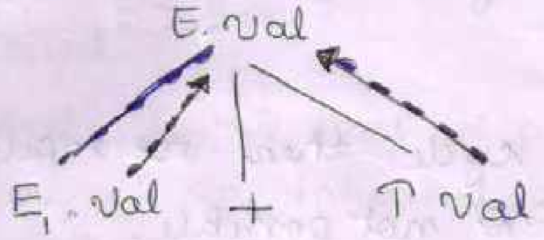
3) If a semantic rule associated with a product^n 'p' defines the value of inherited attribute B.c in terms of value of X.a then the

dependency graph has edge from X.c to B.c

Eg1:

production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

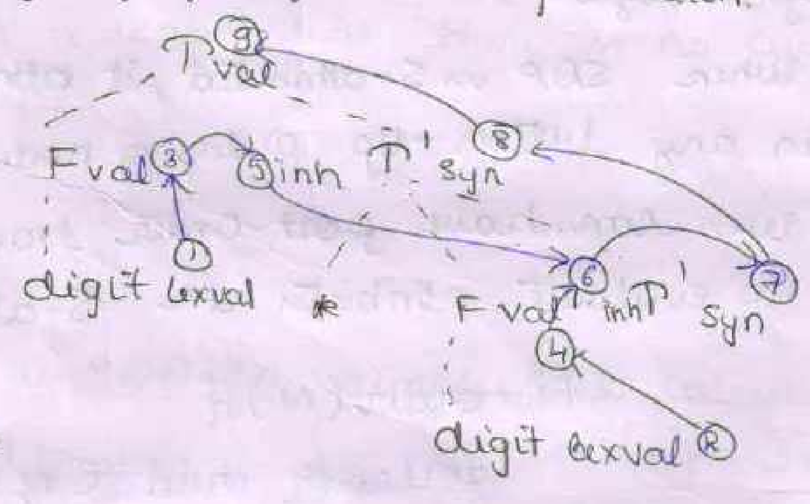
fig: E.val is synthesized from E_1.val & T.val



Eg2:

Production	Semantic Rule
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * FT'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

fig: Dependency graph for above production.



Ordering the evaluation of attributes

- If the dependency graph has an edge from node M to N, then the attribute corresponding to M must be evaluated before attribute of N.

* Thus the only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots N_k$ ∃ if there is an edge of the dependency graph from N_i to N_j then $i<j$.

* Such an ordering is called a topological sorting of a graph

* If there is any cycle then no topological sorts, i.e., evaluation of SDD not possible.

* Eg: For dependency graph for (Eg 2) in previous page
topological sorting: 1, 2, 3, 4, 5, 6, 7, 8, 9

or

1, 3, 5, 2, 4, 6, 7, 8, 9

S-Attributed Definations

→ An SDD is S-attributed if every attribute is synthesized.

→ When SDD is S-attributed, it attributes evaluated in any bottom-up order of nodes of parse trees.

→ we can have post-order traversal of parse tree to evaluate attributes in S-attributed definat^n.

```
Postorder(N) {
    for(each child C of N, from the left)
        postorder(C);
    evaluate the attributes associated with
    Node N;
}
```

→ S-Attributed definations can be implemented during bottom up parsing without the need to explicitly create parse trees.

L-Attributed Definations

- ★ A SDD is L-attributed if the edges in dependency graph goes from Left to Right but not from Right to left.
- ★ More precisely, each attribute must be either
  - → Synthesized.
  - → Inherited, but if there is a production $A \to X_1 X_2 \ldots X_n$, & there is an inherited attribute $X_i.a$ Computed by a rule associated with this production then the rule many only use:
    - (a) Inherited attributes associated with the head A.
    - (b) Either inherited or synthesized attr associated with the occurance of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.
    - (c) Inherited/Synthesized attr associated with the, occurance of X_i itself but only in such a way that there is no cycle in the graph.

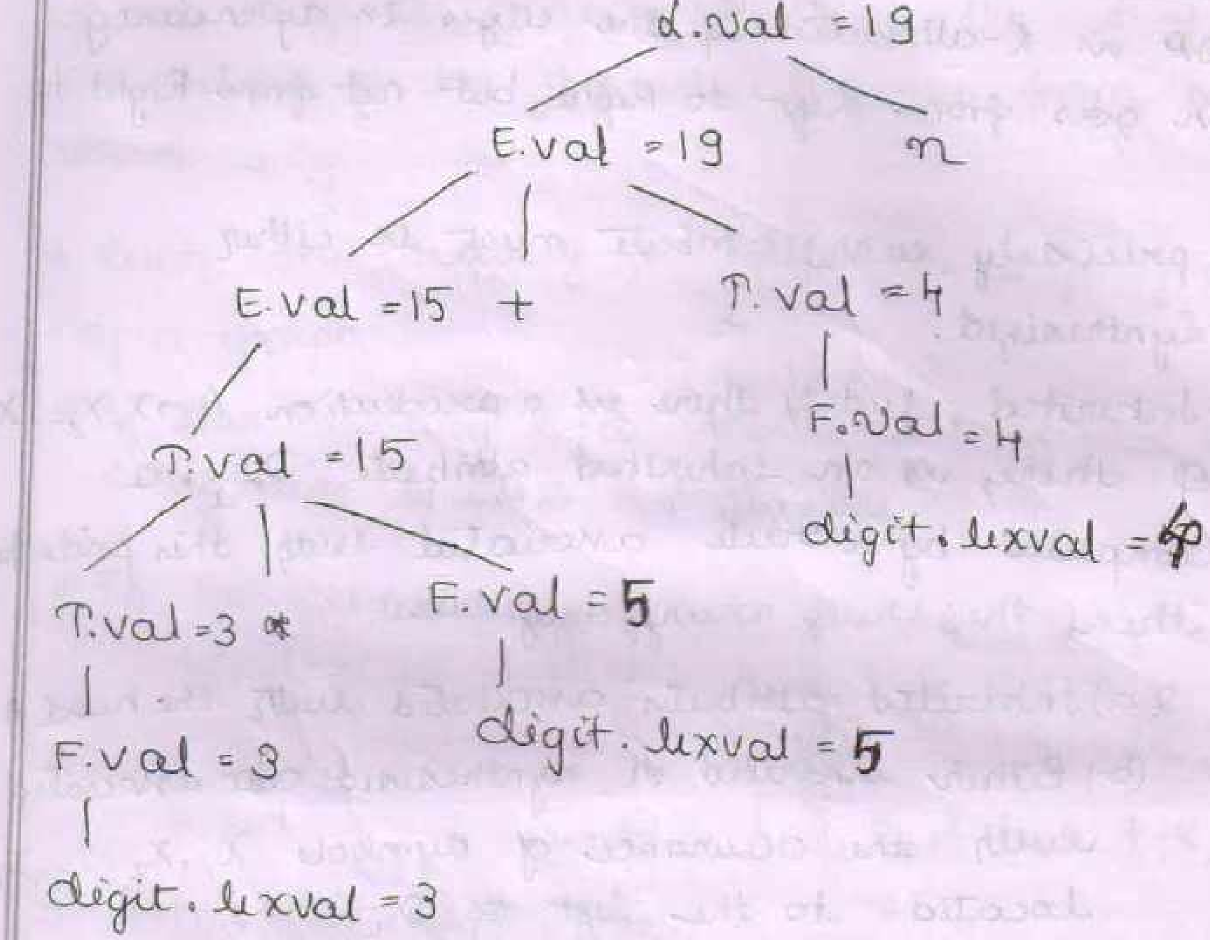
PROBLEMS

(5.1) Write a SDD for simple desk calculator

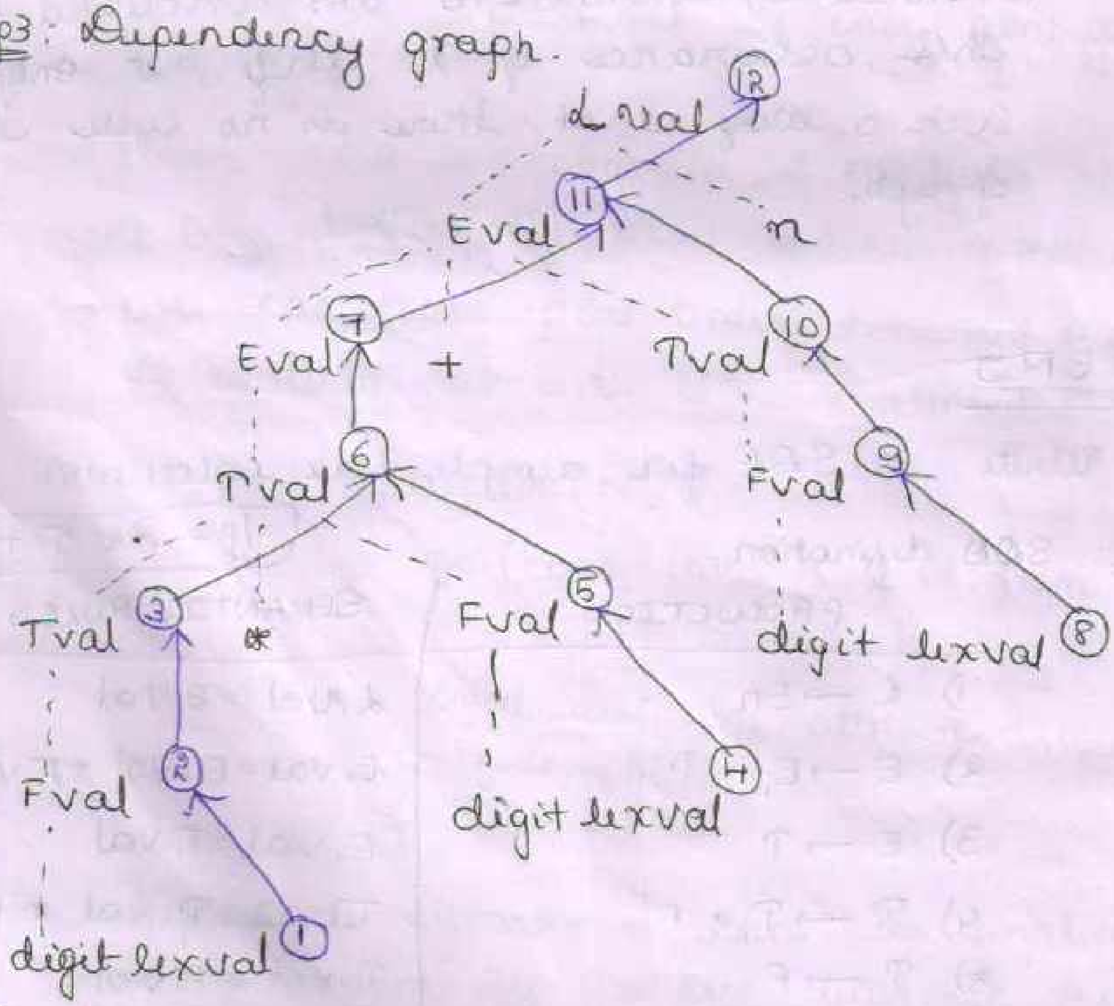
Soln: SDD defination {i/p = 3*5+4n}

PRODUCTION	SEMANTIC RULES
1) $L \to En$	L.val = E.val
2) $E \to E_1 + T$	E.val = E_1.val + T.val
3) $E \to T$	E.val = T.val
4) $T \to T_1 * F$	T.val = T_1.val * F.val
5) $T \to F$	T.val = F.val
6) $F \to (E)$	F.val = E.val
7) $F \to digit$	F.val = digit.lexval

Step2: Annotated Parse tree



Step3: Dependency graph.



Step4: Topological Ordering : ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫

) write the SDD & construct Annotated parse tree, dependency graph.

a) (3 + 4) * (5+6) n

b) 1 * 2 * 3 * (4+5) n

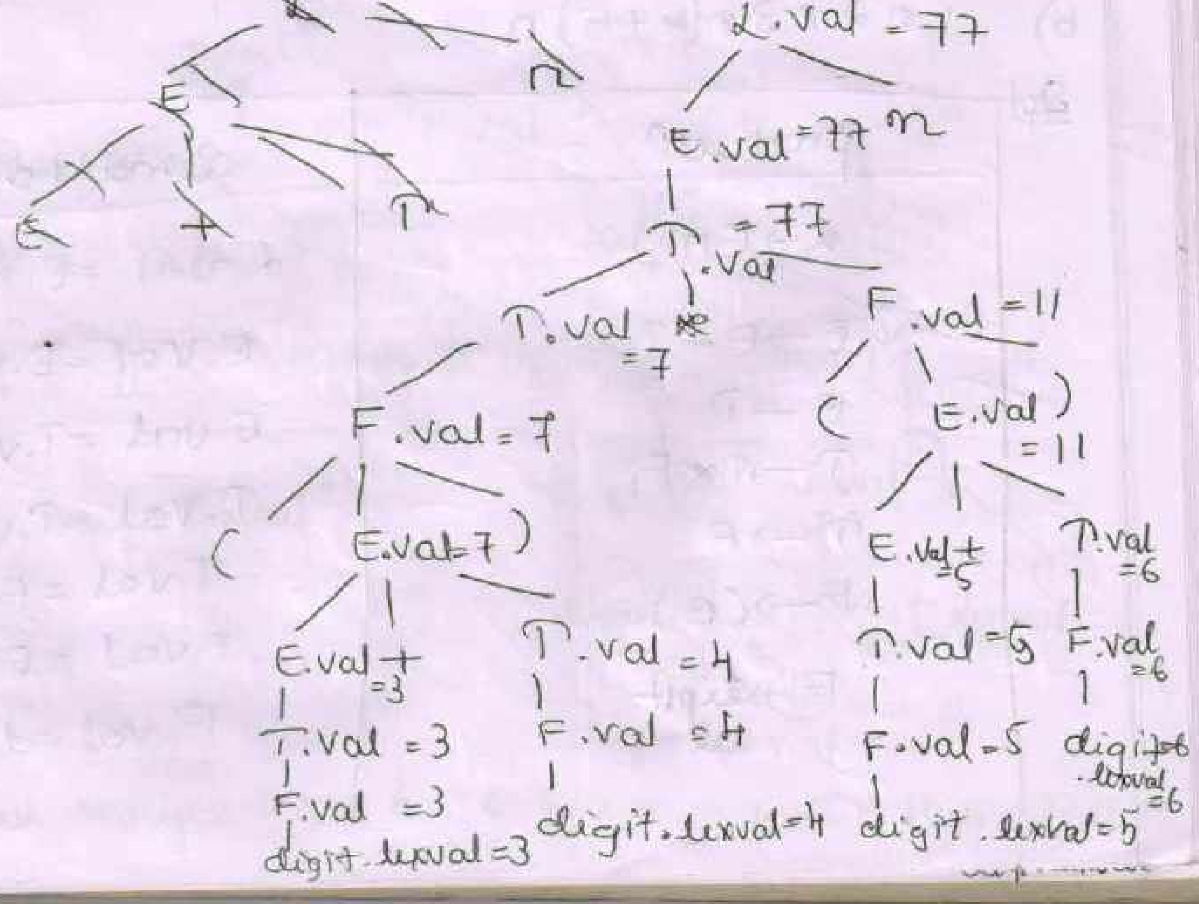
c) (9 + 8 *(7 + 6) + 5) * 4 n

a) (3 + 4) * (5+6) n

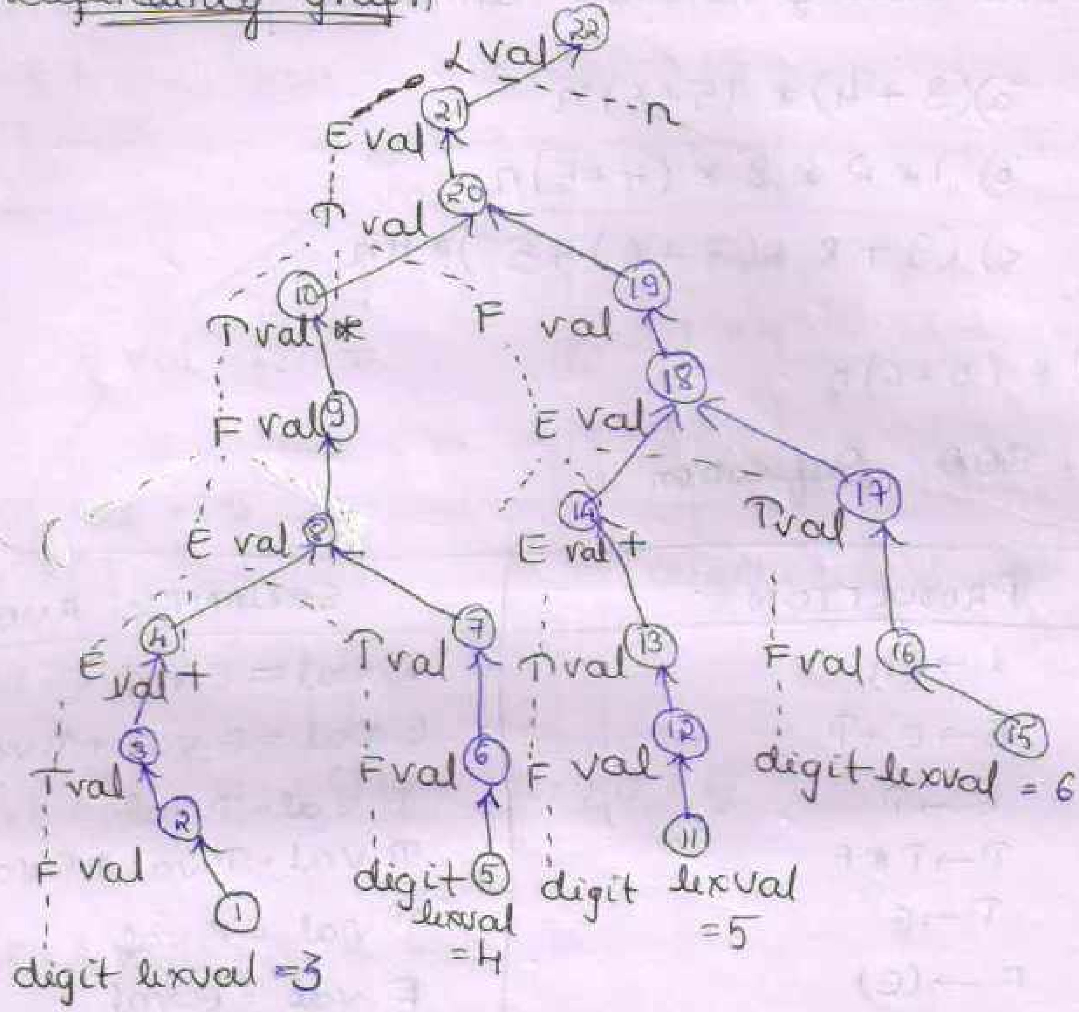
Step1 : SDD Defination

PRODUCTION	SEMANTIC RULES
L → En	L.val = E.val
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Step2 : Annotated Parse tree



