System Software

Semester : VI	**Course Code :** 15CS63
Course Title : System Software AND Compiler Design	
Faculty : Niranjan Murthy C	
Dept : Computer Science & engineering	

Prerequisites:	Basic concepts of microprocessors (10CS45)
Description	This **course** gives an introduction to the design and implementation of various types of system software. A central theme of the course is the relationship between machine architecture and system software. The design of an assembler or an operating system is greatly influenced by the architecture of the machine on which it runs. These influences are emphasized and demonstrated through the discussion of actual pieces of system softare fo a variety of real machines.

Outcomes

The students should be able to:

1. Student able to Define System Sotware such as Assembler and Macroprocessor.
2. Student able to Define System Sotware such as Loaders and Linkers
3. Student able to lexical analysis and syntax analysisFamiliaize with source file ,object and executable file structures and libraries
4. Describe the front and back end phases of compiler and their importance to students

MODULE- 1

- **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, ->10 Hours**

MACHINE ARCHITECTURE

System Software:

- System software consists of a variety of programs that support the operation of a computer.
- Application software focuses on an application or problem to be solved.
- System softwares are the machine dependent softwares that allows the user to focus on the application or problem to be solved, without bothering about the details of how the machine works internally.

Examples: Operating system, compiler, assembler, macroprocessor, loader or linker, debugger, text editor, database management systems, etc.

Difference between System Software and application software

System Software	Application Software
System software is machine dependent	Application software is not dependent on the underlying hardware.
System software focus is on the computing system.	Application software provides solution to a problem
Examples: Operating system, compiler, assembler	Examples: Antivirus, Microsoft office

SIC – Simplified Instructional Computer

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. There are two versions of SIC, they are, standard model (SIC), and, extension version (SIC/XE) (extra equipment or extra expensive).

SIC Machine Architecture:

We discuss here the SIC machine architecture with respect to its Memory and Registers, Data Formats, Instruction Formats, Addressing Modes, Instruction Set, Input and Output.

Memory:

There are 215 bytes in the computer memory, that is 32,768 bytes. It uses Little Endian format to store the numbers, 3 consecutive bytes form a word , each location in memory contains 8-bit bytes.

Registers:

There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

Data Formats:

Integers are stored as 24-bit binary numbers. 2's complement representation is used for negative values, characters are stored using their 8-bit ASCII codes.No floating-point hardware on the standard version of SIC.

Instruction Formats:

Opcode(8)	x	Address (15)

X is used to indicate indexed-addressing mode.

All machine instructions on the standard version of SIC have the 24-bit format as shown above.

Addressing Modes:

Only two modes are supported: Direct and Indexed

Mode	Indication	Target address calculation
Direct	x= 0	TA = address
Indexed	x= 1	TA = address + (x)

() are used to indicate the content of a register.

Instruction Set

- Load and store registers (LDA, LDX, STA, STX)
- Integer arithmetic (ADD, SUB, MUL, DIV), all involve register A and a word in memory.
- Comparison (COMP), involve register A and a word in memory.
- Conditional jump (JLE, JEQ, JGT, etc.)
- Subroutine linkage (JSUB, RSUB)

Input and Output

- One byte at a time to or from the rightmost 8 bits of register A.
- Each device has a unique 8-bit ID code.
- Test device (TD): test if a device is ready to send or receive a byte of data.
- Read data (RD): read a byte from the device to register A
- Write data (WD): write a byte from register A to the device.

SIC/XE Machine Architecture:

Memory

- Maximum memory available on a SIC/XE system is 1 Megabyte (2 20 bytes).

Registers

- Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC.

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

Floating-point data type:

- There is a 48-bit floating-point data type, F*2(e-1024)

Instruction Formats :

The new set of instruction formats fro SIC/XE machine architecture are as follows.

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, F is replaced by hex 6).

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 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.

Addressing Modes:

Five possible addressing modes plus the combinations are as follows.

1. 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.

2. 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)

3. 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)

4. Indirect(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

5. 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.

The various flag bits used in the above formats have the following meanings

e - > e = 0 means format 3, e = 1 means format 4

Bits x,b,p : Used to calculate the target address using relative, direct, and indexed addressing Modes.

Bits i and n: Says, how to use the target address b and p - both set to 0, disp field from format 3 instruction is taken to be the target address.

For a format 4 bits b and p are normally set to 0, 20 bit address is the target address

x -x is set to 1, X register value is added for target address calculation

i=1, n=0 Immediate addressing, TA: TA is used as the operand value, no memory reference

i=0, n=1 Indirect addressing, ((TA)): The word at the TA is fetched. Value of TA is taken as the address of the operand value

i=0, n=0 or i=1, n=1 Simple addressing, (TA):TA is taken as the address of the operand value

Two new relative addressing modes are available for use with instructions assembled using format 3.

Instruction Set:

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating-point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction : RMO, Register-to-register arithmetic operations, ADDR, SUBR, MULR, DIVR and, Supervisor call instruction : SVC.

Input and Output:

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

Example programs SIC:

Example 1: Simple data and character movement operation

```
            LDA         FIVE
            STA         ALPHA
            LDCH        CHARZ
            STCH        C1
ALPHA       RESW        1
FIVE        WORD        5
CHARZ       BYTE        C’Z’
C1          RESB        1
```

Example 2: Arithmetic operations

```
            LDA         ALPHA
            ADD         INCR
            SUB         ONE
            STA         BETA
……..
```

```
……..
……..
ONE         WORD        1
ALPHA       RESW        1
BEETA       RESW        1
INCR        RESW        1
```

Example 3: Looping and Indexing operation

```
            LDX         ZERO                ;       X = 0
MOVECH      LDCH        STR1, X
            STCH        STR2, X
            TIX         ELEVEN
            JLT         MOVECH
. . . . . .
. . . . . .
. . . . . .
STR1        BYTE        C ‘HELLO WORLD’
STR2        RESB        11
ZERO        WORD        0
ELEVEN      WORD        11
```

Example 4: Input and Output operation

```
INLOOP          TD          INDEV       ; TEST INPUT DEVICE
                JEQ         INLOOP      ; LOOP UNTIL DEVICE IS READY
                RD          INDEV       ; READ ONE BYTE INTO A
                STCH        DATA        ; STORE A TO DATA
.
.
OUTLP           TD          OUTDEV      ; TEST OUTPUT DEVICE
                JEQ         OUTLP       ; LOOP UNTIL DEVICE IS READY
                LDCH        DATA        ; LOAD DATA INTO A
                WD          OUTDEV      ; WRITE A TO OUTPUT DEVICE
.
.
INDEV           BYTE        X ‘F5’      ; INPUT DEVICE NUMBER
OUTDEV          BYTE        X ‘08’      ; OUTPUT DEVICE NUMBER
DATA            RESB        1           ; ONE-BYTE VARIABLE
```

Example 5: To transfer two hundred bytes of data from input device to memory

```
                LDX         ZERO
CLOOP           TD          INDEV
                JEQ         CLOOP
                RD          INDEV
                STCH        RECORD, X
                TIX         B200
                JLT         CLOOP
.
.
INDEV           BYTE        X ‘F5’
RECORD          RESB        200
ZERO            WORD        0
B200            WORD        200
```

Example Programs (SIC/XE)

Example 1: Simple data and character movement operation

```
                LDA         #5
                STA         ALPHA
                LDA         #90
.
.
.
ALPHA           RESW        1
C1              RESB        1
```

Example 2: Arithmetic operations

```
                LDS         INCR
                LDA         ALPHA
                ADD         S,A
                SUB         #1
                STA         BETA
…………
…………
ALPHA           RESW        1
BETA            RESW        1
INCR            RESW        1
```

Example 3: Looping and Indexing operation

```
                LDT         #11
                LDX         #0              ;X = 0
MOVECH          LDCH        STR1, X         ; LOAD A FROM STR1
                STCH        STR2, X         ; STORE A TO STR2
                TIXR        T
                JLT         MOVECH
.
.
STR1            BYTE        C ‘HELLO WORLD’
STR2            RESB        11
```

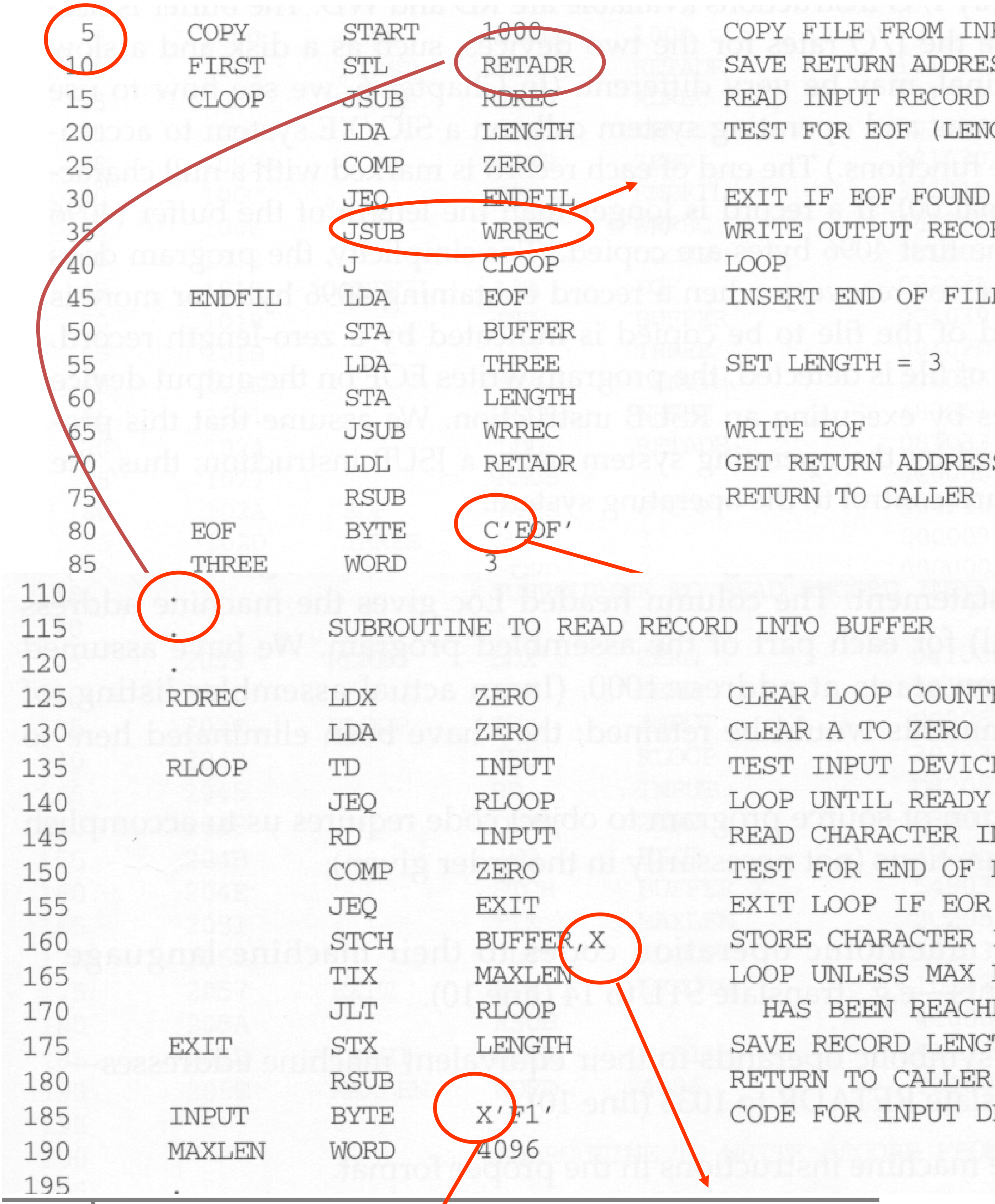
Assemblers - 1

A Simple Two-Pass Assembler

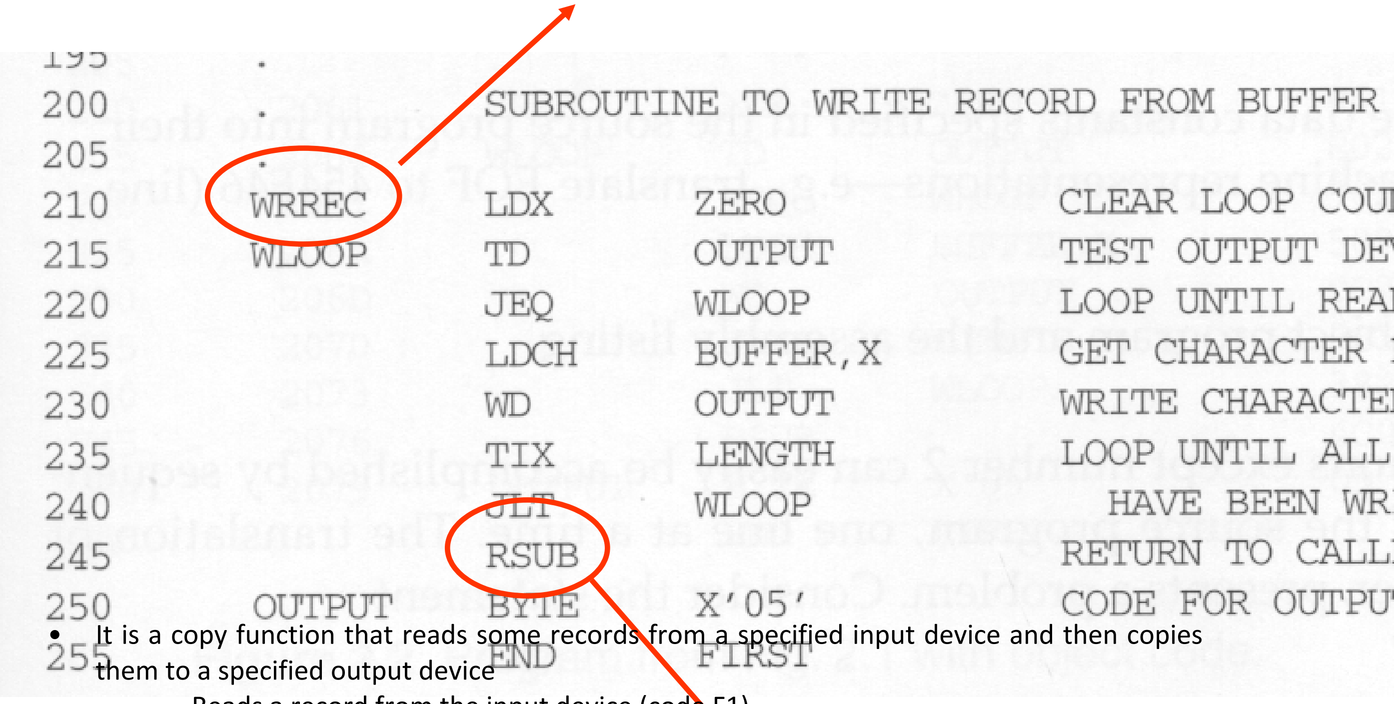
Main Functions

- Translate mnemonic operation codes to their machine language equivalents
- Assign machine addresses to symbolic labels used by the programmers
- Depend heavily on the source language it translates and the machine language it produces.
- E.g., the instruction format and addressing modes

Basic Functions of an Assembler



```
  5    COPY     START   1000        COPY FILE FROM IN
 10    FIRST    STL     RETADR      SAVE RETURN ADDRES
 15    CLOOP    JSUB    RDREC       READ INPUT RECORD
 20             LDA     LENGTH      TEST FOR EOF (LENG
 25             COMP    ZERO
 30             JEQ     ENDFIL      EXIT IF EOF FOUND
 35             JSUB    WRREC       WRITE OUTPUT RECOR
 40             J       CLOOP       LOOP
 45    ENDFIL   LDA     EOF         INSERT END OF FIL
 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
110    .
115    .        SUBROUTINE TO READ RECORD INTO BUFFER
120    .
125    RDREC    LDX     ZERO        CLEAR LOOP COUNT
130             LDA     ZERO        CLEAR A TO ZERO
135    RLOOP    TD      INPUT       TEST INPUT DEVIC
140             JEQ     RLOOP       LOOP UNTIL READY
145             RD      INPUT       READ CHARACTER I
150             COMP    ZERO        TEST FOR END OF
155             JEQ     EXIT        EXIT LOOP IF EOR
160             STCH    BUFFER,X    STORE CHARACTER
165             TIX     MAXLEN      LOOP UNLESS MAX
170             JLT     RLOOP         HAS BEEN REACH
175    EXIT     STX     LENGTH      SAVE RECORD LENG
180             RSUB                RETURN TO CALLER
185    INPUT    BYTE    X'F1'       CODE FOR INPUT D
190    MAXLEN   WORD    4096
195    .
```



```
195         .
200         .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205         .
210   WRREC         LDX     ZERO              CLEAR LOOP COU
215   WLOOP         TD      OUTPUT            TEST OUTPUT DE
220                 JEQ     WLOOP             LOOP UNTIL REA
225                 LDCH    BUFFER,X          GET CHARACTER
230                 WD      OUTPUT            WRITE CHARACTE
235                 TIX     LENGTH            LOOP UNTIL ALL
240                 JLT     WLOOP               HAVE BEEN WR
245                 RSUB                      RETURN TO CALL
250   OUTPUT        BYTE    X'05'             CODE FOR OUTPU
255                 END     FIRST
```

- It is a copy function that reads some records from a specified input device and then copies them to a specified output device
  - Reads a record from the input device (code F1)
  - Copies the record to the output device (code 05)
  - Repeats the above steps until encountering EOF.
  - Then writes EOF to the output device
  - Then call RSUB to return to the caller
  -

RDREC and WRREC

- Data transfer
  - A record is a stream of bytes with a null character (0016) at the end.
  - If a record is longer than 4096 bytes, only the first 4096 bytes are copied.
  - EOF is indicated by a zero-length record. (I.e., a byte stream with only a null character.
  - Because the speed of the input and output devices may be different, a buffer is used to temporarily store the record
- Subroutine call and return
  - On line 10, “STL RETADDR” is called to save the return address that is already stored in register L.
  - Otherwise, after calling RD or WR, this COPY cannot return back to its caller.

Assembler Directives

- Assembler directives are pseudo instructions
  - They will not be translated into machine instructions.
  - They only provide instruction/direction/information to the assembler.
- Basic assembler directives :
  - START : Specify name and starting address for the program
  - END : Indicate the end of the source program, and (optionally) the first executable instruction in the program. Assembler Directives (cont’d)
  - BYTE : Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

  - WORD : Generate one-word integer constant
  - RESB : Reserve the indicated number of bytes for a data area
  - RESW : Reserve the indicated number of words for a data area

An Assembler's Job

- Convert mnemonic operation codes to their machine language codes
- Convert symbolic (e.g., jump labels, variable names) operands to their machine addresses
- Use proper addressing modes and formats to build efficient machine instructions
- Translate data constants into internal machine representations
- Output the object program and provide other information (e.g., for linker and loader)

Object Program Format

- Header

  Col. 1 H

  Col. 2~7 Program name

  Col. 8~13 Starting address of object program (hex)

  Col. 14-19 Length of object program in bytes (hex)

- Text

  Col.1 T

  Col.2~7 Starting address for object code in this record (hex)

  Col. 8~9 Length of object code in this record in bytes (hex)

  Col. 10~69 Object code, represented in hexa (2 col. per byte)

- End

  Col.1 E

  Col.2~7 Address of first executable instruction in object program (hex)

The Object Code for COPY

```
H COPY  001000 00107A

T 001000 1E 141033 482039 001036 281030 301015 482061 3C1003

            00102A 0C1039 00102D

T 00101E 15 0C1036 482061 081044 4C0000 454F46 000003 000000

T 002039 1E 041030 001030 E0205D 30203F D8205D 281030 302057

            549039 2C205E 38203F

T 002057 1C 101036 4C0000 F1 001000 041030 E02079 302064 509039

            DC2079 2C1036
```

T 002073 07 382064 4C0000 05

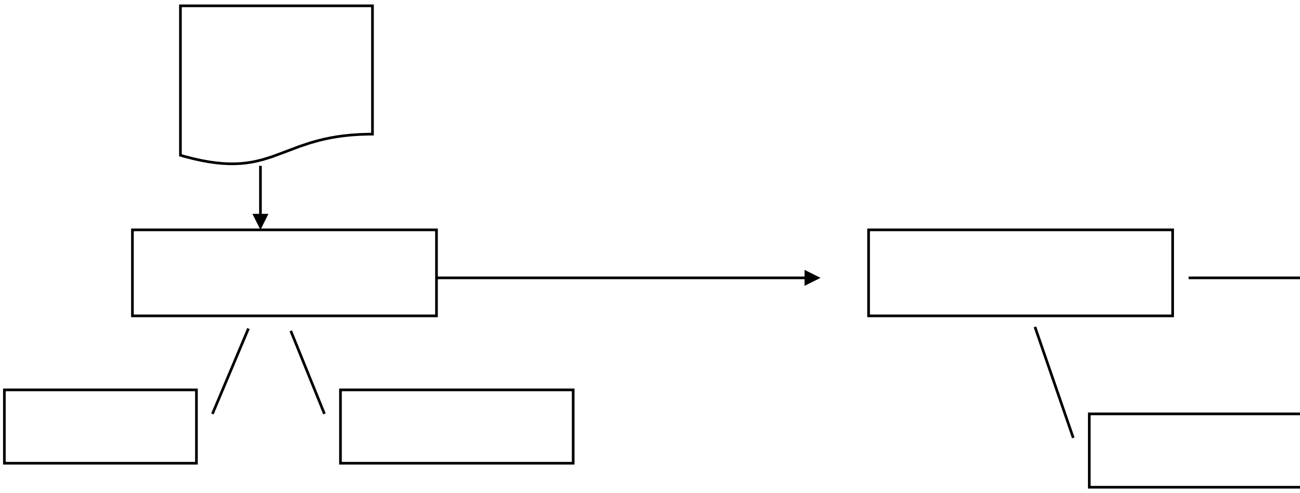
E 001000

NOTE: There is no object code corresponding to addresses 1033-2038. This storage is simply reserved by the loader for use by the program during execution.

Two Pass Assembler

- Pass 1
    - Assign addresses to all statements in the program
    - Save the values (addresses) assigned to all labels (including label and variable names) for use in Pass 2 (deal with forward references)
    - Perform some processing of assembler directives (e.g., BYTE, RESW, these can affect address assignment)
- Pass 2
    - Assemble instructions (generate opcode and look up addresses)
    - Generate data values defined by BYTE, WORD
    - Perform processing of assembler directives not done in Pass 1
    - Write the object program and the assembly listing

A Simple Two Pass Assembler Implementation



Algorithms and Data Structures

Three Main Data Structures

- Operation Code Table (OPTAB)
- Location Counter (LOCCTR)
- Symbol Table (SYMTAB)

OPTAB (operation code table)

- Content
    - The mapping between mnemonic and machine code. Also include the instruction format, available addressing modes, and length information.
- Characteristic
    - Static table. The content will never change.
- Implementation

  - Array or hash table. Because the content will never change, we can optimize its search speed.
- In pass 1, OPTAB is used to look up and validate mnemonics in the source program.
- In pass 2, OPTAB is used to translate mnemonics to machine instructions.

Location Counter (LOCCTR)

- This variable can help in the assignment of addresses.
- It is initialized to the beginning address specified in the START statement.
- After each source statement is processed, the length of the assembled instruction and data area
- to be generated is added to LOCCTR.
- Thus, when we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

Symbol Table (SYMTAB)

- Content
  – Include the label name and value (address) for each label in the source program.
  – Include type and length information (e.g., int64)
  – With flag to indicate errors (e.g., a symbol defined in two places)
- Characteristic
  – Dynamic table (I.e., symbols may be inserted, deleted, or searched in the table)
- Implementation
  – Hash table can be used to speed up search – Because variable names may be very similar (e.g., LOOP1, LOOP2), the selected hash function must perform well with such non-random keys.

The Pseudo Code for Pass 1

```
Begin
    read first input line
    if OPCODE = ‘START’ then begin
        save #[Operand] as starting addr
        initialize LOCCTR to starting address
        write line to intermediate file
        read next 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
        (if symbol)
        search OPTAB for OPCODE
        if found then
            add 3 (instr 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
        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}
```

The Pseudo Code for Pass 2

```
Begin
    read 1st input line
    if OPCODE = 'START' then
    begin
        write listing line
        read next input line
    end
    write Header record to object program
    initialize 1st Text record
    while OPCODE != 'END' do
    begin
        if this is not 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 field then
                    if found then
                    begin
                    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
                else if OPCODE = ‘BYTE’ or ‘WORD” then
                    convert constant to object code
                if object code doesn’t fit into current Text record then
                begin
                    Write text record to object code
                    initialize new Text record
                end
                add object code to Text record
            end {if not comment}
            write listing line
            read next input line
        end
        write listing line
        read next input line
        write last listing line
End {Pass 2}
```

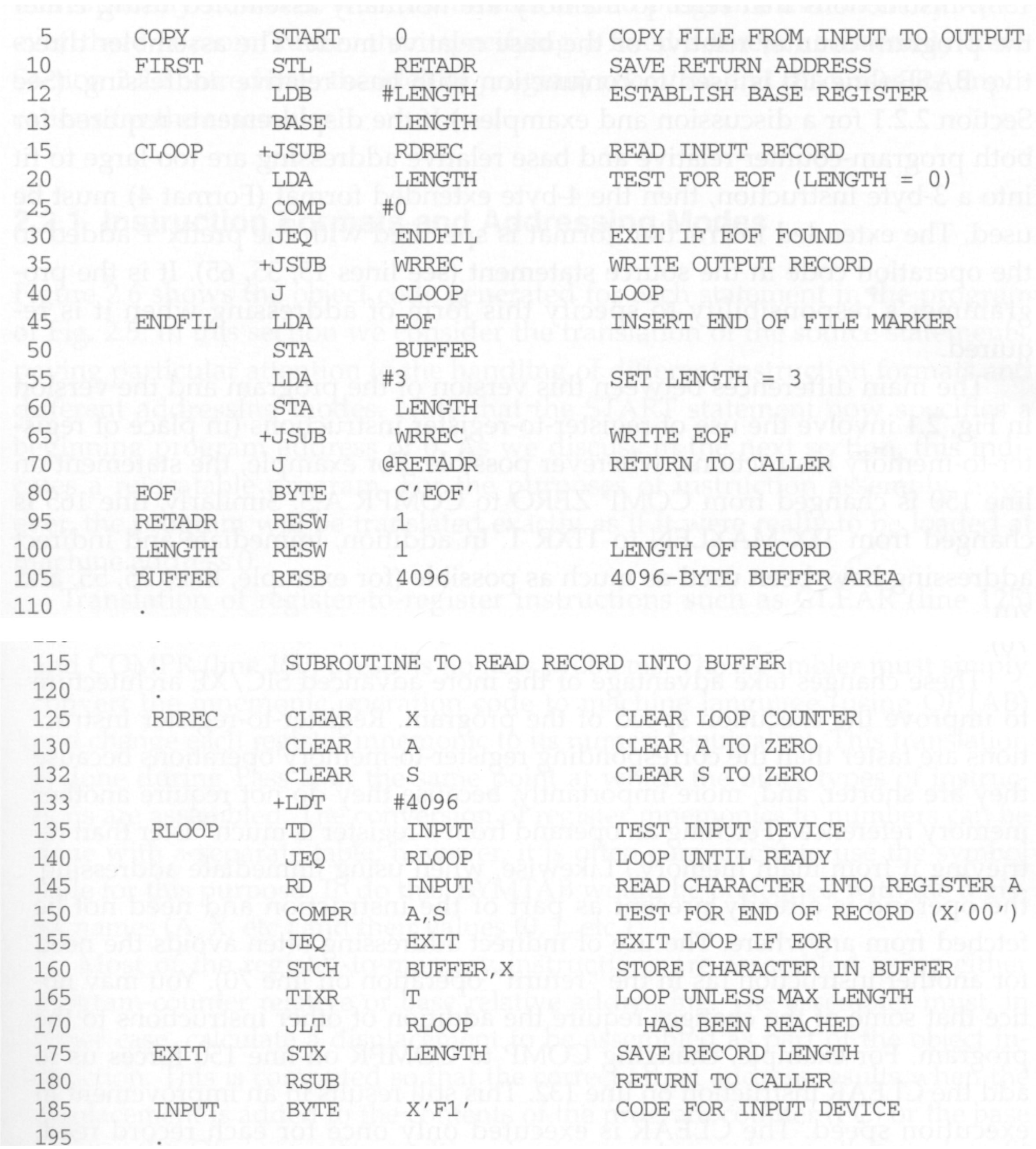
Machine dependent Assembler Features

Assembler Features

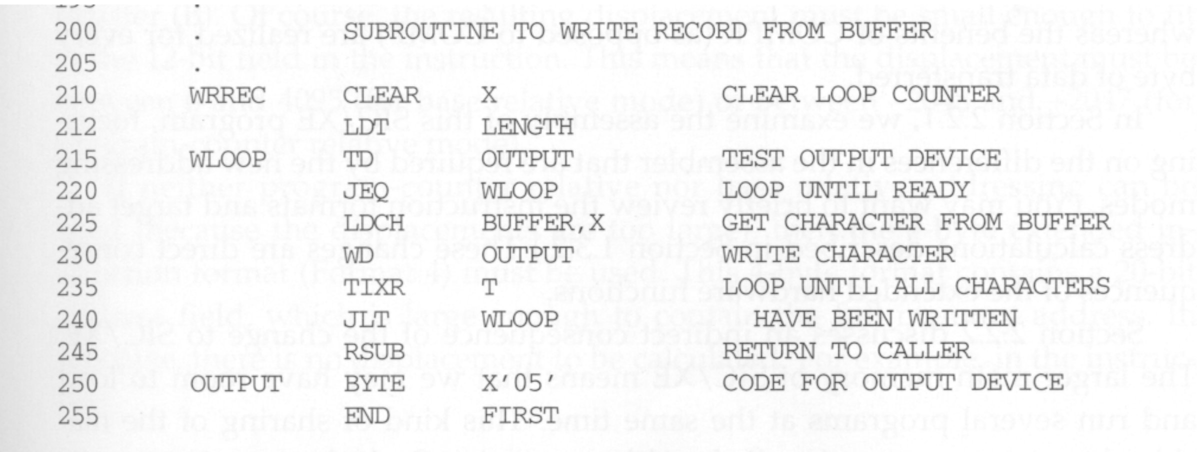
- Machine Dependent Assembler Features
    - Instruction formats and addressing modes (SIC/XE)
    - Program relocation
- Machine Independent Assembler Features
    - Literals
    - Symbol-defining statements
    - Expressions

- Program blocks
- Control sections and program linking

A SIC/XE Program



```
COPY     START    0          COPY FILE FROM INPUT TO OUTPUT
FIRST    STL      RETADR     SAVE RETURN ADDRESS
         LDB      #LENGTH    ESTABLISH BASE REGISTER
         BASE     LENGTH
CLOOP   +JSUB     RDREC      READ INPUT RECORD
         LDA      LENGTH     TEST FOR EOF (LENGTH = 0)
         COMP     #0
         JEQ      ENDFIL     EXIT IF EOF FOUND
        +JSUB     WRREC      WRITE OUTPUT RECORD
         J        CLOOP      LOOP
ENDFIL   LDA      EOF        INSERT END OF FILE MARKER
         STA      BUFFER
         LDA      #3         SET LENGTH = 3
         STA      LENGTH
        +JSUB     WRREC      WRITE EOF
         J        @RETADR    RETURN TO CALLER
EOF      BYTE     C'EOF'
RETADR   RESW     1
LENGTH   RESW     1          LENGTH OF RECORD
BUFFER   RESB     4096       4096-BYTE BUFFER AREA
.

.        SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC    CLEAR    X          CLEAR LOOP COUNTER
         CLEAR    A          CLEAR A TO ZERO
         CLEAR    S          CLEAR S TO ZERO
        +LDT      #4096
RLOOP    TD       INPUT      TEST INPUT DEVICE
         JEQ      RLOOP      LOOP UNTIL READY
         RD       INPUT      READ CHARACTER INTO REGISTER A
         COMPR    A,S        TEST FOR END OF RECORD (X'00')
         JEQ      EXIT       EXIT LOOP IF EOR
         STCH     BUFFER,X   STORE CHARACTER IN BUFFER
         TIXR     T          LOOP UNLESS MAX LENGTH
         JLT      RLOOP        HAS BEEN REACHED
EXIT     STX      LENGTH     SAVE RECORD LENGTH
         RSUB                RETURN TO CALLER
INPUT    BYTE     X'F1'      CODE FOR INPUT DEVICE
.
```



```
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
```

SIC/XE Instruction Formats and Addressing Modes

- PC-relative or Base-relative (BASE directive needs to be used) addressing: **op m**
- Indirect addressing: **op @m**
- Immediate addressing: **op #c**
- Extended format (4 bytes): **+op m**
- Index addressing: **op m,X**
- Register-to-register instructions

Relative Addressing Modes

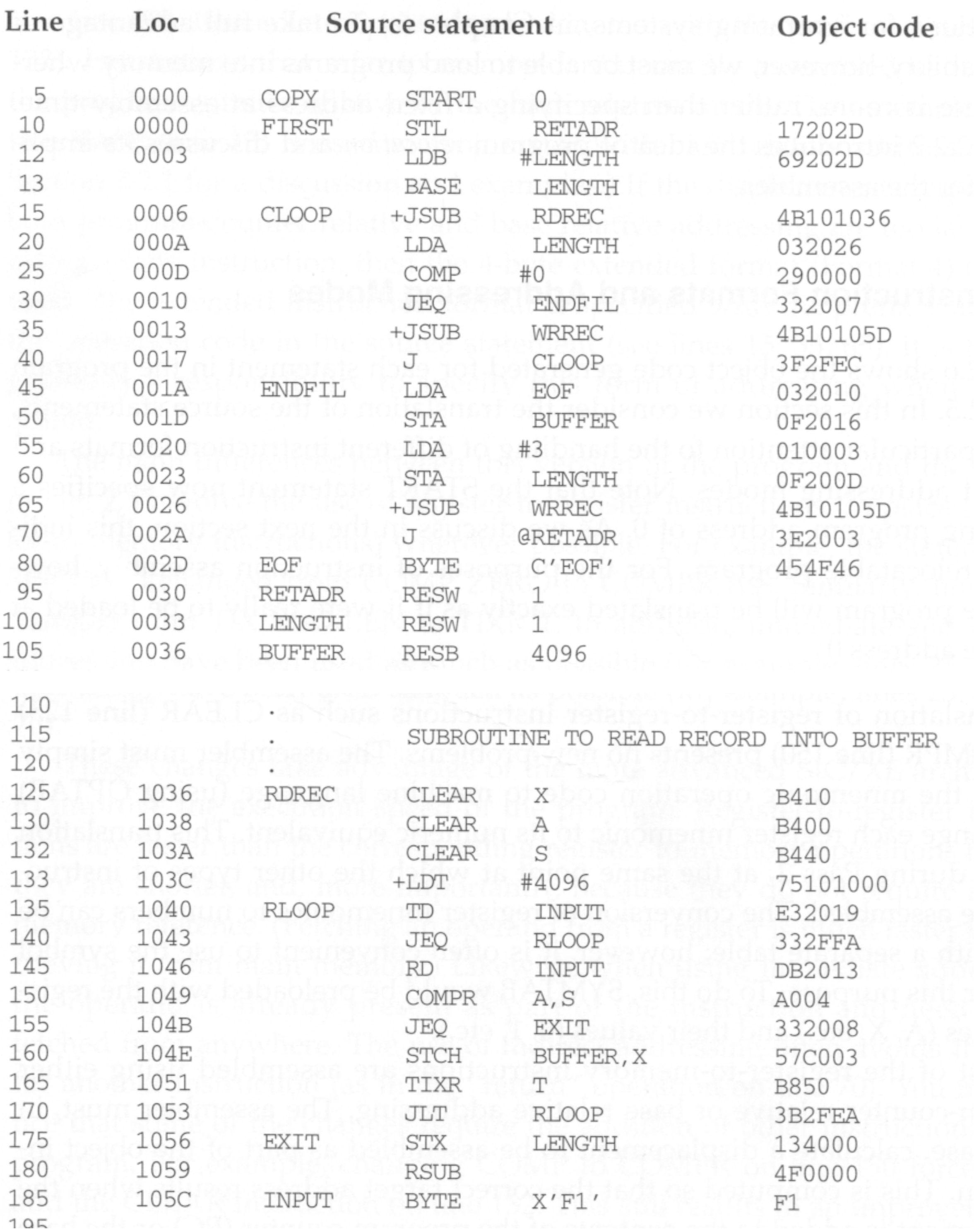
- PC-relative or base-relative addressing mode is preferred over direct addressing mode.
  - Can save one byte from using format 3 rather than format 4.
    - Reduce program storage space
    - Reduce program instruction fetch time
  - Relocation will be easier.

The Differences Between the SIC and SIC/XE Programs

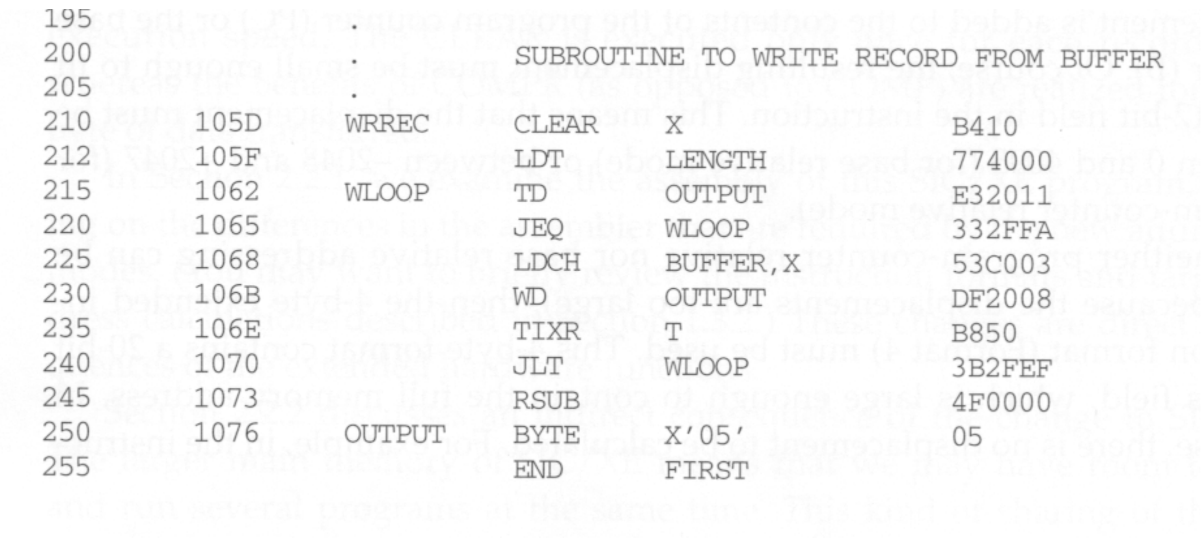
- Register-to-register instructions are used whenever possible to improve execution speed.
  - Fetch a value stored in a register is much faster than fetch it from the memory.
- Immediate addressing mode is used whenever possible.
  - Operand is already included in the fetched instruction. There is no need to fetch the operand from the memory.
- Indirect addressing mode is used whenever possible.

- Just one instruction rather than two is enough.

The Object Code



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					



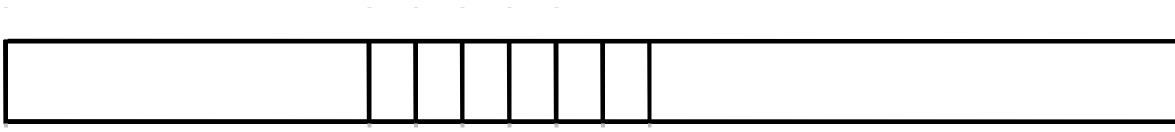
```
195               .
200               .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205               .
210   105D  WRREC      CLEAR     X                 B410
212   105F             LDT       LENGTH            774000
215   1062  WLOOP      TD        OUTPUT            E32011
220   1065             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                    END       FIRST
```

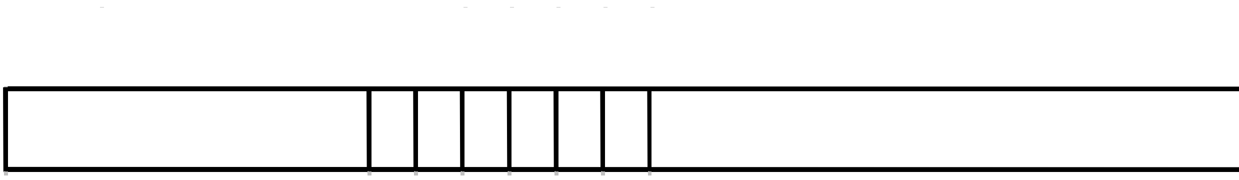
Generate Relocatable Programs

- Let the assembled program starts at address 0 so that later it can be easily moved to any place in the physical memory.
  - Actually, as we have learned from virtual memory, now every process (executed program) has a separate address space starting from 0.
- Assembling register-to-register instructions presents no problems. (e.g., line 125 and 150)
  - Register mnemonic names need to be converted to their corresponding register numbers.
  - This can be easily done by looking up a name table.

PC or Base-Relative Modes

- Format 3: 12-bit displacement field (in total 3 bytes)
  - Base-relative: 0~4095
  - PC-relative: -2048~2047
- Format 4: 20-bit address field (in total 4 bytes)
- The displacement needs to be calculated so that when the displacement is added to PC (which points to the following instruction after the current instruction is fetched) or the base register (B), the resulting value is the target address.
- If the displacement cannot fit into 12 bits, format 4 then needs to be used. (E.g., line 15 and 125)
  - Bit e needs to be set to indicate format 4.
  - A programmer must specify the use of format 4 by putting a + before the instruction. Otherwise, it will be treated as an error.

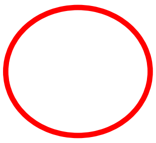


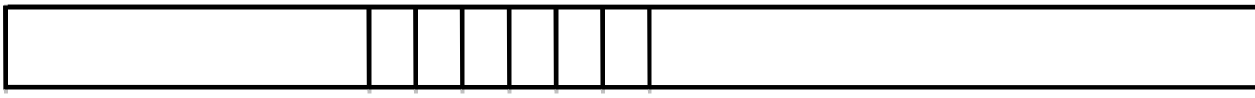


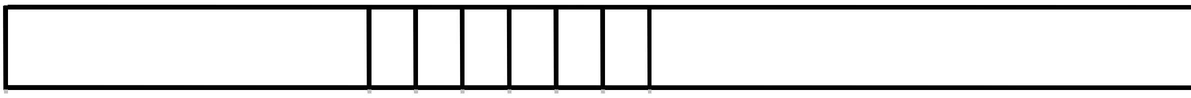
Base-Relative v.s. PC-Relative

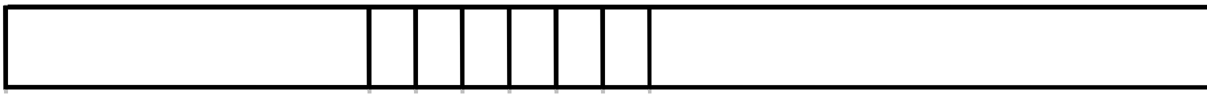
- The difference between PC and base relative addressing modes is that the assembler knows the value of PC when it tries to use PC-relative mode to assembles an instruction. However, when trying to use base-relative mode to assemble an instruction, the assembler does not know the value of the base register.

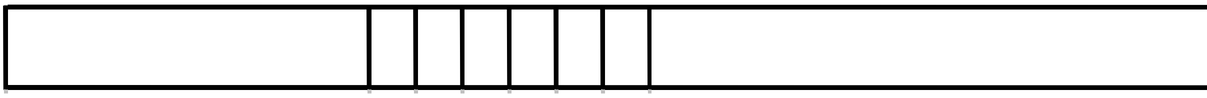
- Therefore, the programmer must tell the assembler the value of register B.
- This is done through the use of the BASE directive. (line 13)
- Also, the programmer must load the appropriate value into register B by himself.
- Another BASE directive can appear later, this will tell the assembler to change its notion of the current value of B.
- NOBASE can also be used to tell the assembler that no more base-relative addressing mode should be used.









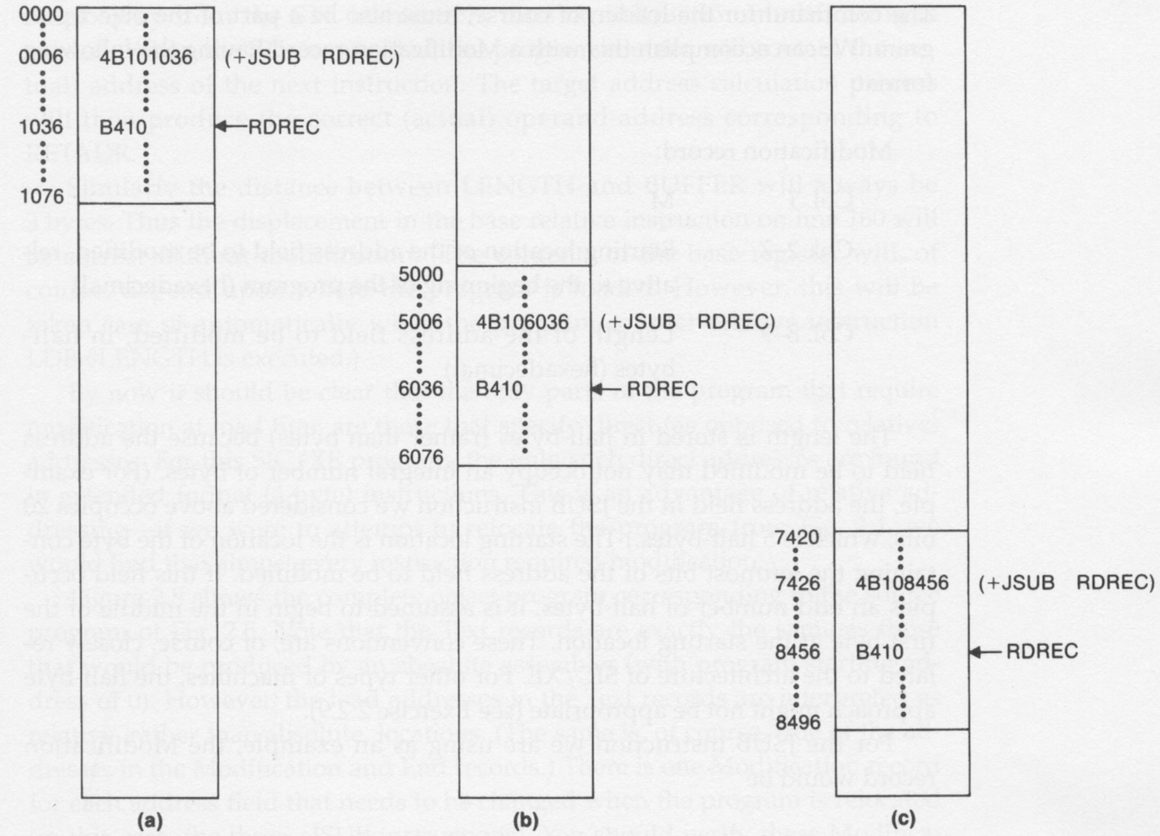


Relocatable Is Desired

- The program in Fig. 2.1 specifies that it must be loaded at address 1000 for correct execution. This restriction is too inflexible for the loader.
- If the program is loaded at a different address, say 2000, its memory references will access wrong data! For example:
    - 55 101B LDA THREE 00102D
- Thus, we want to make programs relocatable so that they can be loaded and execute correctly at any place in the memory.

Address Modification Is Required

If we can use a hardware relocation register (MMU), software relocation can be avoided here. However, when linking multiple object Programs together, software relocation is still needed.

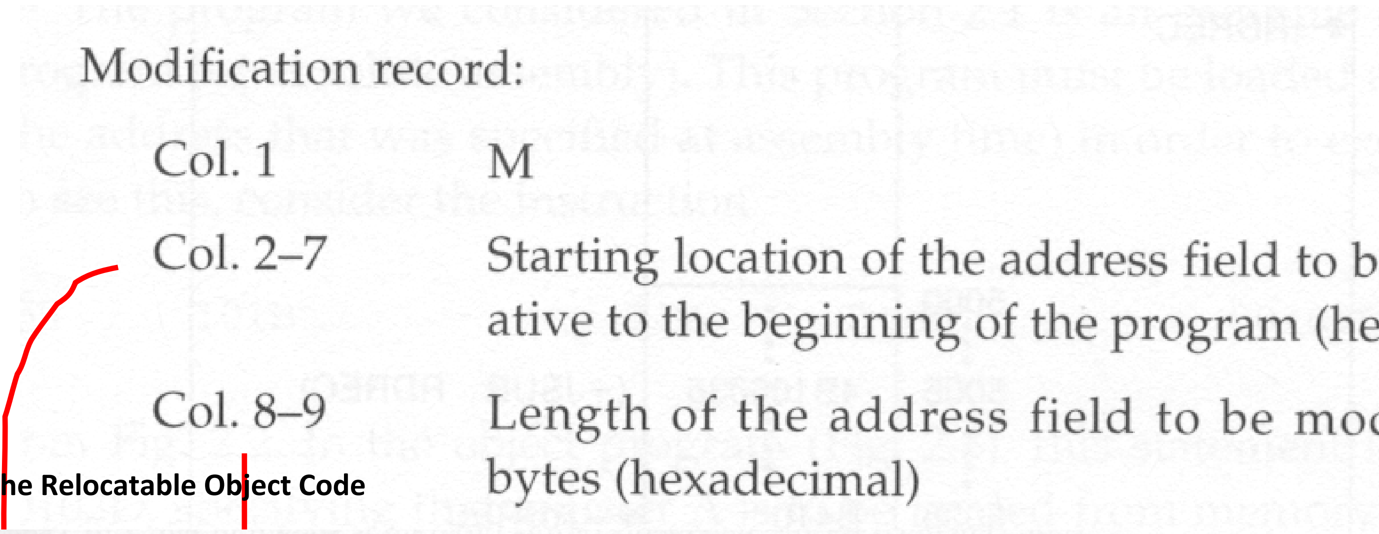


What Instructions Needs to be Modified?

- Only those instructions that use absolute (direct) addresses to reference symbols.
- The following need not be modified:
    - Immediate addressing (no memory references)
    - PC or Base-relative addressing (Relocatable is one advantage of relative addressing, among others.)
    - Register-to-register instructions (no memory references)

The Modification Record

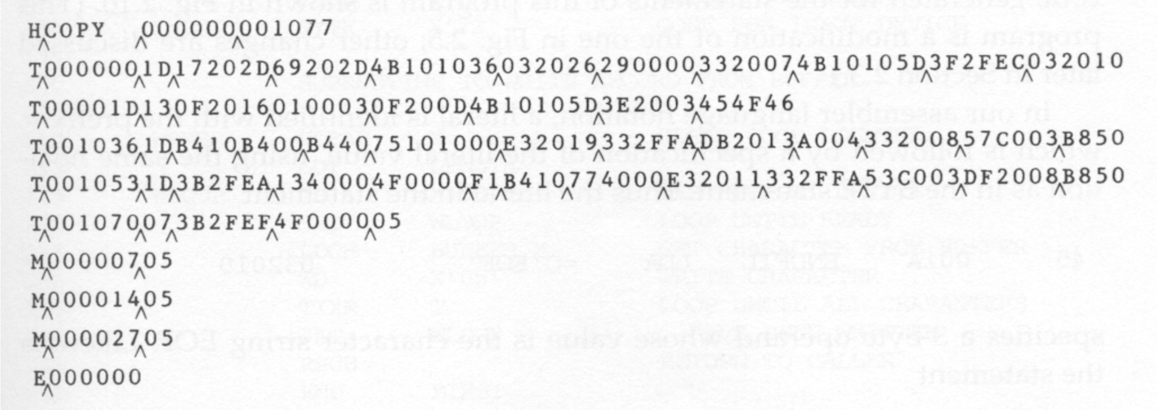
- When the assembler generate an address for a symbol, the address to be inserted into the instruction is relative to the start of the program.
- The assembler also produces a modification record, in which the address and length of the need-to-be-modified address field are stored.
- The loader, when seeing the record, will then add the beginning address of the loaded program to the address field stored in the record.



Modification record:

Col. 1	M
Col. 2–7	Starting location of the address field to b ative to the beginning of the program (he
Col. 8–9	Length of the address field to be mod bytes (hexadecimal)

The Relocatable Object Code



```
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^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

MODULE-2

- **Loaders and Linkers: Basic Loader Functions,**
- **Machine Dependent Loader**
- **Features, Machine Independent Loader Features,**
- **Loader Design Options,**
- **Implementation Examples.**

Machine Independent Assembler Features

These are the features which do not depend on the architecture of the machine. These are:

- Literals
- Expressions
- Program blocks
- Control sections

Literals

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

45 001A ENDFIL LDA =C‟EOF‟ 032010

\-

\-

93 002D * LTORG =C‟EOF‟ 454F46

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010).

As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

215 1062 WLOOP TD =X‟05‟ E32011

It is important to understand the difference between a constant defined as a literal and a constant defined as an immediate operand. In case of literals the assembler generates the specified value as a constant at some other memory location. In immediate mode the operand value is assembled as part of the instruction itself. Example

55 0020 LDA #03 010003

All the literal operands used in a program are gathered together into one or more literal pools. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

A literal table is created for the literals which are used in the program. The literal table contains the literal name, operand value and length. The literal table is usually created as a hash table on the literal name.

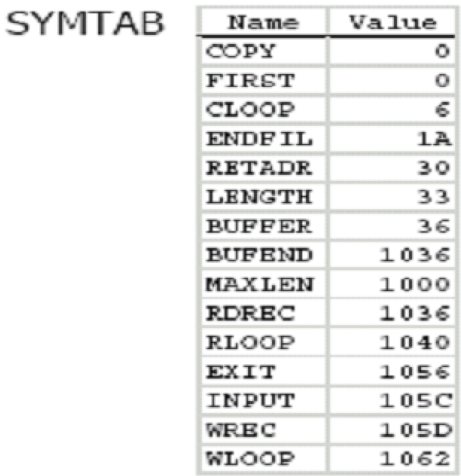
Implementation of Literals:

During Pass-1:

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value, it waits till it encounters LTORG for literal definition. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

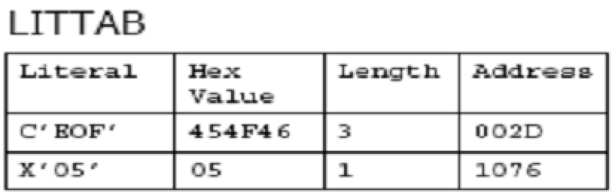
During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITTAB.



SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062



LITTAB

Literal	Hex Value	Length	Address
C'EOF'	454F46	3	002D
X'05'	05	1	1076

Symbol-Defining Statements:

EQU Statement:

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this EQU (Equate). The general form of the statement is

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any othersymbol which is already defined. One common usage is to define symbolic names that can be used to improve readability in place of numeric values.

For example

```
+LDT    #4096
```

This loads the register T with immediate value 4096, this does not clearly show what exactly this value indicates. If a statement is included as:

```
MAXLEN  EQU    4096                                      and then

             +LDT    #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

ORG Statement:

This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin). Its general format is:

```
ORG    value
```

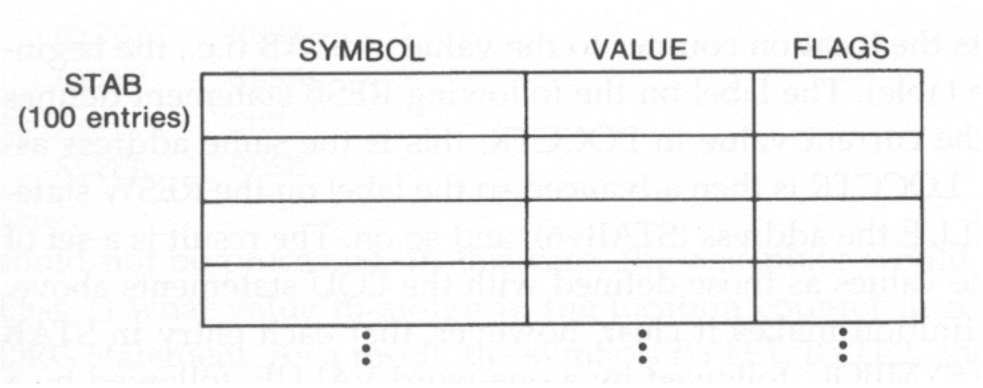
where value is a constant or an expression involving constants and previously defined symbols.

When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program.Sometimes altering the values may result in incorrect assembly.

ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

SYMBOL	6 Bytes
VALUE	3 Bytes
FLAG	2 Bytes

The table looks like the one given below.



The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the table can be reserved by the statement:

```
STAB        RESB        1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

```
SYMBOL      EQU         STAB

VALUE       EQU         STAB+6

FLAGS       EQU         STAB+9
```

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
            LDA         VALUE, X
```

The same thing can also be done using ORG statement in the following way:

```
STAB        RESB        1100

            ORG         STAB

SYMBOL      RESB         6

VALUE       RESW         1

FLAG        RESB        2

            ORG         STAB+1100
```

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. For example for the sequence of statements below:

```
            ORG         ALPHA
```

```
BYTE1        RESB        1

BYTE2        RESB         1

BYTE3        RESB        1

             ORG

ALPHA        RESB         1
```

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

EXPRESSIONS:

Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /. Division is usually defined to produce an integer result. Individual terms may be constants, user-defined symbols, or special terms. The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement

```
 BUFFEND  EQU    *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either absolute expression or relative expressions depending on the type of value they produce.

Absolute Expressions:

The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. Example:

```
 MAXLEN   EQU    BUFEND-BUFFER
```

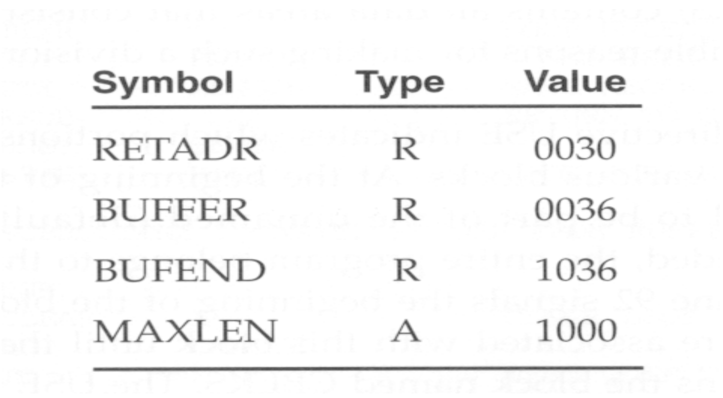
In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute immaterial o the relocation of the program. The expression can have only absolute terms. Example:

```
 MAXLEN   EQU    1000
```

Relative Expressions: All the relative terms except one can be paired as described in “absolute”. The remaining unpaired relative term must have a positive sign. Example:

```
STAB    EQU    OPTAB + (BUFEND – BUFFER)
```

Handling the type of expressions: to find the type of expression, we must keep track the type of symbols used. This can be achieved by defining the type in the symbol table against each of the symbol as shown in the table below:



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

Program Blocks:

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

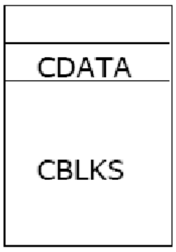
Assembler Directive USE:

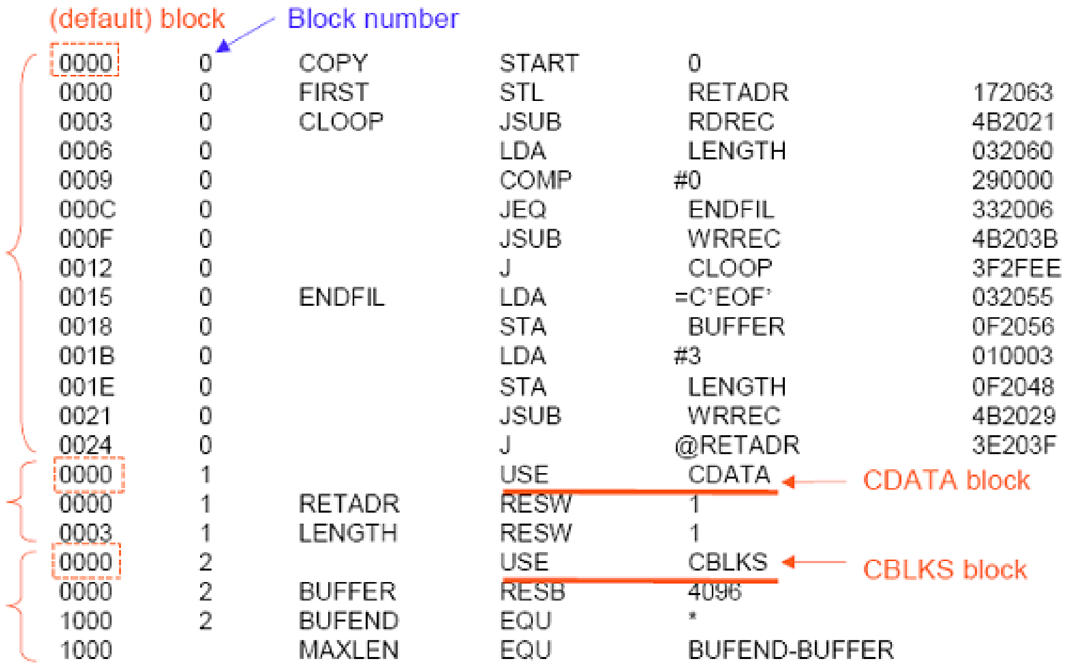
USE [blockname]

At the beginning, 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. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. Program readability is better if data areas are placed in the source program close to the statements that reference them.

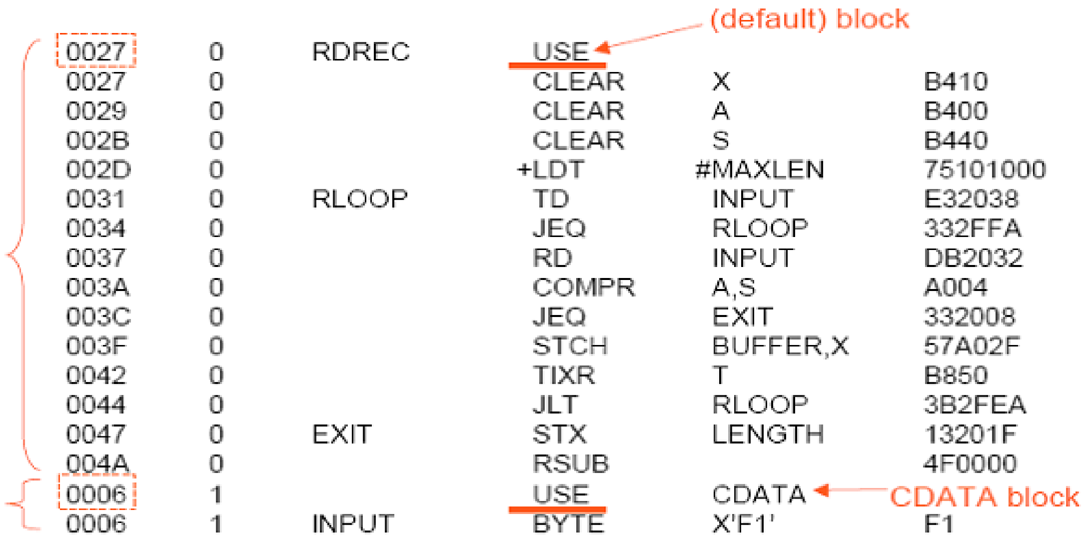
In the example below three blocks are used :

- Default: executable instructions
- CDATA: all data areas that are less in length
- CBLKS: all data areas that consists of larger blocks of memory

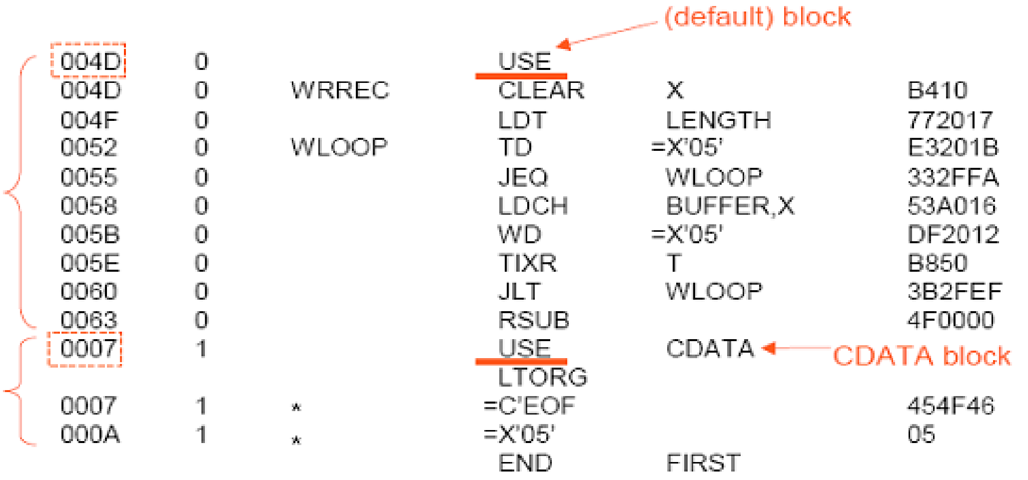




```
0000   0   COPY     START   0
0000   0   FIRST    STL     RETADR        172063
0003   0   CLOOP    JSUB    RDREC         4B2021
0006   0            LDA     LENGTH        032060
0009   0            COMP    #0            290000
000C   0            JEQ     ENDFIL        332006
000F   0            JSUB    WRREC         4B203B
0012   0            J       CLOOP         3F2FEE
0015   0   ENDFIL   LDA     =C'EOF'       032055
0018   0            STA     BUFFER        0F2056
001B   0            LDA     #3            010003
001E   0            STA     LENGTH        0F2048
0021   0            JSUB    WRREC         4B2029
0024   0            J       @RETADR       3E203F
0000   1            USE     CDATA
0000   1   RETADR   RESW    1
0003   1   LENGTH   RESW    1
0000   2            USE     CBLKS
0000   2   BUFFER   RESB    4096
1000   2   BUFEND   EQU     *
1000       MAXLEN   EQU     BUFEND-BUFFER
```



```
0027   0   RDREC    USE
0027   0            CLEAR   X             B410
0029   0            CLEAR   A             B400
002B   0            CLEAR   S             B440
002D   0            +LDT    #MAXLEN       75101000
0031   0   RLOOP    TD      INPUT         E32038
0034   0            JEQ     RLOOP         332FFA
0037   0            RD      INPUT         DB2032
003A   0            COMPR   A,S           A004
003C   0            JEQ     EXIT          332008
003F   0            STCH    BUFFER,X      57A02F
0042   0            TIXR    T             B850
0044   0            JLT     RLOOP         3B2FEA
0047   0   EXIT     STX     LENGTH        13201F
004A   0            RSUB                  4F0000
0006   1            USE     CDATA
0006   1   INPUT    BYTE    X'F1'         F1
```



```
004D   0            USE
004D   0   WRREC    CLEAR   X             B410
004F   0            LDT     LENGTH        772017
0052   0   WLOOP    TD      =X'05'        E3201B
0055   0            JEQ     WLOOP         332FFA
0058   0            LDCH    BUFFER,X      53A016
005B   0            WD      =X'05'        DF2012
005E   0            TIXR    T             B850
0060   0            JLT     WLOOP         3B2FEF
0063   0            RSUB                  4F0000
0007   1            USE     CDATA
                    LTORG
0007   1   *        =C'EOF'               454F46
000A   1   *        =X'05'                05
                    END     FIRST
```

Arranging code into program blocks:

Pass 1

A separate location counter for each program block is maintained.

Save and restore LOCCTR when switching between blocks.

At the beginning of a block, LOCCTR is set to 0.

Assign each label an address relative to the start of the block.

Store the block name or number in the SYMTAB along with the assigned relative address of the label

Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1

Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

Calculate the address for each symbol relative to the start of the object program by adding The location of the symbol relative to the start of its block

The starting address of this block

Control Sections:

A 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 others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called external references.

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive assembler directive: CSECT

The syntax :

secname CSECT

separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

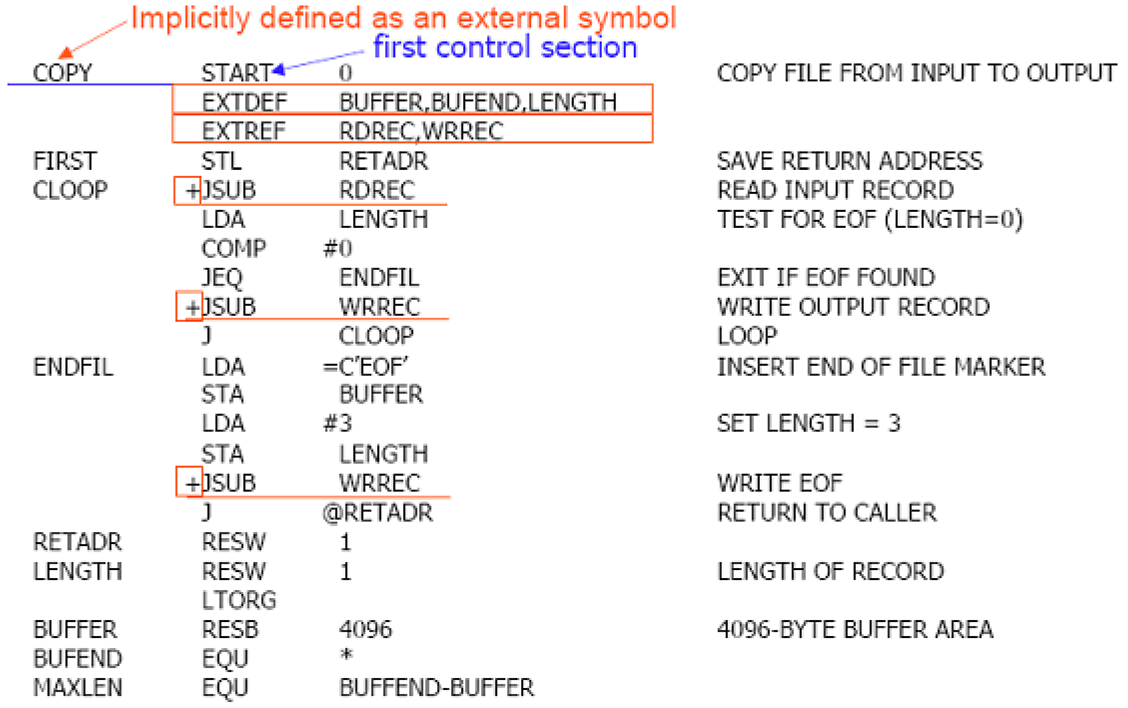
EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

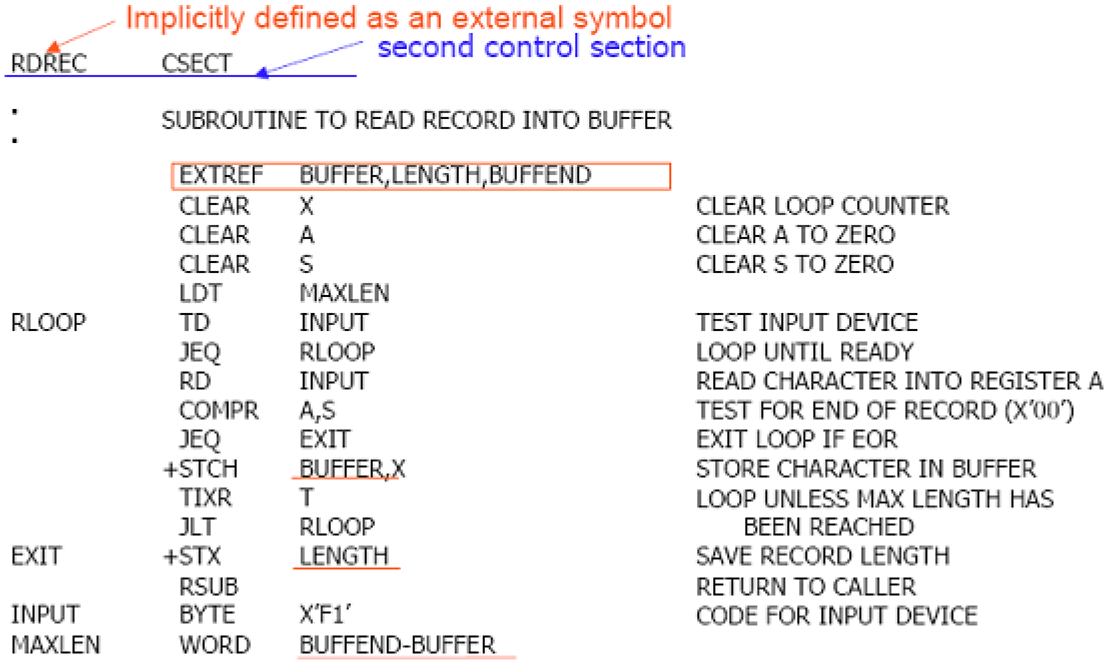
EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

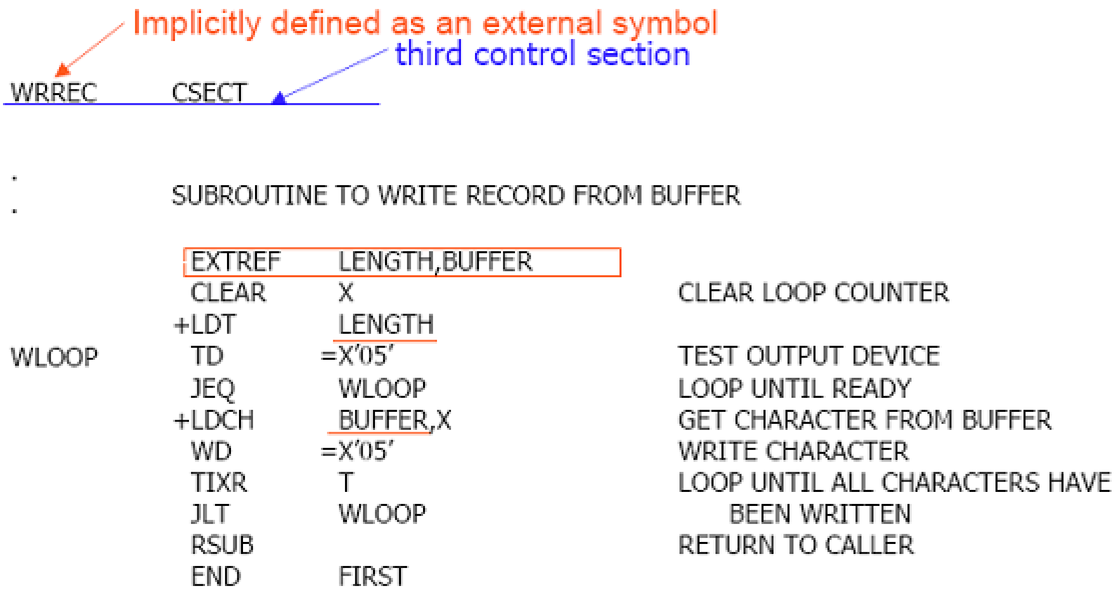
The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.