Step3: Dependency Graph



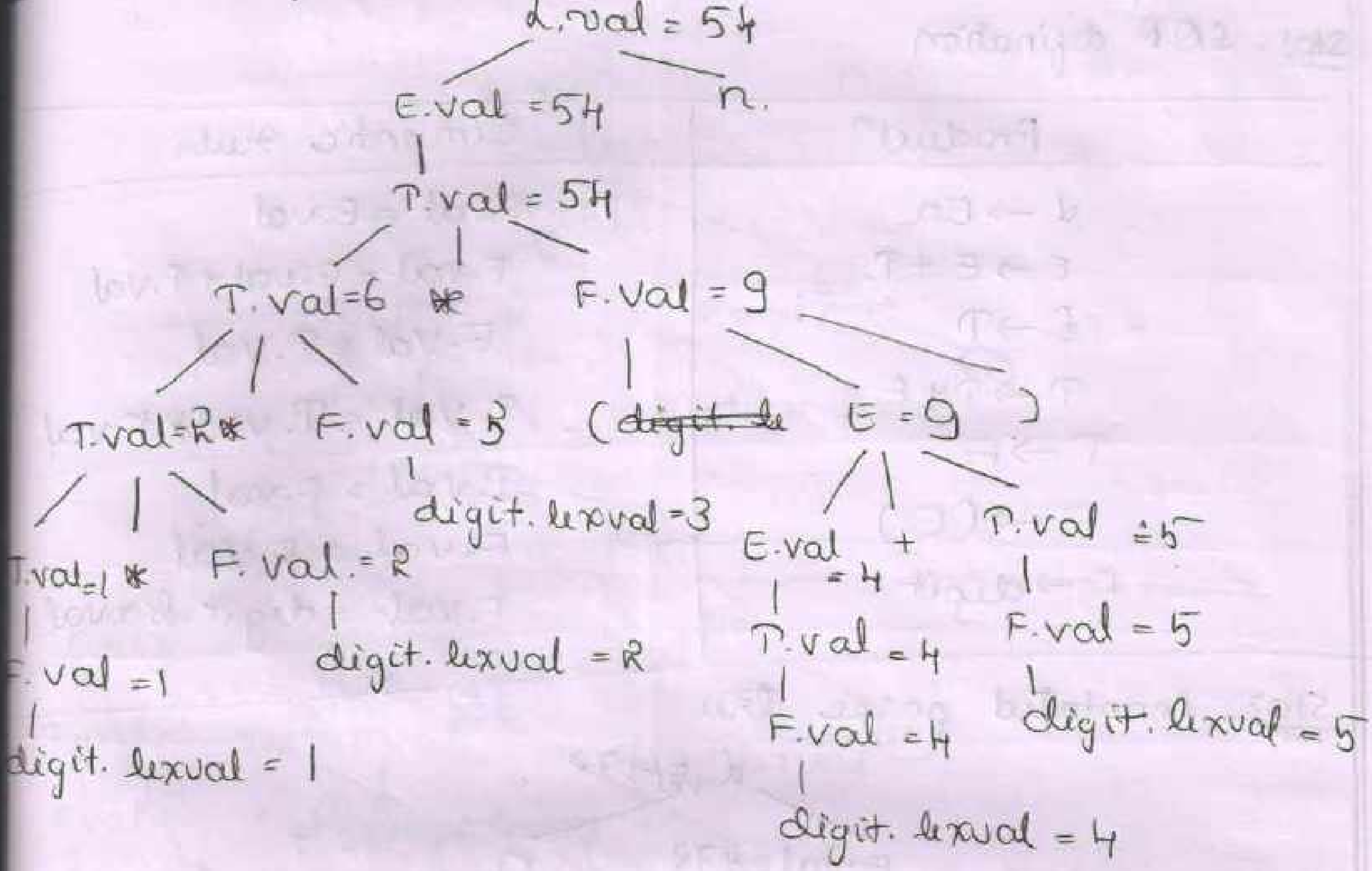
Step4 : Topological sorting : ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ ⑯ ⑰ ⑱ ⑲ ⑳ ㉑ ㉒.

b) 1 * 2 * 3 *(4+5) n

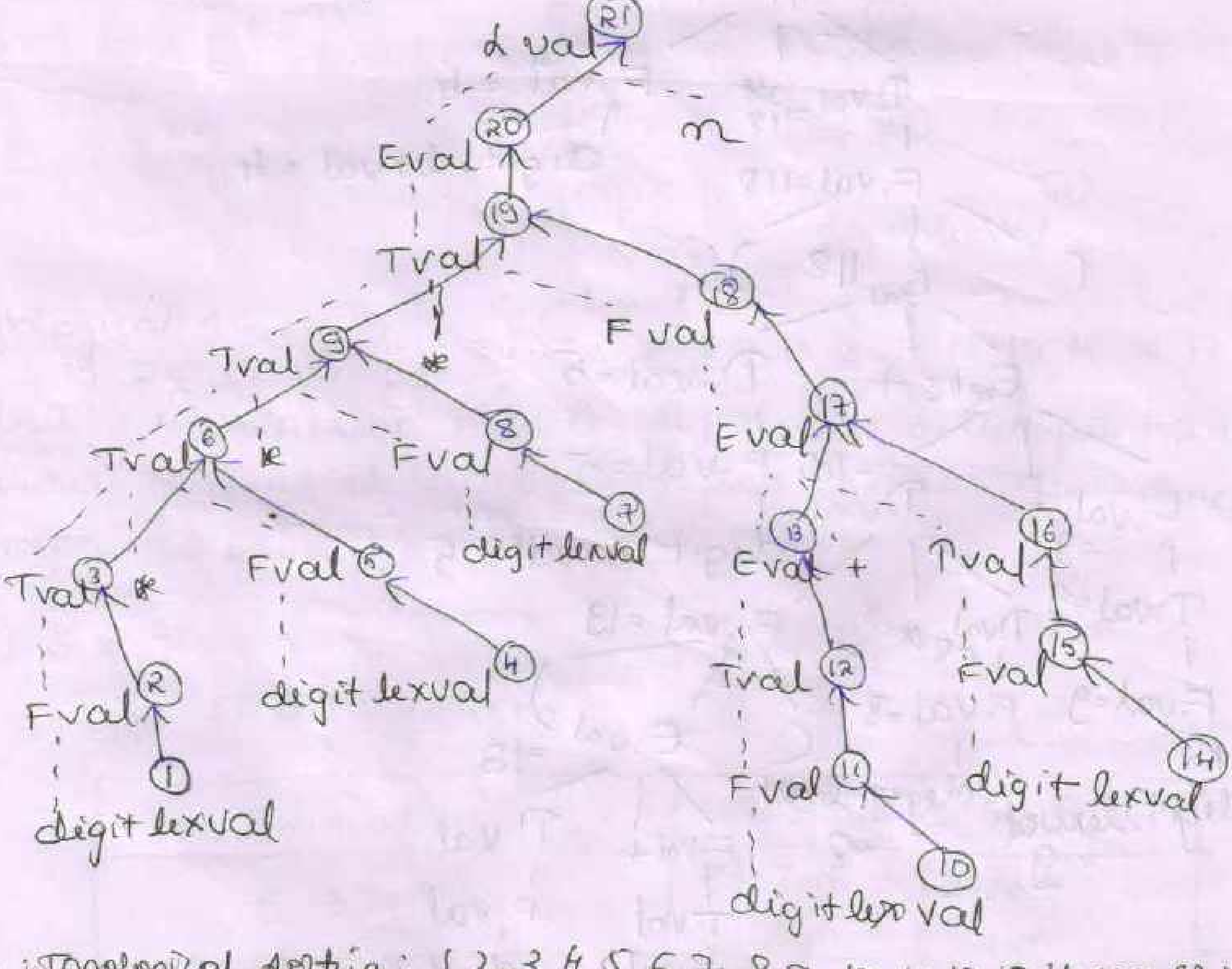
Step1

productⁿ	Semantic rules
L → En	L.val = E.val
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit. lexval

2: Anotated parse tree



Step3 : Dependency graph



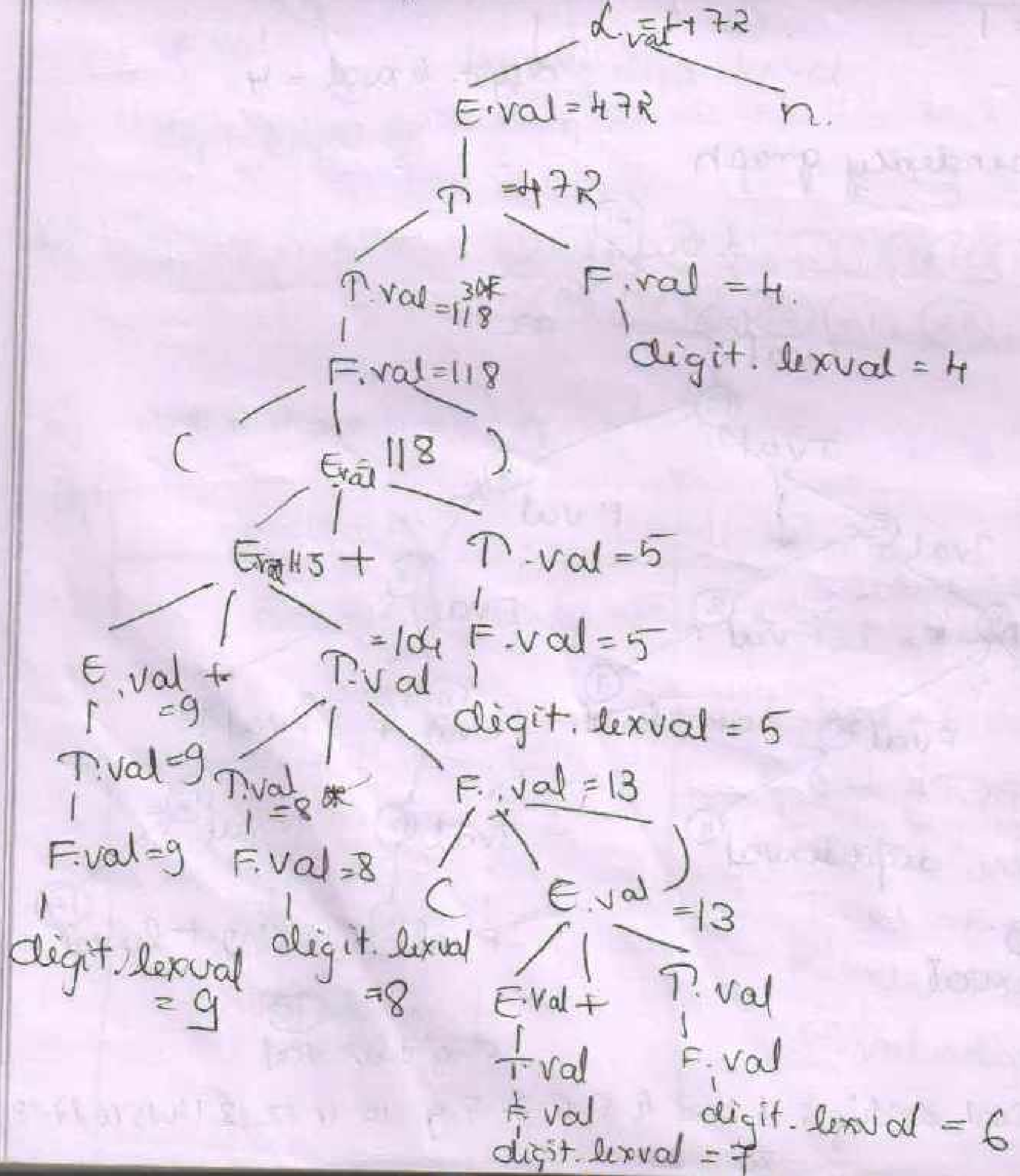
Step4 : Topological sorting : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

c) (9 $\underset{+}{\cancel{*}}$ 8 * (7+6) + 5) * 4n

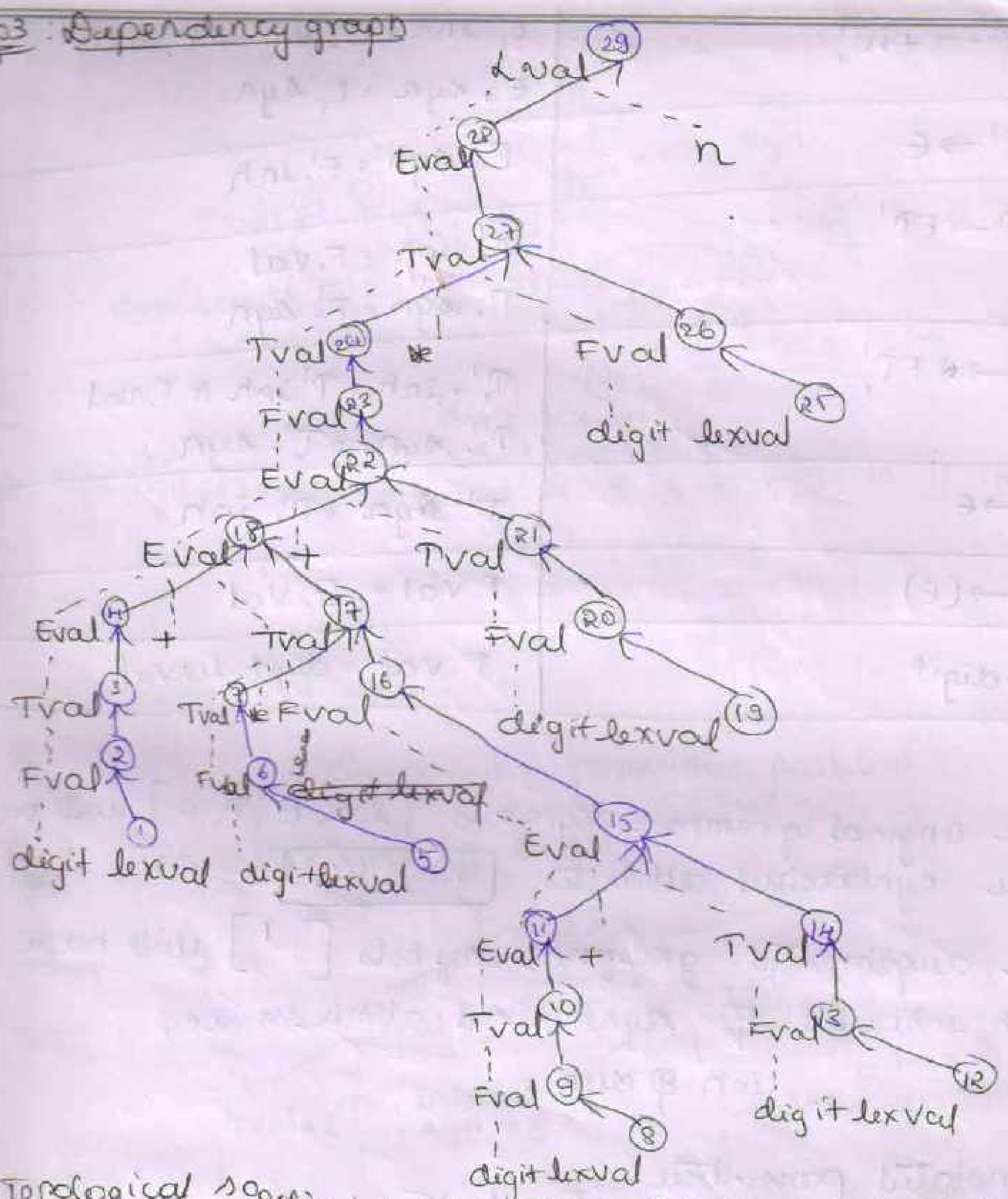
Stp1: SDD defination

Productn	Semantic rules.
L → En	L.val = E.val
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Stp2: Annotated parse tree



Step3 : Dependency graph



Topological sorting : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

II Write L-attributed SDD (or) write a SDD for top down parser & construct annotated parse tree, dependency graph for given i/p.

① 3 * 5

Step1 : SDD Definition

Product^n	Semantic rules
E → TE'	E'.inh = T.val E. ~~syn~~ val = E'.syn

$E' \to +TE'_1$	$E'_1.inh = E'.inh + T.inh$ $E'.syn = E'_1.syn$
$E' \to \epsilon$	$E'.syn = E'.inh$
$T \to FT'$	$T'.inh = F.val$ $T.$ ~~syn~~ val $= T'.syn$
$T' \to * FT'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to (E)$	$F.val = E.val$
$E \to digit$	$F.val = digit.lexval$

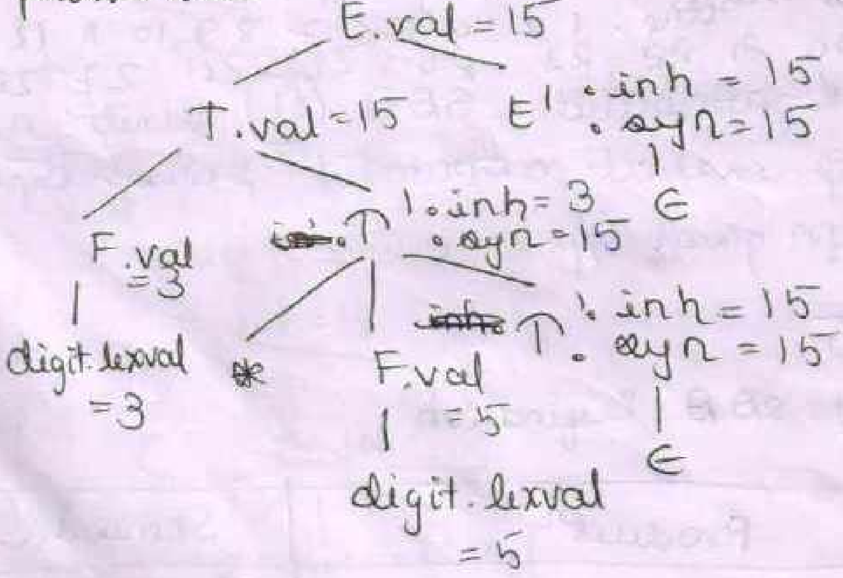
NOTE

① The original grammer symbols [i.e., E, T, F] will only have synthesized attributes $\boxed{\text{i.e., val}}$

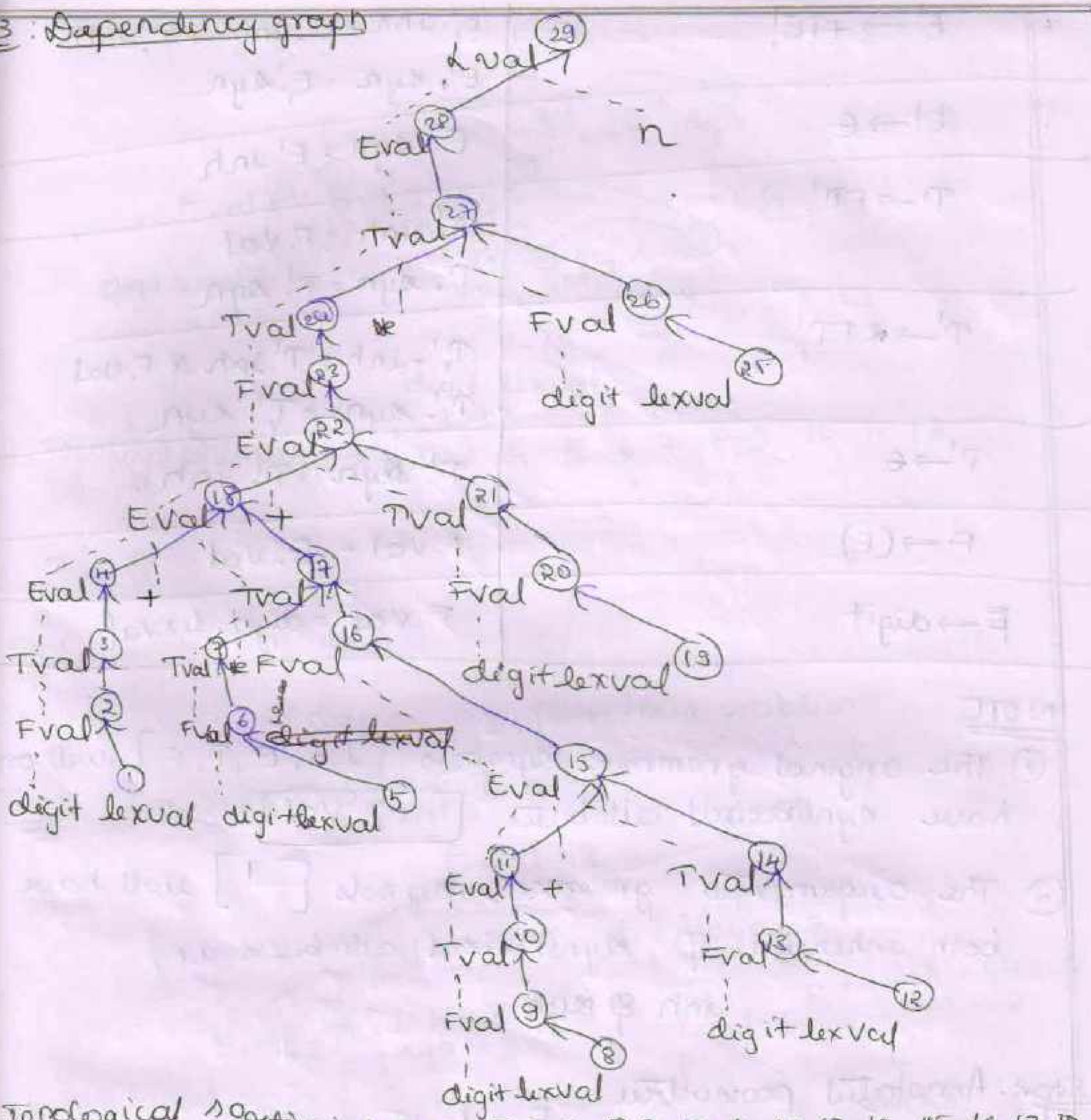
② The augmented grammer sybols ['] will have both inherited & synthesized attributes i.e.,

inh & syn

Step 2: Annotated parse tree



Q3: Dependency graph



Topological sorting: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

Write L-attributed SDD (or) write a SDD for top down parser & construct annotated parse tree, dependency graph for given i/p.