```
COPY      START     0               COPY FILE FROM INPUT TO OUTPUT
          EXTDEF    BUFFER,BUFEND,LENGTH
          EXTREF    RDREC,WRREC
FIRST     STL       RETADR          SAVE RETURN ADDRESS
CLOOP     +JSUB     RDREC           READ INPUT RECORD
          LDA       LENGTH          TEST FOR EOF (LENGTH=0)
          COMP     #0
          JEQ       ENDFIL          EXIT IF EOF FOUND
          +JSUB     WRREC           WRITE OUTPUT RECORD
          J         CLOOP           LOOP
ENDFIL    LDA      =C'EOF'          INSERT END OF FILE MARKER
          STA       BUFFER
          LDA      #3               SET LENGTH = 3
          STA       LENGTH
          +JSUB     WRREC           WRITE EOF
          J        @RETADR          RETURN TO CALLER
RETADR    RESW      1
LENGTH    RESW      1               LENGTH OF RECORD
          LTORG
BUFFER    RESB      4096            4096-BYTE BUFFER AREA
BUFEND    EQU       *
MAXLEN    EQU       BUFFEND-BUFFER
```



```
RDREC     CSECT
.
.         SUBROUTINE TO READ RECORD INTO BUFFER
.
          EXTREF    BUFFER,LENGTH,BUFFEND
          CLEAR     X               CLEAR LOOP COUNTER
          CLEAR     A               CLEAR A TO ZERO
          CLEAR     S               CLEAR S TO ZERO
          LDT       MAXLEN
RLOOP     TD        INPUT           TEST INPUT DEVICE
          JEQ       RLOOP           LOOP UNTIL READY
          RD        INPUT           READ CHARACTER INTO REGISTER A
          COMPR     A,S             TEST FOR END OF RECORD (X'00')
          JEQ       EXIT            EXIT LOOP IF EOR
         +STCH      BUFFER,X        STORE CHARACTER IN BUFFER
          TIXR      T               LOOP UNLESS MAX LENGTH HAS
          JLT       RLOOP             BEEN REACHED
EXIT     +STX       LENGTH          SAVE RECORD LENGTH
          RSUB                      RETURN TO CALLER
INPUT     BYTE      X'F1'           CODE FOR INPUT DEVICE
MAXLEN    WORD      BUFFEND-BUFFER
```

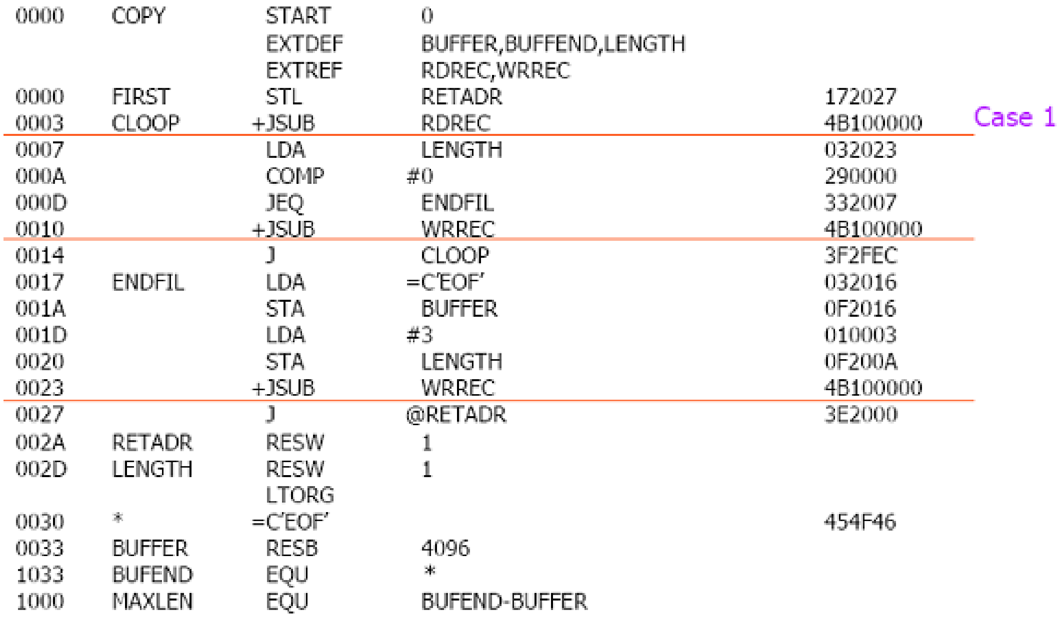


Implicitly defined as an external symbol

third control section

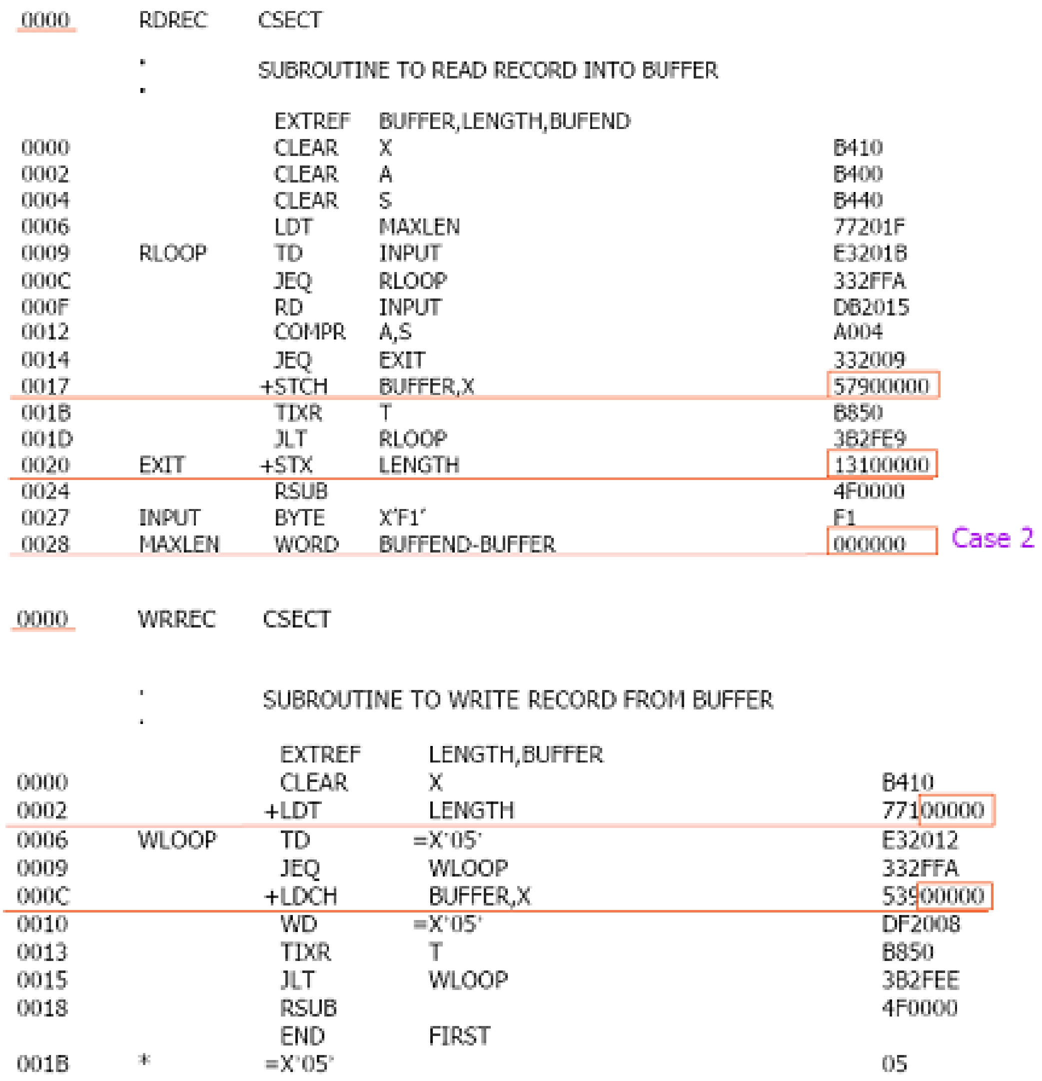
```
WRREC      CSECT

.
.          SUBROUTINE TO WRITE RECORD FROM BUFFER

           EXTREF   LENGTH,BUFFER
           CLEAR    X                   CLEAR LOOP COUNTER
          +LDT      LENGTH
WLOOP      TD      =X'05'               TEST OUTPUT DEVICE
           JEQ      WLOOP               LOOP UNTIL READY
          +LDCH     BUFFER,X            GET CHARACTER FROM BUFFER
           WD      =X'05'               WRITE CHARACTER
           TIXR     T                   LOOP UNTIL ALL CHARACTERS HAVE
           JLT      WLOOP                   BEEN WRITTEN
           RSUB                         RETURN TO CALLER
           END      FIRST
```

Object Code for the example program:



Case 1

```
0000   COPY     START    0
                EXTDEF   BUFFER,BUFFEND,LENGTH
                EXTREF   RDREC,WRREC
0000   FIRST    STL      RETADR                  172027
0003   CLOOP   +JSUB     RDREC                   4B100000
0007            LDA      LENGTH                  032023
000A            COMP    #0                       290000
000D            JEQ      ENDFIL                  332007
0010           +JSUB     WRREC                   4B100000
0014            J        CLOOP                   3F2FEC
0017   ENDFIL   LDA     =C'EOF'                  032016
001A            STA      BUFFER                  0F2016
001D            LDA     #3                       010003
0020            STA      LENGTH                  0F200A
0023           +JSUB     WRREC                   4B100000
0027            J       @RETADR                  3E2000
002A   RETADR   RESW     1
002D   LENGTH   RESW     1
                LTORG
0030   *       =C'EOF'                           454F46
0033   BUFFER   RESB     4096
1033   BUFEND   EQU      *
1000   MAXLEN   EQU      BUFEND-BUFFER
```



```
0000    RDREC    CSECT
         .
         .       SUBROUTINE TO READ RECORD INTO BUFFER
                  EXTREF  BUFFER,LENGTH,BUFEND
0000              CLEAR   X                              B410
0002              CLEAR   A                              B400
0004              CLEAR   S                              B440
0006              LDT     MAXLEN                         77201F
0009    RLOOP     TD      INPUT                          E3201B
000C              JEQ     RLOOP                          332FFA
000F              RD      INPUT                          DB2015
0012              COMPR   A,S                            A004
0014              JEQ     EXIT                           332009
0017             +STCH    BUFFER,X                       57900000
001B              TIXR    T                              B850
001D              JLT     RLOOP                          3B2FE9
0020    EXIT     +STX     LENGTH                         13100000
0024              RSUB                                   4F0000
0027    INPUT     BYTE    X'F1'                          F1
0028    MAXLEN    WORD    BUFFEND-BUFFER                 000000   Case 2

0000    WRREC    CSECT

         .
         .       SUBROUTINE TO WRITE RECORD FROM BUFFER

                  EXTREF    LENGTH,BUFFER
0000              CLEAR     X                            B410
0002             +LDT       LENGTH                       77100000
0006    WLOOP     TD       =X'05'                        E32012
0009              JEQ       WLOOP                        332FFA
000C             +LDCH      BUFFER,X                     53900000
0010              WD       =X'05'                        DF2008
0013              TIXR      T                            B850
0015              JLT       WLOOP                        3B2FEE
0018              RSUB                                   4F0000
                  END       FIRST
001B    *        =X'05'                                  05
```

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

Define record (EXTDEF)

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 symbols

Refer record (EXTREF)

Col. 1 R

Col. 2-7 Name of external symbol referred to in this control section

Col. 8-73 Name of other external reference symbols

Modification record

Col. 1 M

Col. 2-7 Starting address of the field to be modified (hexadecimal)

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

Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.

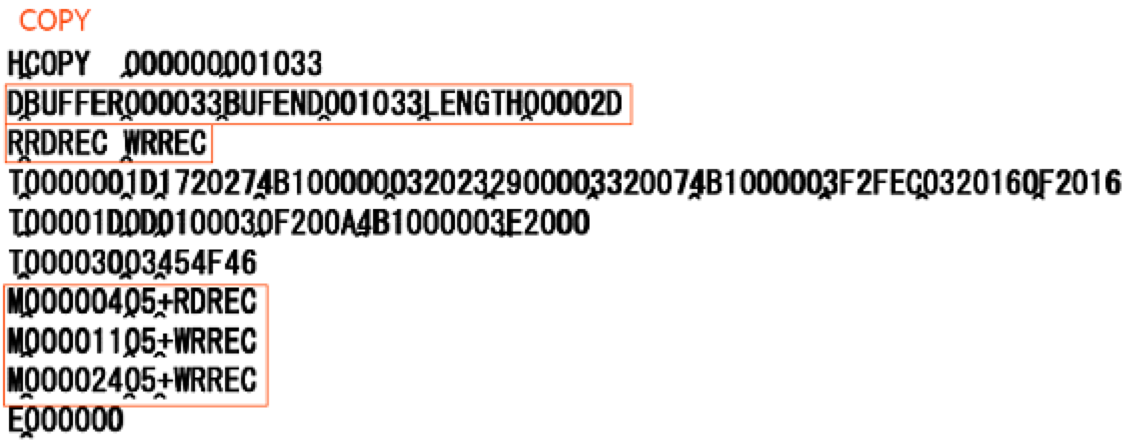
A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification my be defined either in this control section or in another section.

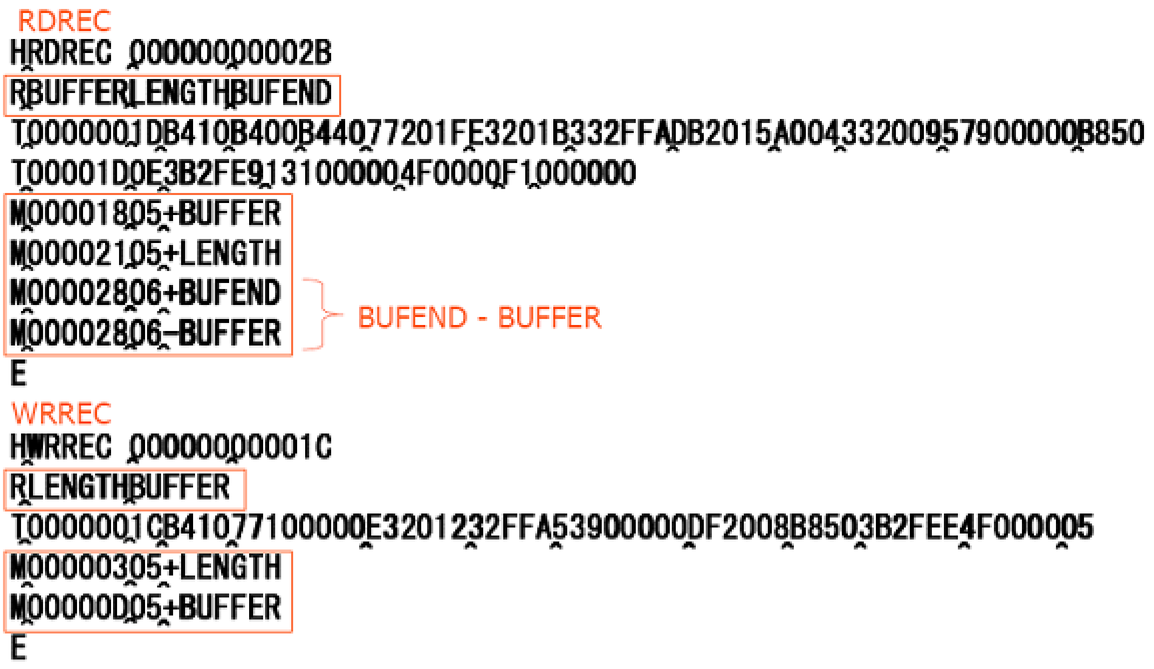
The object program is shown below. There is a separate object program for each of the control sections. In the Define Record and refer record the symbols named in EXTDEF and EXTREF are included.

In the case of Define, the record also indicates the relative address of each external symbol within the control section.

For EXTREF symbols, no address information is available. These symbols are simply named in the Refer record.



```
COPY
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100003^3F2FEC^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
```



```
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

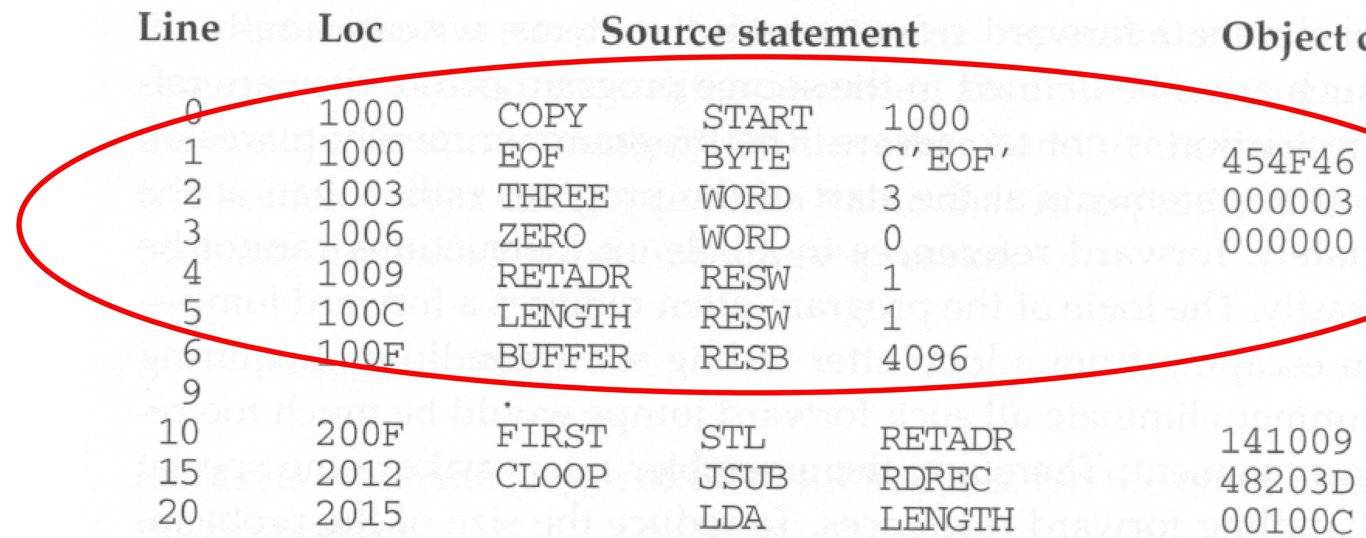
Assembler Design Options

One and Multi-Pass Assembler

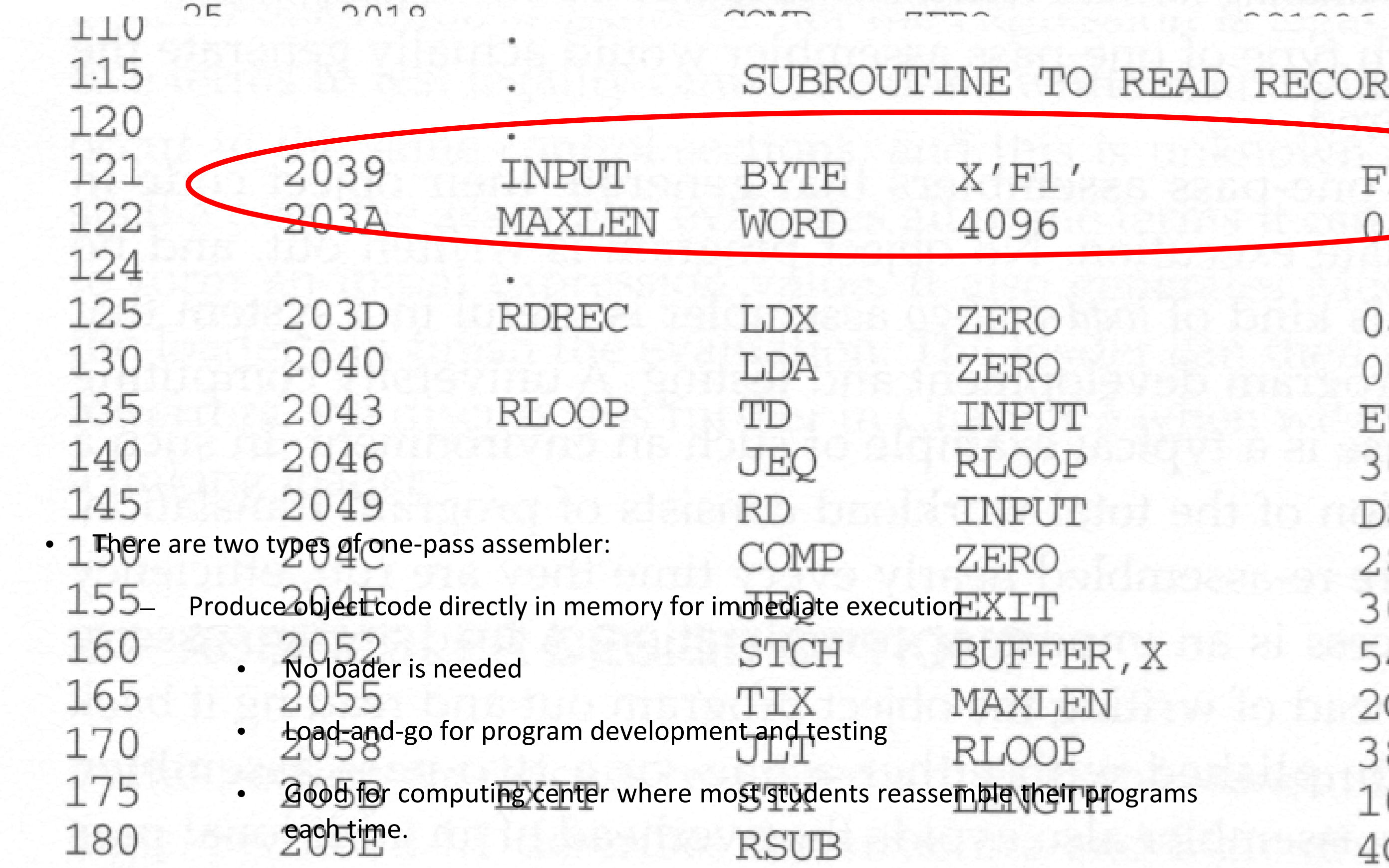
- So far, we have presented the design and implementation of a two-pass assembler.
- Here, we will present the design and implementation of
  - One-pass assembler
    - If avoiding a second pass over the source program is necessary or desirable.
  - Multi-pass assembler
    - Allow forward references during symbol definition.

One-Pass Assembler

- The main problem is about forward reference.
- Eliminating forward reference to data items can be easily done.
  - Simply ask the programmer to define variables before using them.
- However, eliminating forward reference to instruction cannot be easily done.
  - Sometimes your program needs a forward jump.
  - Asking your program to use only backward jumps is too restrictive.



```
Line   Loc    Source statement                  Object c
  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
```

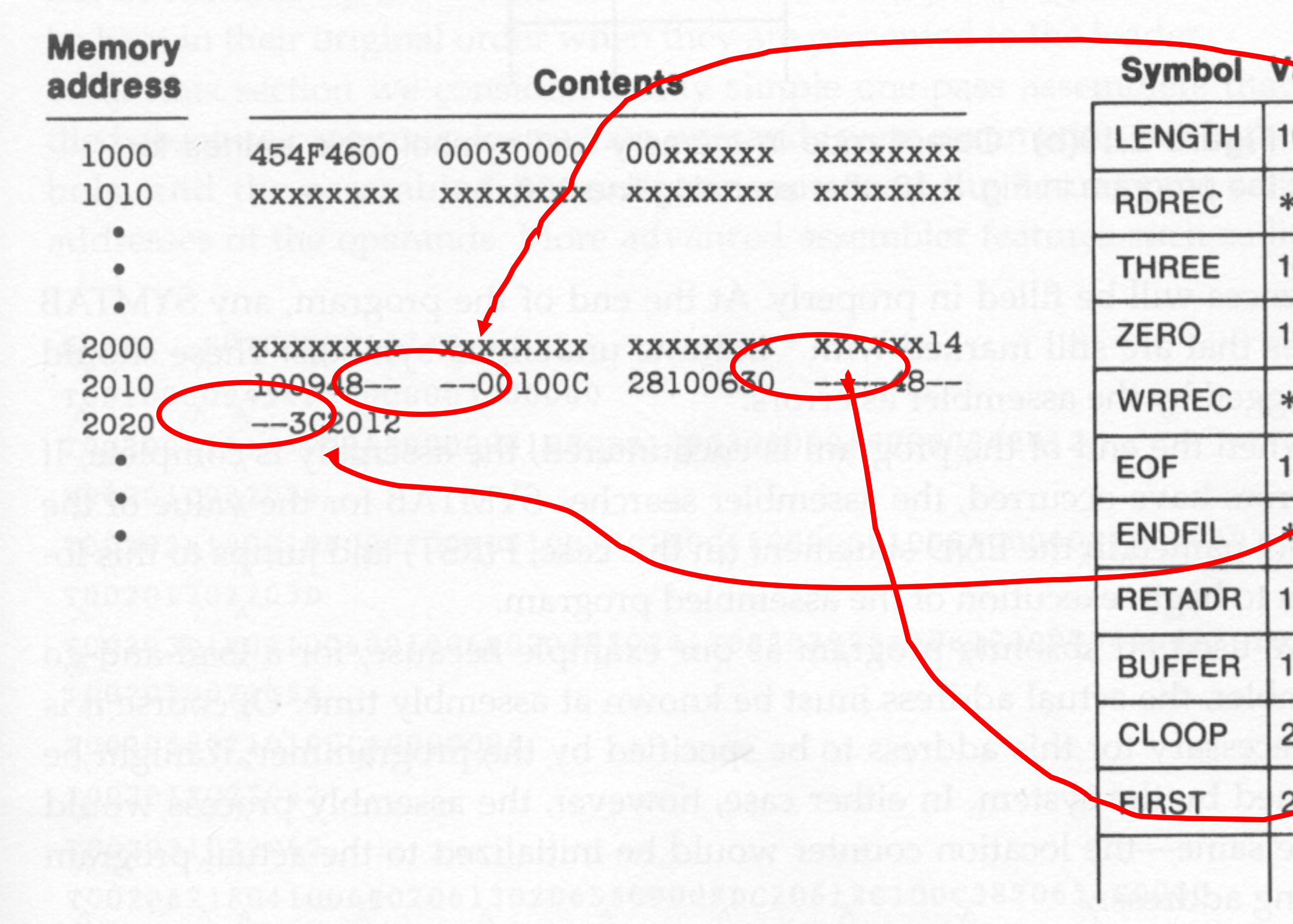


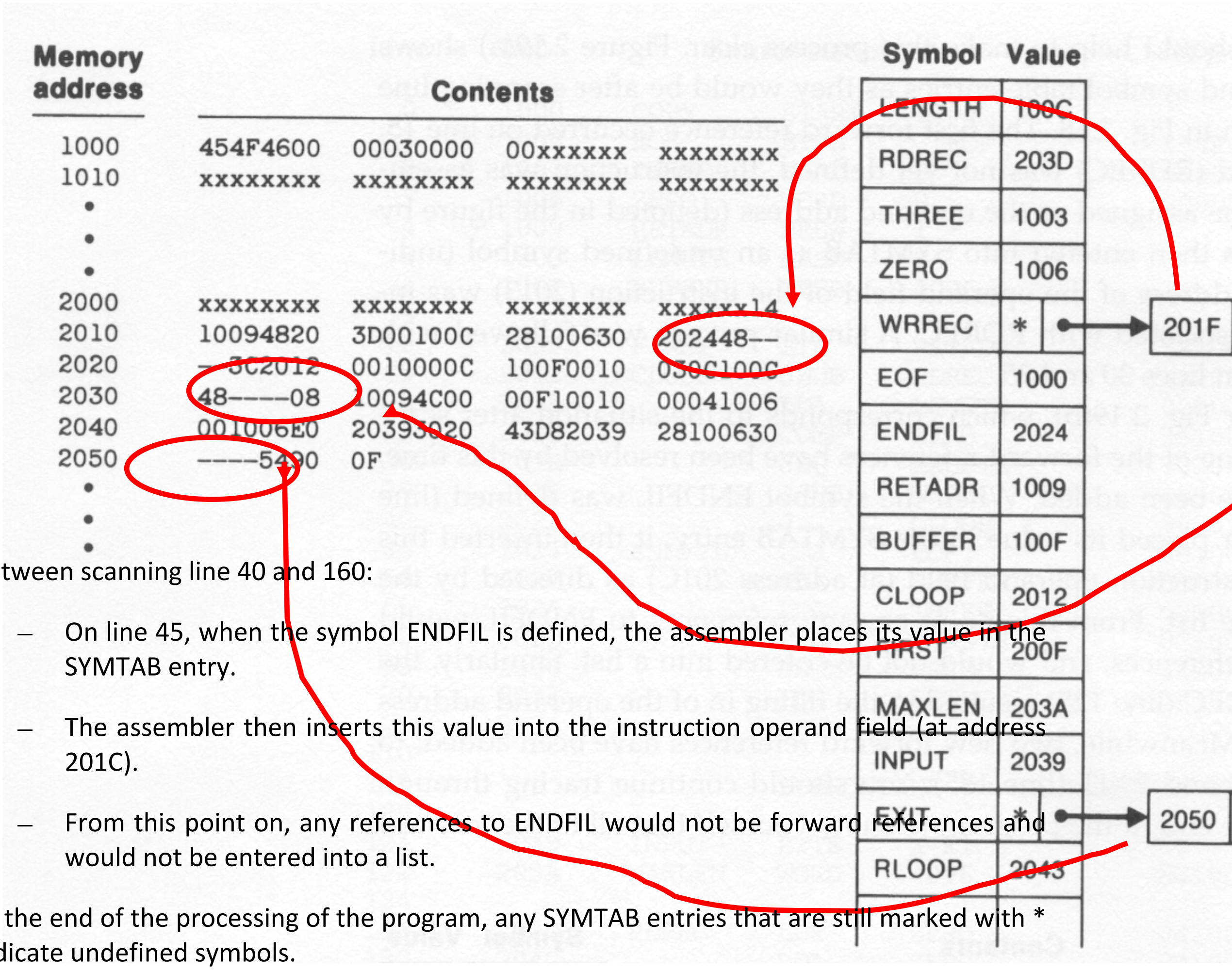
```
110          .
115          .       SUBROUTINE TO READ RECOR
120          .
121   2039   INPUT   BYTE    X'F1'       F
122   203A   MAXLEN  WORD    4096        0
124          .
125   203D   RDREC   LDX     ZERO        0
130   2040           LDA     ZERO        0
135   2043   RLOOP   TD      INPUT       E
140   2046           JEQ     RLOOP       3
145   2049           RD      INPUT       D
150   204C           COMP    ZERO        2
155   204F           JEQ     EXIT        3
160   2052           STCH    BUFFER,X    5
165   2055           TIX     MAXLEN      2
170   2058           JLT     RLOOP       3
175   205B   EXIT    STX     LENGTH      1
180   205E           RSUB                4
```

- There are two types of one-pass assembler:
  - Produce object code directly in memory for immediate execution
    - No loader is needed
    - Load-and-go for program development and testing
    - Good for computing center where most students reassemble their programs each time.
    - Can save time for scanning the source code again
  - Produce the usual kind of object program for later execution

Internal Implementation

- The assembler generate object code instructions as it scans the source program.
- If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled.
- The symbol used as an operand is entered into the symbol table.
- This entry is flagged to indicate that the symbol is undefined yet.
- 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.
- When the definition of the symbol is encountered, the forward reference list for that symbol is scanned, and the proper address is inserted into any instruction previously generated.





- Between scanning line 40 and 160:
  - On line 45, when the symbol ENDFIL is defined, the assembler places its value in the SYMTAB entry.
  - The assembler then inserts this value into the instruction operand field (at address 201C).
  - From this point on, any references to ENDFIL would not be forward references and would not be entered into a list.
- At the end of the processing of the program, any SYMTAB entries that are still marked with * indicate undefined symbols.
  - These should be flagged by the assembler as errors.