① 3 * 5

Step1: SDD Defination

Production	Semantic rules.
E → TE'	E'.inh = T.val E.syn = E'.syn

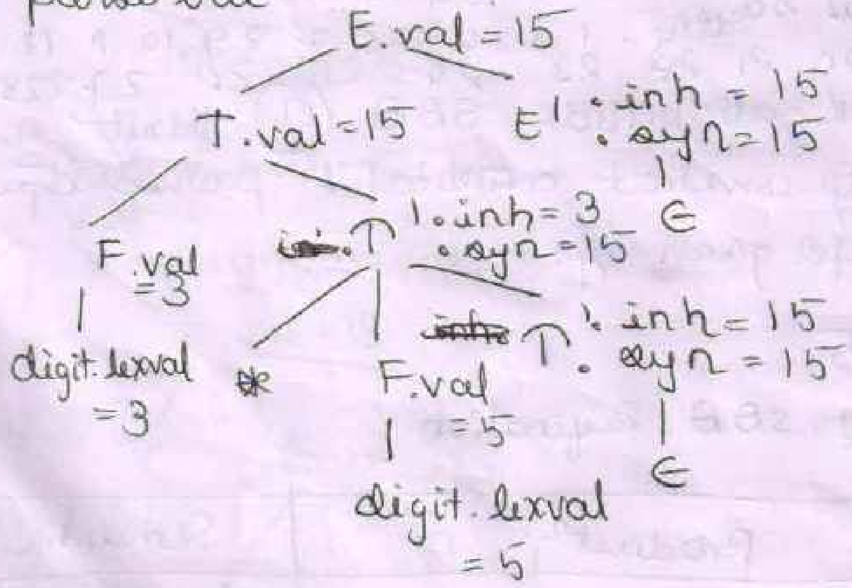
$E' \rightarrow +TE'_1$	$E'_1.inh = E'.inh + T.inh$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow FT'$	$T'.inh = F.val$ $T.syn = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh * F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow (E)$	$F.val = E.val$
$E \rightarrow digit$	$F.val = digit.lexval$

NOTE

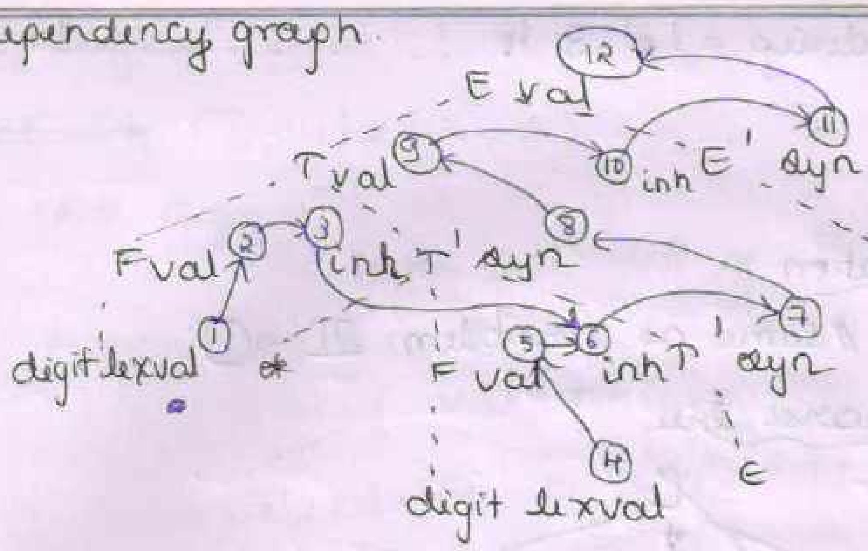
① The original grammer symbols [i.e., E, T, F] will only have synthesized attributes [i.e., val]

② The augumented grammer sybols ['] will have both inherited & synthesised attributes i.e., inh & syn

Step2. Annotated parse tree



Step3: Dependency graph.



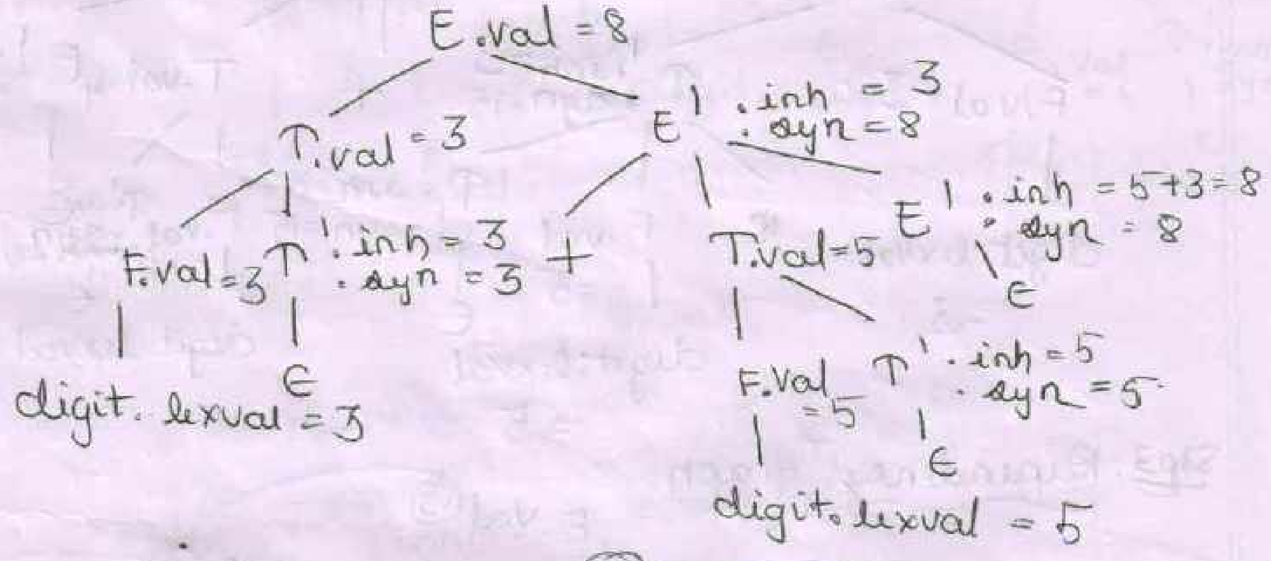
Step4 : Topological Order → 1 2 3 4 5 6 7 8 9 10 11 12.

(2) 3 + 5

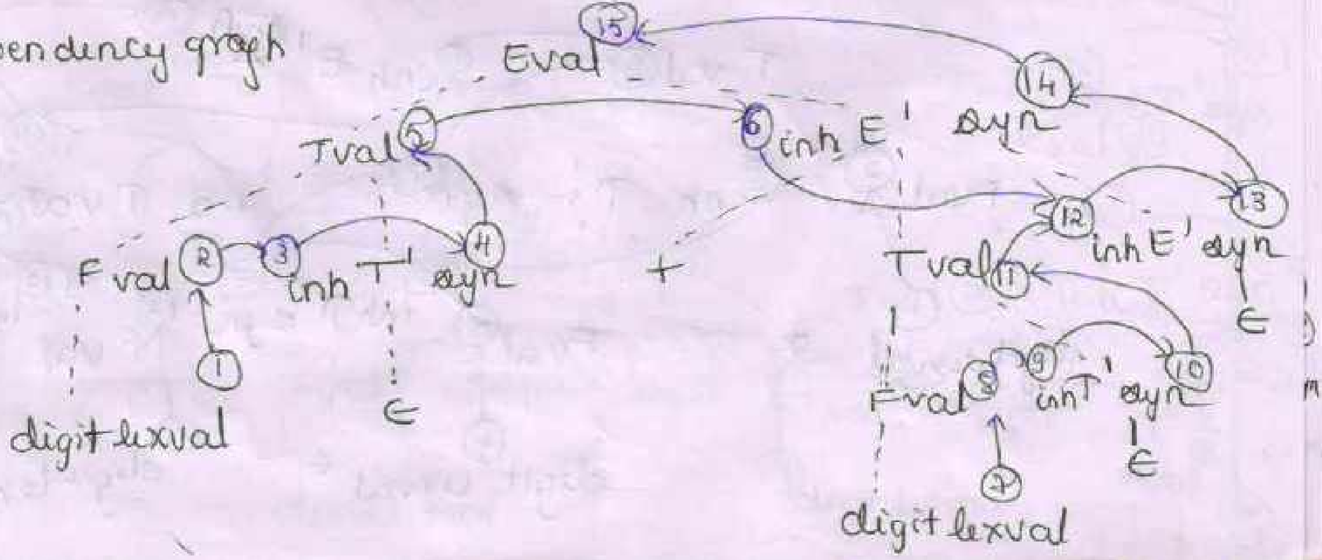
Step1: SDD definatn

// Same as previous problem.

Step2: Annotated parse tree



Step3: Dependency graph



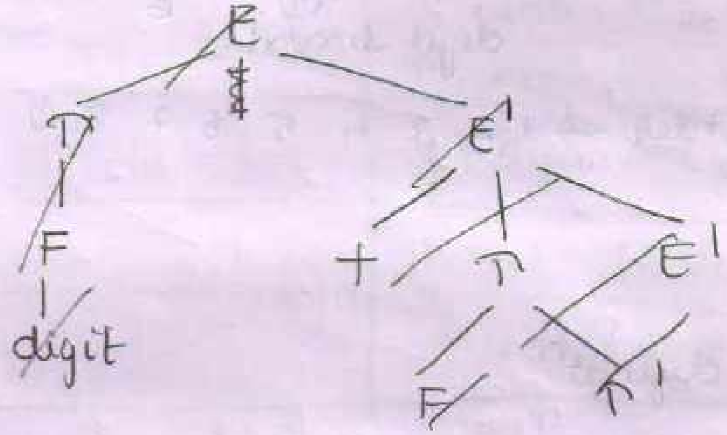
Step4: Topological Ordering = 1 2 3 4 .. .. - - - - - - 14 15

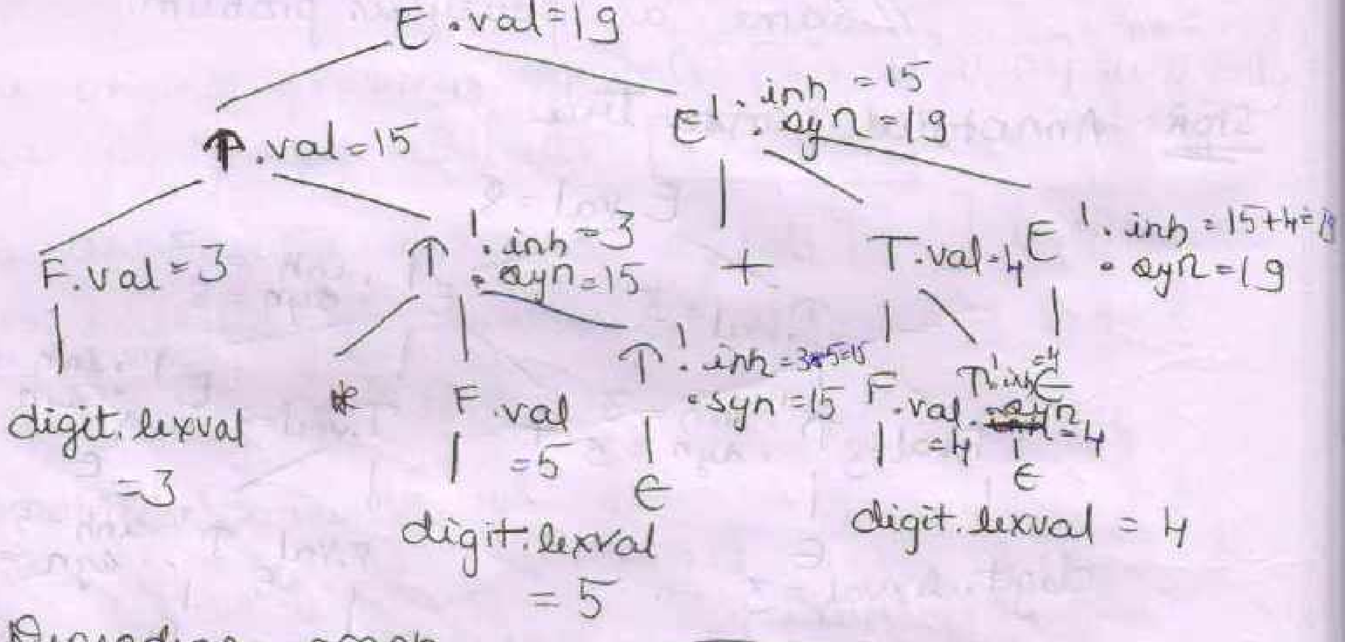
③ 3 * 5 + 4

Step1: SDD defination

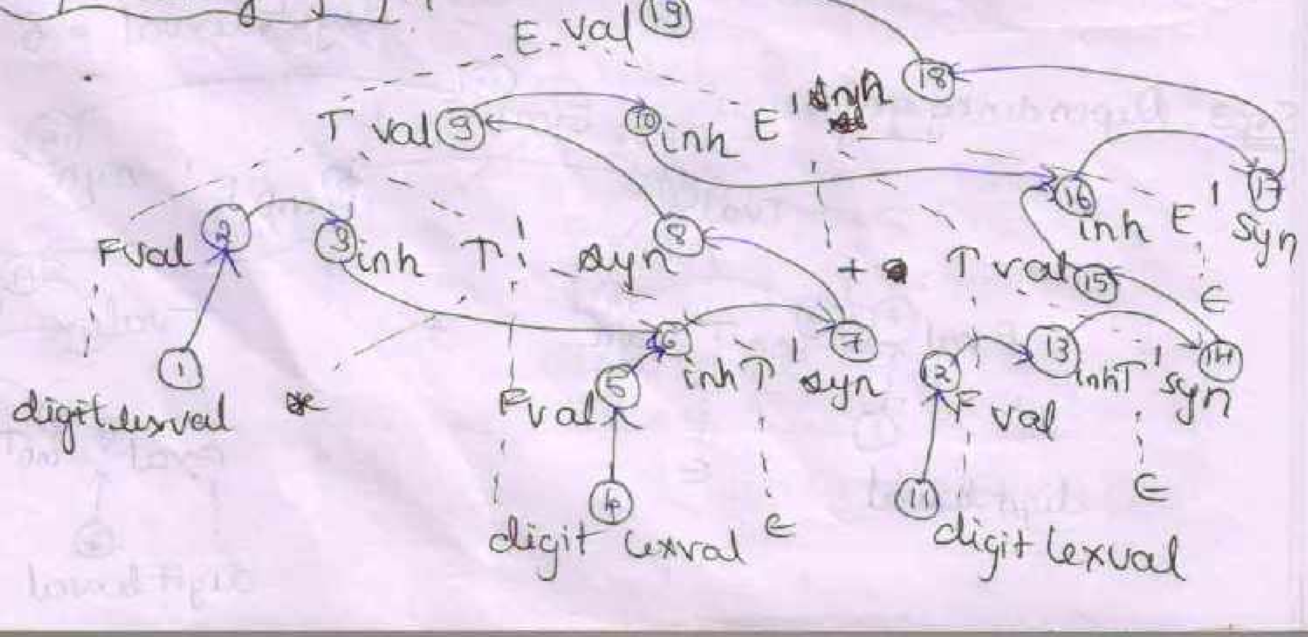
// Same as problem II → ①

Step2: Annotated parse tree





Step3: Dependency graph

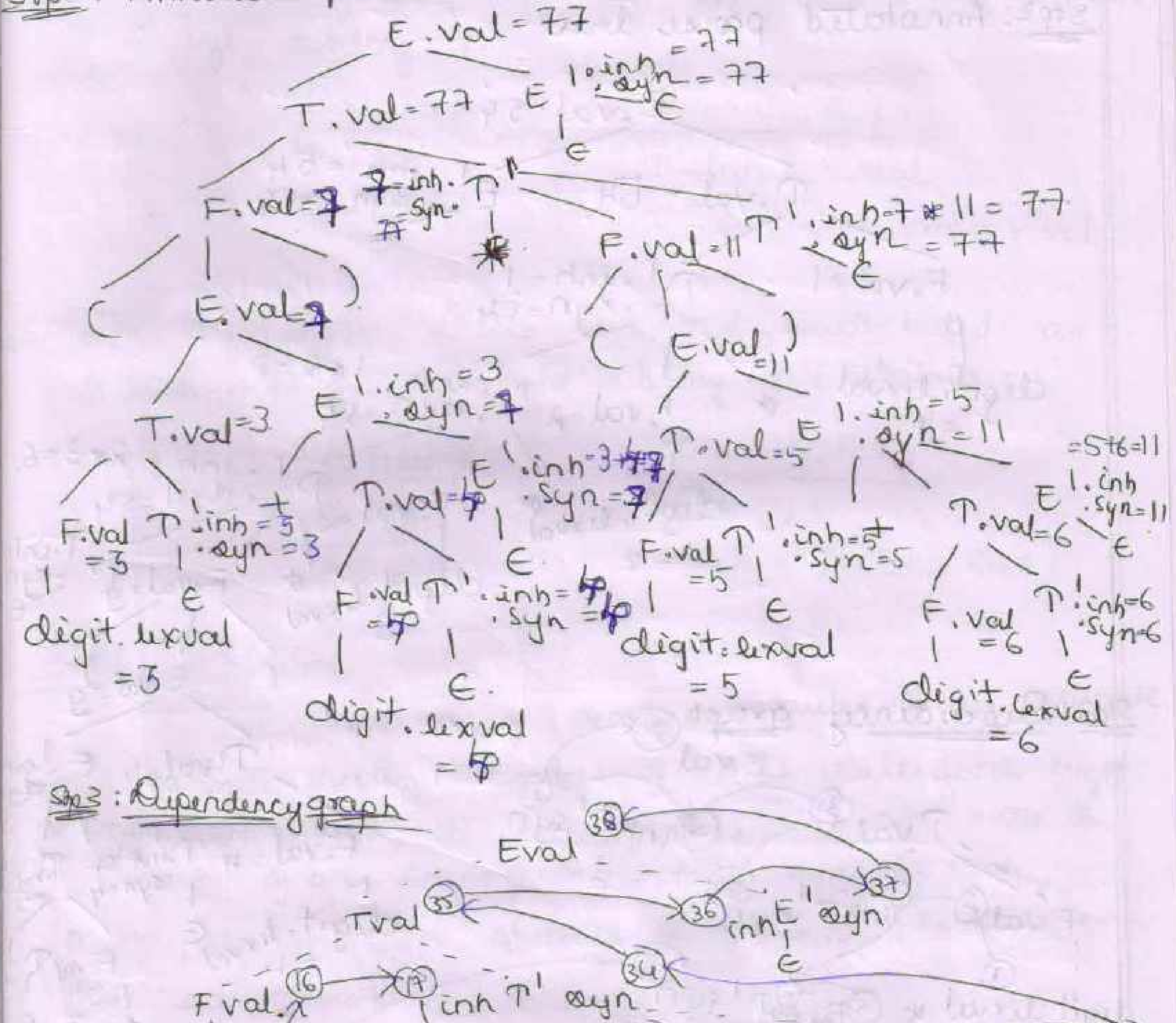


Topological Order: 1 2 3 4 19

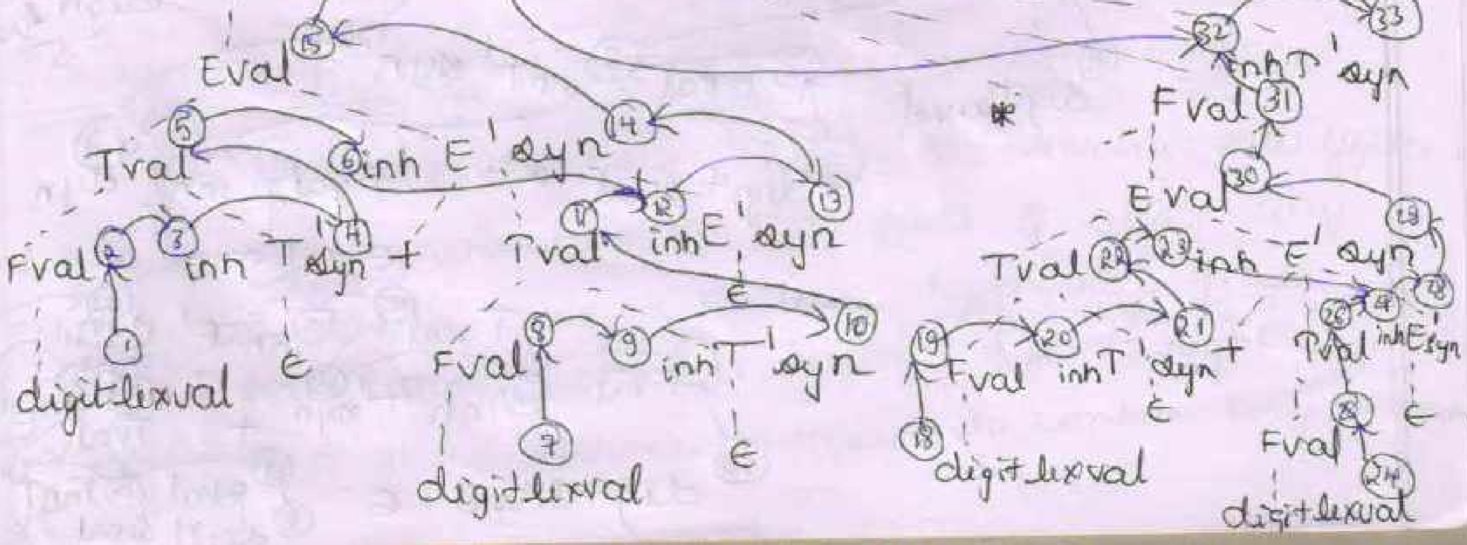
3 ~~* 5 + 4~~. (3+4) * (5+6)

Step1: SDD definatn
// same as SDD of II -> (1)

Step2: Annotated parse tree



Step3: Dependency graph



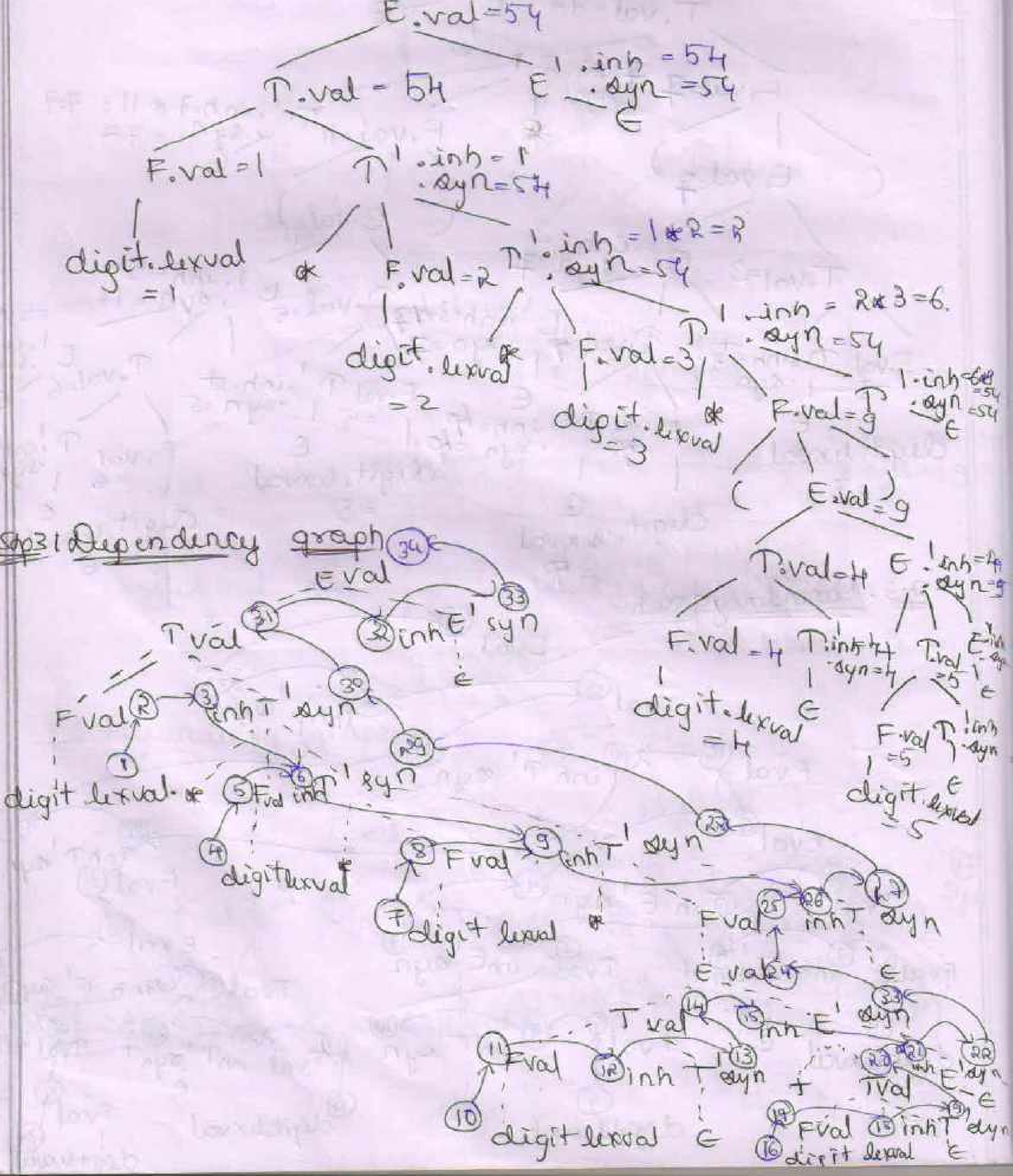
~~4)~~ Topological Ordering. ① ② ③ ④ - - - - - - - - ㊲ ㊳

⑤ 1*2*3*(4+5)

Step1: SDD definatⁿ

// same as II → ①

Step2: Annotated parse Tree



Step3: Dependency graph

appy · Topological sorting : ① ②

Eg1: The following defination is L attributed. Here the inherited attribute of T' gets its value from its left sibling F. Similarly T_1' gets its value from its parent T' & left sibling F

Production	Semantic Rules
$T \to FT'$	$T'.inh = F.val$
$T' \to *FT_1'$	$T_1'.inh = T'.inh * F.val$

Eg2: The definations below are not L-attributed as B.i depends on its right sibling C's attribute.

Production	Semantic Rules
$A \to BC$	$A.s = B.b$ $B.i = f(C.c, A.s)$

SIDE EFFECTS

Evaluation of semantic rules may generate intermediate codes, ~~may~~ Eg: A desk calculator might print a result, a code generator might enter the type of an identifier into a symbol table, may perform type checking & may issue error msgs. These are known as side effects.

$5+4=9$

SEMANTIC RULES WITH CONTROLLED SIDE EFFECTS

In practise translation involves side effects. Attribute grammers has no side effects & allow any evaluation order consistent with dependency graph whereas translation schemes impose left to right evaluation & allow scheme actions to contain any program fragment.

Ways to Control Side Effects

1. permit incidential side effects that donot constrain attribute evaluation.

   In other words, permit side effects when attr evaluat^n based on any topological sort of the dependcy graph produces a correct translation.

2. Impose constraints on allowable evaluation orders, so that the same translation is produced for any allowable order.

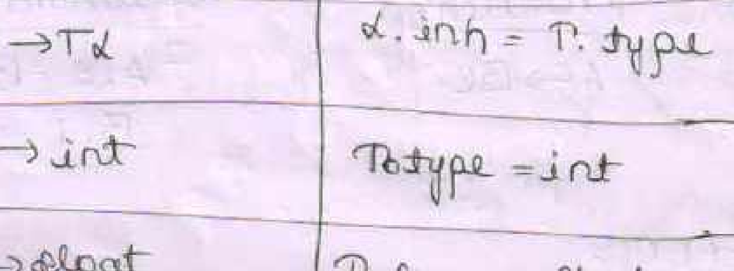
Write an SDD for simple type declaration

a) i/p: $\boxed{\text{int a}}$

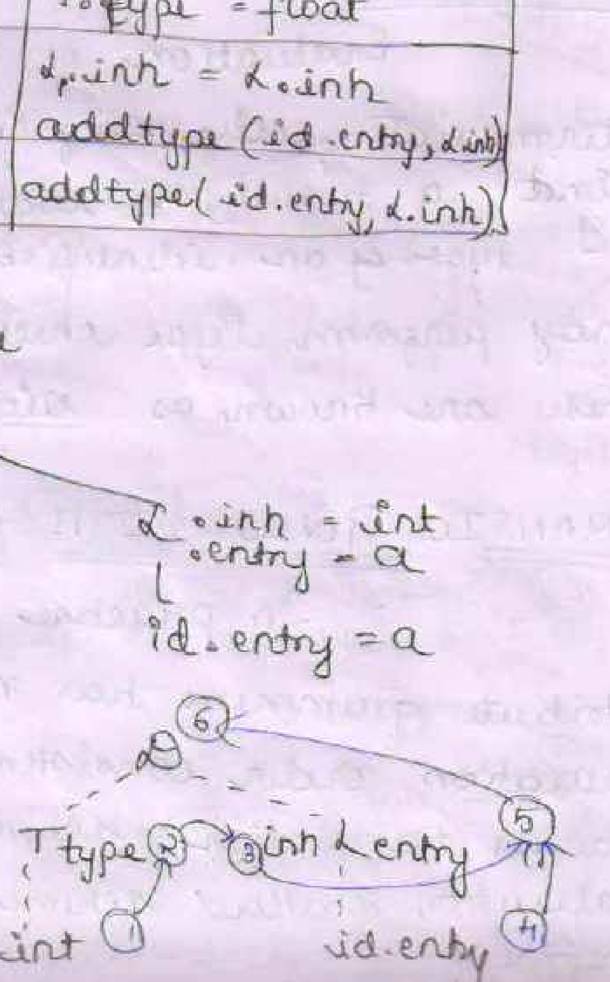
Step1:

Production	Semantic rules.
$D \to TL$	$L.inh = T.type$
$T \to int$	$T.type = int$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$ addtype(id.entry, L.inh)
$L \to id$	addtype(id.entry, L.inh)

Step2: Annotated parse tree



Step3: Dependency graph



explanation:

Non terminal D represents a declaration, which from product^n 1, consists of a type T followed by a list L of identifiers. T has one attribute: T.type, which is the type in the declaration D. Nonterminal L has one attribute, which call inh to emphasize that it is an inherited attribute. The purpose of L.inh is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol table entries. Product^n 2 & 3 each evaluate the synthesized attribute T.type giving it the appropriate value, integer or float. This type is passed to the attribute L.inh in the rule of product^n 1. Product^n 4 passes L.inh down the parse tree i.e., the value of L1.inh is computed at a parse tree node by copying the value of L1.inh from the parent of that node, the parent corresponds to the head of product^n. Product^n 4 & 5 also have a rule in which a funct^n addtype is called with 2 arguments:

1) id.entry a lexical value that points to a symbol table object.

2) L.inh, the type being assigned to every identifier on the list.

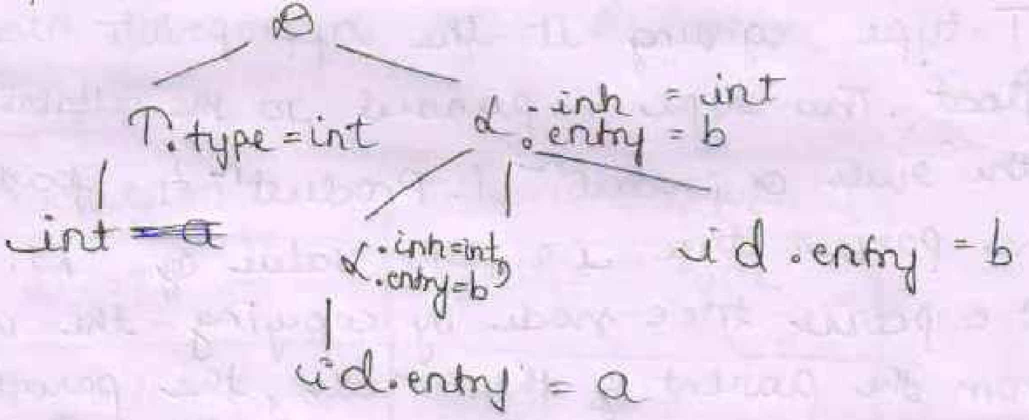
The function addType properly installs the type L.inh as the type of the represented identifier. Note that the side effect, adding the type info to the table, doesnot affect the evaluation order.

(b) int a, b

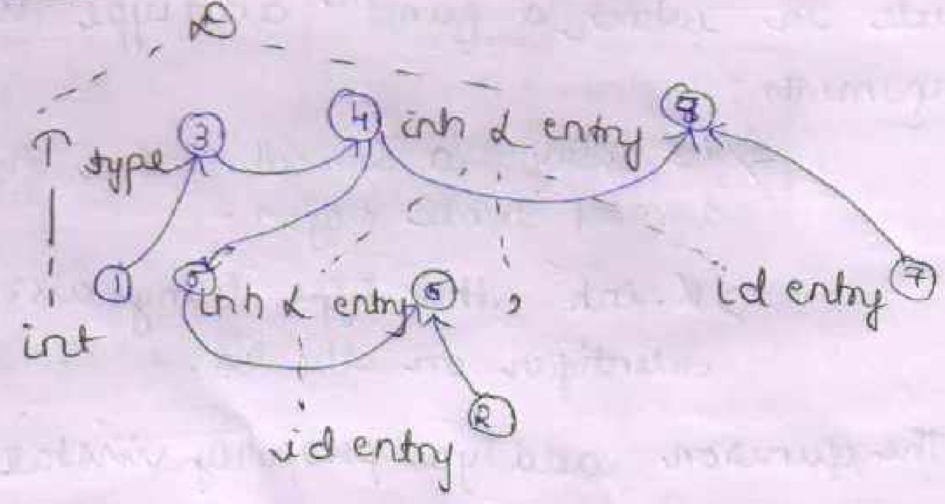
Step1

Product^n	Semantic rules.
$D \rightarrow TL$	$L.inh = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L, id$	addtype(id.entry, L.in) $L_1.inh = L.inh$
$L \rightarrow id$	addtype(id.entry, L.inh)

Step2: Annotated parse tree



Step3: Dependency graph



Step4: Topological order ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨

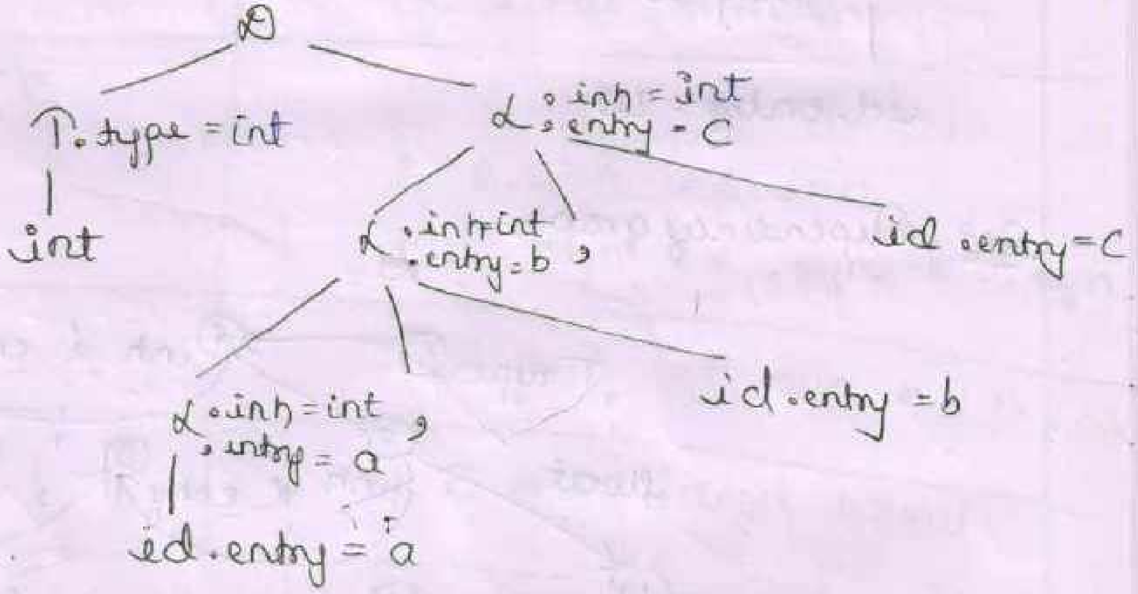
UNIT-5 [continued . . . . .]