Multi-Pass Assembler

- If we use a two-pass assembler, the following symbol definition cannot be allowed.

```
ALPHA  EQU    BETA
BETA   EQU    DELTA
DELTA  RESW   1
```

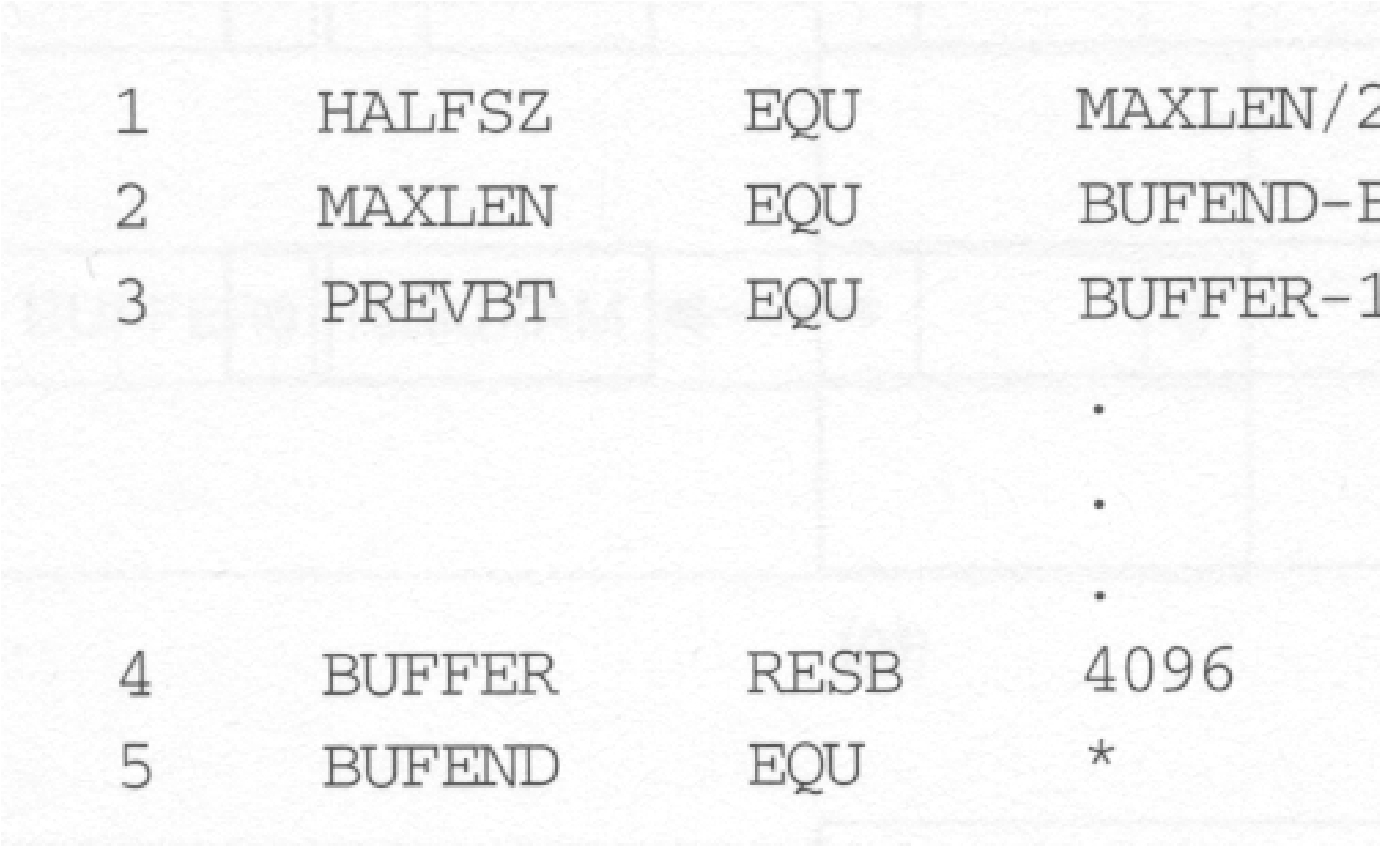
- This is because ALPHA and BETA cannot be defined in pass 1. Actually, if we allow multi-pass processing, DELTA is defined in pass 1, BETA is defined in pass 2, and ALPHA is defined in pass 3, and the above definitions can be allowed.
- This is the motivation for using a multi-pass assembler.

- It is unnecessary for a multi-pass assembler to make more than two passes over the entire program.
- Instead, only the parts of the program involving forward references need to be processed in multiple passes.
- The method presented here can be used to process any kind of forward references.

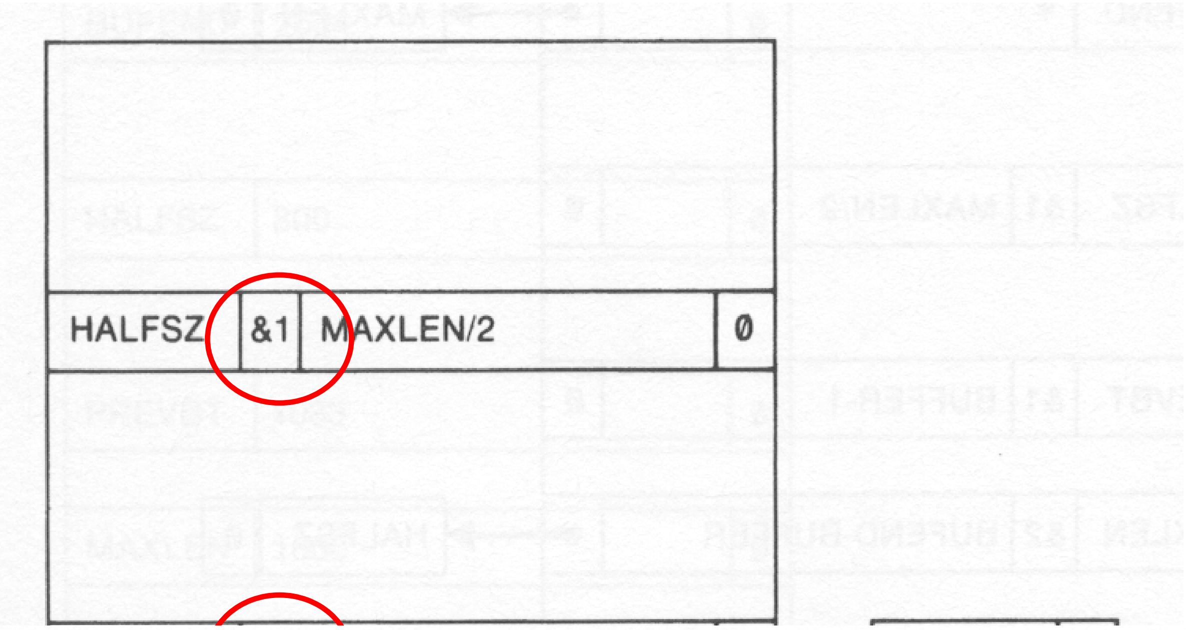
Multi-Pass Assembler Implementation

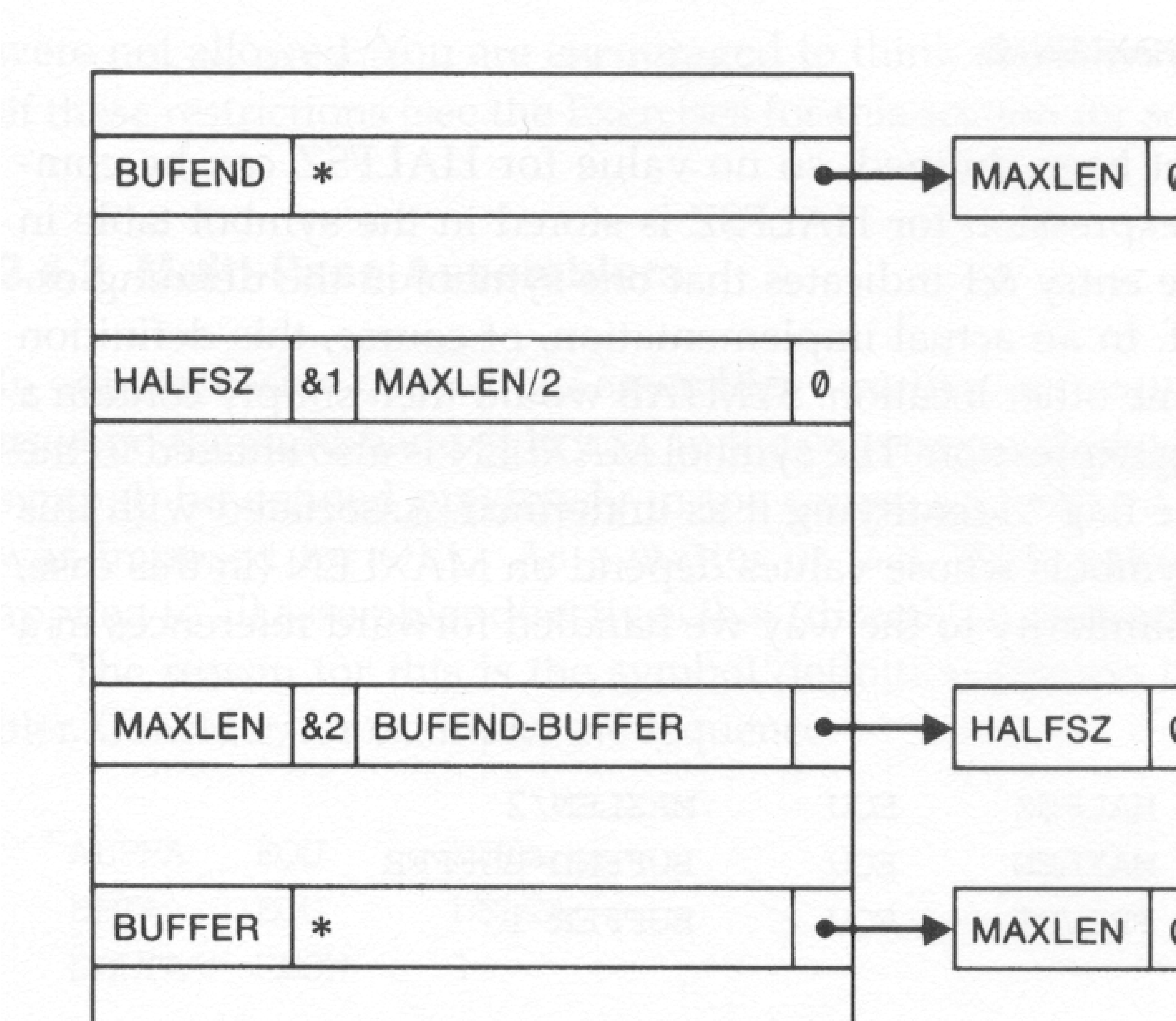
Steps:

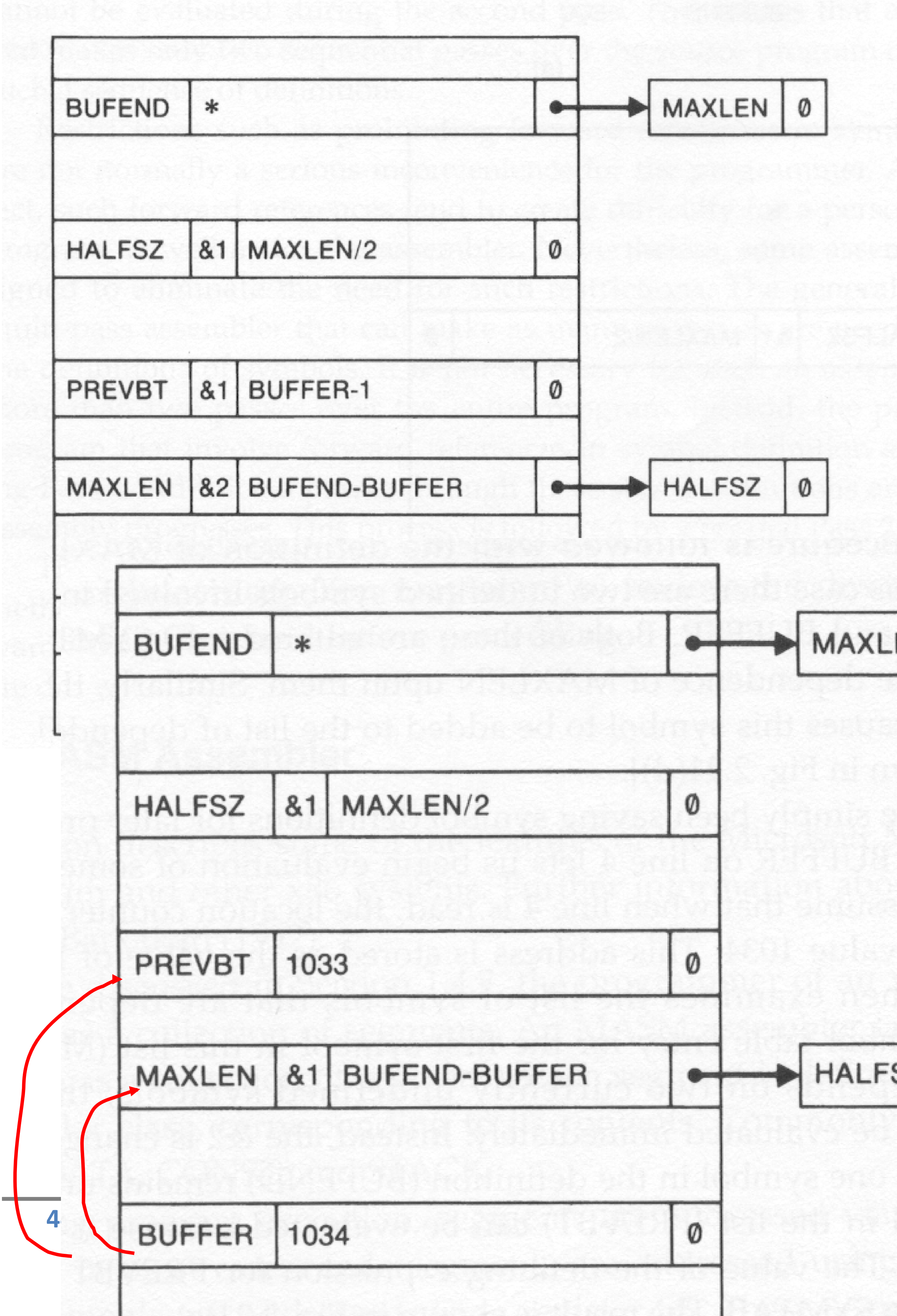
- Use a symbol table to store symbols that are not totally defined yet.
- For a undefined symbol, in its entry,
  - We store the names and the number of undefined symbols which contribute to the calculation of its value.
  - We also keep a list of symbols whose values depend on the defined value of this symbol.
- When a symbol becomes defined, we use its value to reevaluate the values of all of the symbols that are kept in this list.
- The above step is performed recursively.

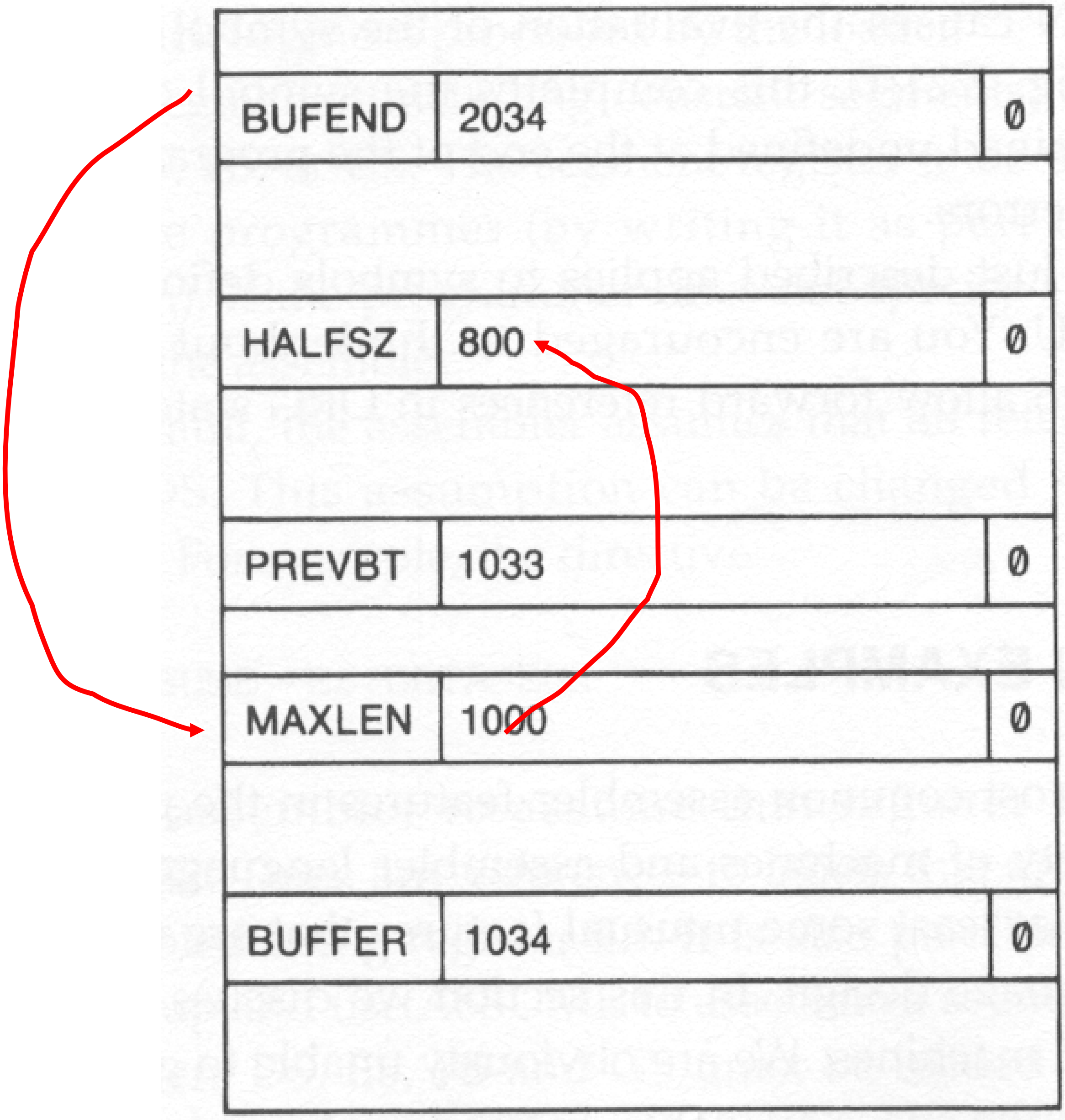


```
1    HALFSZ    EQU    MAXLEN/2
2    MAXLEN    EQU    BUFEND-B
3    PREVBT    EQU    BUFFER-1
                      .
                      .
                      .
4    BUFFER    RESB   4096
5    BUFEND    EQU    *
```


HALFSZ
&1
MAXLEN/2
0


BUFEND
*
MAXLEN
HALFSZ
&1
MAXLEN/2
0
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
BUFFER
*
MAXLEN


BUFEND
*
MAXLEN
0
HALFSZ
&1
MAXLEN/2
0
PREVBT
&1
BUFFER-1
0
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
0
BUFEND
*
MAXL
HALFSZ
&1
MAXLEN/2
0
PREVBT
1033
0
MAXLEN
&1
BUFEND-BUFFER
HALFS
BUFFER
1034
0



BUFEND	2034	0
HALFSZ	800	0
PREVBT	1033	0
MAXLEN	1000	0
BUFFER	1034	0

MODULE-3

Lexical Analysis

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer

Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

- **Attributes for tokens**

  E = M * C ** 2

  <id, pointer to symbol table entry for E>

  <assign-op>

  <id, pointer to symbol table entry for M>

  <mult-op>

  <id, pointer to symbol table entry for C>

  <exp-op>

  <number, integer value 2>

- **Lexical errors**

  Some errors are out of power of lexical analyzer to recognize:
  - **fi (a == f(x)) …**

  However it may be able to recognize errors like:
  - **d = 2r**

  Such errors are recognized when no pattern for tokens matches a character sequence

- **Error recovery**

1. Panic mode: successive characters are ignored until we reach to a well formed token
2. Delete one character from the remaining input
3. Insert a missing character into the remaining input

4. Replace a character by another character
5. Transpose two adjacent characters

- **Input buffering**

Sentinels

Specification of tokens

1. In theory of compilation regular expressions are used to formalize the specification of tokens
2. Regular expressions are means for specifying regular languages
3. Example:

   i. Letter_(letter_ | digit)*

4. Each regular expression is a pattern specifying the form of strings

Regular expressions

1. Ɛ is a regular expression, L(Ɛ) = {Ɛ}
2. If a is a symbol in ∑then a is a regular expression, L(a) = {a}
3. (r) | (s) is a regular expression denoting the language L(r)   L(s)
4. (r)(s) is a regular expression denoting the language L(r)L(s)
5. (r)* is a regular expression denoting (L(r))*

6. (r) is a regular expression denoting L(r)

➢ Regular definitions

1. d1 -> r1
2. d2 -> r2
3. …
4. dn -> rn

5. Example:

6. letter_ -> A | B | … | Z | a | b | … | Z | _
7. digit -> 0 | 1 | … | 9
8. id -> letter_ (letter_ | digit)*

➢ Extensions

One or more instances: (r)+

Zero of one instances: r?

Character classes: [abc]

Example:

letter_ -> [A-Za-z_]

digit -> [0-9]

id -> letter_(letter|digit)*

➢ Recognition of tokens

Starting point is the language grammar to understand the tokens:

stmt -> if expr then stmt
| if expr then stmt else stmt
| Ɛ
expr -> term relop term
| term
term -> id
| number

➢ Recognition of tokens (cont.)

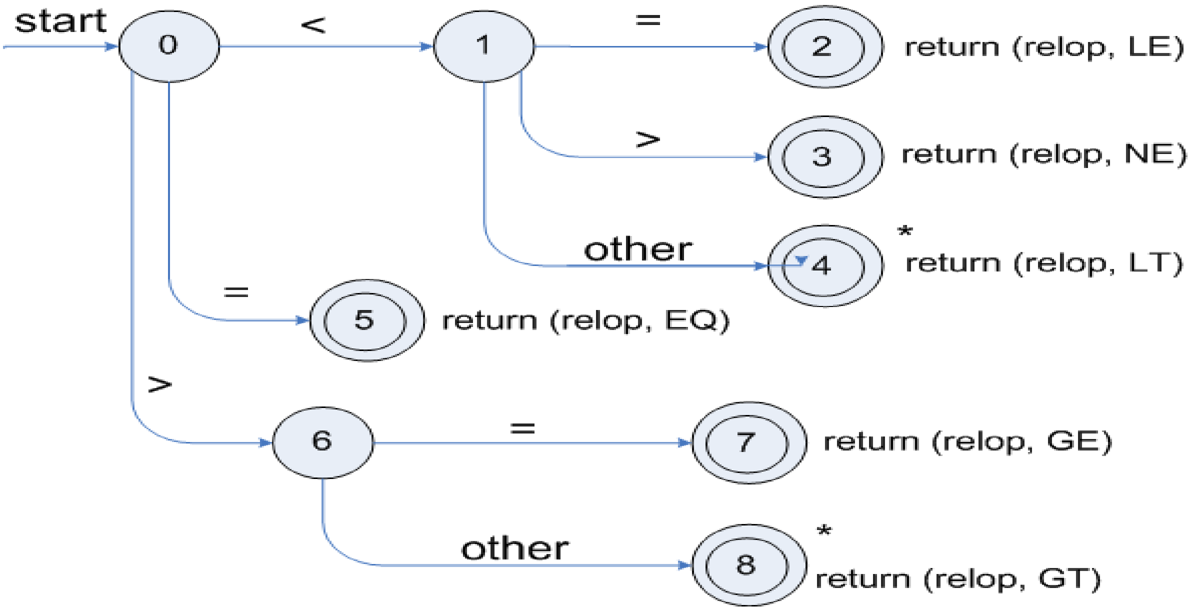
The next step is to formalize the patterns:

digit **-> [0-9]**
Digits **-> digit+**
number **-> digit(.digits)? (E[+-]? Digit)?**
letter **-> [A-Za-z_]**
id **-> letter (letter|digit)***
If **-> if**
Then **-> then**
Else **-> else**
Relop **-> < | > | <= | >= | = | <>**

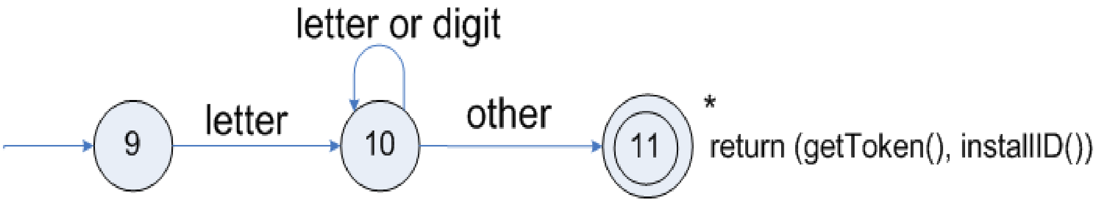
We also need to handle whitespaces:

ws **-> (blank | tab | newline)+**

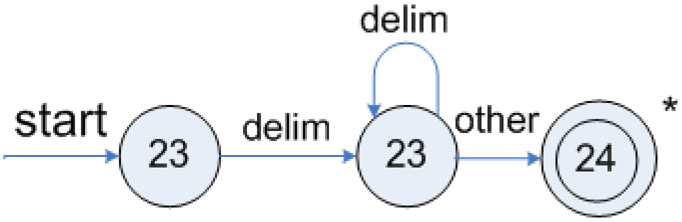
- **Transition diagrams**



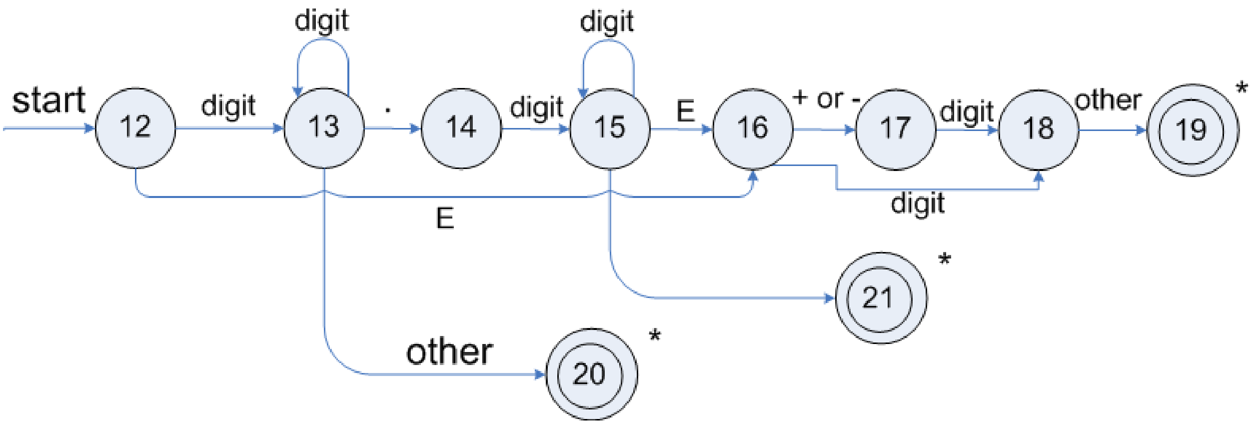
- **Transition diagrams (cont.)**



- **Transition diagram for whitespace**



- **Transition diagram for unsigned numbers**



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
        TOKEN retToken = new (RELOP)
        while (1) {         /* repeat character processing until a
                            return or failure occurs      */
        switch(state) {
          case 0: c= nextchar();
                  if (c == '<') state = 1;
                  else if (c == '=') state = 5;
                  else if (c == '>') state = 6;
                  else fail();    /* lexeme is not a relop */
                  break;
          case 1: …
          …
          case 8: retract();
                  retToken.attribute = GT;
                  return(retToken);
        }
```

- **Finite Automata**
- Regular expressions = specification

- Finite automata = implementation
- A finite automaton consists of
    - An input alphabet
    - A set of states S
    - A start state n
    - A set of accepting states F   S
    - A set of transitions state ^{input} state

- Transition

s_1 ^{a} s_2

- Is read

In state s_1 on input “a” go to state s_2

- If end of input
    - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Example

- Alphabet still { 0, 1 }

The operation of the automaton is not completely defined by the input

On input “11” the automaton could be in either state

MODULE-4

- **Syntax Analysis: Introduction,**
- **Role Of Parsers, Context Free Grammars,**
- **Writing a grammar,**
- **Top Down Parsers,**
- **Bottom-Up Parsers,**
- **Operator-Precedence Parsing**

The role of parser

Uses of grammars

E -> E + T | T

T -> T * F | F

F -> (E) | **id**

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

F -> (E) | **id**

Error handling

- Common programming errors
    - Lexical errors
    - Syntactic errors
    - Semantic errors
    - Logical errors
- Error handler goals
    - Report the presence of errors clearly and accurately
    - Recover from each error quickly enough to detect subsequent errors
    - Add minimal overhead to the processing of correct progrms

Context free grammars

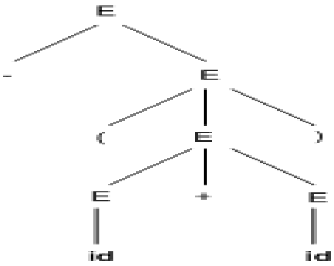
- Terminals
- Nonterminals
- Start symbol
- Productions

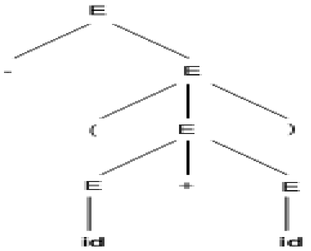
Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
    - E -> E + E | E * E | -E | (E) | **id**
    - Derivations for **–(id+id)**
        - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id+id**)

Parse trees

- **-(id+id)**
        - **E => -E => -(E) => -(E+E) => -(id+E)=>-(id+id)**



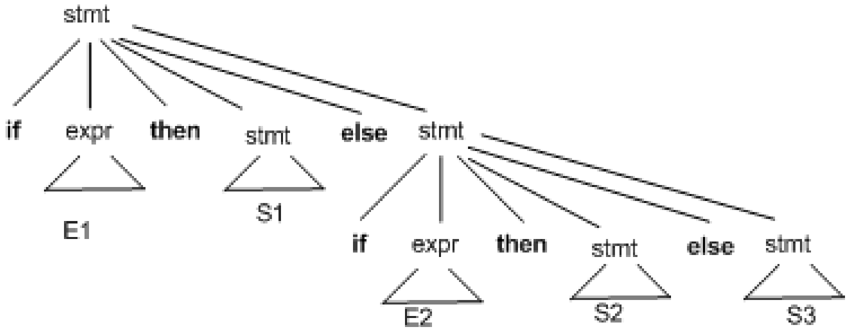


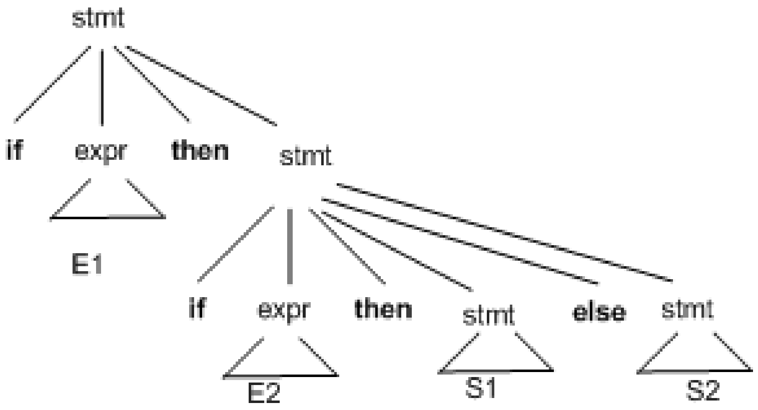
Elimination of ambiguity

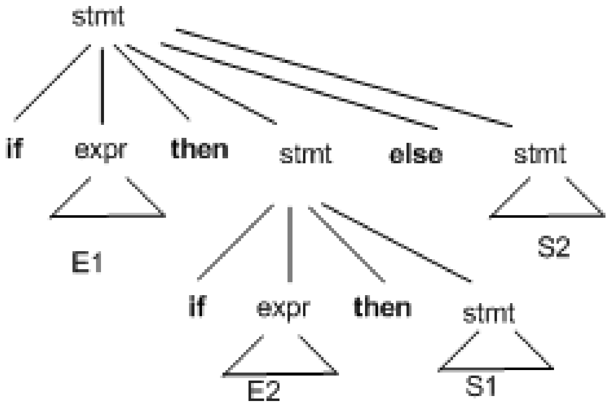
stmt → If expr **then** stmt

| If expr **then** stmt **else** stmt

| **other**







Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation A=> Aα
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:

    - A -> A α|β
  - We may replace it with
    - A -> β A’
    - A’ -> α A’ | ε

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> if expr then stmt else stmt
  - | if expr then stmt
- On seeing input if it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A->αβ1 | αβ2 then we replace it with
    - A -> αA’
    - A’ -> β1 | β2

➢ TOP DOWN PARSING

A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right

It can be also viewed as finding a leftmost derivation for an input string

Example: id+id*id

E -> TE’

E’ -> +TE’ | Ɛ

T -> FT’

T’ -> *FT’ | Ɛ

F -> (E) | **id**

}

Recursive descent parsing

Consists of a set of procedures, one for each nonterminal

Execution begins with the procedure for start symbol

A typical procedure for a non-terminal

```
void A() {
          choose an A-production, A->X1X2..Xk
          for (i=1 to k) {
              if (Xi is a nonterminal
                  call procedure Xi();
              else if (Xi equals the current input symbol a)
                  advance the input to the next symbol;
              else /* an error has occurred */
          }
}
```

Example

S->cAd

A->ab | a

Input: cad

First and Follow

- First() is set of terminals that begins strings derived from
- If α=>ε then is also in First(ε)
    - In predictive parsing when we have A-> α|β, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
    - If we have S => αAaβ for some αand βthen a is in Follow(A)

If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
    1. If X is a terminal then First(X) = {X}.
    2. If X is a nonterminal and X->Y1Y2…Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y1),…,First(Yi-1) that is Y1…Yi-1 => ε. if ε is in First(Yj) for j=1,…,k then add ε to First(X).
    3. If X-> ε is a production then add ε to First(X)
- Example!

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
    1. Place $ in Follow(S) where S is the start symbol

2. If there is a production A-> αBβ then everything in First(β) except ε is in Follow(B).

3. If there is a production A->B or a production A->αBβ where First(β) contains ε, then everything in Follow(A) is in Follow(B)

- Example**!**

LL(1) Grammars

Predictive parsers are those recursive descent parsers needing no backtracking

Grammars for which we can create predictive parsers are called LL(1)

The first L means scanning input from left to right

The second L means leftmost derivation

And 1 stands for using one input symbol for lookahead

A grammar G is LL(1) if and only if whenever A-> α|βare two distinct productions of G, the following conditions hold:

For no terminal a do αandβ both derive strings beginning with a

At most one of α or βcan derive empty string

If α=> ε then βdoes not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

For each production A->α in grammar do the following:

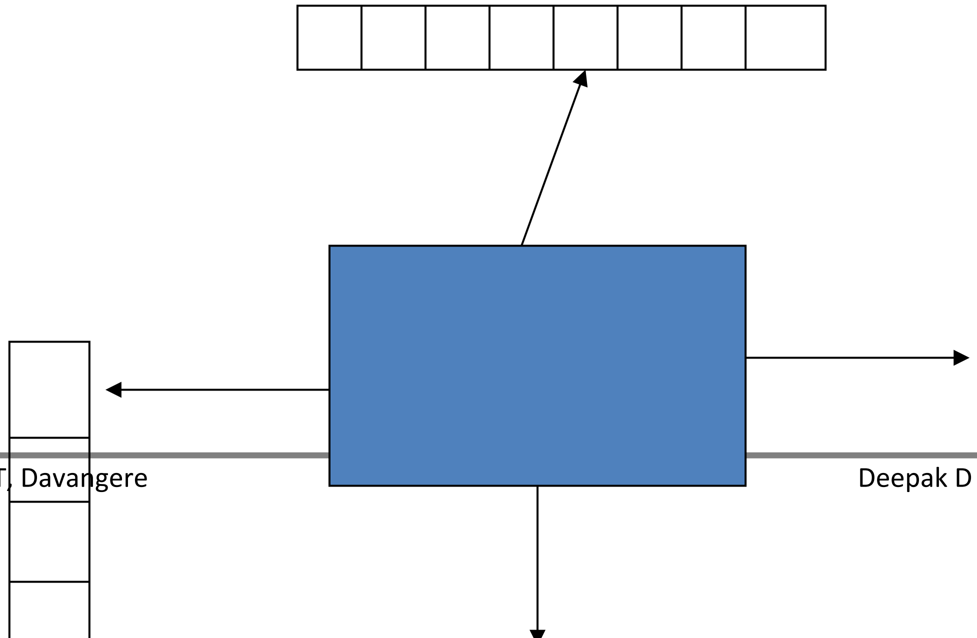
For each terminal a in First(α) add A-> in M[A,a]

If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well

If after performing the above, there is no production in M[A,a] then set M[A,a] to error .

Example

Non-recursive predicting parsing



Predictive parsing algorithm