ⓒ float a, b, c. or int a, b, c <Exercise 5.2.2>

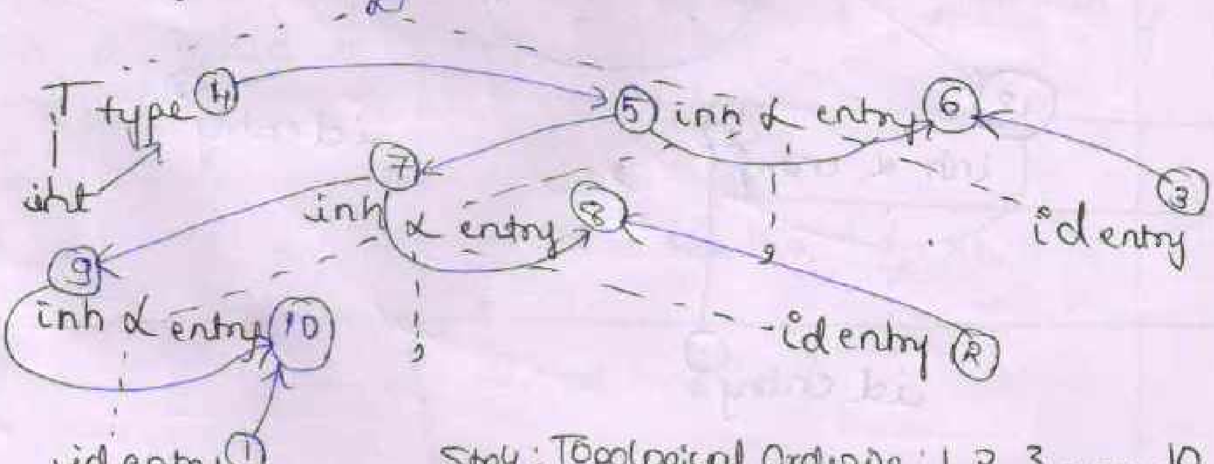
Stp1 : SDD defination

Production	SDD
$D \rightarrow TL$	$L.inh = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow float$	$T.type = float$
$L \rightarrow L, id$	$L_1.inh = L.inh$ addtype (id.entry , L.inh)
$L \rightarrow id$	addtype (id.entry , L.inh)

Stp2 : Annotated parse tree



Stp3 : Dependency graph



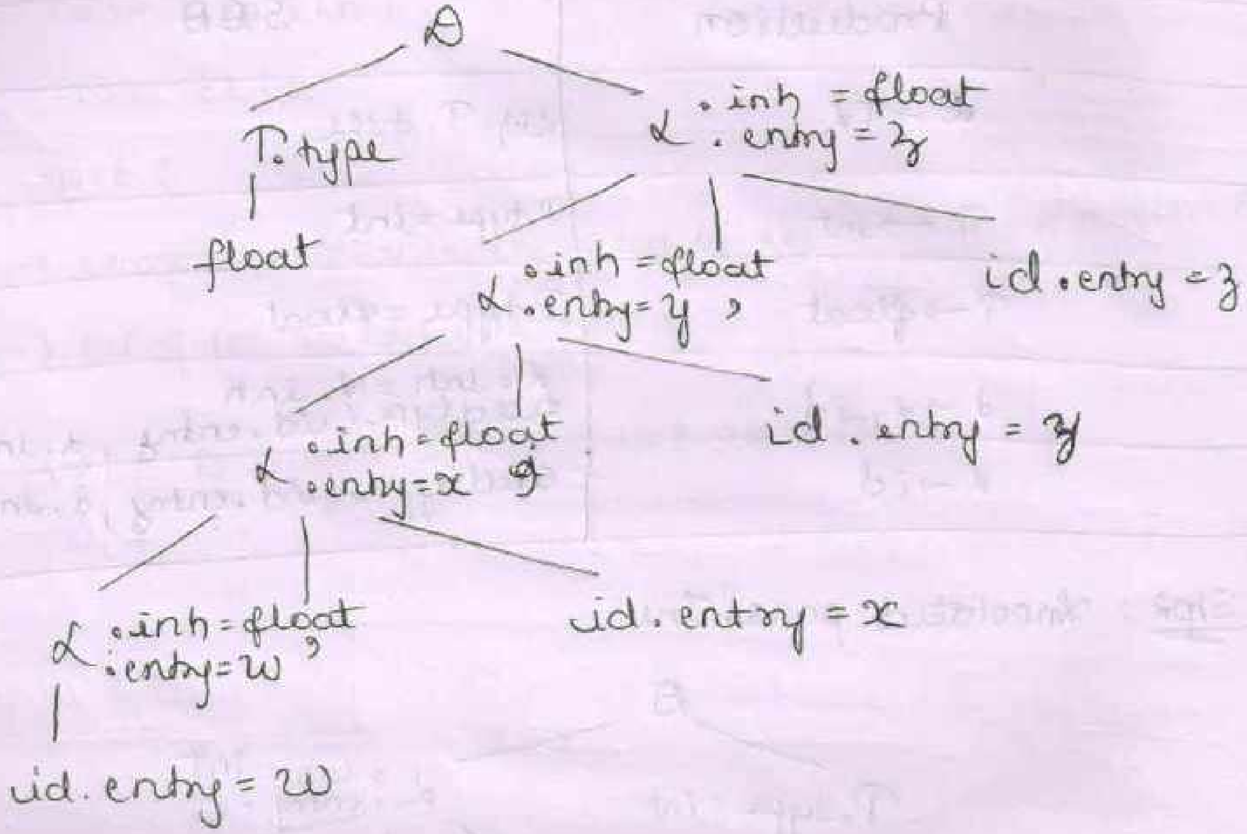
Stp4 : Topological Ordering : 1 2 3 10

(d) float w, x, y, z

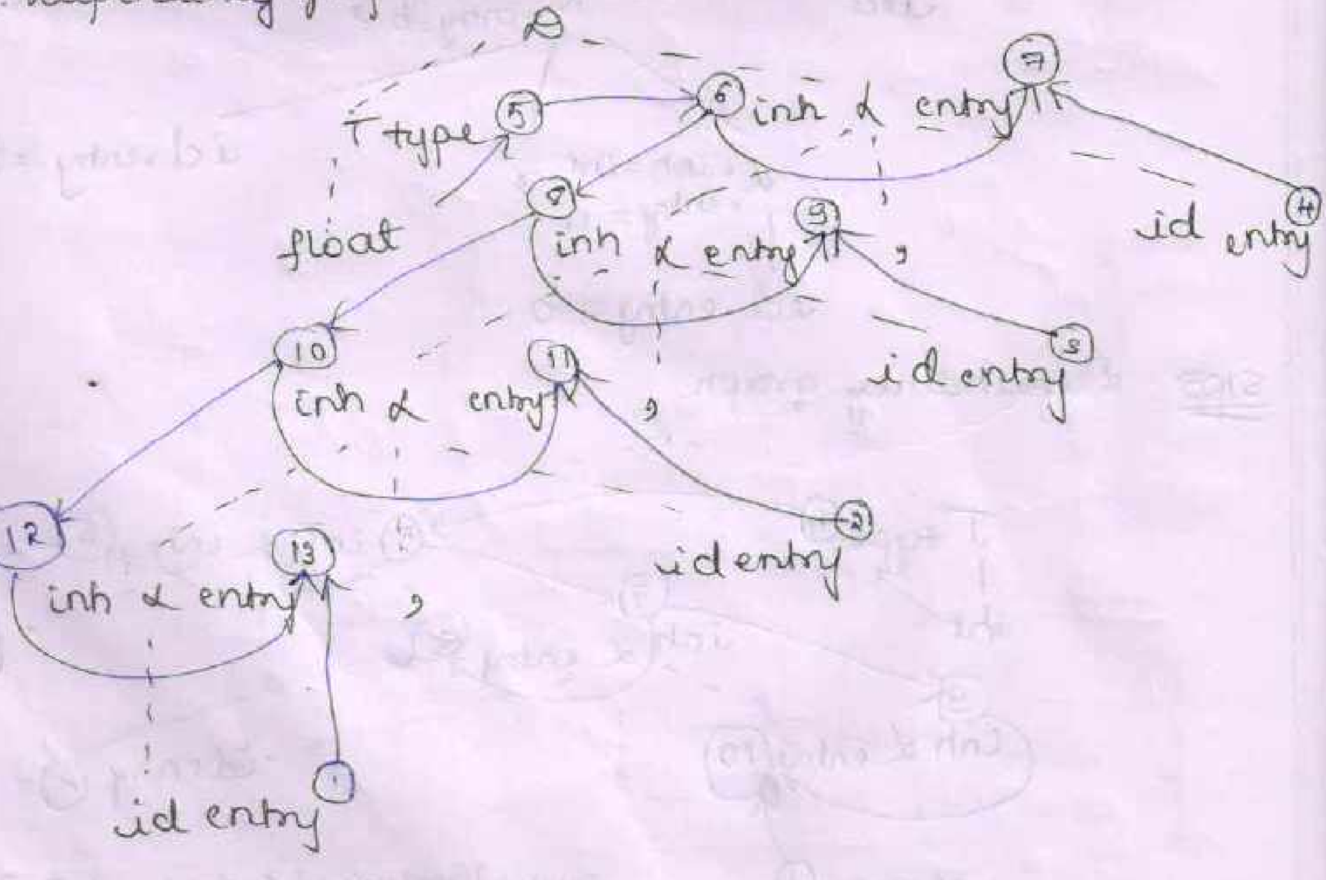
Step1 SDD definat^n

// Same as previous problem.

Step2 : Annotated parse tree



Step3 : Dependency graph



Exercise 5.2.4 This grammer generates binary no with a decimal point

$S \rightarrow L . L \mid L$

$L \rightarrow L B \mid B$

$B \rightarrow 0 \mid 0$

Design an L attributed SDD to compute S.val, the decimal no value of i/p string. For eg, translat^n g string 101.101 should be the decimal no 5.625.

Soln

Product^n	SDD
1) $S \rightarrow L . L_1$	1. L.inh = 0 2. L_1.inh = -1 3. S.val = L.syn + L_1.syn
2) $S \rightarrow L$	1. L.inh = 0 2. S.val = L.syn
3) $L \rightarrow L_1 B$	1. L_1.inh = L.inh + 1 2. B.inh = L.inh 3. L.syn = L_1.syn * B.syn
4) $L \rightarrow B$	1. L.syn = B.syn * 2 ^ L.inh
5) $B \rightarrow 0 \mid 1$	1. B.syn = digit.lexval

Exercise 5.2.5 Design an S-attributed SDD for grammer of translation described in 5.2.4

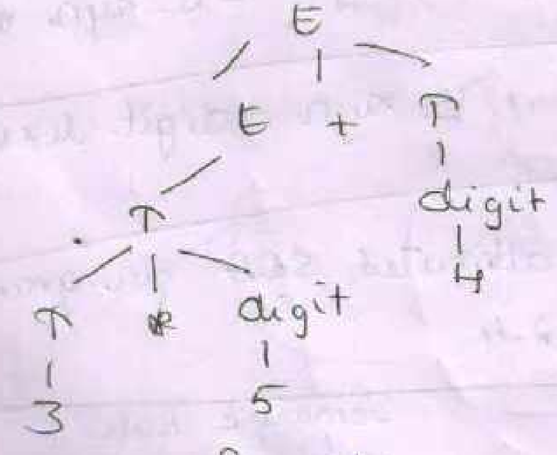
Prodution	Semantic Rule.
1) $S \rightarrow L . L_1$	S.val = L.lhs + L_1.rhs
2) $S \rightarrow L$	S.val = L.lhs

3) $\alpha \rightarrow \alpha_{1} B$	1) α.lhs = α_1.lhs + (2 ^ α.lhs_exponent * B.val) 2) α.rhs = α_1.rhs + (2 ^ α.rhs_exponent * B.val) 3) α.lhs_exponent = α_1.lhs_exponent + 1 4) α.rhs_exponent = α_1.rhs_exponent - 1
4) $\alpha \rightarrow B$	1) α.lhs = 2 ^ α.lhs exponent * B.val 2) α.rhs = 2 ^ α.rhs exponent * B.val 3) α.lhs_exponent = 0 4) α.rhs_exponent = -1
5) B → 0 | 1	B.val = digit.lexval

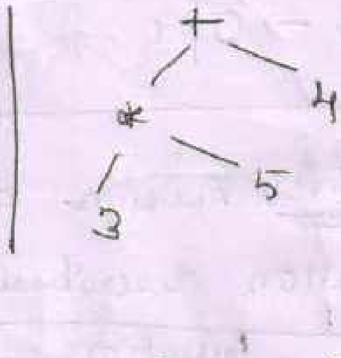
Application Of Syntax - Directed Translation

1. Construction Of Syntax Tree

SDD's are useful for construct^n of syntax tree. A syntax tree is condensed form of parse tree



Parsetree



Syntax tree

A Syntax trees are useful for representing programing language constructs like expressions & statements.

- They help Compiler design by decoupling parsing from translatn.

- Each node of a syntax tree represent a construct; the children of the node represent the meaningful components of a construct

  Eg: A syntax tree node representing an expression E1 + E2 has label + & 2 children representing the sub expression E1 & E2

- Each node is implemented by object with suitable no of fields; each object will have an opfield that is the label of node with additional fields as follow:

  a) If the node is a leaf, an addition field holds the lexical value for the leaf. This is created by function Leaf (op, val).

  b) If the node is an interior node, there are as many fields as the node has children in syntax tree. This is created by function Node(op, c1, c2, ---ck)

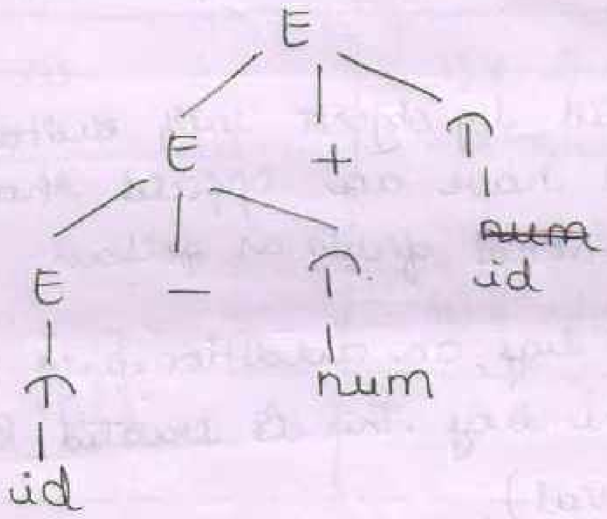
Example: The S-attributed definatn in fig below constructs syntax trees for a simple expn grammer involving only binary operators + & -. As usual these operators are at the same precedence level & are jointly left associative. All nonterminals have one synthesized attr node, which represents a node of the syntax tree.

a-4+c

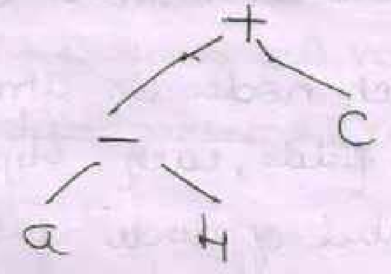
	PRODUCTION	SEMANTIC RULES
1.	$E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
2.	$E \rightarrow E_1 - T$	E.node = new Node ('-', E_1.node, T.node)
3.	$E \rightarrow T$	E.node = T.node

T → (E)	T.node = E.node
T → id	T.node = newLeaf(id, id.entry)
T → num	T.node = newLeaf(num, num.val)

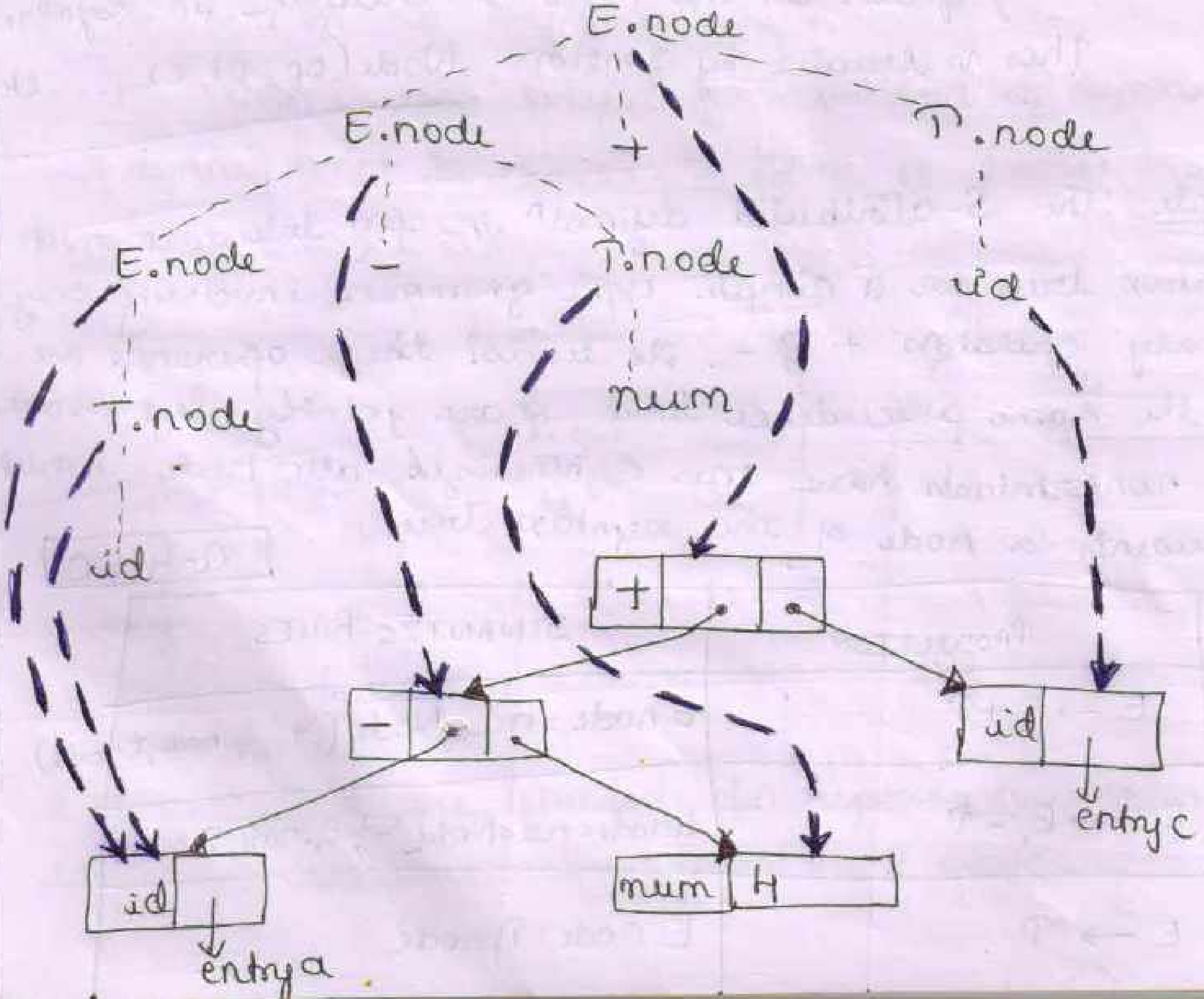
Step2 : Parse tree



Syntax tree



Step3: Syntax tree for a-4+C using above SDD. :



Steps in Construction Of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reduct^n during a bottom up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

1) P_1 = new Leaf (id, entry-a)

2) P_2 = new Leaf (num, 4)

3) P_3 = newNode ('-', P_1, P_2)

4) P_4 = new Leaf (id, entry-c)

5) P_5 = new Node ('+', P_3, P_4)

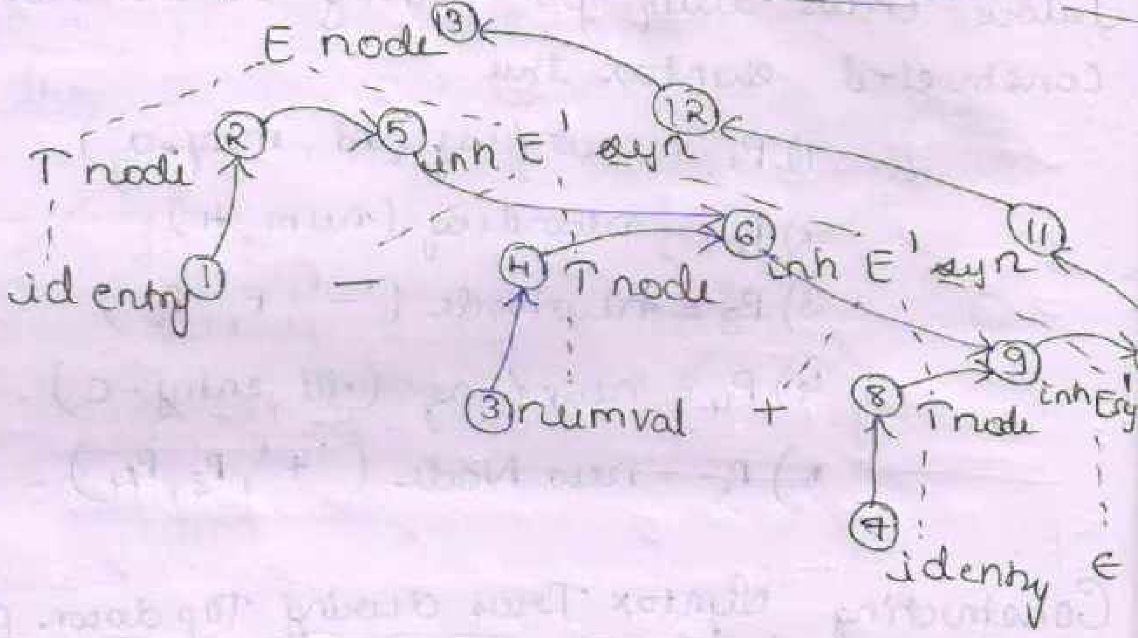
Constructing Syntax Trees during Top down parsing

With a grammer designed for top-down parsing, the syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees. The L-attributed defination below performs the same translat^n as the S attributed defination shown before.

1) $E \to TE'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \to +TE_1'$	$E_1'.inh =$ ~~$E'.inh + P_1$~~ new Node('+', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
3) $E' \to -TE_1'$	$E'.inh$ = new Node('-', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
4) $E' \to \epsilon$	$E'.syn = E'.inh$
5) $T \to (E)$	$T.node = E.node$

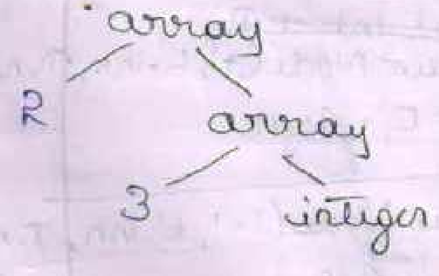
6) $T \to id$	T.node = new Leaf (id, id.entry)
7) $T \to num$	T.node = new Leaf (num, num.val)

Dependency Graph for a-4+c with L-attributed SDD



STRUCTURE OF A TYPE

This is an example of how inherited attribute can be used to carry info from one part of the parse tree to another. In C the type int[2][3] can be read as "array of 2 arrays of 3 integer". The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below



productn	Semantic rules

1) $T \to BC$	T.t = C.t
	C.b = B.t

2) $B \to int$	B.t = integer
3) $B \to float$	B.t = float
4) $C \to [num] C_1$	C.t = array (num.val, C C_1.b = C.b
5) $C \to \epsilon$	C.t = C.b

→ The non terminals B & T have a synthesized attribute t representing a type

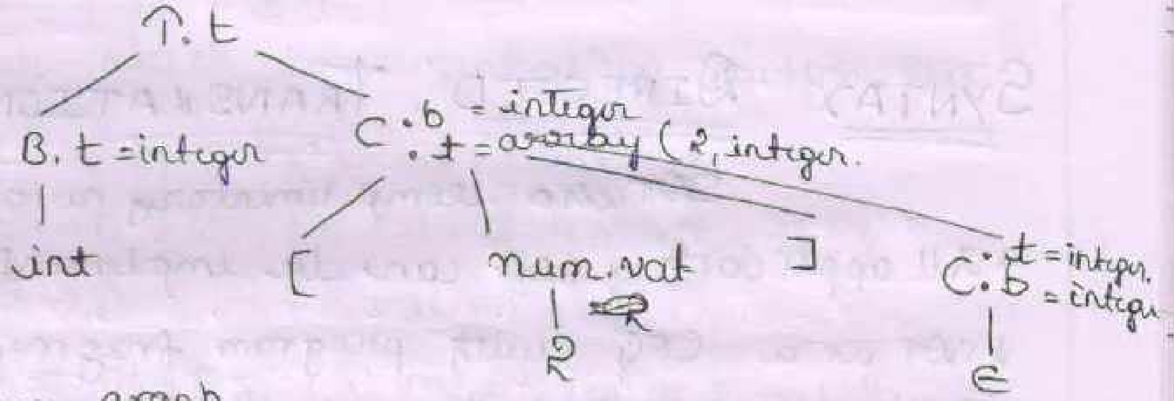
→ The non terminal C has 2 attributes: an inheritted attr (b) & a synthesized attr (t).

→ The inheritted attribute, b pass a basic type down the tree

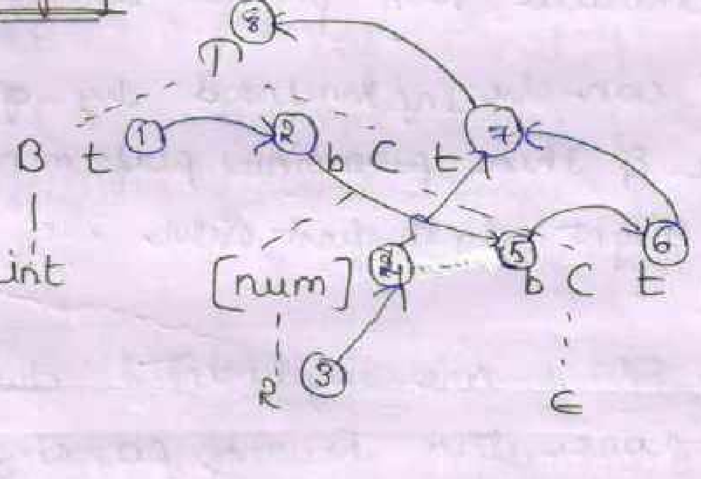
→ They synthesized attribute, t accumulate the result.

→ An anotated parse tree for i/p : int [2]

(i) int [2]

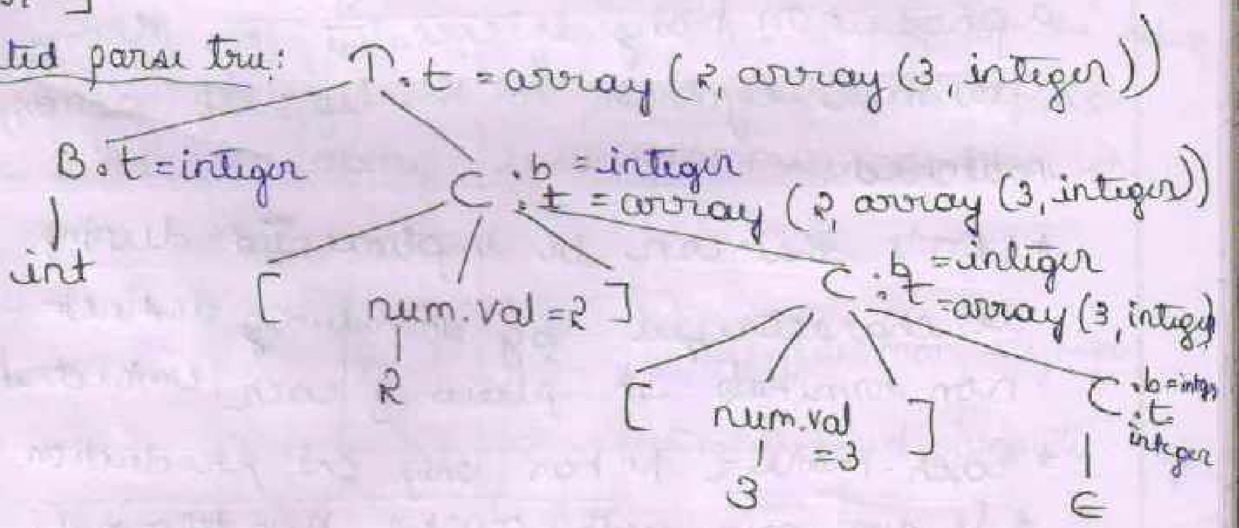


<u>Dependency graph</u>

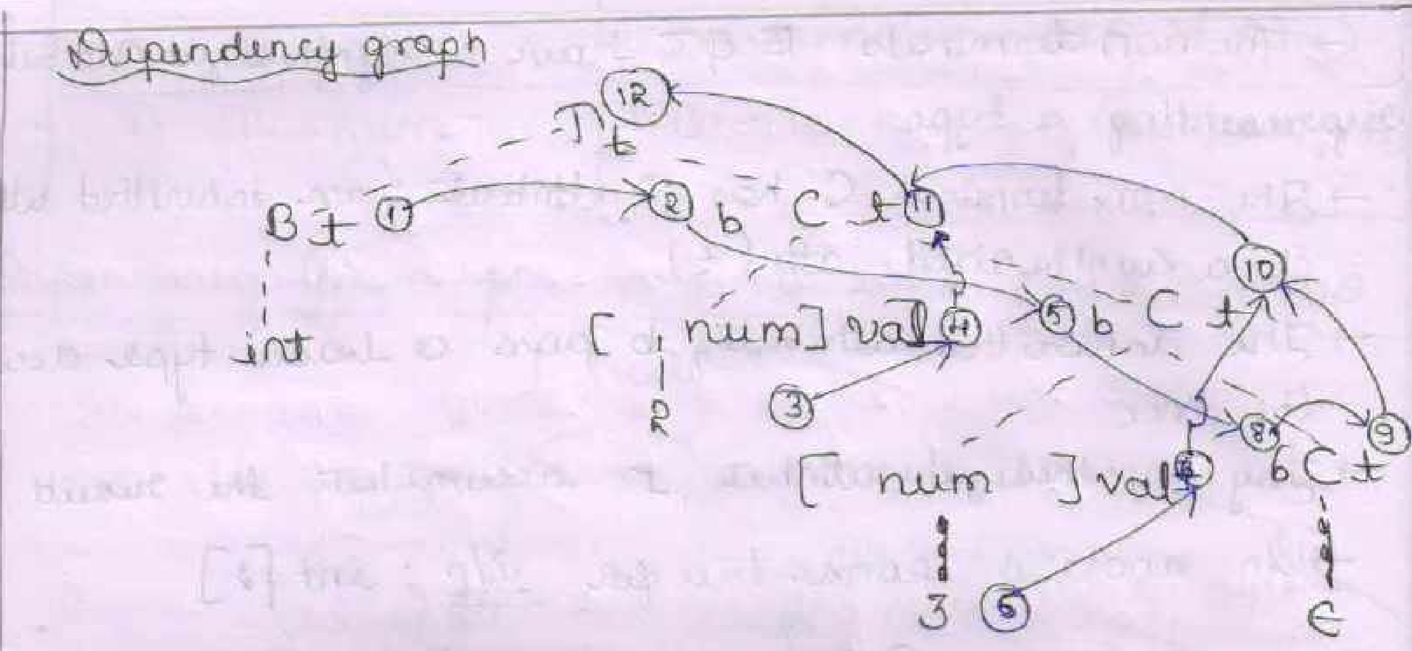


(ii) int [2][3]

Annotated parse tree:



Dependency graph



SYNTAX DIRECTED TRANSLATION SCHEME:

SDT is a complementary notation to SDD.

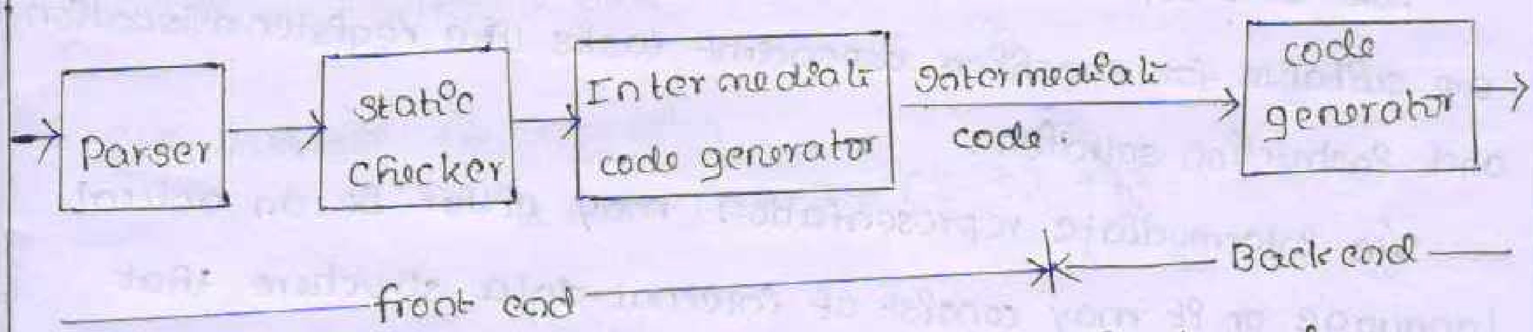
- All applicat^n of SDD can be implemented using SDT.
- SDT is a CFG with program fragments called seman actions embedded with product^n bodies.
- Any SDT can be implemented by first building a parse tree & then ~~[illegible]~~ performing the actions in a left to right, depth first order i-e, during preorder traversal.
- Typically SDT's are implemented during parsing without building parse tree. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of action have been matched.
- SDT's that can be implemented during parsing can be characterized by introducing distinct marker non terminals in place of each embedded action.
- Each marker M has only one production M → ε.
- If grammar with marker non terminals can be parsed by a given method, then SDT can be implemented

Unit-6

Intermediate Code Generation

Intermediate Code generation.

In the analyses-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.



Logical structure of a compiler front end

Parsing, static checking and intermediate code generation are done sequentially; sometimes they can be combined and folded into parsing.

Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing.

Ex: It ~~ensures~~ assures that a break-statement in C is enclosed within a while-, for- or switch-statement; an error is reported if such an enclosing statement does not exist.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representation as

Source program → High level Intermediate Representation → ... → Low level Intermediate Representation → Target code

High level representations are close to the source language and are well suited to tasks like static type checking.

Ex: Syntax tree

Low level representations are close to the target machine & are suitable for machine dependent tasks like register allocation and instruction selection.

An intermediate representation may either be an actual language or it may consist of internal data structure that are shared by phases of the compiler.

Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.

A directed acyclic graph (DAG) for an expression identifies the common subexpressions of the expression.

Directed Acyclic Graphs for Expressions.

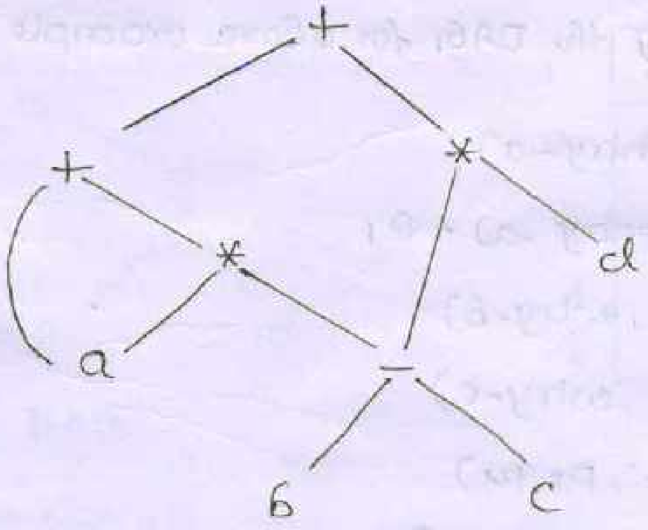
In DAG leaves represents the atomic operands and interior nodes represents the operators. as in the syntax tree.

A node N in a DAG has more than one parent if N represents a common subexpression; But in the syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.

DAG gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Ex: DAG for the expression

$$a + a * (b - c) + (b - c) * d.$$



↳ The leaf for 'a' has 2 parents, because 'a' appears twice in the expression

↳ The 2 occurance of the common subexpression b-c are represented by one node, the node labeled '-'

SDD to produce DAG.

PRODUCTION	SEMANTIC RULES
(i) $E \rightarrow E_1 + T$	E.node = new Node ('+', E_1.node, T.node)
(ii) $E \rightarrow E_1 - T$	E.node = new Node ('–', E_1.node, T.node)
(iii) $E \rightarrow T$	E.node = T.node
(iv) $T \rightarrow (E)$	T.node = E.node
(v) $T \rightarrow id$	T.node = new Leaf (id, id.entry)
(vi) $T \rightarrow num$	T.node = new Leaf (num, num.val)

It will construct a DAG, before creating a new node, these functions first check whether an identical node already exists.

If a previously created identical ~~ex~~ node exists, the existing node is returned.

Steps for constructing the DAG for above example.