```
Set ip 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 and advance ip;
        else if (X is a terminal) error();
        else if (M[X,a] is an error entry) error();
        else if (M[X,a] = X->Y1Y2..Yk) {
            output the production X->Y1Y2..Yk;
            pop the stack;
            push Yk,…,Y2,Y1 on to the stack with Y1 on top;
        }
        set X to the top stack symbol;
}
```

BOTTOMUP PARSING

Shift-reduce parser

The general idea is to shift some symbols of input to the stack until a reduction can be applied

At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production

The key decisions during bottom-up parsing are about when to reduce and about what production to applyA reduction is a reverse of a step in a derivation

The goal of a bottom-up parser is to construct a derivation in reverse: E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

- **A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation**

Shift reduce parsing (cont.)

Basic operations:

Shift,Reduce,Accept, Error Example: id*id

LR Parsing

The most prevalent type of bottom-up parsers

LR(k), mostly interested on parsers with k<=1

Why LR parsers?

Table driven

Can be constructed to recognize all programming language constructs

Most general non-backtracking shift-reduce parsing method

Can detect a syntactic error as soon as it is possible to do so

Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

States represent set of items

An LR(0) item of G is a production of G with the dot at some position of the body:

For A->XYZ we have following items

A->.XYZ

A->X.YZ

A->XY.Z

A->XYZ.

In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.

What about A->X.YZ?

Constructing canonical LR(0) item sets

Augmented grammar:

G with addition of a production: S’->S

Closure of item sets:

If I is a set of items, closure(I) is a set of items constructed from I by the following rules:

Add every item in I to closure(I)

If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).

Example: E’->E

E -> E + T | T

T -> T * F | F, F -> (E) | **id**

Closure algorithm

```
SetOfItems CLOSURE(I) {
        J=I;
        repeat
            for (each item A-> α.Bβ in J)
                for (each prodcution B->γ of G)
                    if (B->.γ is not in J)
                        add B->.γ to J;
        until no more items are added to J on one round;
        return J;
```

GOTO Algorithm

```
SetOfItems  GOTO(I,X) {
  J=empty;
        if (A-> α.X β is in I)
            add CLOSURE(A-> αX. β ) to J;
```

```
        return J;
}
```

Canonical LR(0) items

```
Void items(G') {
        C= CLOSURE({[S'->.S]});
        repeat
              for (each set of items I in C)
                for (each grammar symbol X)
                  if (GOTO(I,X) is not empty and not in C)
                     add GOTO(I,X) to C;
        until no new set of items are added to C on a round;
}
```

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F-
(3)	03	$F	*id$	Reduce by T-
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F-
(7)	02710	$T*F	$	Reduce by T-
(8)	02	$T	$	Reduce by E-
(9)	01	$E	$	accept

a1	…	ai	…	an	$

Sm

Sm-1

…

$

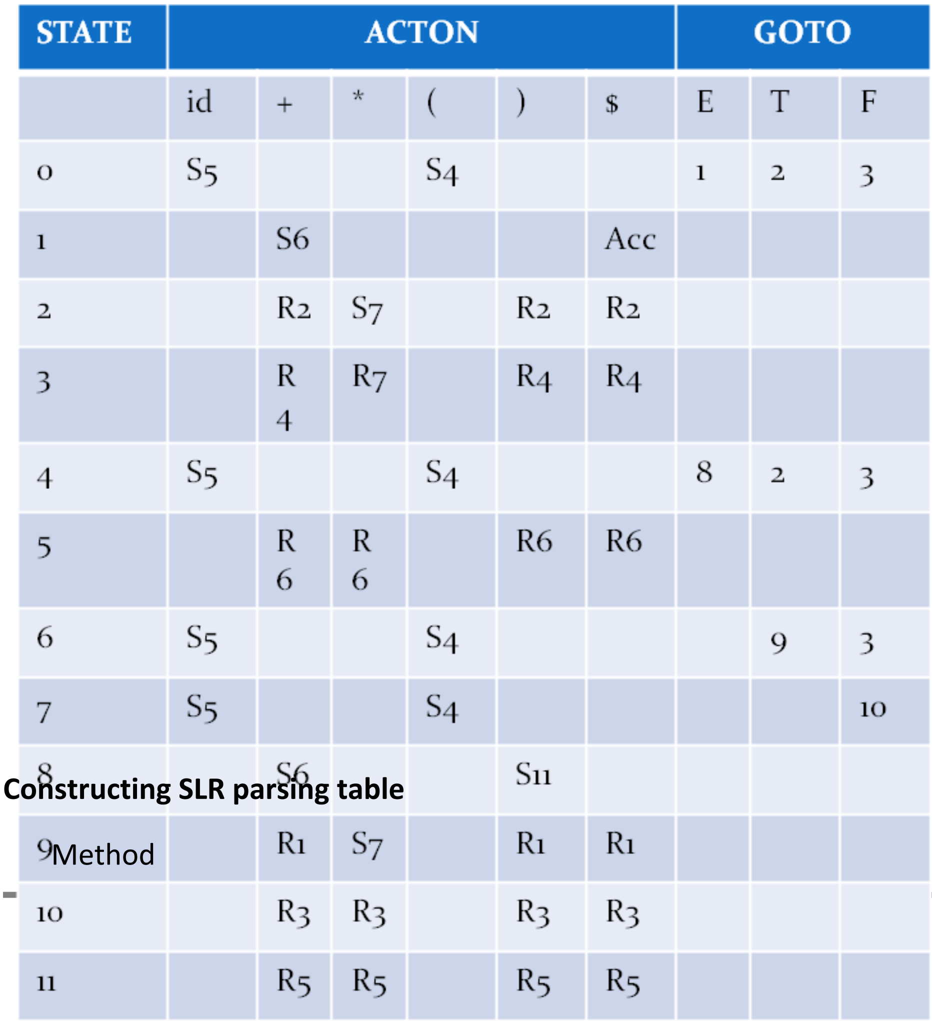
LR parsing algorithm

let a be the first symbol of w$;

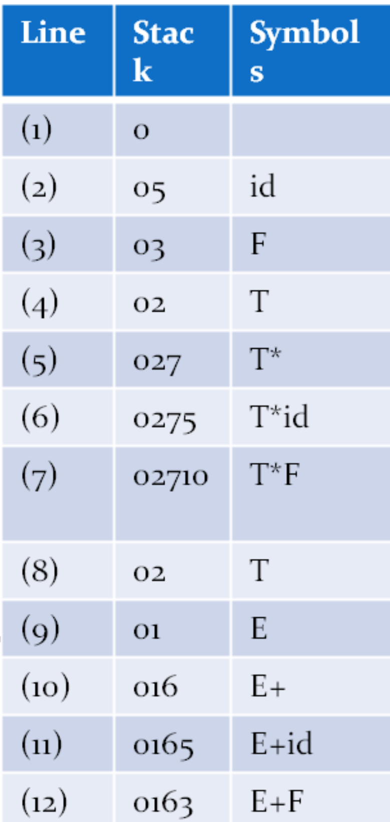
while(1) { /*repeat forever */

ACTION	GOTO

```
        let s be the state on top of the stack;
        if (ACTION[s,a] = shift t) {
                push t onto the stack;
                let a be the next input symbol;
        } else if (ACTION[s,a] = reduce A->β) {
                pop |β| symbols of the stack;
                let state t now be on top of the stack;
                push GOTO[t,A] onto the stack;
                output the production A->β;
        } else if (ACTION[s,a]=accept) break; /* parsing is done */
        else call error-recovery routine;
}
```



STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R 4	R7		R4	R4			
4	S5			S4			8	2	3
5		R 6	R 6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			



Line	Stack	Symbols
(1)	0	
(2)	05	id
(3)	03	F
(4)	02	T
(5)	027	T*
(6)	0275	T*id
(7)	02710	T*F
(8)	02	T
(9)	01	E
(10)	016	E+
(11)	0165	E+id
(12)	0163	E+F

Constructing SLR parsing table

Method

Construct C={I0,I1, … , In}, the collection of LR(0) items for G’

State i is constructed from state Ii:

If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”

If [A->α.] is in Ii, then set ACTION[i,a] to “reduce A->α” for all a in follow(A)

If {S’->.S] is in Ii, then set ACTION[I,$] to “Accept”

If any conflicts appears then we say that the grammar is not SLR(1).

If GOTO(Ii,A) = Ij then GOTO[i,A]=j

All entries not defined by above rules are made “error”

The initial state of the parser is the one constructed from the set of items containing [S’->.S]

MODULE-5

- **Syntax Directed Translation**
- **Intermediate code generation**
- **Code generation**

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Ordering the evaluation of attributes

If dependency graph has an edge from M to N then M must be evaluated before the attribute of N

Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j

Such an ordering is called a topological sortof a graph

Example!

S-Attributed definitions

An SDD is S-attributed if every attribute is synthesized

We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
    for (each child C of N, from the left) postorder(C);
    evaluate the attributes associated with node N;
}
```

S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- 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 us a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
        - Inherited attributes associated with the head A
        - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
        - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Construction of syntax trees
    - Leaf nodes: Leaf(op,val)
    - Interior node: Node(op,c1,c2,…,ck)

Example:

Production

E -> E1 + T

E -> E1 - T

E -> T

T -> (E)

T -> id

T -> num

Semantic RULE

E.node=new node('+', E1.node,T.node)

E.node=new node('-', E1.node,T.node)

E.node = T.node

T.node = E.node

T.node = new Leaf(id,id.entry)

T.node = new Leaf(num,num.val)

Syntax tree for L-attributed definition

Syntax directed translation schemes

An SDT is a Context Free grammar with program fragments embedded within production bodies

Those program fragments are called semantic actions

They can appear at any position within production body

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order

Typically SDT's are implemented during parsing without building a parse tree .

Postfix translation schemes

Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed

For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production

SDT's with all actions at the right ends of the production bodies are called postfix SDT's

Parse-Stack implementation of postfix SDT's

In a shift-reduce parser we can easily implement semantic action using the parser stack

For each nonterminal (or state) on the stack we can associate a record holding its attributes

Then in a reduction step we can execute the semantic action at the end of a production to evaluate the attribute(s) of the non-terminal at the leftside of the production

And put the value on the stack in replace of the rightside of production

EXAMPLE

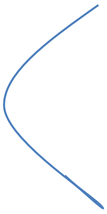
```
L -> E n        {print(stack[top-1].val);
                               top=top-1;}

E -> E1 + T    {stack[top-2].val=stack[top-2].val+stack.val;
                               top=top-2;}

E -> T

T -> T1 * F     {stack[top-2].val=stack[top-2].val+stack.val;
                               top=top-2;}

T -> F

F -> (E)      {stack[top-2].val=stack[top-1].val
                       top=top-2;}

F -> digit
```

Intermediate Code Generation

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation.

Variants of syntax trees

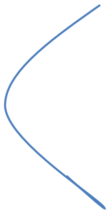
- It is sometimes beneficial to crate a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- Example: a+a*(b-c)+(b-c)*d



SDD for creating DAG’sSDD for creating DAG’s

Value-number method for constructing DAG's

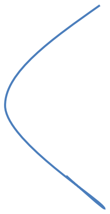
- Algorithm
    - Search the array for a node M with label op, left child l and right child r
    - If there is such a node, return the value number M
    - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table



Three address code

- **In a three address code there is at most one operator at the right side of an instruction**

Example:



Data structures for three address codes

- Quadruples
    - Has four fields: op, arg1, arg2 and result
- Triples
    - Temporaries are not used and instead references to instructions are made
- Indirect triples
    - In addition to triples we use a list of pointers to triples.

Type Expressions

Example: int[2][3]

array(2,array(3,integer))

A basic type is a type expression

A type name is a type expression

A type expression can be formed by applying the array type constructor to a number and a type expression.

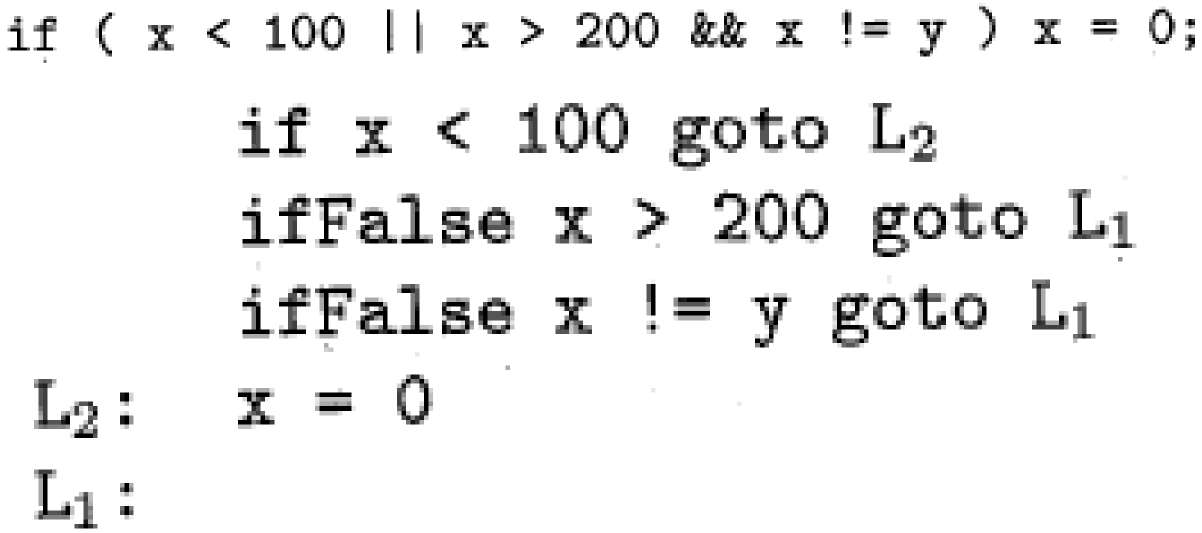
A record is a data structure with named field

A type expression can be formed by using the type constructor g for function types

If s and t are type expressions, then their Cartesian product s**t* is a type expression

Type expressions may contain variables whose values are type expressions.

Short-Circuit Code



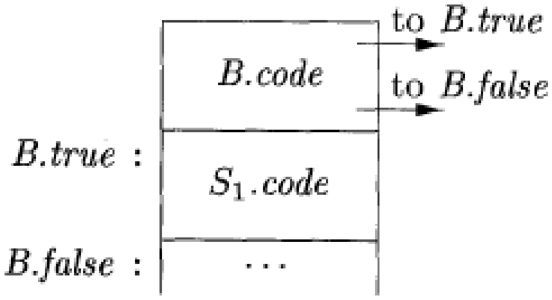
```
if ( x < 100 || x > 200 && x != y ) x = 0;

        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

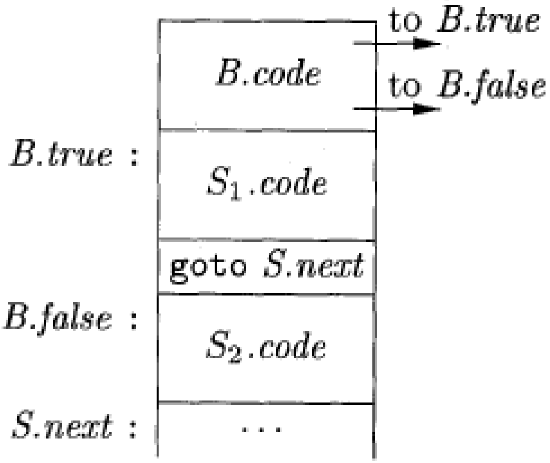
$S \rightarrow$ **if** (B) S_1

$S \rightarrow$ **if** (B) S_1 **else** S_2

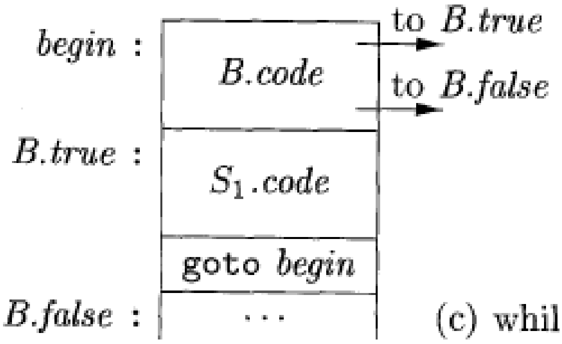
$S \rightarrow$ **while** (B) S_1



(a) if



(b) if-else



(c) while

Regular Expressions.

any lang which is accepted by FA we can represent it using regular expressions

① ✓ union (+)

✓ concat (.)

✓ kleene closure (*)

(a) $\emptyset$, ϵ , $a \in \Sigma$ (Primitive)

$$\begin{array}{cccc} \emptyset & \epsilon & a & b \\ \downarrow & \downarrow & \downarrow & \\ \{\ \} & \{\epsilon\} & \{a\} & \{b\} \end{array}$$

(b) $r_1 + r_2, r_1 . r_2, r_1^*$ where r_1, r_2 regular expressions

eg: $\emptyset = \{\ \}$

λ = ~~$\{\epsilon\}$~~ $\{\lambda\}$

$a = \{a\}$

$a^* = \{$~~ϵ~~$, \lambda, a, aa, \ldots\}$

$a^+ = a . a^*$

$= \{a, aa, aaa, \ldots\}$

$(a+b)^* = \{\lambda, a, b, aa, ab, ba, bb, \ldots\}$

✓ **eg:** set of all strings whose length is exactly 2 over $\Sigma = \{a,b\}$

$L = \{aa, ab, ba, bb\}$

so regular expressions will be

$= aa + ab + ba + bb$ ($\because$ lang. is finite)

$= a(a+b) + b(a+b)$ or union

$= (a+b)(a+b)$

✓ **eg:** length is atleast 2

$L_1 = \{aa, ab, ba, bb, aaa, \ldots\}$

(lang. is infinite)

regular expression $= (a+b)(a+b)(a+b)^*$

LectureNotes.in

eg exactly 2 nos. of 'a' // b can be any numbers.

RE = $b^*ab^*ab^*$

$b^*ab^*ab^*$

eg: atleast 2 nos. of 'a'

RE = $b^*ab^*a(a+b)^*$

eg atmost 2 nos. of 'a' // 0 'a', 1 'a', 2 'a's

RE = $b^*(\lambda+a)b^*(\lambda+a)b^*$

↓

there can be 'a'

or there can be no 'a'

eg: even number of a's

RE = $(b^*ab^*ab^*)^* + b^*$

eg: set of all strings starting with a

RE = $a(a+b)^*$

eg: set of all strings ends with a

RE = $(a+b)^*a$

eg: set of all strings containing a

RE = $(a+b)^*a(a+b)^*$

eg: set of all strings starting and ending with diff. symbols

RE = $a(a+b)^*b + b(a+b)^*a$

eg: starting and ends with same symbol

i.e. L = $\{\lambda, a, b, aa, bb, \ldots\}$

LectureNotes.in

Scanned by CamScanner

$\therefore RE = (ab)^*a + (ab)^* + b(ab)^*a + b(ab)^*$

$= (ab)^*(a+\lambda) + b(ab)^*(a+\lambda)$

~~$= (ab)^*(a+\lambda)$~~ $= (\lambda+b)(ab)^*(a+\lambda)$

OR

$RE = (\lambda+a)(ba)^*(\lambda+b)$

Identities of RE

(1) $\phi + R = R + \phi = R$

(2) $\phi . R = R . \phi = \phi$

(3) $\lambda . R = R . \lambda = R$

(4) $\lambda^* = \lambda$

* (5) $\phi^* = \lambda$

* (6) $\lambda + RR^* = R^*R + \lambda = R^*$

* (7) $(a+b)^* = a^*(ba^*)^* = b^*(ab)^*$

eg: $L = \{a^m, b^n, m \geq 0, n \geq 0\}$

$L = \{\lambda, a, b, ab, ba, ...\}$

$RE = a^*b^*$

eg: $L = \{a^m b^n, m \geq 1, n \geq 1\}$

$L = \{a, b, ab, ba, ...\}$

$RE = a^+b^+$ OR $RE = (aa^*)(bb^*)$

eg: $L = \{a^m b^n$, $m+n$ is even $m \geq 0, n \geq 0\}$

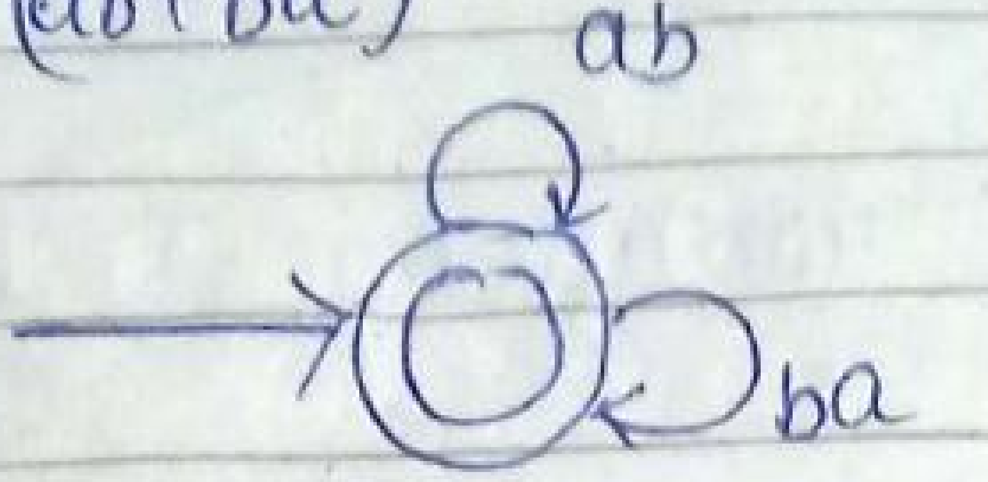
$RE = (aa)^*(bb)^* + a(aa)^*b(bb)^*$

(for both even) (for both odd)

LectureNotes.in

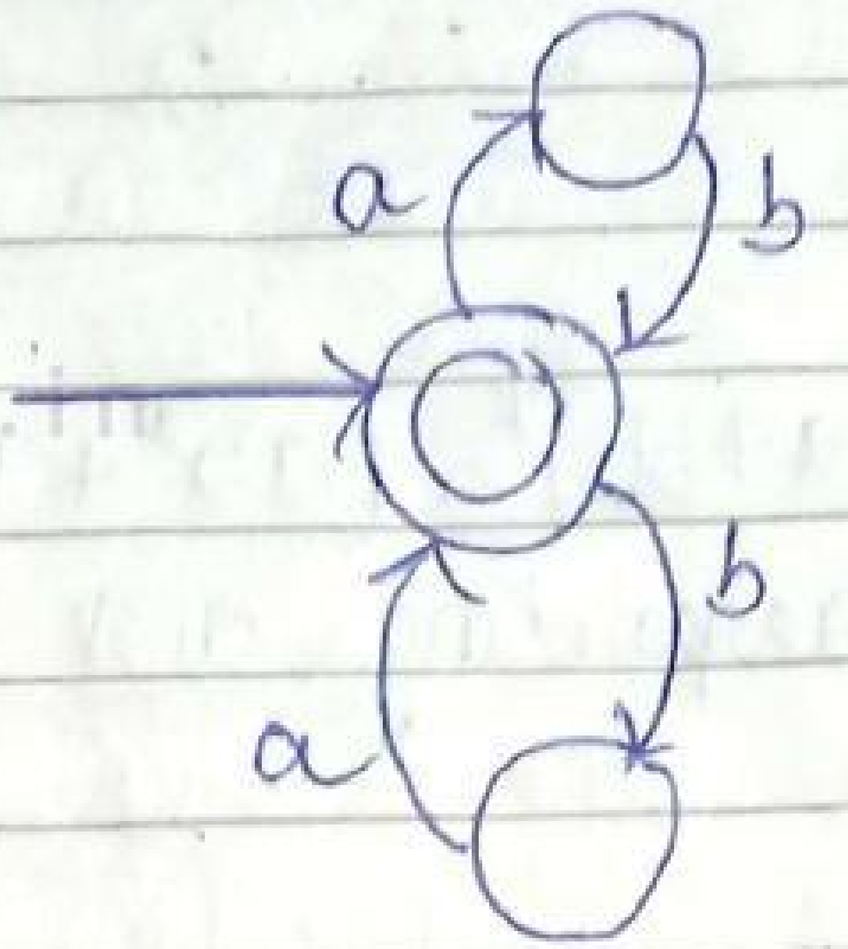
Scanned by CamScanner

eg: $a'(ab+ba)^*$

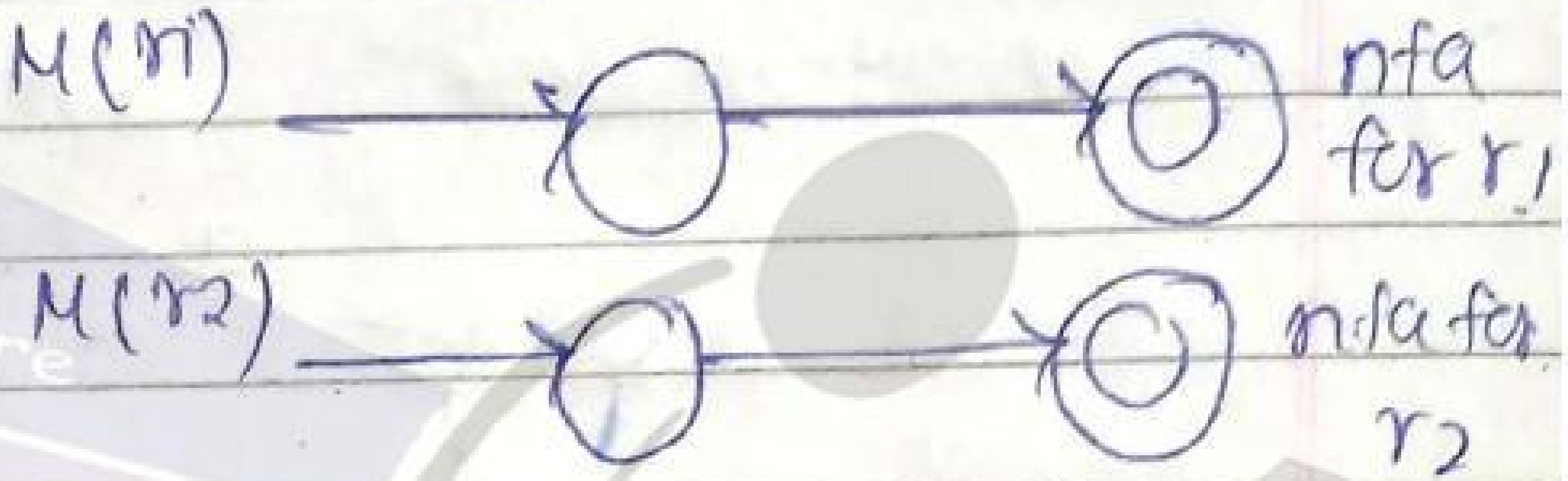


OR

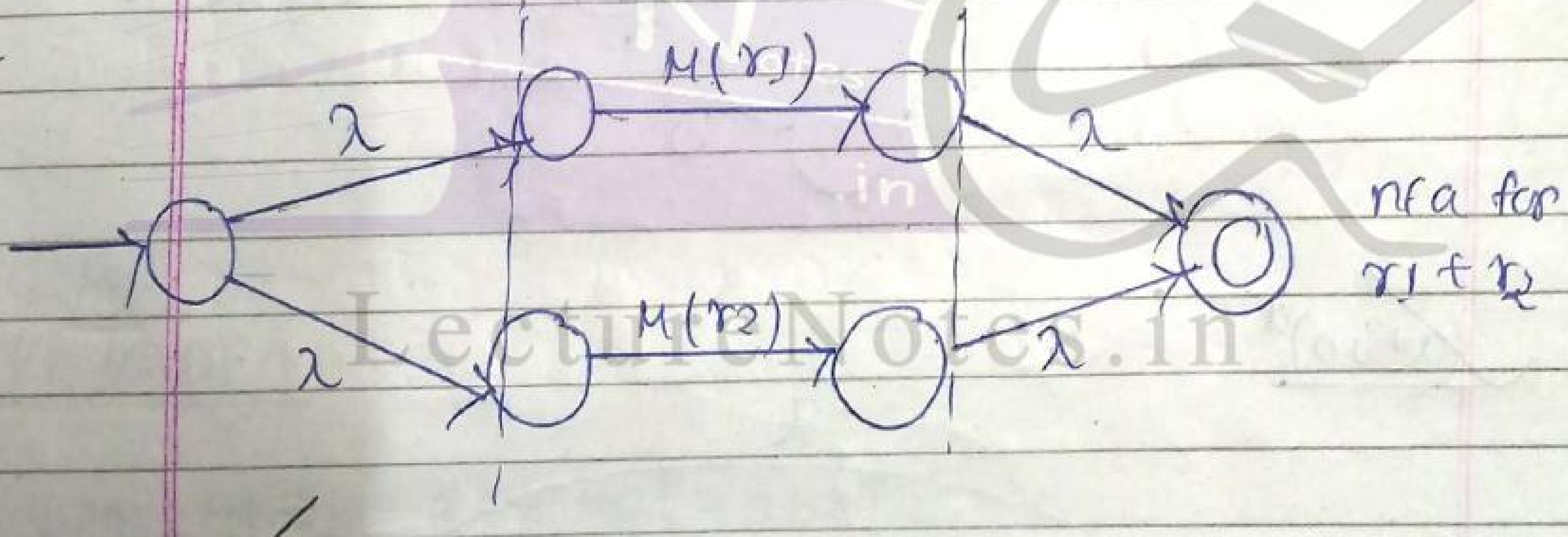
state creation method



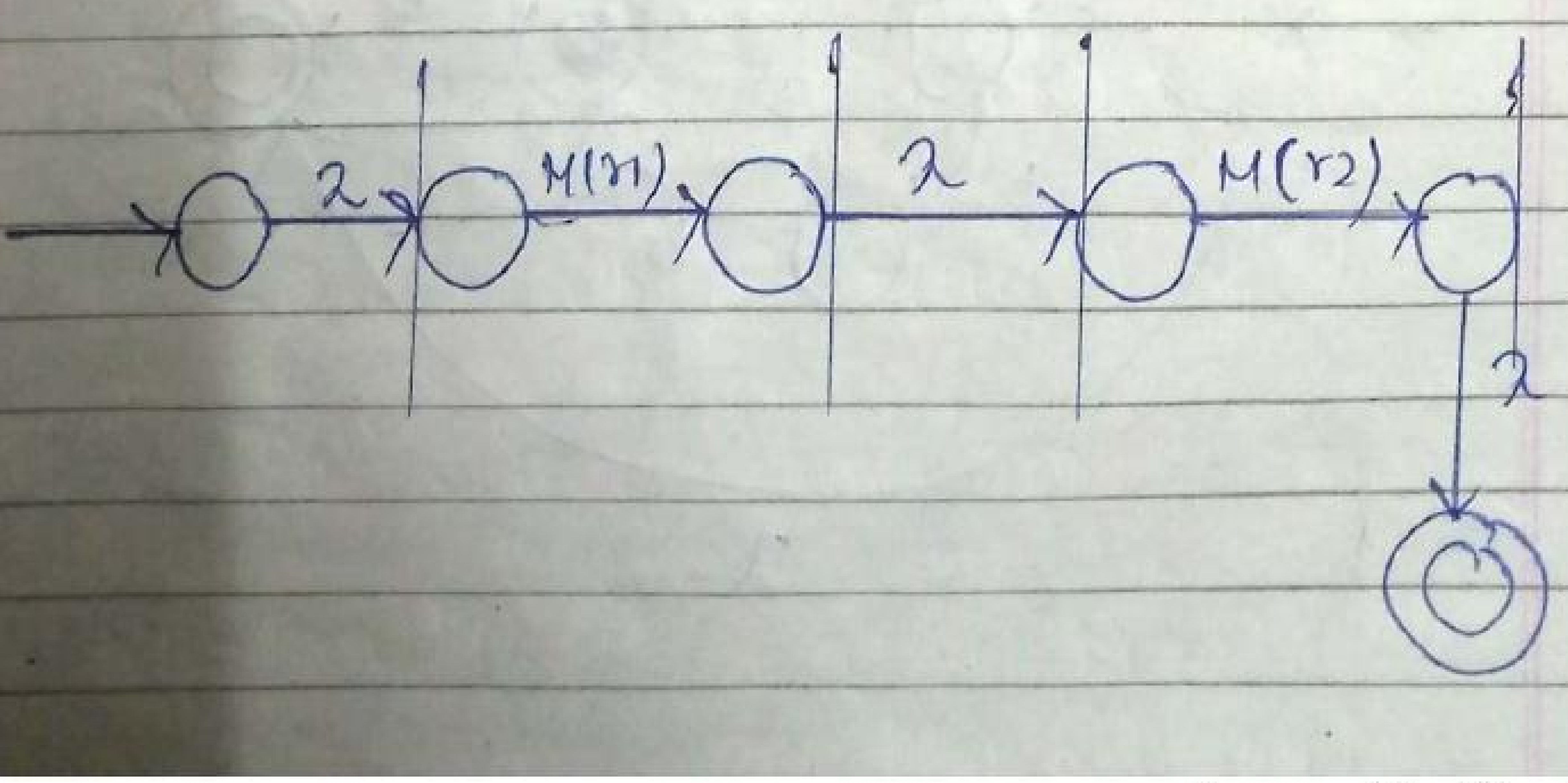
RE to NFA



eg: (a) r_1+r_2



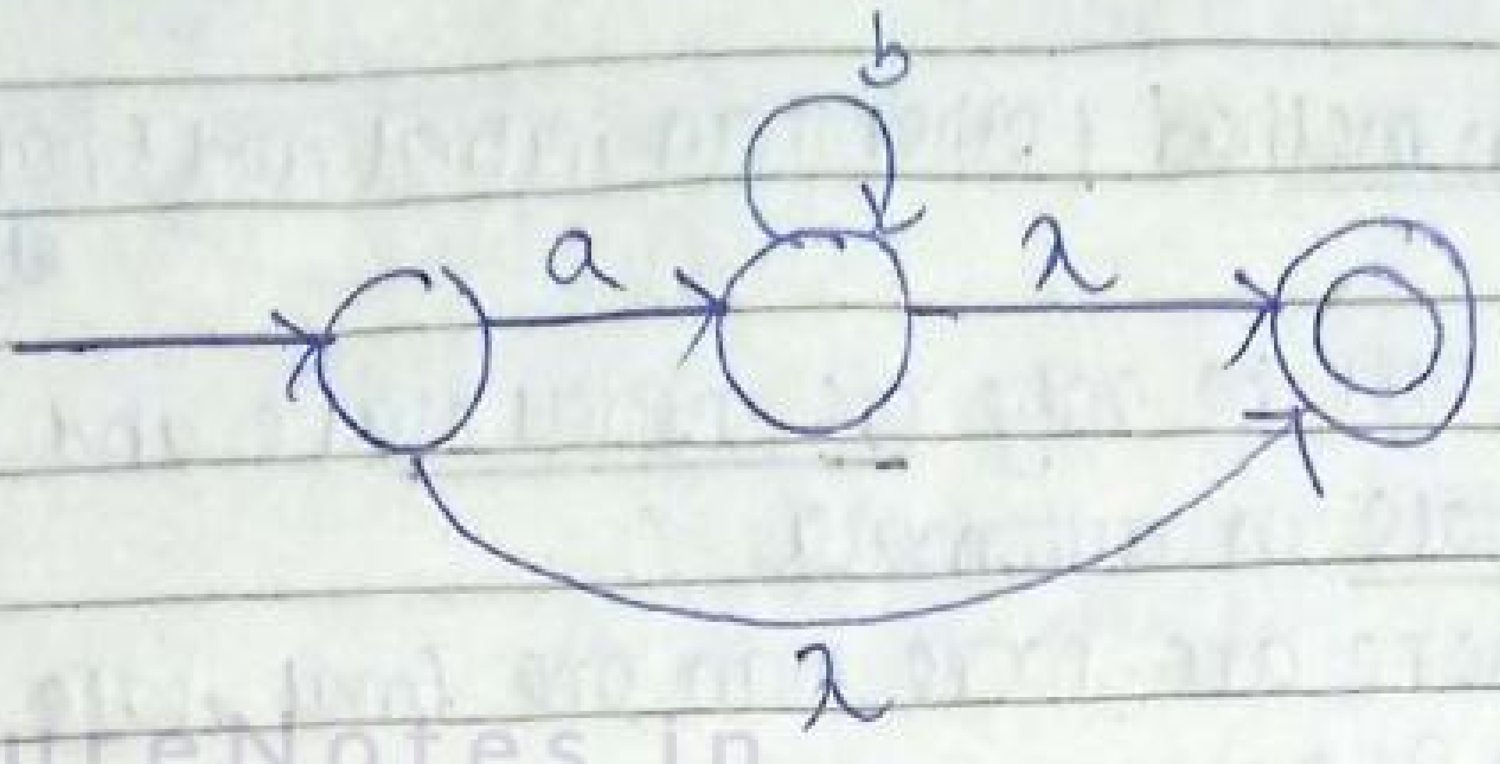
eg: (b) $r_1 . r_2$



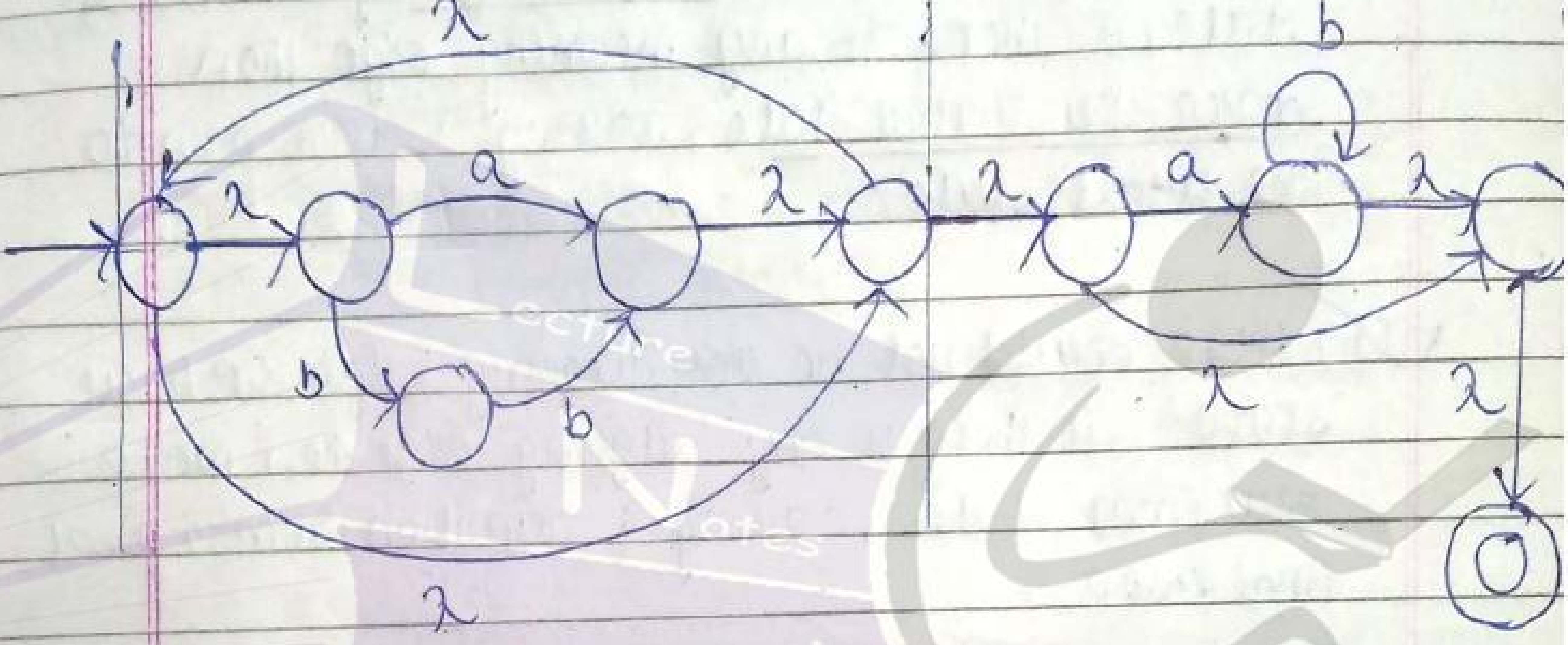