(i) P_1 = Leaf (id, entry-a)

(ii) P_2 = Leaf (id, entry-a) = P_1

(iii) P_3 = Leaf (id, entry-b)

(iv) P_4 = Leaf (id, entry-c)

(v) P_5 = Node ('–', P_3, P_4)

(vi) P_6 = Node ('x', P_1, P_5)

(vii) P_7 = Node ('+', P_1, P_6)

(viii) P_8 = Leaf (id, entry-b) = P_3

(ix) P_9 = Leaf (id, entry-c) = P_4

(x) P_{10} = Node ('–', P_3, P_4) = P_5

(xi) P_{11} = Leaf (id, entry-d)

(xii) P_{12} = Node ('*', P_5, P_{11})

(xiii) P_{13} = Node ('+', P_7, P_{12})

When the call to Leaf (id, entry-a) is repeated at step 2, the node created by the previous call is returned, so $P_2 = P_1$.

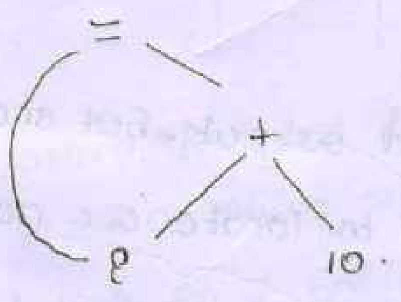
The Value-Number Method for Constructing DAG's

* The nodes of a DAG are stored in an array of records

* Each row of array represents one record & therefore one node.

* In each record, the first field is an operation code, indicating the label of the node. Leaves have one additional field which holds the lexical value and interior nodes have 2 additional fields indicating the left and right children.

Ex: DAG for i=i+10 allocated in an Array



DAG

1	id		
2	num	10	
3	+	1	2
4	=	1	3
5			

to entry for i

Array

* In this array, we refer to nodes by giving the integer index of the record for that node within the array.

* This integer historically has been called the "value number" for the node or for the expression represented by the node

* For above example node labeled + has value number 3 & its left & right children have value numbers 1 & 2 respectively.

Suppose that nodes are stored in an array & each node is referred to by its value numbers. Let the signature of an interior node be the triple <op, l, r> where op is the label, l is its left child's value number & r its right child's value number. A unary operator may be assumed to have r=0.

ALGORITHM: To construct the nodes of a DAG using value number method.

INPUT: Label op, node l and node r

OUTPUT: The value number of a node in the array with signature <op, l, r>

METHOD: Search the array for a node M with label op, left child l and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l and right child r & return its value number.

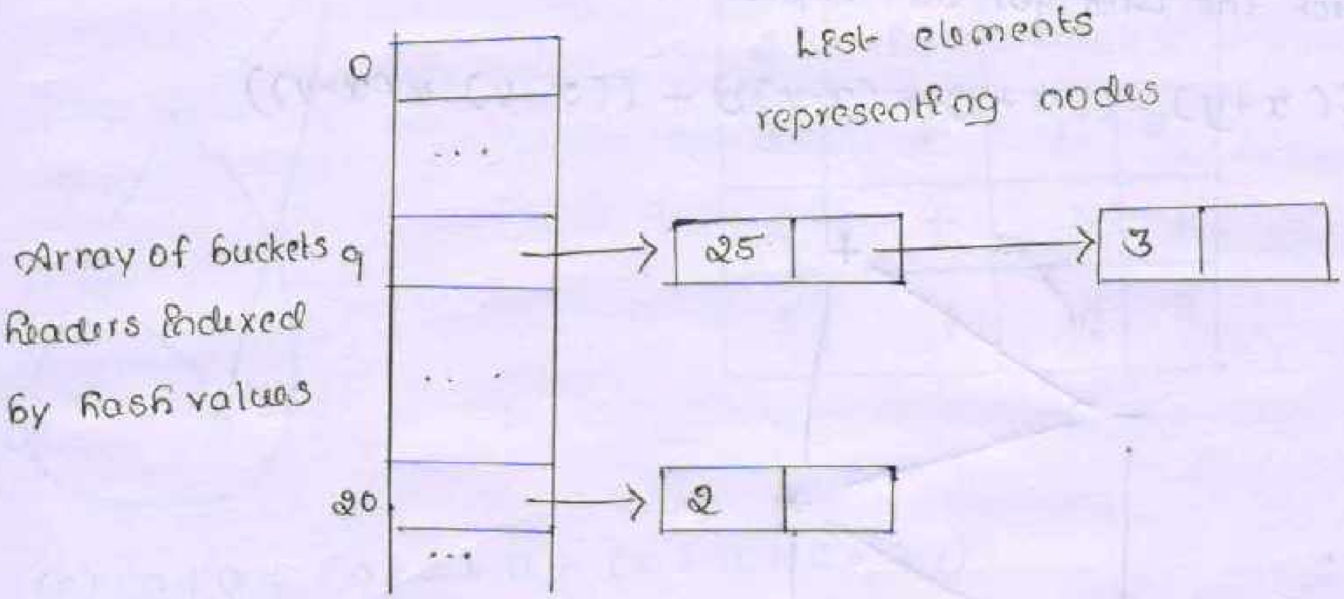
Above algorithm yields the desired output, but searching the entire array every time we are asked to locate one node is expensive.

A more efficient approach is to use a hash table, in which the nodes are put into "buckets" each of which typically will have only a few nodes. It supports dictionaries. which is an abstract data type that allows us to insert & delete elements of a set & to determine whether a given elemen is currently in the set.

To construct a hash table for the nodes of a DAG, we need a hash function h that computes the index of the bucket for a signature <op, l, r>.

The bucket index h(op, l, r) is computed deterministically from op, l & r so that we may repeat the calculation & always get to the same bucket index for node <op, l, r>

The buckets can be implemented as linked list as,


0
...
9
...
20
...
List elements representing nodes
Array of buckets headers indexed by hash values
25
3
2

An array indexed by hash value, holds the bucket headers, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node <op, l, r> can be found on the list whose header is at index h(op, l, r) of the array.

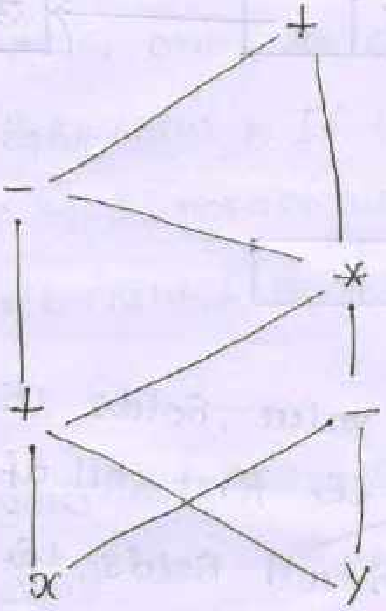
Thus, given the ~~output~~ input node op, l & r we compute the bucket index h(op, l, r) & search the list of cells in this bucket for the given input node.

For each value number 'v' found in a cell, we must check whether the signature <op,l,r> of the input node matches the node with value number v in the list of the cells If we find a match, we return v. If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index h(op,l,r) & return the value number in that new cell.

Problems.

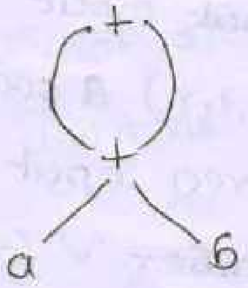
Construct the DAG for the expression.

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$



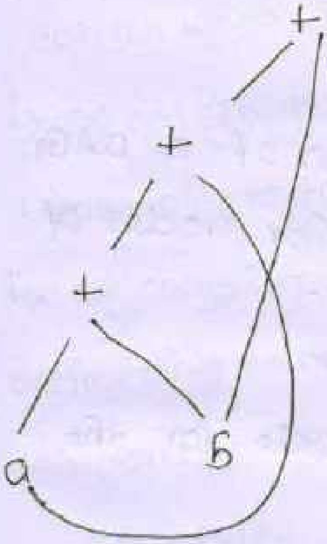
Construct the DAG & identify the value number for the sub expressions of the following expressions, assuming + associa from the left.

(a) $a+b+(a+b)$



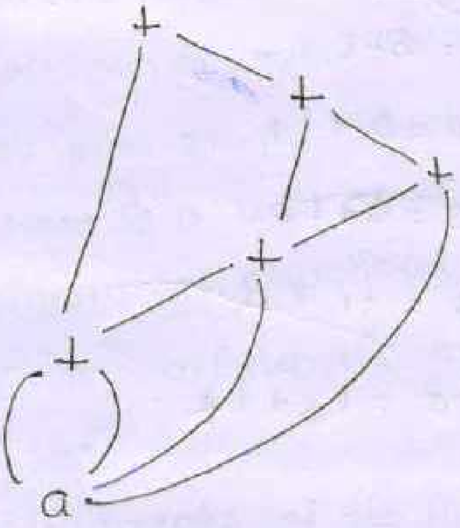
1	ld	a	
2	ld	b	
3	+	1	2
4	+	3	3

(b) a+ b+ a+ b



1	ld	a	
2	ld	b	
3	+	1	2
4	+	3	1
5	+	4	2

(c) a+a + (a+a+ a+ (a+ a + a+a))



1	ld	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

Three-Address Code

In 3-address code, there is atmost one operator on the right R side of an instruction, i.e, no built up arithmetic expression are permitted.

Thus a source-language expression like $x+y*z$ might be translated into the sequence of 3 address instructions

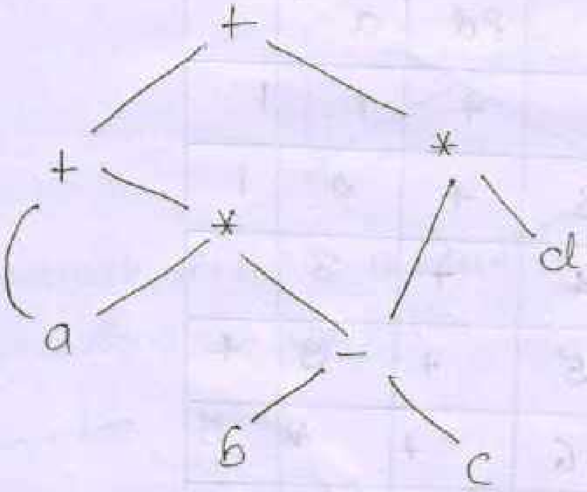
$t_1 = y*z$

$t_2 = x+t_1$

where t_1 & t_2 are compiler generated names.

3 address code is a linearized representation of a DAG in which explicit names correspond to the interior nodes of the graph.

Ex: Write DAG & its corresponding 3 address code for the expression $a + a*(b-c) + (b-c)*d$.



$t_1 = b-c$

$t_2 = a*t_1$

$t_3 = a+t_2$

$t_4 = t_1*d$

$t_5 = t_3+t_4$.

DAG

3-address code

Addresses and Instructions.

3-address code is built from 2 concepts: address & instructions.

An address can be one of the following.

↳ A name: For convenience, we allow source program names to appear as addresses in 3-address code. In an implementation, a source name is replaced by a pointer to its symbol table entry, where all information about the name is kept.

↳ A constant: A compiler must deal with many different types of constants and variables.

↳ A compiler generated temporary: It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.

Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a 3-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching"

Here is a list of the common 3-address instruction forms

(i) Assignment instructions of the form $x = y \text{ op } z$ where op is a binary arithmetic or logical operation & x, y & z are addresses.

(ii) Assignments of the form $x = \text{op } y$, where op is a unary operation

(iii) Copy instructions of the form $x = y$, where x is assigned the value of y.

(iv) An unconditional jump goto L. The 3-address instruction with label L is the next to be executed.

(v) Conditional jumps of the form if x goto L and if False x goto L. These instructions execute the instruction with label L next if x is true and false respectively.

(vi) Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >= etc) to x & y & execute the instruction with label L next if x stands in relation relop to y. If not, the 3 address instruction following if x relop y goto L is executed next, in sequenc

(vii) Procedures calls & returns are implemented using the following instructions: param x for parameters; call p, n & y = call p, n for procedure & function calls respectively & return y, where y representing a returned value, is optional

```
param x1
param x2
...
param xn
call p, n
```

The integer n indicating the no. of actual parameters in call p, n is not redundant because calls can be nested.

(xiii) Indexed copy instructions of the form x = y[i] and x[i] = y. The instruction x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i] = y sets the contents of the location i units beyond x to the value of y.

(ix) Address & pointer assignments of the form $x = \&y$, $x = *y$ and $*x = y$.

Ex: Consider the statement

```
do i = i+1 ; while ( a[i] < v);
```

2 possible translations of this statement are

```
L : t1 = i+1
    i = t1
    t2 = i*8
    t3 = a[t2]
    if t3 < v goto L
```

(a) symbolic labels

```
100 : t1 = i+1
101 : i = t1
102 : t2 = i*8
103 : t3 = a[t2]
104 : if t3 < v goto 100
```

(b) position number.

The translation in (a) uses a symbolic label L, attached to the first instruction. The translation in (b) shows position number for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication i*8 is appropriate for an array of elements that each take 8 units of space.

Quadruples.

A quadruples has 4 fields, which we call op, arg1, arg2, & result. The op field contains an internal code for the operator.

Ex: $x = y + z$ is represented by placing + in op, y in arg1, z in arg2 & x in result.

The following are some exceptions to this rule.

(i) Instructions with unary operators like x = minus y or $x = y$ do not use arg2. Note that for a copy statement like $x = y$, op is =, while for most other operations, the assignment operator is implied.

(ii) Operators like param use neither arg2 nor result.

(iii) Conditional & unconditional jumps put the target label in result.

Ex: Write quadruples for $a = b * - c + b * - c$.

t_1 = minus c

$t_2 = b * t_1$

t_3 = minus c

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

(a) 3-address code

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

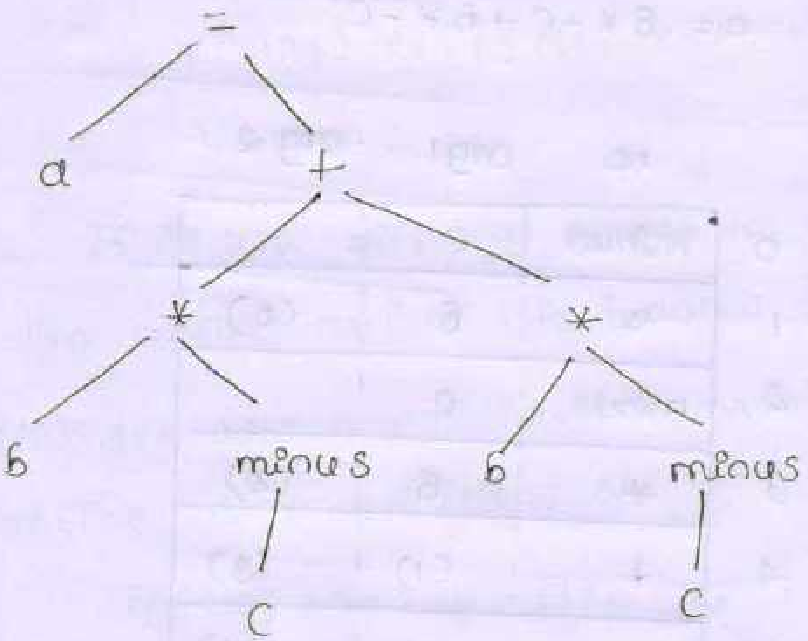
(b) Quadruples.

The special operator minus is used to distinguish the unary minus operator.

Triples

A triples has only 3 fields, op, arg1 & arg2. Note that the result field in quadruples is used primarily for temporary names. Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary names.

Ex: Write triples for a= b*-c + b*-c



(a) Syntax tree

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

(b) Triples.

A ternary operator like x[i]= y requires 2 entries in the triple structure, for example, we can put x & i in one triple & y in the next.

A benifits of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With qudruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

Indirect triples.

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list without affecting the triples themselves.

Ex: Write indirect triples for $a = 6 * -c + 6 * -c$.

instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg1	arg2
0	minus	c	
1	*	6	(0)
2	minus	c	
3	*	6	(2)
4	+	(1)	(3)
5.	=	a	(4)
	...		

Static Single-Assignment Form.

Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.

2 distinctive aspects of SSA that distinguish SSA from 3-address code

(i) All assignments in SSA are to variables with distinct names.

Ex:

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$.
3- address code	static single assignment form

The same variable may be defined in 2 different control flow paths in a program. For example, the source program

```
if (flag) x = -1; else x = 1;
y = x*a;
```

If we use different names for x in the true part & false then conflict arises which name should use in y = x*a.

(ii) SSA uses a notational convention called φ-function to combine the 2 definitions of x

```
if (flag) x1 = -1; else x2 = 1;
x3 = φ (x1, x2);
```

Here $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional & the value x_2 if the control flow passes through the false part.

Translate the arithmetic expression $a + -(b+c)$ into

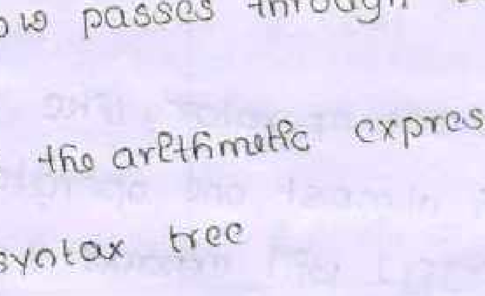
(a) A syntax tree

(b) Quadruples

(c) Triples

(d) Indirect triples.

(a) Syntax tree



$t_1 = b + c$

$t_2 = \text{minus } t_1$

$t_3 = a + t_2$

(b) Quadruples

	op	arg1	arg2	result
0				
1 0	+	b	c	t_1
2 1	minus	t_1		t_2
3 2	=	a	t_2	t_3

(c) triples

	op	arg1	arg2
0	+	b	~~b~~ c
1	minus	(0)	
2	=	a	(1)

(d) Indirect triples.

Instructions

35	(0)
36	(1)
37	(2)

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	=	a	(1)

Translation of Expressions.

An expression with more than one operator, like a+b*c, will translate into instructions with atmost one operator per instruction. An array reference A[i][j] will expand into a sequence of 3-address instructions that calculate an address for the reference.

Operations within Expressions

The following syntax-directed definition builds up the 3-address code for an assignment statement S using attribute code for S & attributes addr & code for an express E. Attributes S.code & E.code denotes the 3 address code for S & E respectively. Attribute E.addr denotes the address

QUESTIONS

1. Define quadruples, triples and static single assignmen

form.

A quadruples has 4 fields, op, arg1, arg2 & re

The op field contains an incremental code for the operat

Ex: the quadruples for $a = (6 * - c) + (6 * - c)$

t_1 = minus c

$t_2 = 6 * t_1$

t_3 = minus c

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

	op	arg1	arg2	result
0	minus	c		t_1
1	*	6	t_1	t_2
2	minus	c		t_3
3	*	6	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

A triples has only 3 fields op, arg1 & arg2

Ex: the triples for $a = 6 * - c + 6 * - c$.

	op	arg1	arg2
0	minus	c	
1	*	6	(0)
2	minus	c	
3	*	6	(2)
4	+	(1)	(3)
5	=	a	(4)

Static single-~~statement~~ assignment form is an intermediate representation that facilitates certain code optimizations

Ex:

$p = a+b$	$p_1 = a+b$
$q = p-c$	$q_1 = p_1 - c$
$p = q*d$	$p_2 = q_1 * d$
$p = e-p$	$p_3 = e-p_2$
$q = p+q$	$q_2 = p_3 + q_1$
(a) 3-address code	(b) static single assignment form.

2. Develop SDD to produce directed acyclic graph for an expression show the steps for constructing the DAG for the expression $a+a*(b-c)+(b-c)*d$.