LectureNotes.in

Scanned by CamScanner

$ab^* + \lambda$

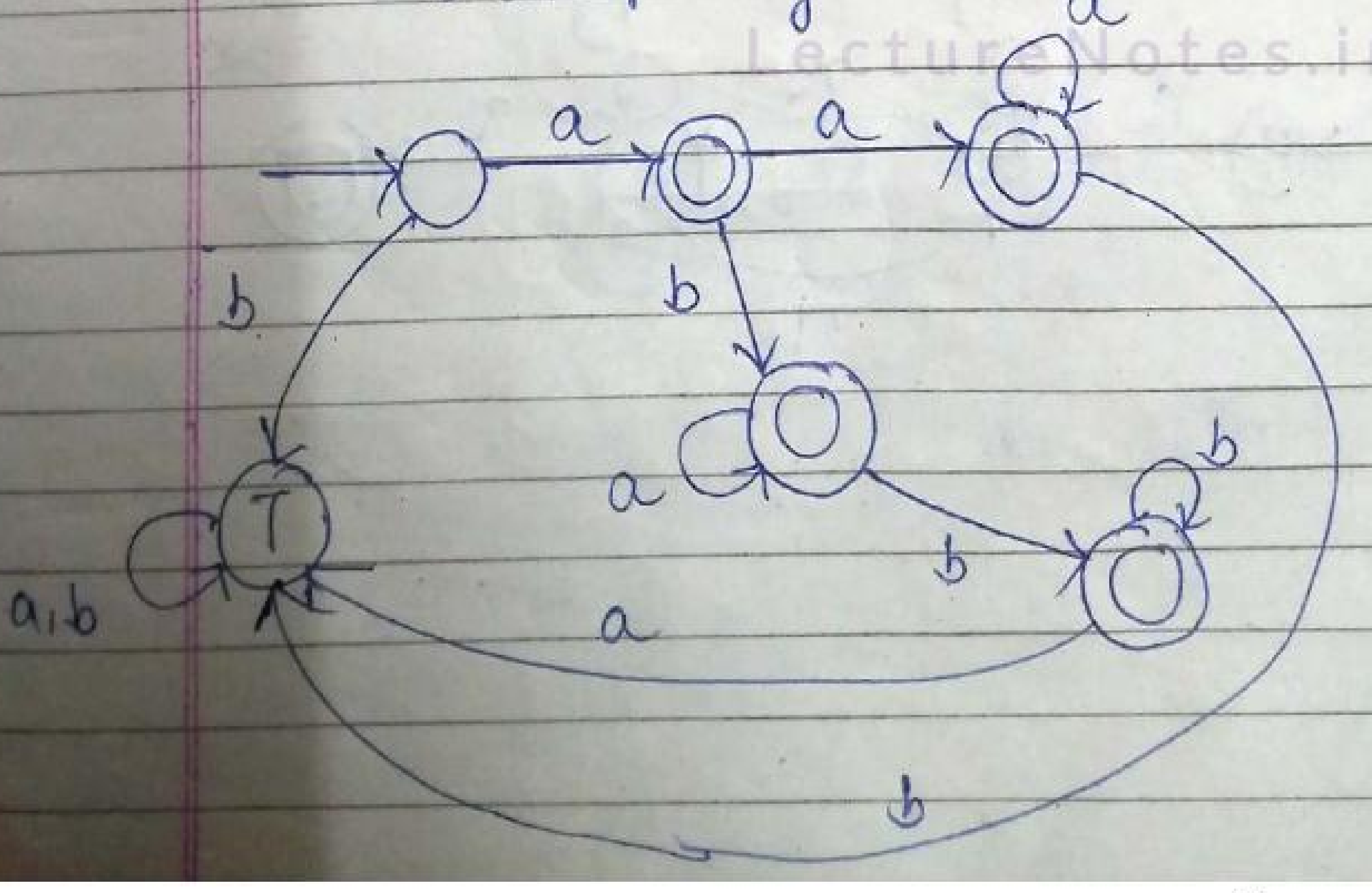


final NFA



RE to DFA

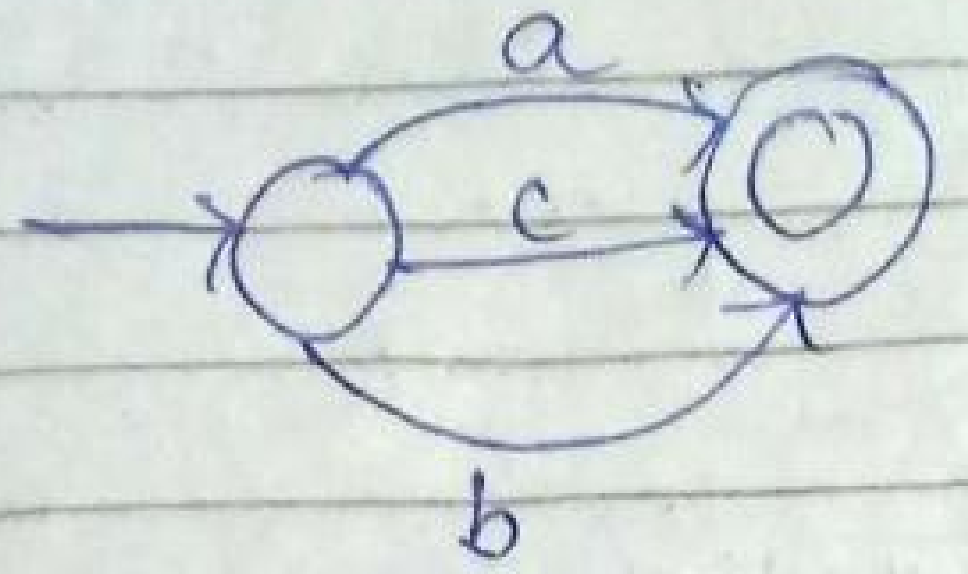
eg: convert RE: $aa^* + aba^*b^*$
to corresponding DFA



LectureNotes.in

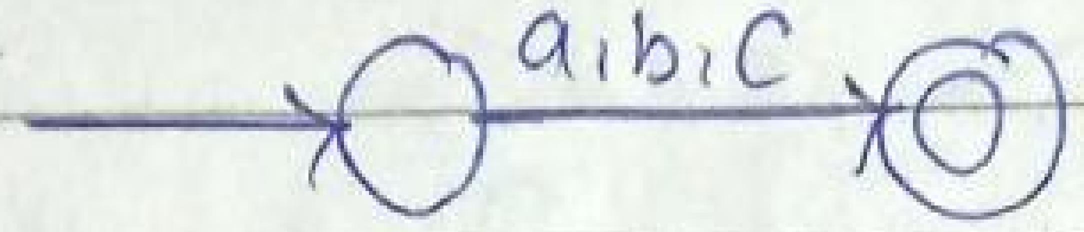
Scanned by CamScanner

eg:



- only 1 initial, final state

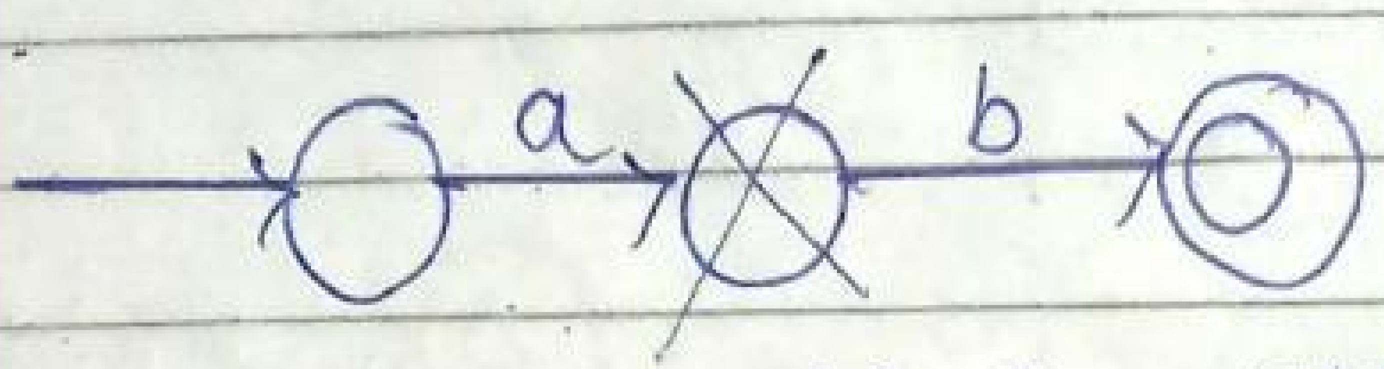
so



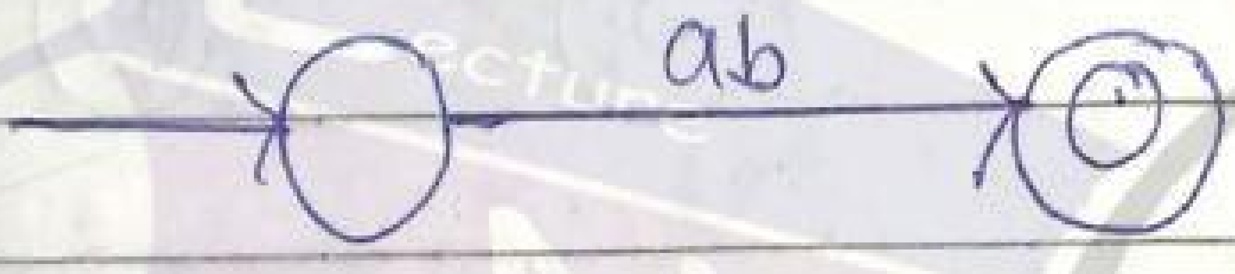
RE: (a+b+c)

LectureNotes.in

eg:

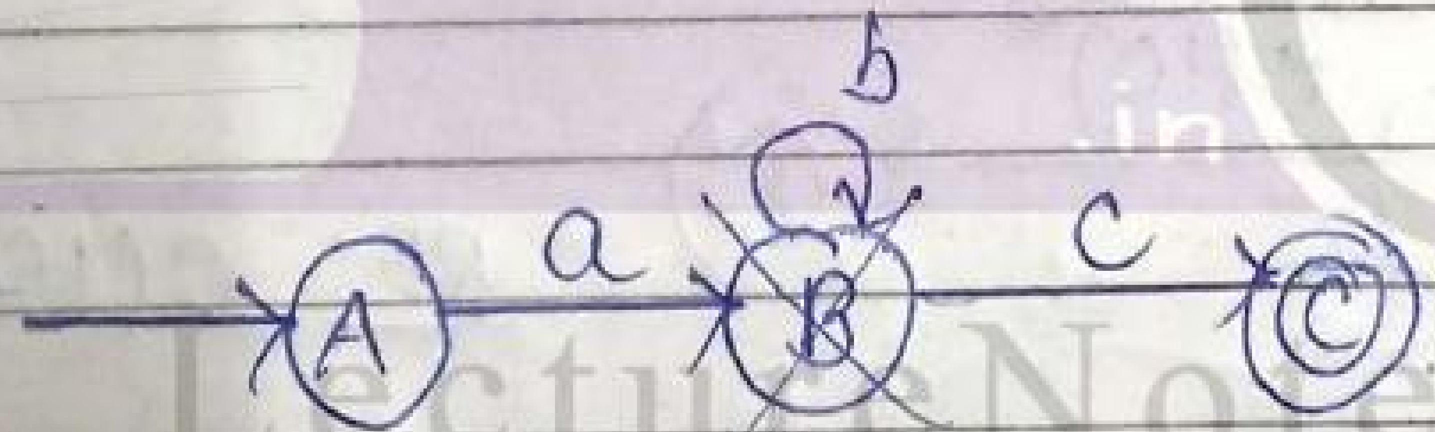


- eliminate this state and all paths remain intact



RE: ab

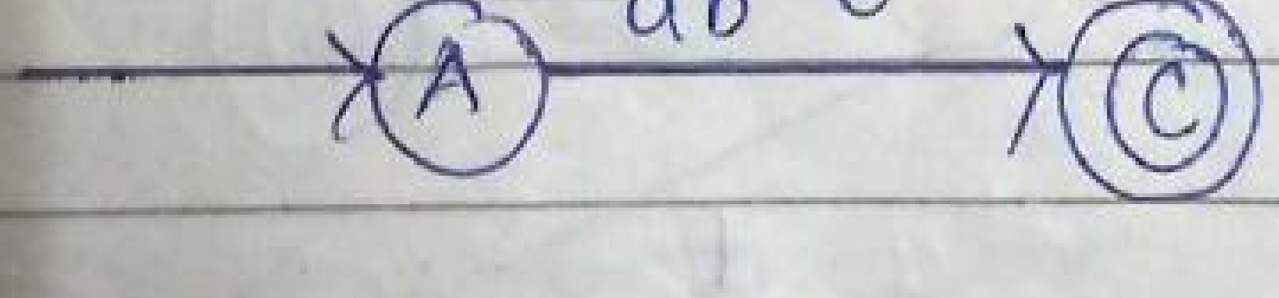
eg:



LectureNotes.in

- eliminate B
then

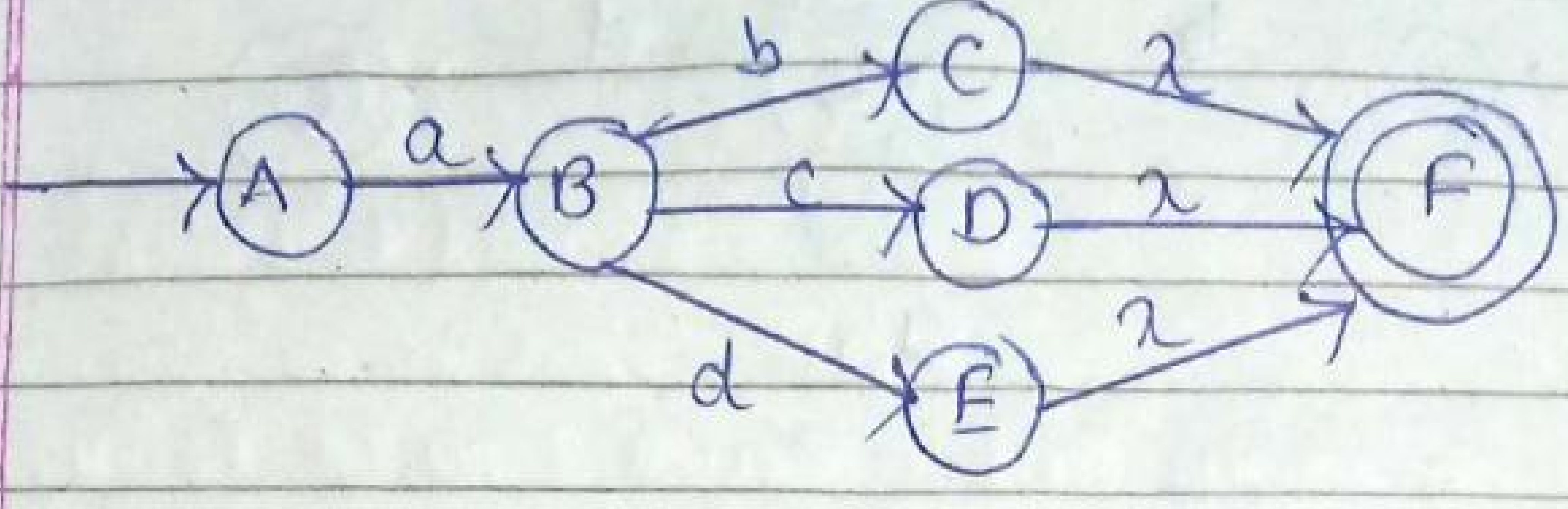
LectureNotes.in



RE: ab*C

Scanned by CamScanner

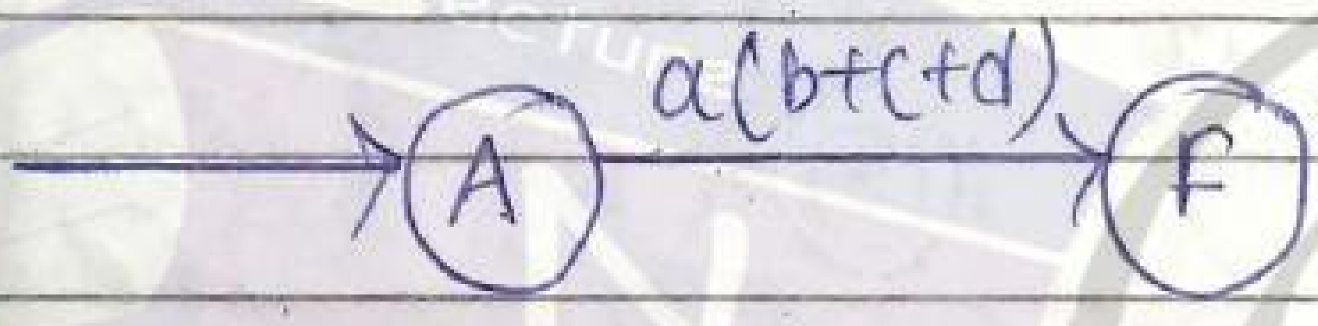
step 1:



step 2: eliminating C, D, E

LectureNotes.in

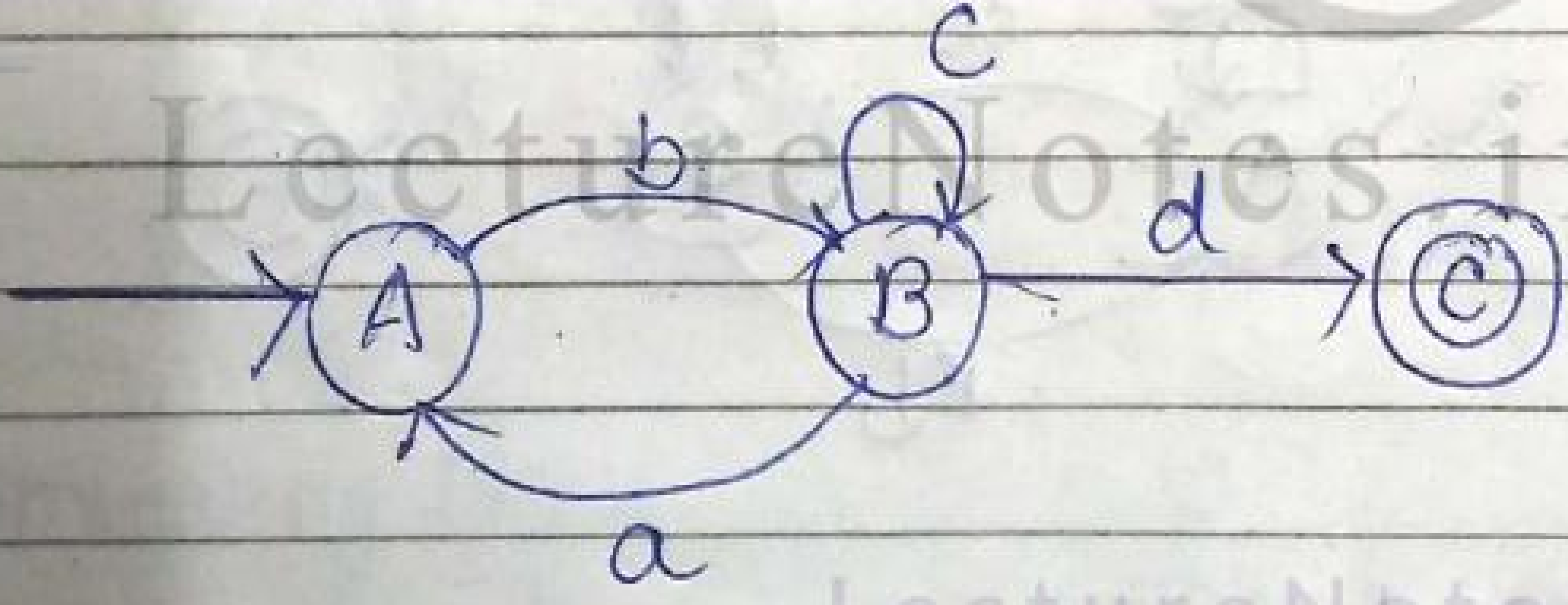
step 3: eliminating B



RE: a(b+c+d)

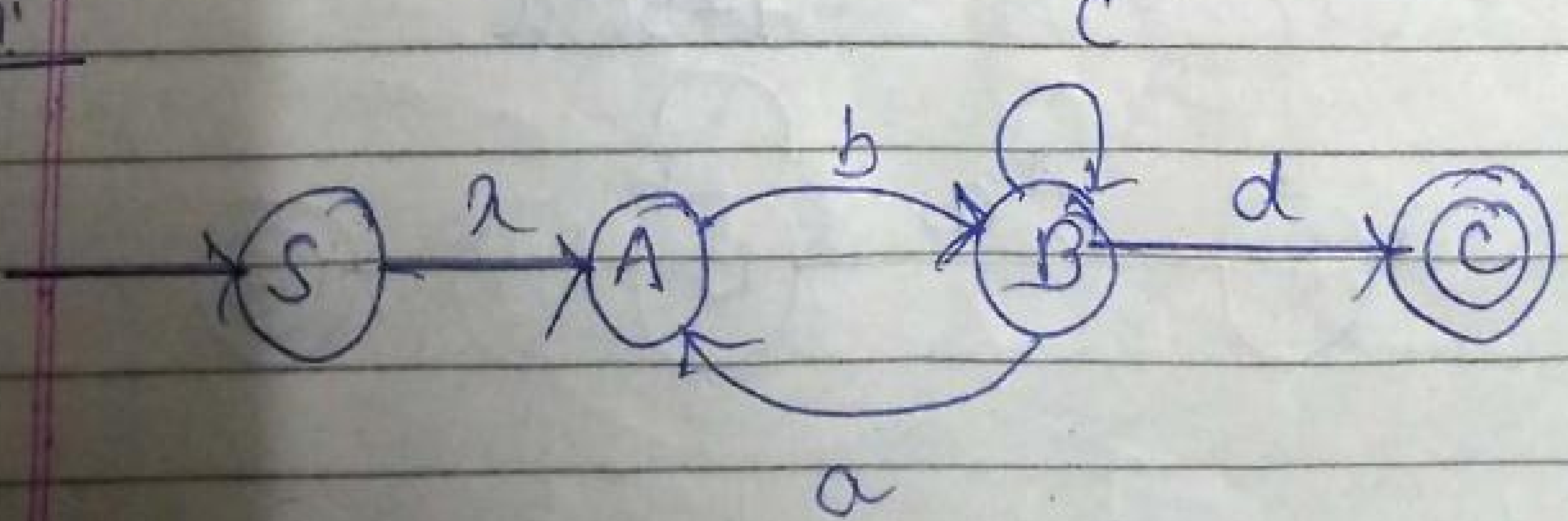
LectureNotes.in

eg:



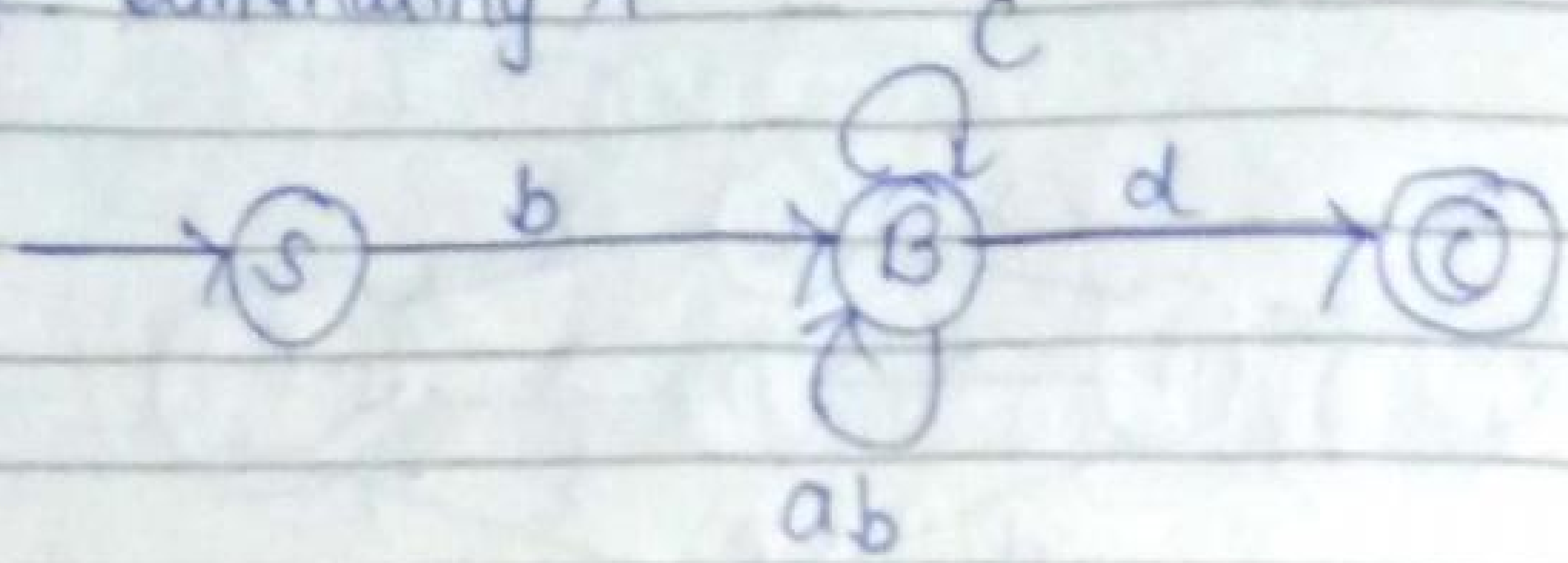
LectureNotes.in

step:

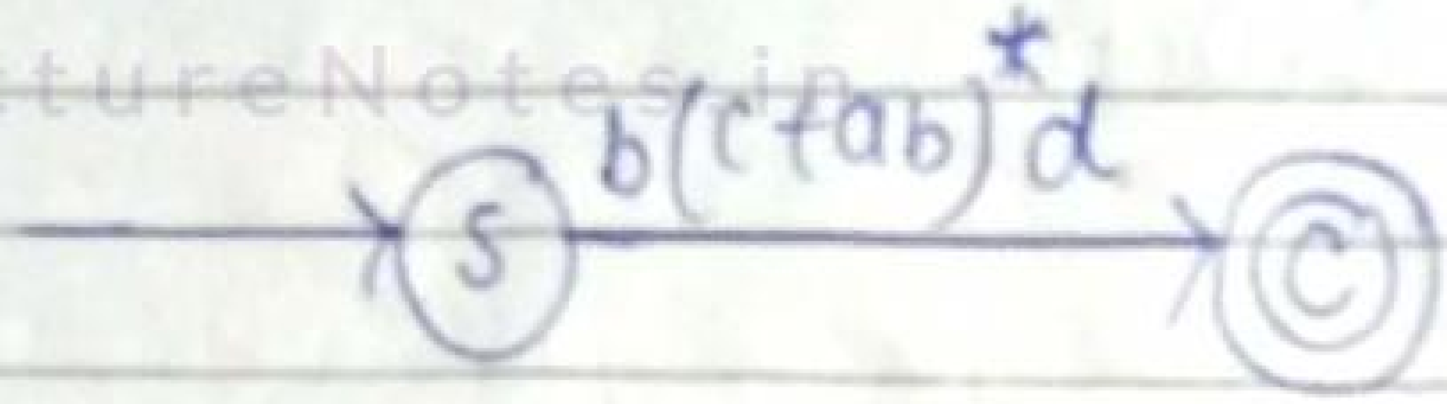


Scanned by CamScanner

step 2 : eliminating A

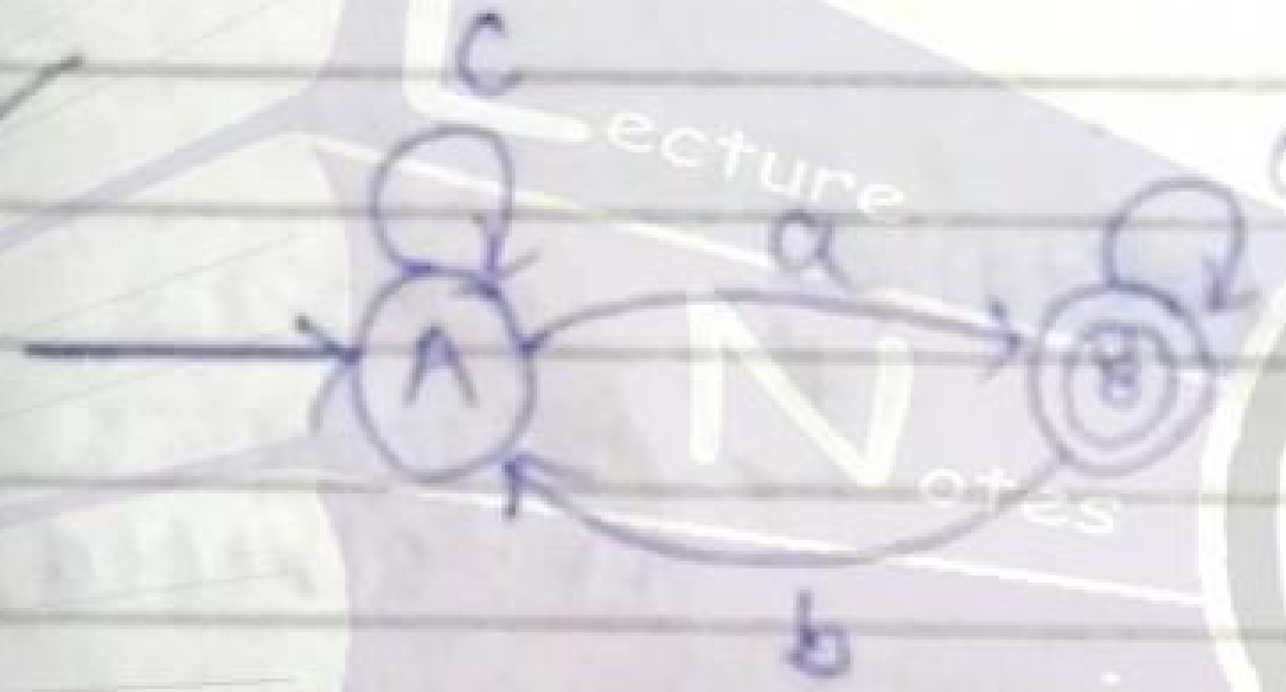


step 3: eliminating B

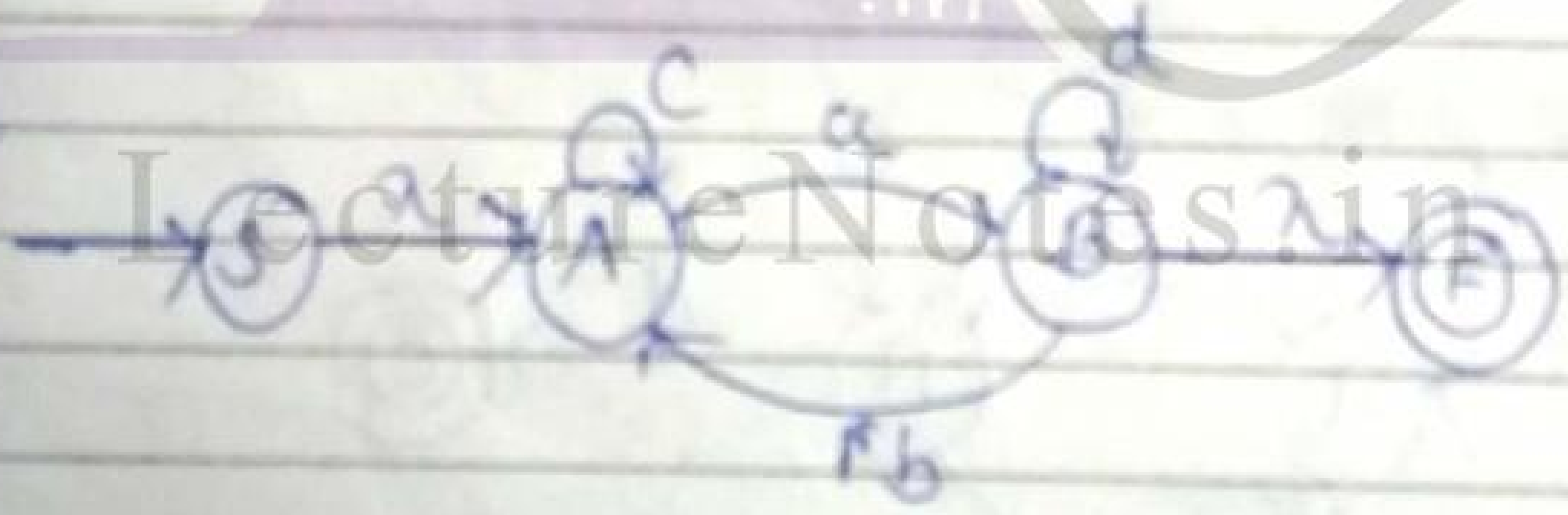


RE: b(c+ab)*d

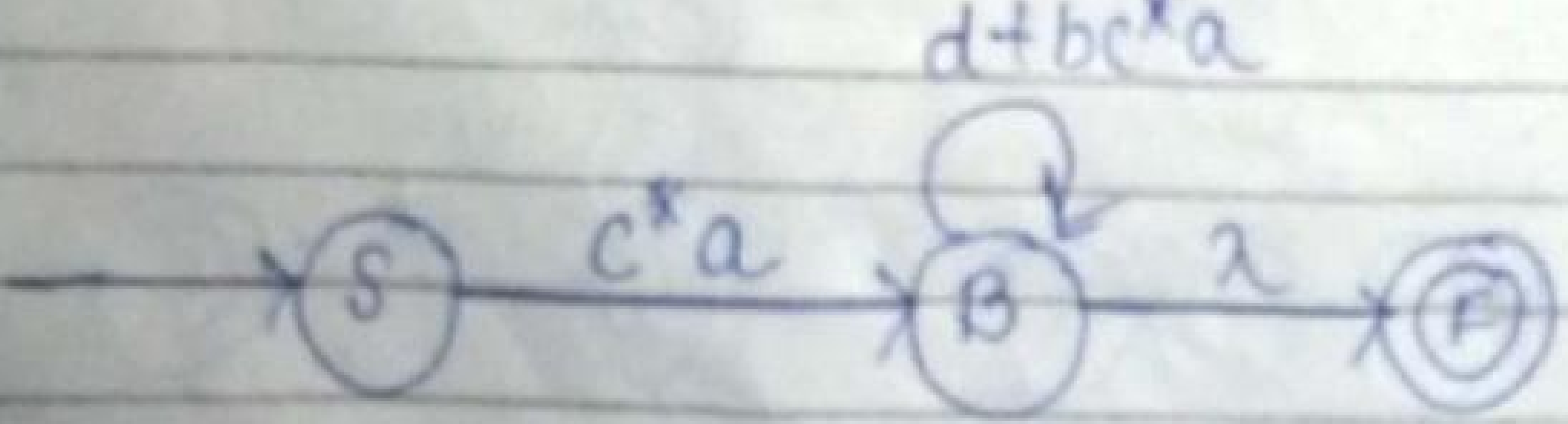
eg:



step 1

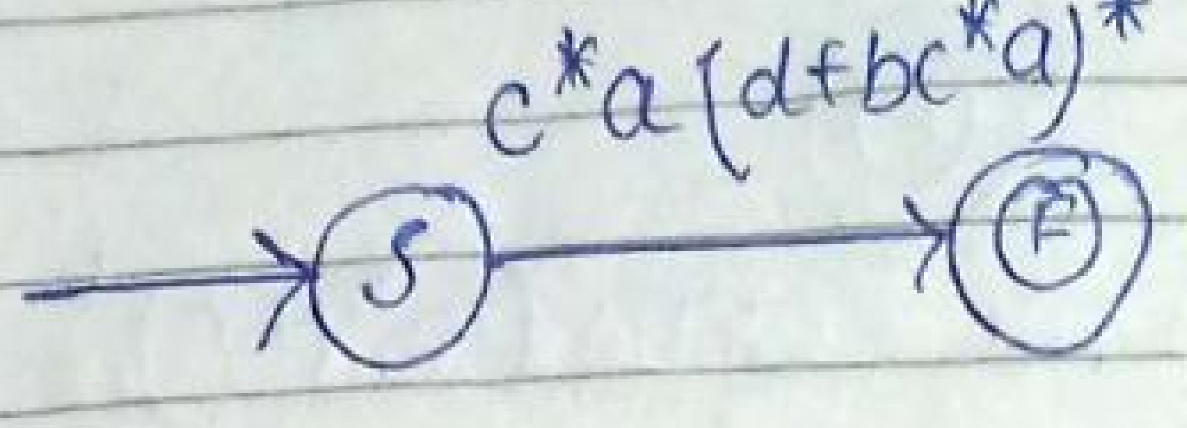


step 2: eliminating A



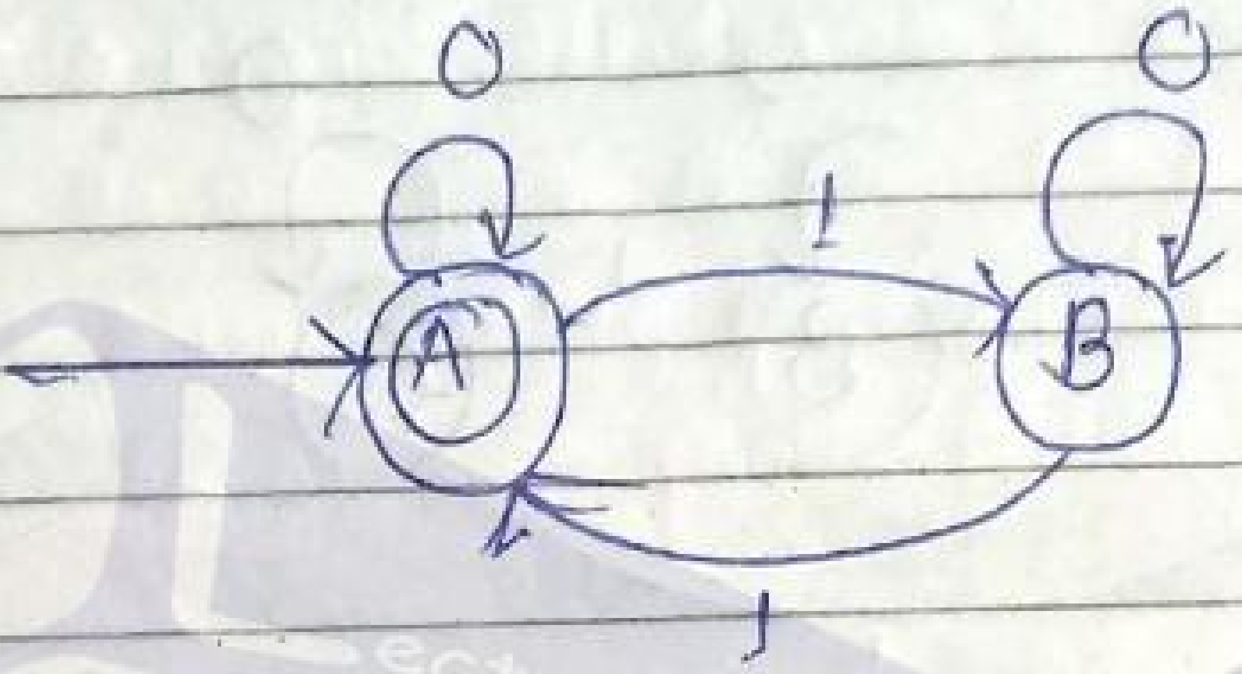
LectureNotes.in

step3 eliminating B



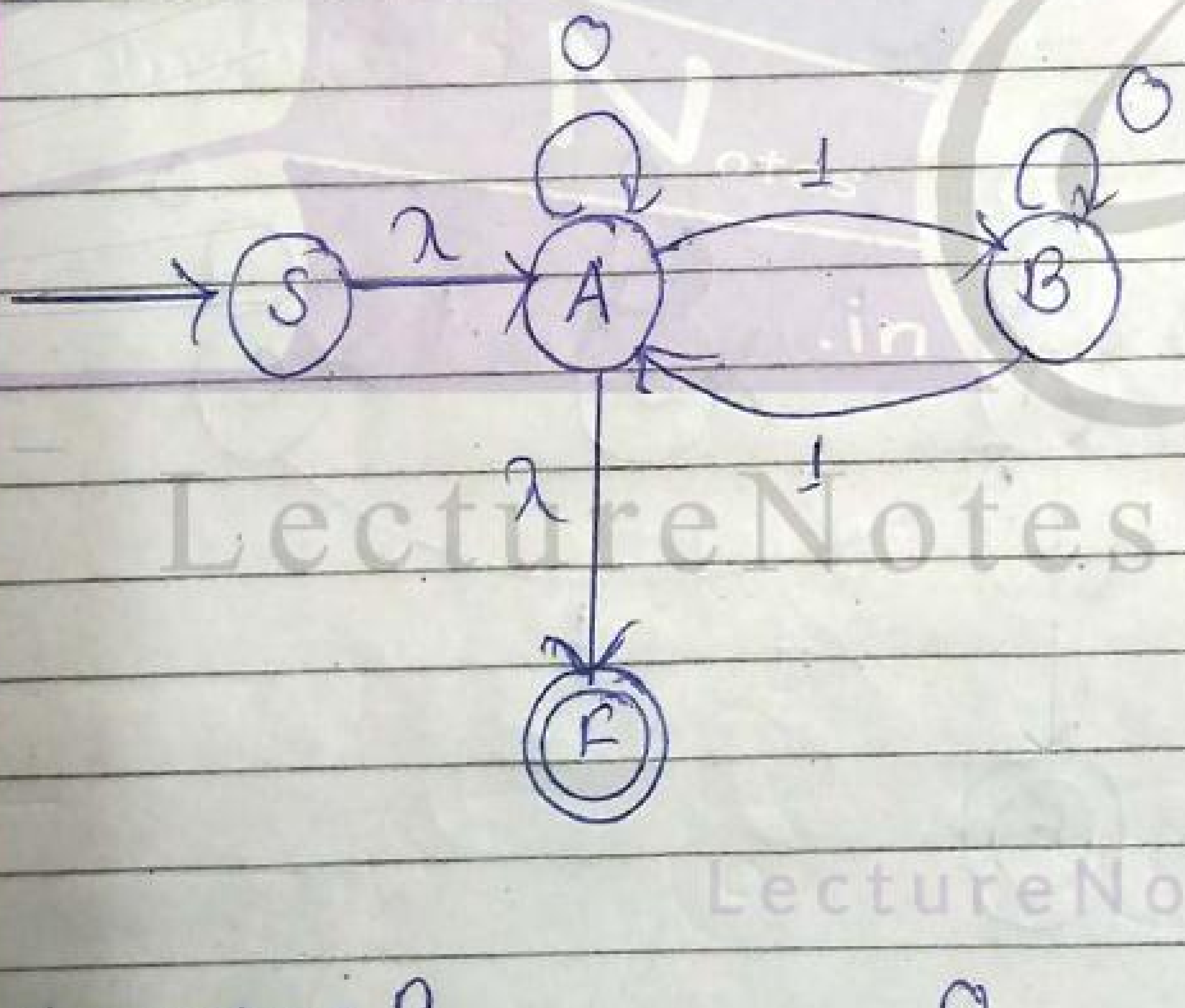
RE : $c^*a(d+bc^*a)^*$

eg:

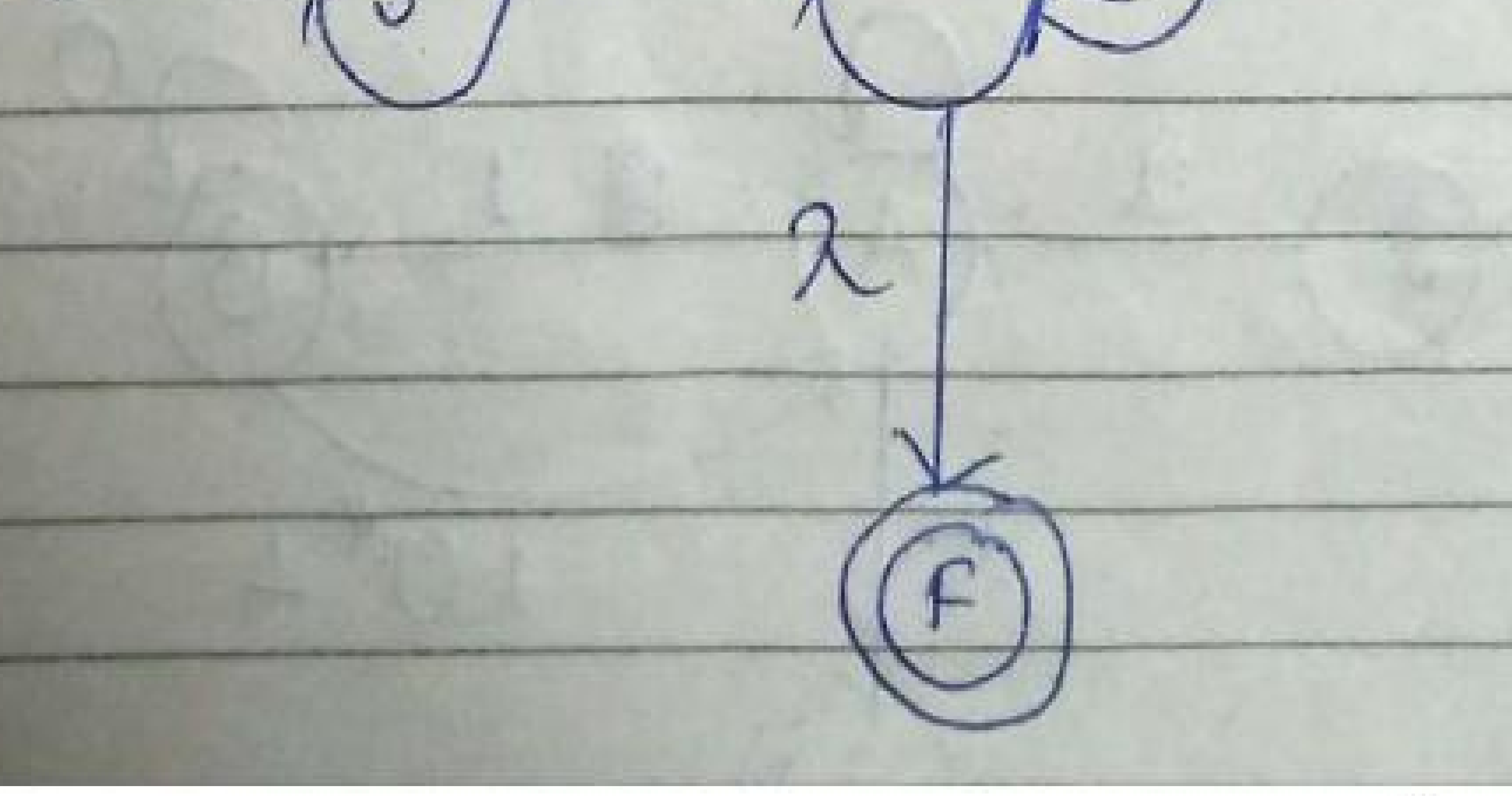


OR set of all strings containing even no. of 1's

step1:

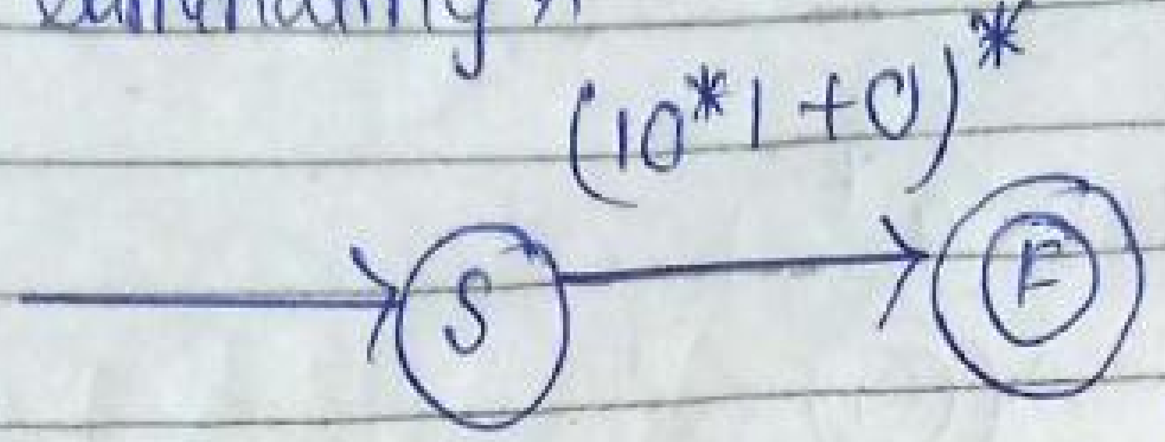


step2: eliminating B



Scanned by CamScanner

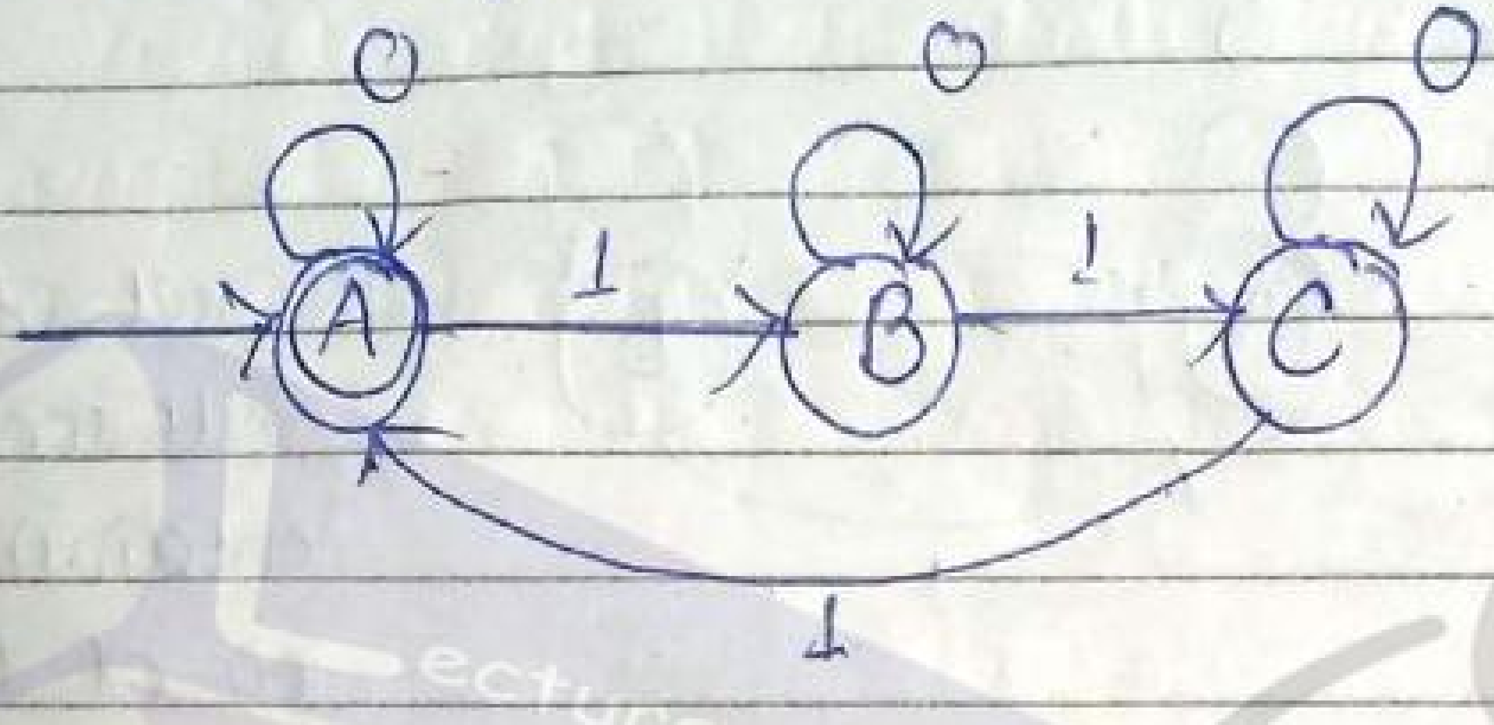
step 3: eliminating A



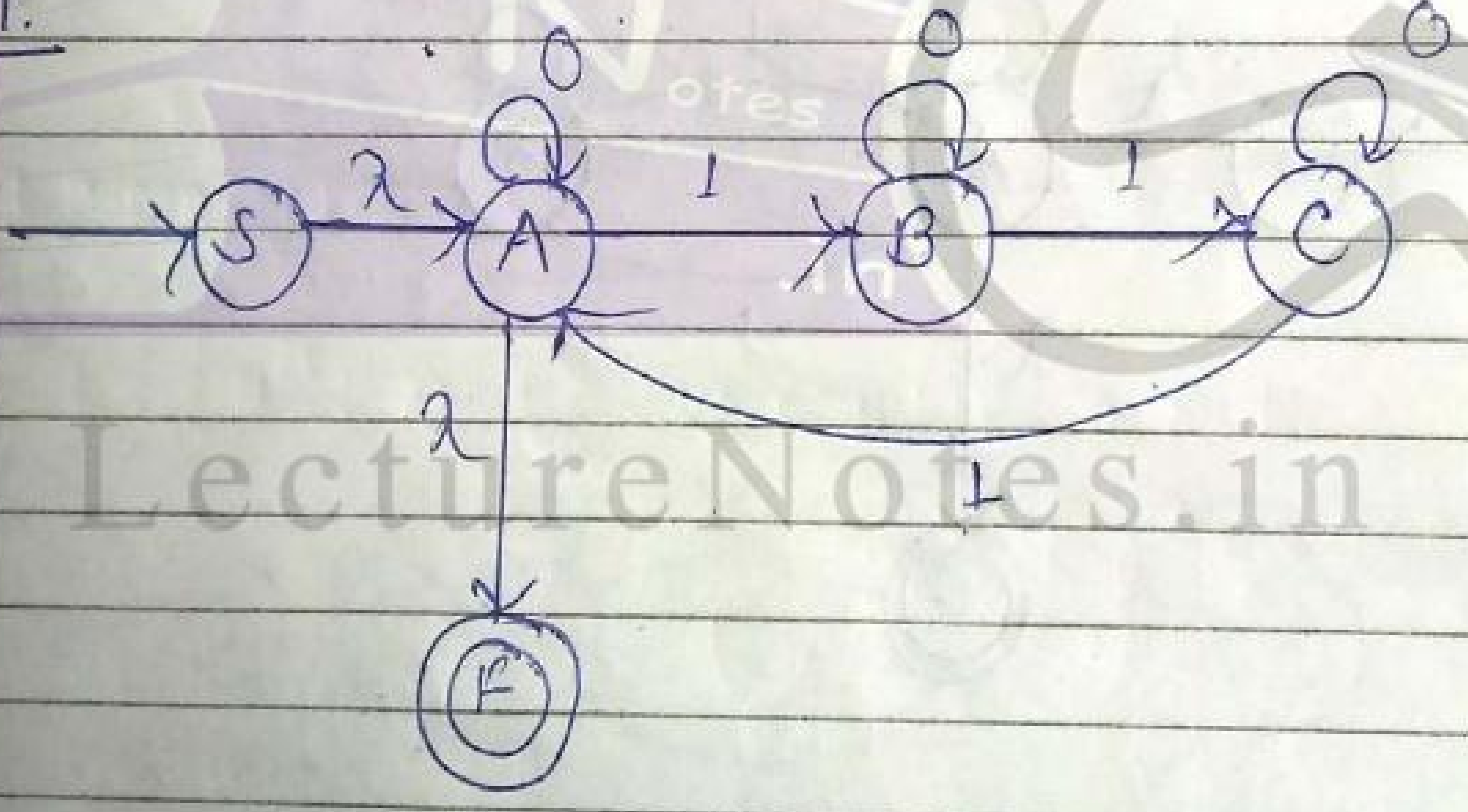
RE: $(10^*1+0)^*$

eg: set of all strings that contains ~~&~~ no. of 1's which are divisible by 3.

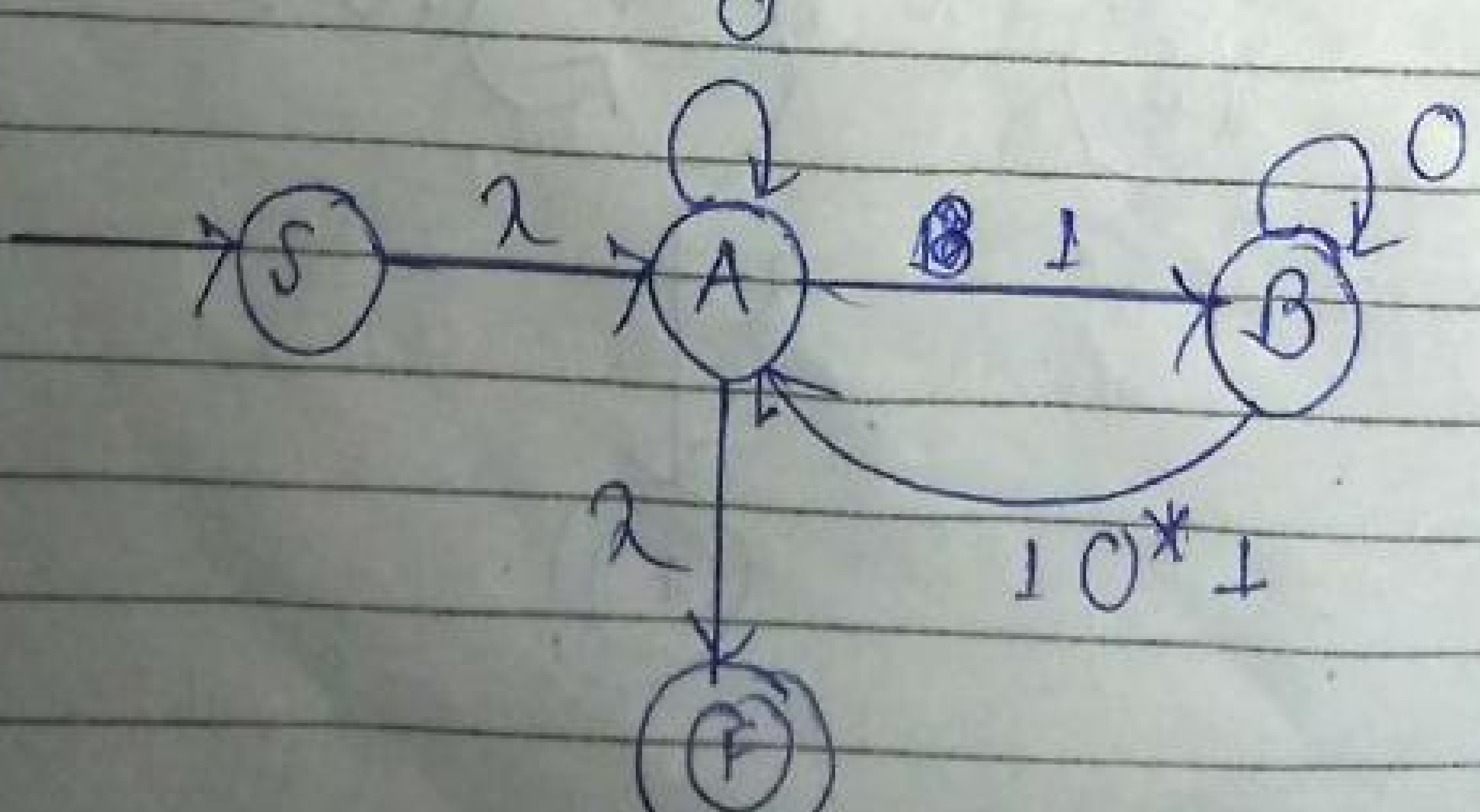
FA:



step 1:



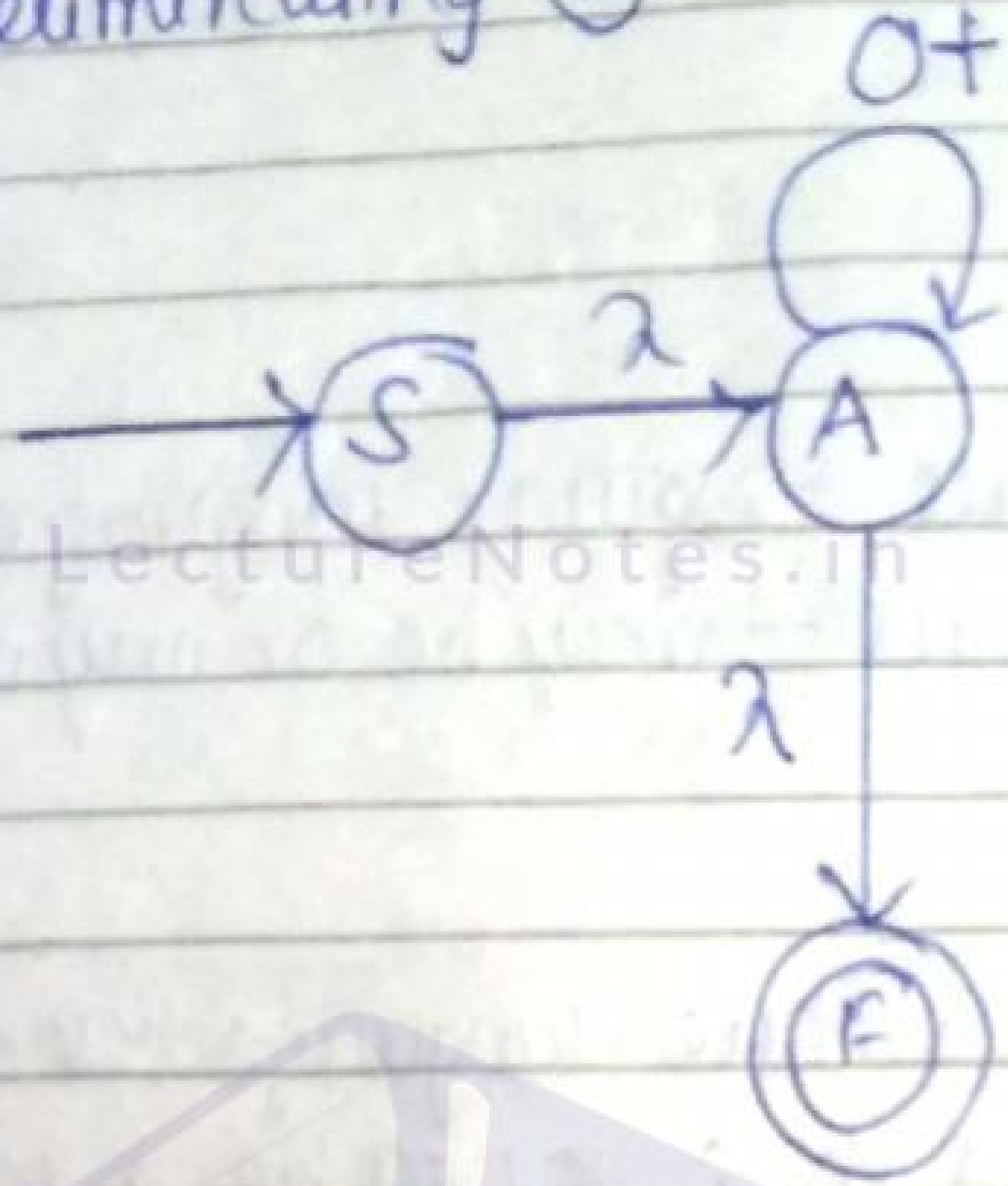
step 2: eliminating C



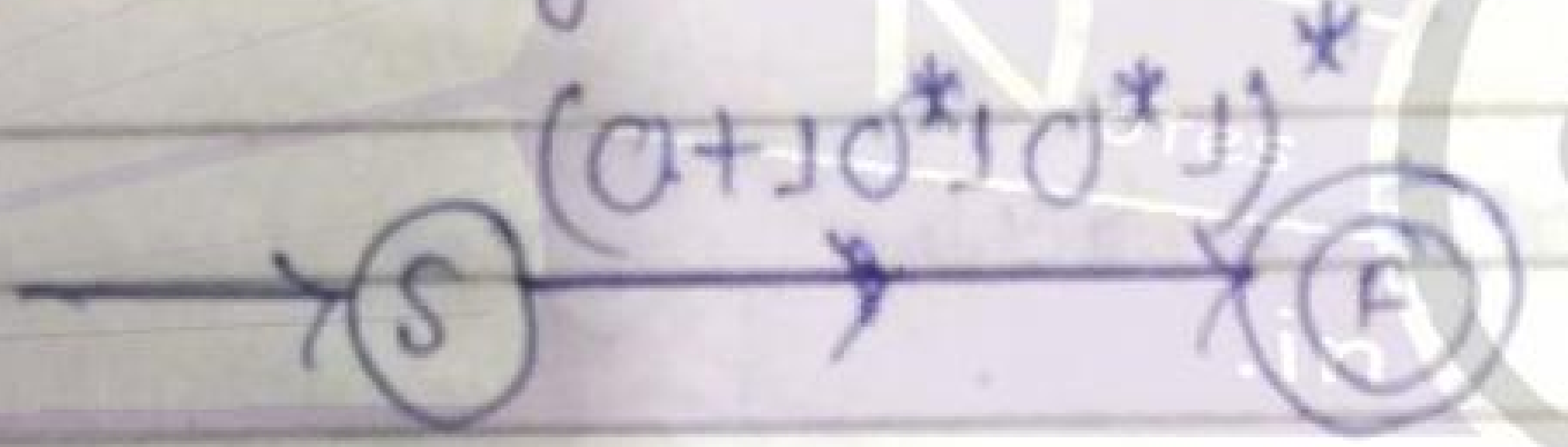
LectureNotes.in

Scanned by CamScanner

step3 eliminating B



step4 eliminating A



RE: $(10^*10^*1+0)^*$

LectureNotes.in

Scanned by CamScanner

eg: $L = \{$set of all strings of length 2.
construct a grammar

i.e. $L = \{aa, ab, ba, bb\}$

$S \to aa/ab/ba/bb$ OR $S \to \underbrace{(a+b)}_{A}\underbrace{(a+b)}_{A}$

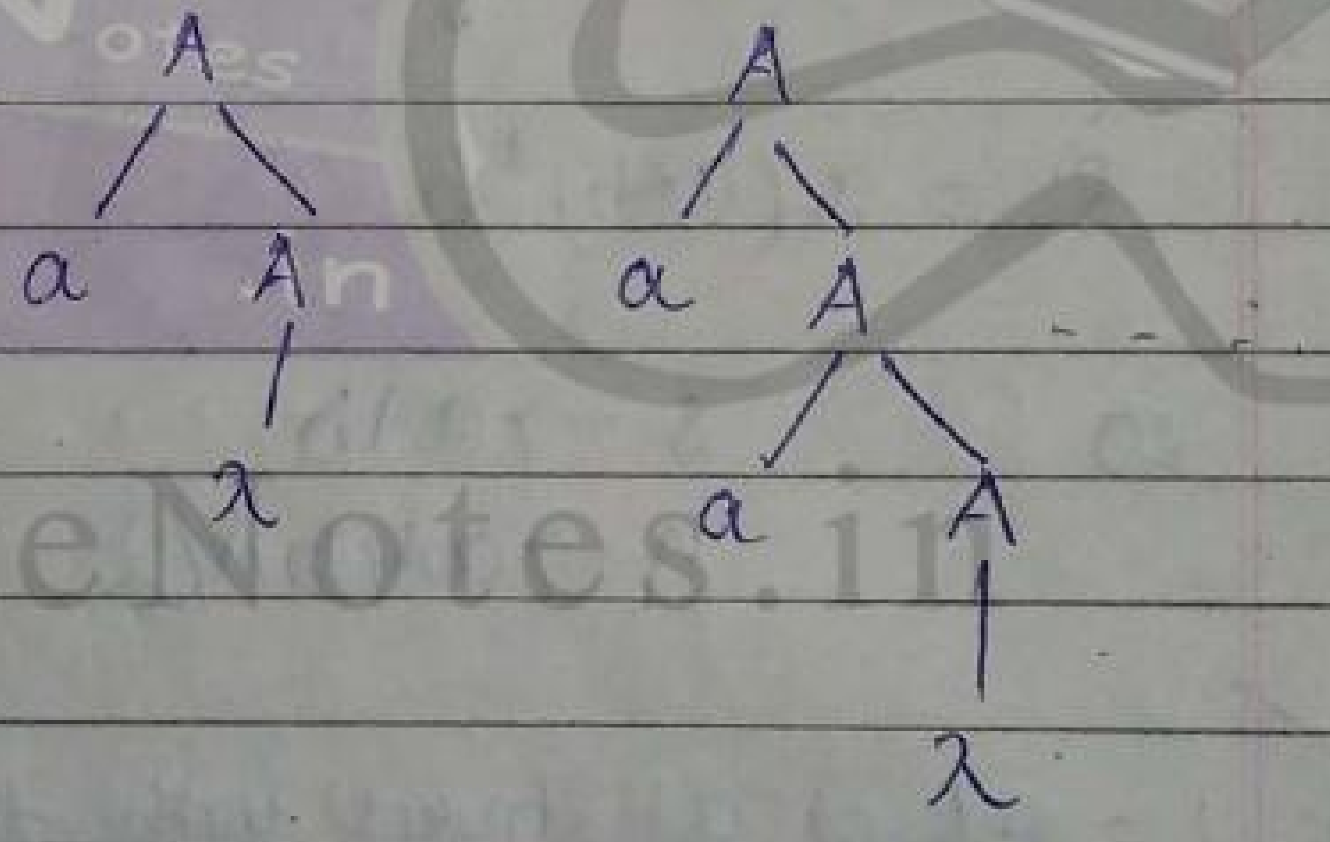
OR

$S \to AA$
$A \to a/b$

eg: $L = \{a^n : n \geq 1\}$

i.e. $L = \{a, aa, aaa, \ldots\}$

so $A \to aA/\lambda \longrightarrow$ represents a^*

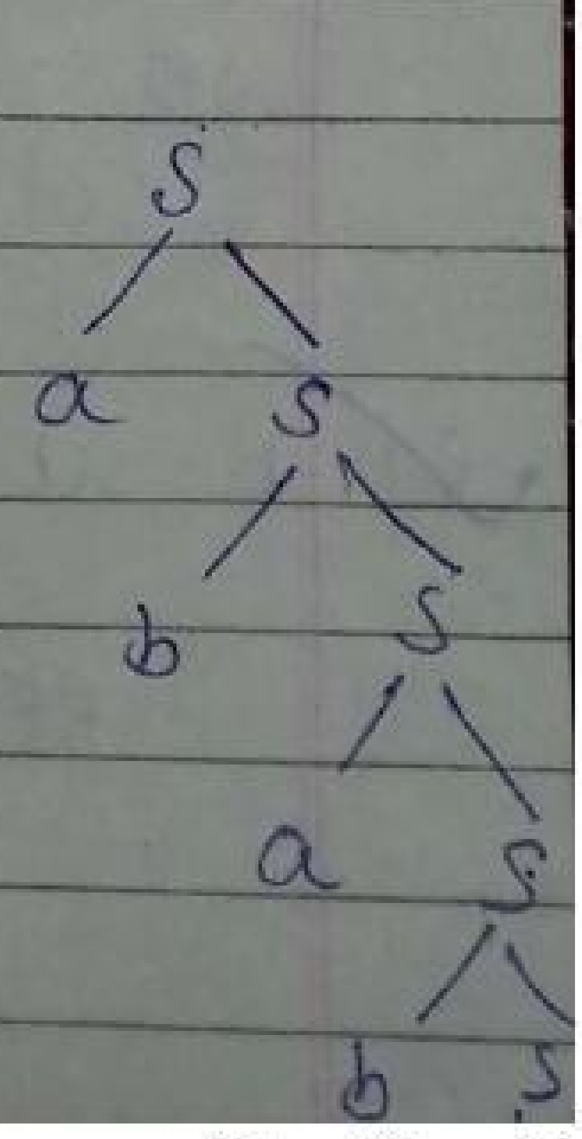


* eg: $L = \{$set of all strings containing a, b$\}$

$L = (a+b)^*$

$\hookrightarrow S \to aS/bS/\lambda$

say: abab



LectureNotes.in

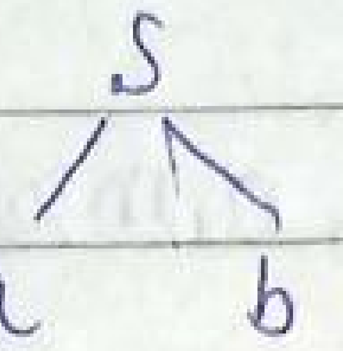
Scanned by CamScanner

SO $S \rightarrow aAa / bAb / a / b / \lambda$

$A \rightarrow aA / bB / \lambda$

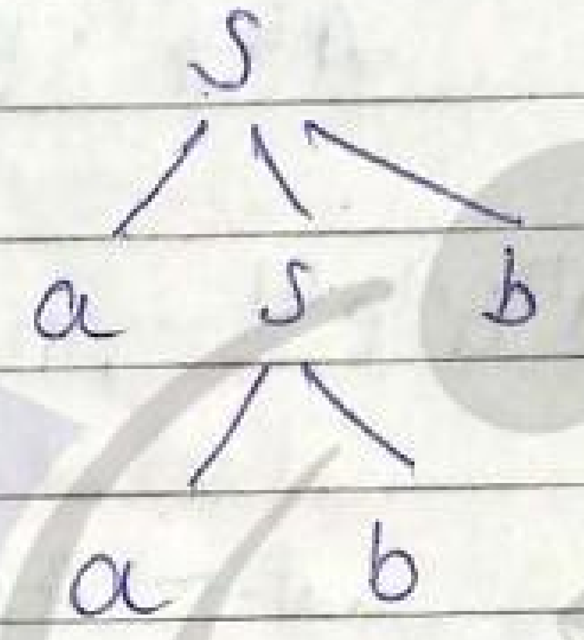
eg: $L = \{a^n b^n, n \geq 1\}$

$S \rightarrow aSb / ab$

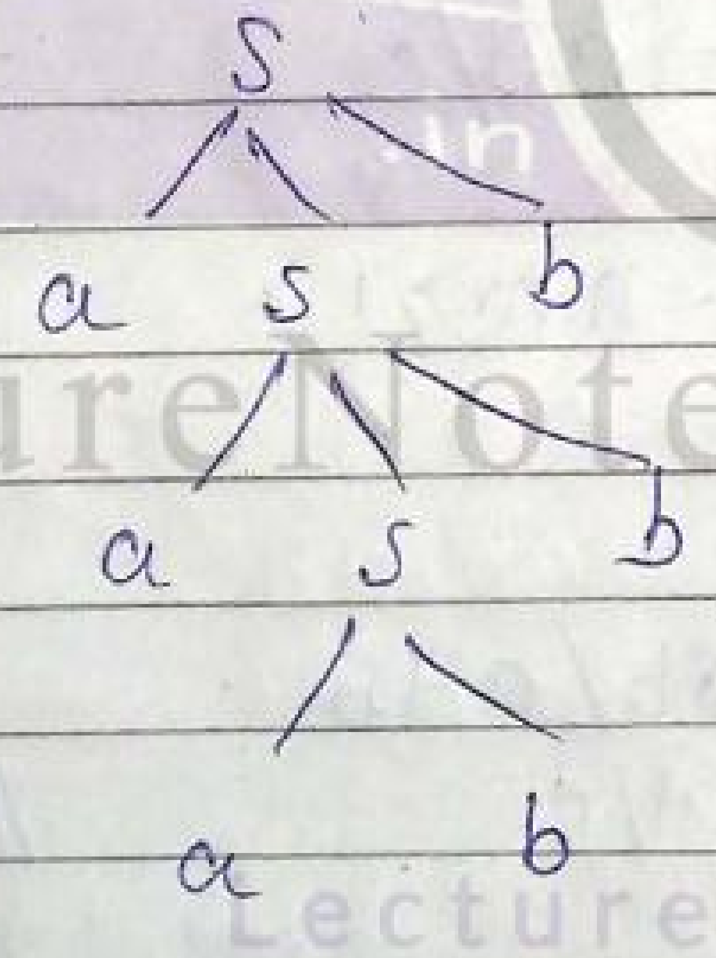


// for exact ab

if we want more than one ab say abab



say for ababab



eg: L = {set of all palindromes}

$L = \{ww^R \cup waw^R \cup wbw^R, w \in (a,b)^*$

$\underbrace{ww^R}_{\text{even}}$ $\underbrace{waw^R}_{\text{odd}}$ $\underbrace{wbw^R}_{\text{odd}}$