Syntax directed definition is,

E→ PRODUCTION	SEMANTIC RULES
(i) $E \to E_1 + T$	E.node = new Node ('+', E1.node, T.node)
(ii) $E \to E_1 - T$	E.node = new Node ('−', E1.node, T.node)
(iii) $E \to T$	E.node = T.node
(iv) $T \to (E)$	T.node = E.node
(v) $T \to id$	T.node = new Leaf (id, id.entry)
(vi) $T \to num$	T.node = new Leaf (num, num.val)

Steps for constructing the DAG

(i) P_1 = Leaf (id, entry-a)

(ii) P_2 = Leaf (id, entry-a) = P_1

(iii) P_3 = Leaf (id, entry-b)

(iv) P_4 = Leaf (id, entry-c)

(v) P_5 = Node ('-', P_3, P_4)

(vi) P_6 = Node ('*', P_1, P_5)

(vii) P_7 = Node ('+', P_1, P_6)

~~(ix)~~ (viii) P_8 = Leaf (id, entry-b) = P_3

(ix) P_9 = Leaf (id, entry-c) = P_4

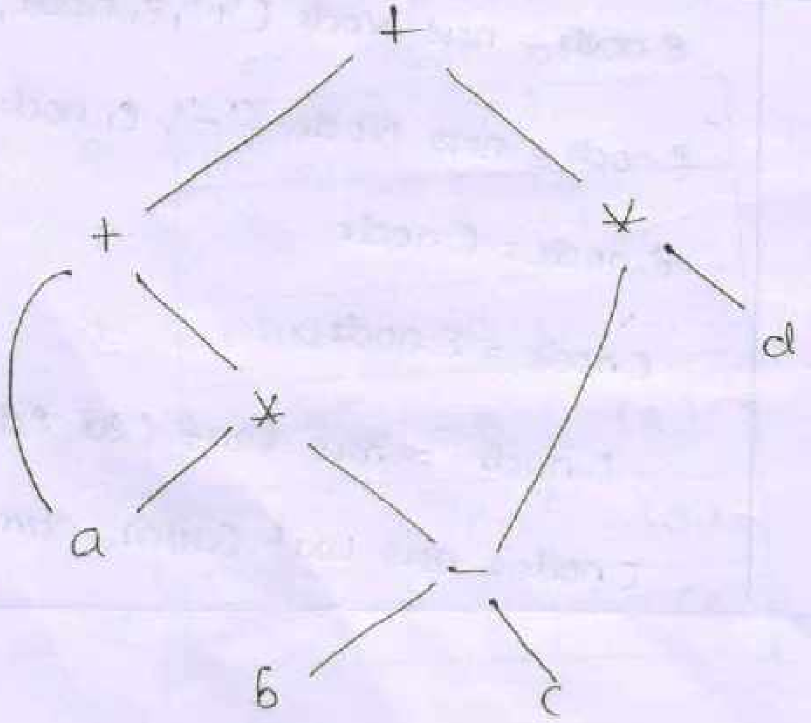
(x) P_{10} = Node ('-', P_3, P_4) = P_5

(xi) P_{11} = Leaf (id, entry-d)

(xii) P_{12} = Node ('*', P_5, P_{11})

(xiii) P_{13} = Node ('+', P_7, P_{12})

DAG



UNIT - 8

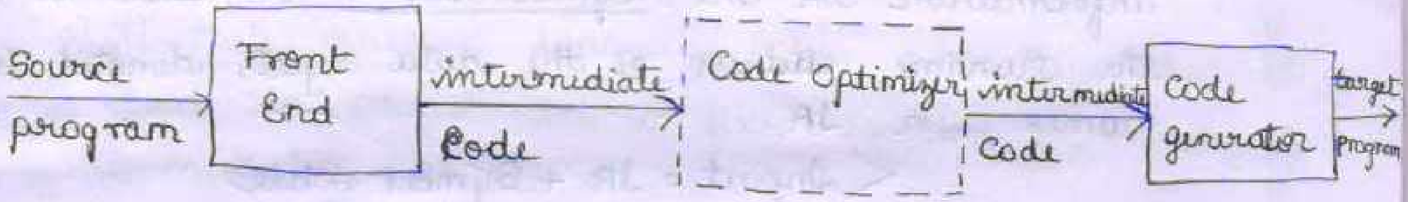
CODE GENERATION

INTRODUCTION :

- Code generation is the final phase in the compiler design

- The code optimizer accepts intermediate code representation which is generated from the front end of the compiler & produces another intermediate code representatn which is optimized.

- Code generator takes intermediate representation produced by code optimizer along with supplementary information in symbol table of the source program & produce as output an equivalent target program.



- Code generator has 3 main tasks:

  1) Instruction selection

  2) Register allocation & assignment

  3) Instruction Ordering

1> INSTRUCTION SELECTION :

Choose a appropriate target machine ~~code~~ instructions to implement the IR [intermediate represent] statements

2> REGISTER ALLOCATION & ASSIGNMENT :

Decide what values to keep in which registers

3> INSTRUCTION ORDERING

Decide in what order. to schedule the execution of instructions.

8.1 ISSUES IN THE DESIGN OF CODE GENERATOR :

1) Input to the code generator
2) The Target program
3) Instruction selection
4) Register Allocation
5) Evaluation Order

1) Input to the Code generator

* Input to the code generator is the intermediate represe of the source program produced by the front end along w information in the symbol table i.e, used to determin the runtime address of the data objects denoted by the names in IR.

< Input = IR + Symbol table>

* IR has several choices

(a) 3-address representation : quadruples, triples, indirect tripl

(b) Virtual machine representat^n : byte codes & stack mach codes

(c) Linear representat^n such as postfix notat^n

(d) Graphical representat^n such as syntax trees & DAG's

* Assumptions made are

(i) Front end produces low-level IR, i.e, values of names in it can be directly manipulated by the machine instruct^n.

(ii) Syntatic & semantic errors have been already detected

2) The Target Program :

- * The Output of code generator is target program.
- * The instn set architecture of the target machine has a significant impact on the design of code generator.
- * Most common architecture are:

(a) CISC : It has few registers, has maximum of 2 operands & variety of addressing mode, variable length instructions & instructn with side effects.

(b) RISC : It has many registers, has maximum of 3 operands with simple addressing modes, & relatively simple instructn set architecture.

- * Output may take variety of forms.

a) Absolute machine language [Executable Code]
b) Relocatable machine language [object files for linker]
c) Assembly language [facilitates debugging]

a) Absolute machine language has advantage that it can be placed in a fixed location in memory & immediately executed.

b) Relocatable machine language program allows subprograms to be compiled seperately.

c) Producing Assembly language program as output makes the process of code generatn somewhat easier.

3) Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine.

* The complexity of performing this mapping is determined by the factors such as:

(i) the level of the IR

(ii) the nature of the instruction set architectures.

(iii) the desired quality of the generated code

(i) the levels of the IR:

> If the IR is high level, use code templates to translate each IR statements into a sequence of machine instruction.

> produces poor code, needs further optimizat^n.

> If the IR is low level, use ~~code~~ ~~this~~ low level informat^n to generate more efficient code sequence.

(ii) the nature of the instruction set architectures has strong effect on difficulty of instruct^n select^n.

> Uniformity & completeness of the instruct^n set are imp factors.

> If we do not care about the efficiency of the target program, instruct^n select^n is straightforward.

> For eg:

```
x = y + z  ⇒  LD   R0, y
              ADD  R0, R0, z
              ST   x, R0
```

∴ produces redundant LD & store

eg2:

```
a = b + c  ⇒  LD   R0, b
d = a + b     ADD  R0, R0, C
              ST   a, R0     ⟶ REDUNDANT
              LD   R0, a
              ADD  R0, R0, C
              ST   d, R0
```

(iii) the quality of the generated Code is determined by its speed & size

>For eg:

$$\boxed{a = a+1 \Rightarrow \begin{array}{l} \text{LD } R_0, a \\ \text{ADD } R_0, R_0, \#1 \\ \text{ST } a, R_0 \end{array} \Bigg\} \xrightarrow[\text{by}]{\text{replaced}} \text{INC } a}$$

4) Register Allocation :

★ Instructⁿ involving register operands are usually shorter & faster then those involving operands in memory.

★ 2 subproblems:

(i) Register allocation: Select the set of variables that will reside in registers at each point in the program.

(ii) Register assignment: Select specific register that a Variable will reside in.

★ Complications imposed by the hardware architecture

Eg. Register pairs for multiplicatⁿ & division.

★ Multiplicatⁿ instⁿ is of the form

$\boxed{M \quad x, y}$

where x ⟶ Multiplicand, is the odd register of an even/odd register pair.

y ⟶ Multiplier , is ~~the~~ a single register

⇒ Product ⟶ occupies the entire even/odd register pair.

★ Division instⁿ is of the form

$\boxed{D \quad x, y}$

where x ⟶ dividend, occupies even register

y ⟶ divisor, occupies odd / even register

⇒ quotient ⟶ stored in ~~even~~ odd register

remainder ⟶ stored in even register

Eg: two 3address code sequences

```
t = a+b          t = a+b
t = t*c          t = t+c
t = t/d          t = t/d
```

Optimal machine-Code sequences

```
L   R1, a        L    R0, a
A   R1, b        A    R0, b
M   R0, c        A    R0, c
D   R0, d        SRDA R0, 32
ST  R1, t        D    R0, d
                 ST   R1, t
```

5) Evaluation Order:

* The order in which computations are performed can effect the efficiency of the Target code.
* when instruct^n are independent their evaluation order can be changed.
* Some computat^n orders require fewer registers to hold intermediate results than others.
* However picking a best order in the general case is a difficult NP-complete problem.

ADDITIONAL INFORMATION: Eg

```
                     t1 = a+b           MOV  R0, a
a+b-(c+d)*e  =>      t2 = c+d    =>     ADD  R0, b
                     t3 = e*t2          MOV  t1, R0
                     t4 = t1-t3         mov  R1, c
                                        ADD  R1, d
           Reorder                      mov  R0, e
                                        MUL  R0, R1
                     t2 = c+d           mov  R1, t1
                     t3 = e*t2          SUB  R1, R0
                     t1 = a+b           MOV  t4, R1
                     t4 = t1-t3  =>
                                        mov  R0, c
                                        ADD  R0, d
                                        mov  R1, e
                                        MUL  R1, R0
                                        mov  R0, a
                                        ADD  R0, b
                                        SUB  R0, R1
                                        mov  t4, R0
```

~~THE~~

8.2 THE TARGET LANGUAGE:

For disigning a good code generator, we need to have familiarity with target machine & its instructⁿ set. Instead of generating code on a specific target machine, a general machine consisting of many registers are considered.

A SIMPLE TARGET MACHINE MODEL:

The characteristics of target machine mode with instructⁿ format & instructⁿ set are shown below:

* Our hypothetical machine:
  (i) It is a 3-address machine with the following format
  OP destination, Source1, Source2

  NOTE: A 3 address instructⁿ can have 2 operands or 1 operands also but it can have max of 3 operands

  (ii) The target machine is byte addressable i.e., it can access 8 bit of info from specific address

  (iii) It has n no of registers denoted by $R_0, R_1, R_2, \ldots, R_{n-1}$

* Various types of instructⁿ that are used by target m/c
  (i) Load Instructⁿ
  (ii) Store Instructⁿ
  (iii) Computational Instructⁿ
  (iv) Unconditional Instructⁿ
  (v) Conditional Instructⁿ

(i) Load Instructⁿ: Used to copy the data into destinatⁿ operand which must be a register.

SYNTAX: LD dst, addr

where addr operand → register or memory locat

(ii) Store instruction: Used to copy the data into memo locat^n specified in the destinat^n operand.

SYNTAX: ST dst, r

where dst → destination of it is a memo location

r → register.

Computational operation.

(iii) Arithmetic or instruction: They are performed using these instruction.

SYNTAX: OP dst, src1, src2.

where 1st operand, dst → destination

2nd & 3rd operand → Operands where R values fetched for Operat^n to be p

Eg1: ADD R0, R1, R2 //R0 = R1 + R2

Eg2: SUB R0, R0, R1 // R0 = R0 - R1

Eg3: MUL R2, R0, R1 //R2 = R0 * R1

(iv) Unconditional Jumps: The branch instruct^n without any condit^n are called unconditional jumps.

SYNTAX: BR label

where BR → BRanch instruct^n

(v) Conditional Jumps: Based on the value stored in a register i.e., whether it is +ve or zero or -ve, if branching takes place, then the branch inst^n are called Conditional jumps.

SYNTAX: Bcond r, label

where B stands from Branch,

Cond can be LT less than, GT Greater than, LTE less than or equal, GTE Greater than or equal r → register, contains value such as 0, +ve or -ve.

Eg1: BL R0,T1 //Branch to T1, if R0 contains -ve value

Eg2: BLTZ R1,T2 //Branch to T2, if R1 contains either 0 or -ve value

* Different addressing modes supported by generalized target machine:

1) Direct addressing mode
2) Indexed —"—
3) Integer Indexed —"—
4) Indirect —"—
5) Immediate —"—

(i) Direct A/M :

Address of the data to be accessed is directly present in the instructⁿ, i.e., location is identified by a variable name x.

Eg: ~~LDP~~ LD R1, x //Load value stored in memory locatⁿ x into R1

(ii) Indexed A/M: The data can be accessed from a memory locatⁿ using index. This addressing mode is useful for accessing arrays, where a is the base address of the array & register holds the index value

Eg: LD R1, a(R2) //Acceps the data stored in R1= Contents(a + contents(R2))

(iii) Indexed A/M where memory locatⁿ is integer

It is same as previous one except that a memory locatⁿ is identified as integer.

Eg: LD R1, 100(R2) //R1 = contents(100+ contents(R2))

(iv) Indirect A/M : Contents of the data can be accessed by deferencing using * operators as shown below:

```
LD   R1, *(R2)        // R2 contains memory locat
                         the data stored in that
                         memory locatn is copied in
                         register R1
```

```
LD   R1, *100(R2)     // R1 = Contents( contents( 100 +
                                           contents(R
```

(v) Immediate A/M : The data to be manipulated is directly present in the instruction & preceded by

```
LD   R1, #100         // R1 ← 100
```

EXERCISE :

1. Generate code for 3 address statement for $x = y - z$

```
LD    R1, y          // R1 = y
LD    R2, z          // R2 = z
ADD   R1, R1, R2     // R1 = R1 + R2
ST    x, R1          // x = R1
```

2. Generate code for 3 address statement $x = *p$

```
LD    R1, p          // R1 ← p
LD    R2, 0(R1)      // R2 = Contents (0 + conte
ST    x, R2          // x = R2
```

(iv) **Indirect A/M**: Contents of the data can be accesed by dereferencing using * operators as shown below:

```
LD  R1, *(R2)        // R2 contains memory locat
                        the data stored in that
                        memory locatn is copied in
                        register R1

LD  R1, *100(R2)     // R1 = contents( contents( 100 +
                                        contents(R
```

(iv) **Immediate A/M**: The data to be manipulated is directly present in the instruction & preceded by #

```
LD  R1, #100         // R1 ← 100
```

EXERCISE:

1. Generate code for 3 address statement for x = y - z

```
LD   R1, y           // R1 = y
LD   R2, z           // R2 = z
ADD  R1, R1, R2      // R1 = R1 + R2
ST   x, R1           // x = R1
```

2. Generate code for 3 address statement x = *p

```
LD   R1, p           // R1 ← p
LD   R2, 0(R1)       // R2 = contents (0 + conte
ST   x, R2           // x = R2
```

3. Generate code for 3 address statement *p = y

```
LD   R1, p            // R1 = p
LD   R2, y            // R2 = y
ST   0(R1), R2        // contents (0 + contents (R1)) = R2.
```

4. Generate m/c code for 3 address statement b = a[i]

```
LD    R1, i           // R1 = i
MUL   R1, R1, 8       // R1 = R1 * 8
LD    R2, a[R1)       // R2 = contents (a + contents (R1))
ST    b, R2           // b = R2
```

5. Generate m/c code for 3 address statement a[j] = c

```
LD   R1, j            // R1 = j
LD   R2, C.           // R2 = C
MUL  R1, R1, 8        // R1 = R1 * C
ST   a[R1], R2        // contents (a + contents (R1)) = R2
```

6. Generate m/c code for 3 address statement
if x < y goto L

```
LD    R1, x           // R1 = x
LD    R2, y           // R2 = y
SUB   R1, R1, R2.     // R1 = R1 - R2
BLTZ  R1, M           // if R1 < 0 jump to M.
```

Program & Instruction Cost

* For simplicity we take the cost of an instruction to be one plus the costs associated with the addressin modes of the operands.

* A/M involves registers have zero additional cost.
* A/M involving memory locatⁿ or constant have additional cost of 1.
* For example:

a) LD R0, R1 ⇒ cost = 1
b) LD R0, M ⇒ Cost = 2.
c) LD R1, *100 (R2) ⇒ cost = 3

* Cost of Addressing mode:

	Mode	Form	Address	Added Cost
①	Absolute direct A/M	M	M	1
②	Register direct A/M	R	R	0
③	Indexed A/M	C(R)	C+contents (R)	1
④	Indirect register A/M	*R	contents (R)	0
⑤	Indirect ~~ind~~ indexed A/M	*C(R)	contents (C+contents (R))	1
⑥	Immediate A/M	#C	N/A	1

NOTE: Cost of each statement = 1 + Cost (Addressing mode)

EXERCISES (8.2)

1. Determine the costs of the following instruct^n sequence

	Cost = 1 + cost(AM)
LD R0, y	cost = 1 + 1 = 2
LD R1, z	Cost = 1 + 1 = 2
ADD R0, R0, R1	Cost = 1 + 0 = 1
ST x, R0	Cost = 1 + 1 = 2
	Total Cost = 7

2.
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

		Cost = Cost (AM) + 1.
LD	R0, i	Cost = 1 + 1 = 2
MUL	R0, R0, 8	Cost = 1 + 1 = 2
LD	R1, a(R0)	Cost = 1 + 1 = 2.
ST	b, R1	Cost = 1 + 1 = 2
		Total cost = 8

3.
```
LD  R0, C
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

	Cost = cost(A.M) + 1
LD R0, C	1 + 1 = 2
LD R1, i	1 + 1 = 2
MUL R1, R1, 8	1 + 1 = 2
ST a(R1), R0	1 + 1 = 2
	Total cost = 8

4. LD R0, p

LD R1, O(R0)

ST x, R1

	Cost = Cost(A.M) + 1
LD R0, p	1 + 1 = 2
LD R1, O(R0)	1 + 1 = 2. // ∵ of constan
ST x, R1	1 + 1 = 2

Total cost = 6

5. LD R0, P

LD R1, x

ST O(R0), R1

	Cost = Cost(A.M) + 1
LD R0, P	1 + 1 = 2
LD R1, x	1 + 1 = 2
ST O(R1), R1	1 + 1 = 2

Total Cost = 6

6. LD R0, x

LD R1, y

SUB R0, R0, R1

BLTZ *R3, R0

	Cost = 1 + cost(A.M)
LD R0, x	1 + 1 = 2
LD R1, y	1 + 1 = 2
SUB R0, R0, R1	1 + 0 = 1
BLTZ *R3, R0	1 + 1 = 2 <∵ indirect A

Total cost = 7