SO

$S \rightarrow aSa / bSb / a / b / \lambda$

LectureNotes.in

✓ **eg:** $L = \{a^n b^n c^m d^m,\ n,m \geq 1\}$

so $S \to AB$

$A \to aAb / ab$ // derives $a^n b^n$

$B \to cBd / cd$ // derives $c^n d^n$

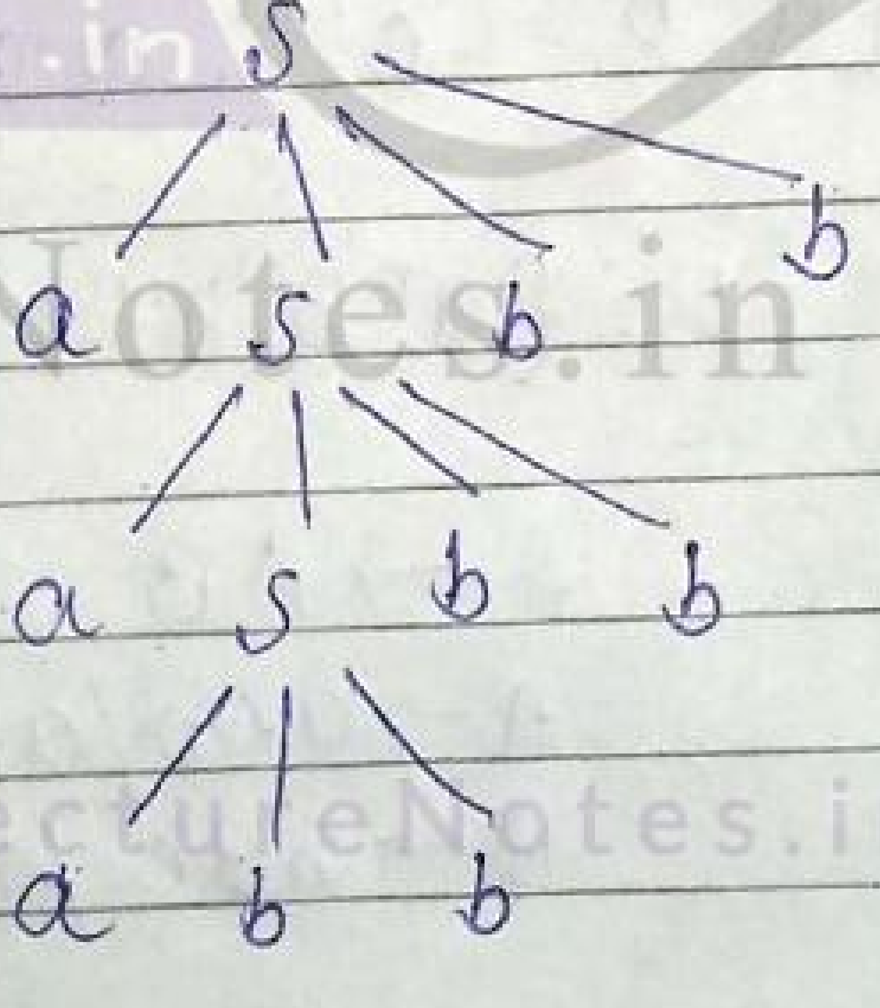
X **eg:** $L = \{a^n b^n c^n,\ n \geq 1\}$

so $S \to aSbc / abc$ X not possible bcs it will generate $a^n (bc)^n$

eg: $L = \{a^n b^{2n},\ n \geq 1\}$

so $S \to aSbb / abb$

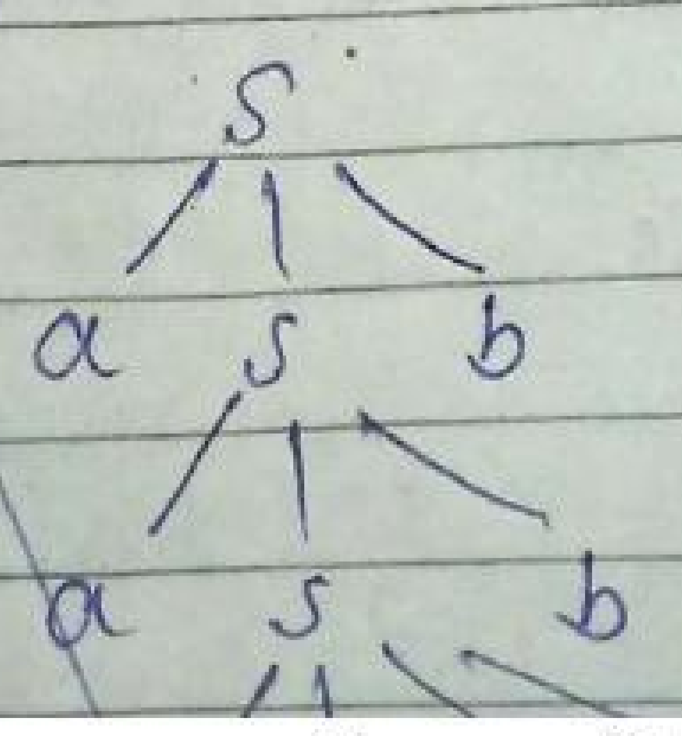
for say $L = aaabbbbbb$



eg: $L = \{a^n b^n,\ n \text{ is even}\}$

so $S \to aSb /$ ~~ab~~ $aabb$

say aaaabbbb



Scanned by CamScanner

*eg: $L = \{a^n b^m , n > m , n, m \geq 0\}$

i.e. $S \to AB$
$B \to aBb \mid \lambda$
$A \to aA / a$
→ because $n > m$ strictly greater

*eg: $L = \{a^n b^m , n < m \quad n, m \geq 0\}$

i.e. $S \to AB$
$A \to aAb \mid \lambda$
$B \to bB / b$

*eg: $L = \{a^n b^m , n \neq m , n, m \geq 0\}$

$S \to AB$
$B \to aBb \mid \lambda$
$A \to aA \mid a$

OR

$S \to PQ$
$P \to aPb \mid \lambda$
$Q \to bB \mid \lambda$

so $S \to AB \mid PQ$

eg: $L = \{a^n b^n : n$ is an even number$\}$

$S \to aaSbb \mid \lambda$

eg: $L = \{a^n b^n : n$ is an odd number$\}$

i.e. $S \to aAb$
$A \to aaAbb \mid \lambda$

OR

$S \to aaSbb \mid ab$

LectureNotes.in

Scanned by CamScanner

eg: $L = \{a^n b^m c^n : n, m \geq 0\}$

i.e. $S \rightarrow$ ~~A~~ $aSC \mid A \mid \lambda$

$A \rightarrow bA \mid \lambda$

~~B~~

Types of Grammar

(a) Type 3 (Right linear) → variables in right side

RLG if $A \rightarrow \alpha\boxed{B} \mid \beta$

where $A, B \in V$

$\alpha, \beta \in T^*$

OR

Type 3 (Left Linear) → left side

LLG if $A \rightarrow \boxed{B}\alpha \mid \beta$

where $A, B \in V$

$\alpha, \beta \in T^*$

eg: $A \rightarrow aB \mid a$

$B \rightarrow aB \mid bB \mid a \mid b$ $\longrightarrow$ RLG

eg: $A \rightarrow Ba \mid a$

$B \rightarrow Ba \mid Bb \mid a \mid b$ $\longrightarrow$ LLG

N.B Type 3 grammars ≡ Regular grammars

LectureNotes.in

Scanned by CamScanner

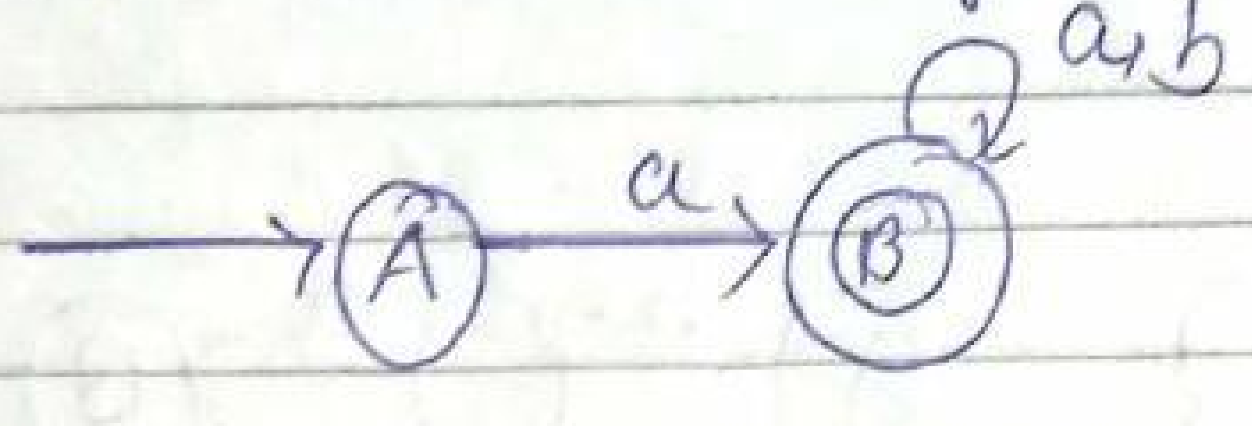
$$\underset{(L)}{FA} \longrightarrow \underset{(L)}{RLG} \xrightarrow{R} \underset{(L^R)}{LLG}$$ X

N.B In order to convert FA to LLG

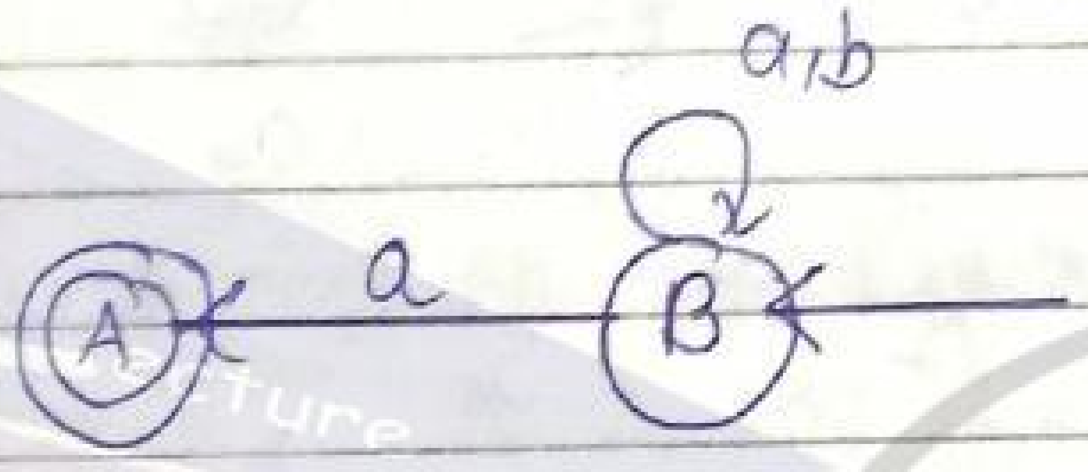
$$\underset{(L)}{FA} \xrightarrow{R} \underset{(L^R)}{FA} \longrightarrow \underset{(L^R)}{RLG} \xrightarrow{R} \underset{(L^R)^R = L}{LLG}$$

eg: find the LLG for all strings with starting a

FA
(L)



FA
(L^R)



RLG
(L^R)

$B \rightarrow aB \mid bB \mid aA$
$A \rightarrow \lambda$

LLG
(L)

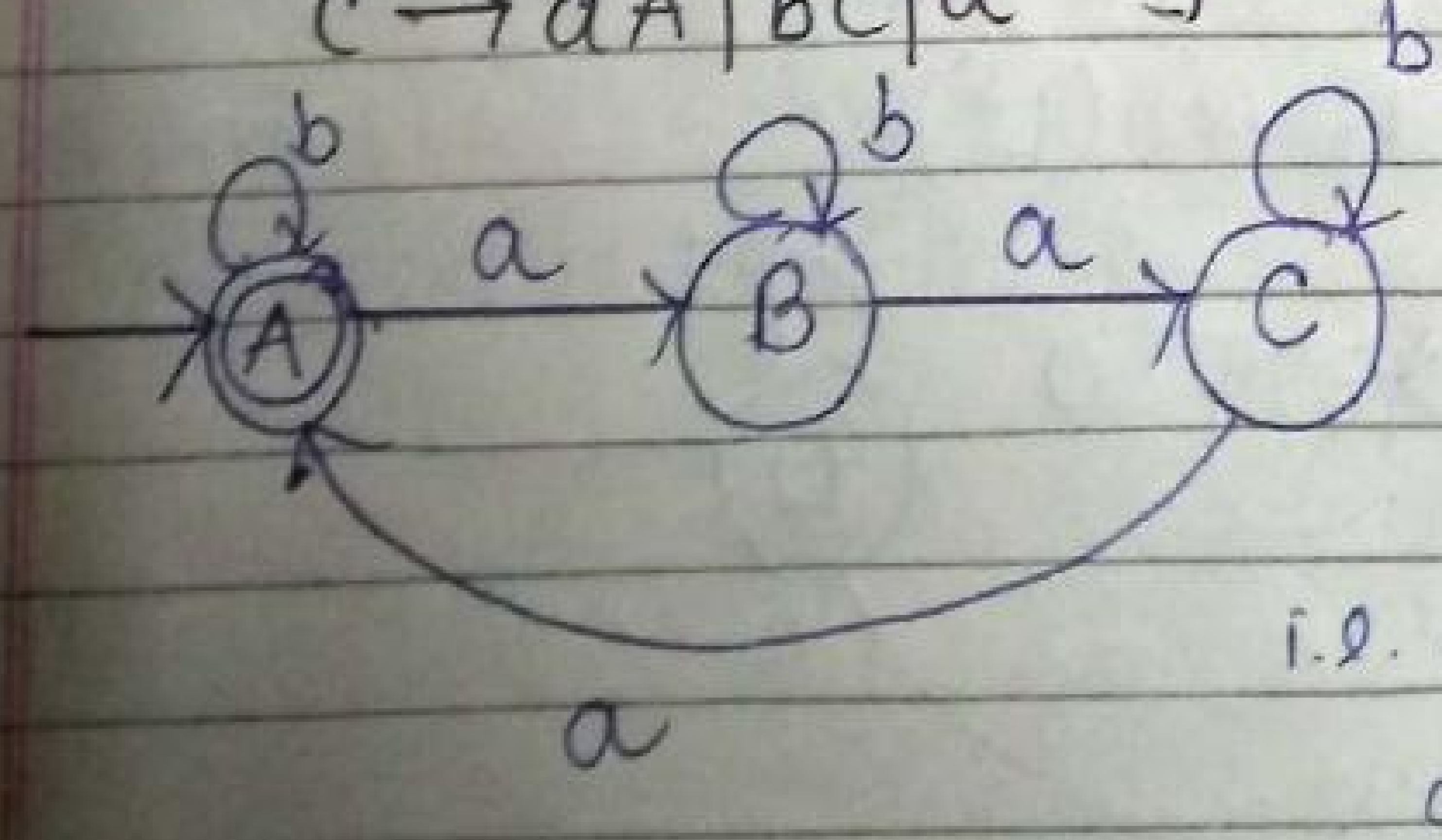
$B \rightarrow Ba \mid Bb \mid Aa$
$A \rightarrow \lambda$

LectureNotes.in

RG to FA

eg:

$$\left.\begin{array}{l} A \rightarrow aB \mid bA \mid b \\ B \rightarrow aC \mid bB \\ C \rightarrow aA \mid bC \mid a \end{array}\right\} RLG$$



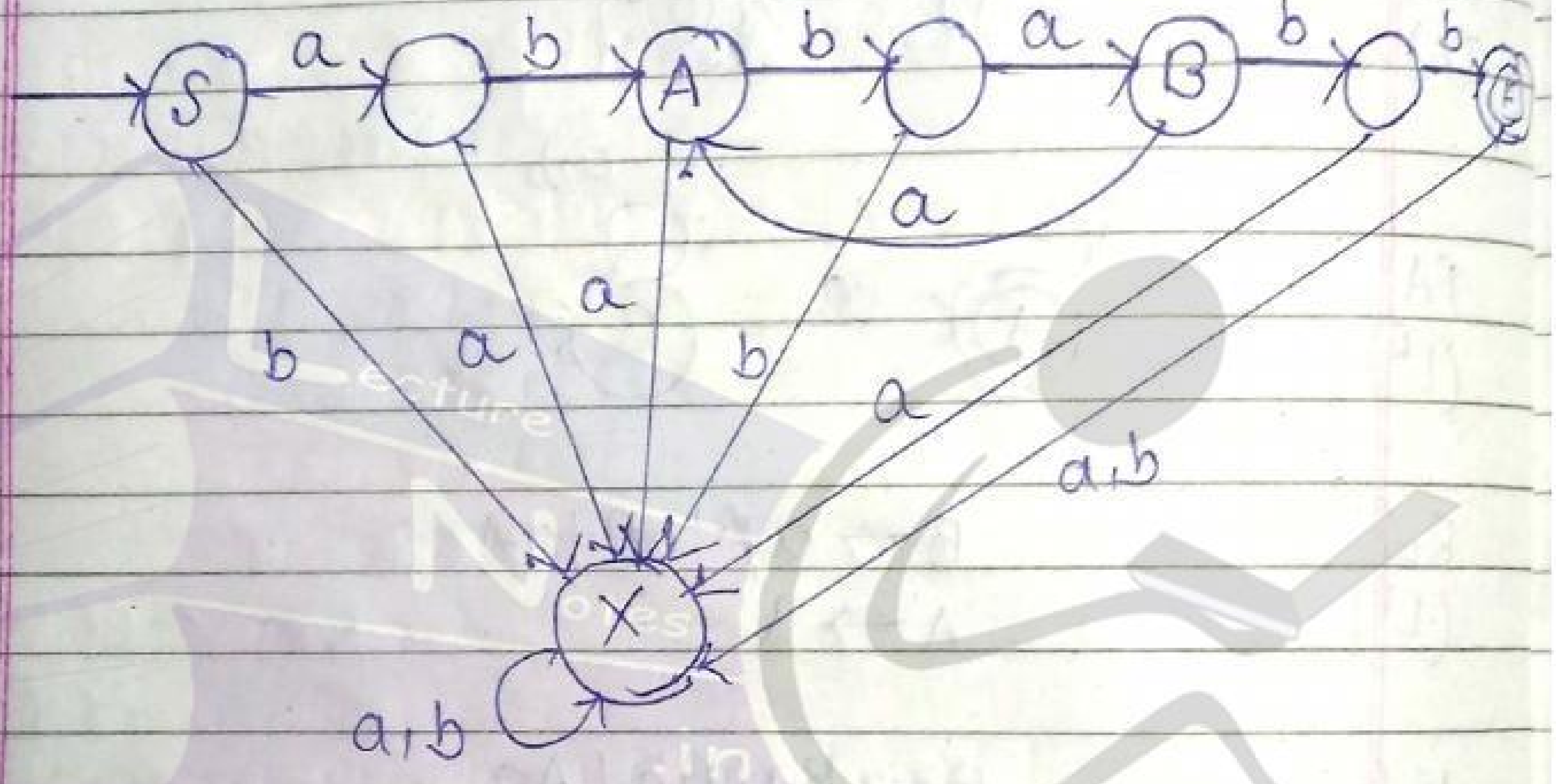
i.e. no' of a's are divisible by 3

Scanned by CamScanner

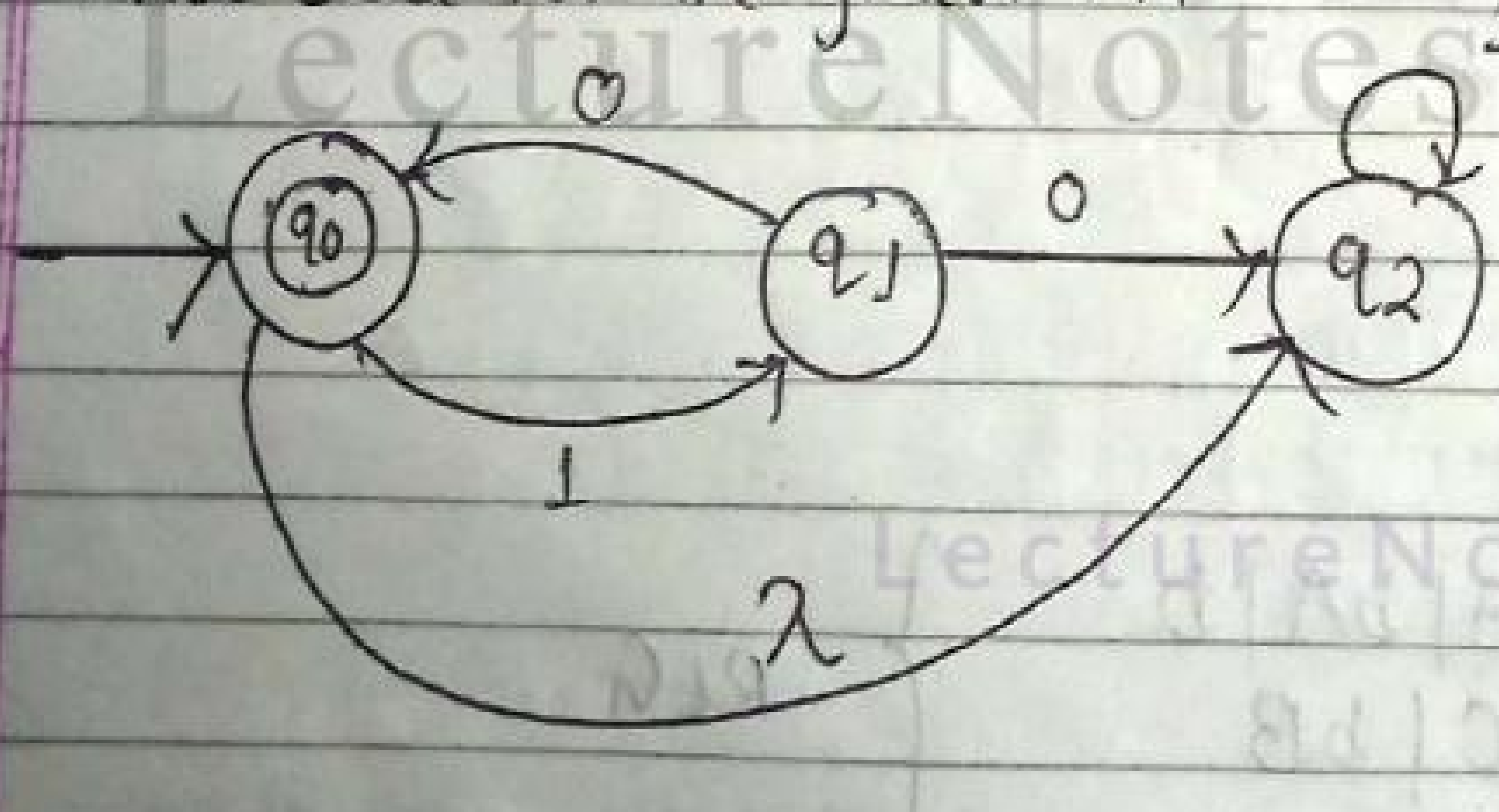
N.B $\underset{(L)}{LLG} \xrightarrow{R} \underset{(L^R)}{RLG} \longrightarrow \underset{(L^R)}{FA} \xrightarrow{R} \underset{(L^R)^R}{FA}$

eg: construct DFA for

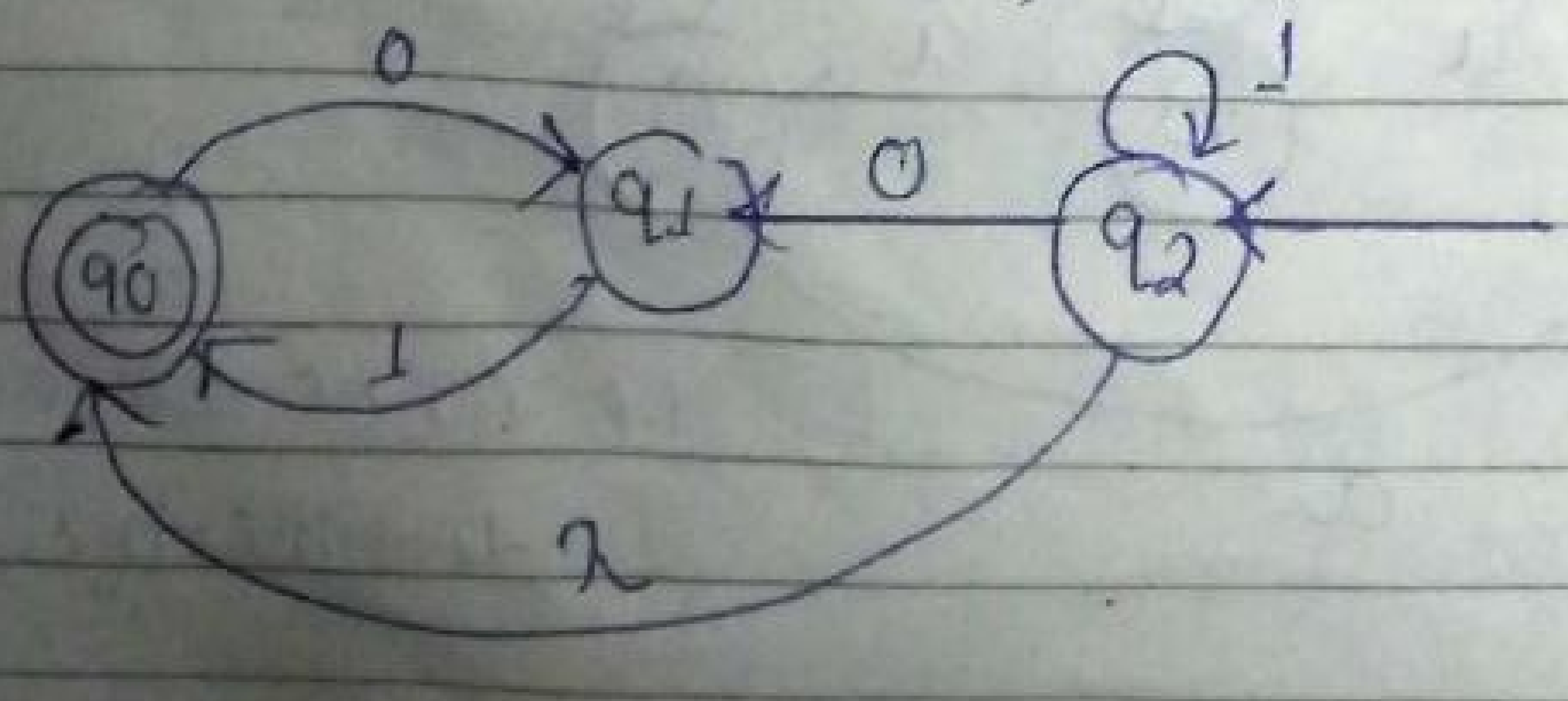
$S \rightarrow abA$
$A \rightarrow baB$
$B \rightarrow aA \mid bb$



eg: Find LLG for the given FA

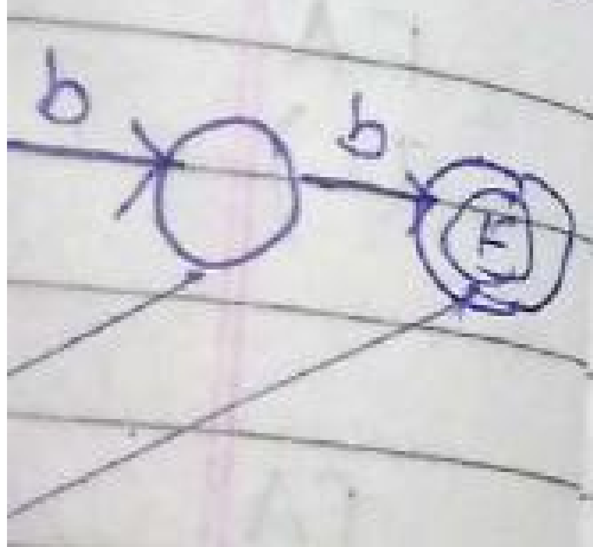


Step1: Find Reverse of FA i.e. (L^R)



→ FA

$(L^R)^R = L$



step 2: Find RLG i.e. L^R

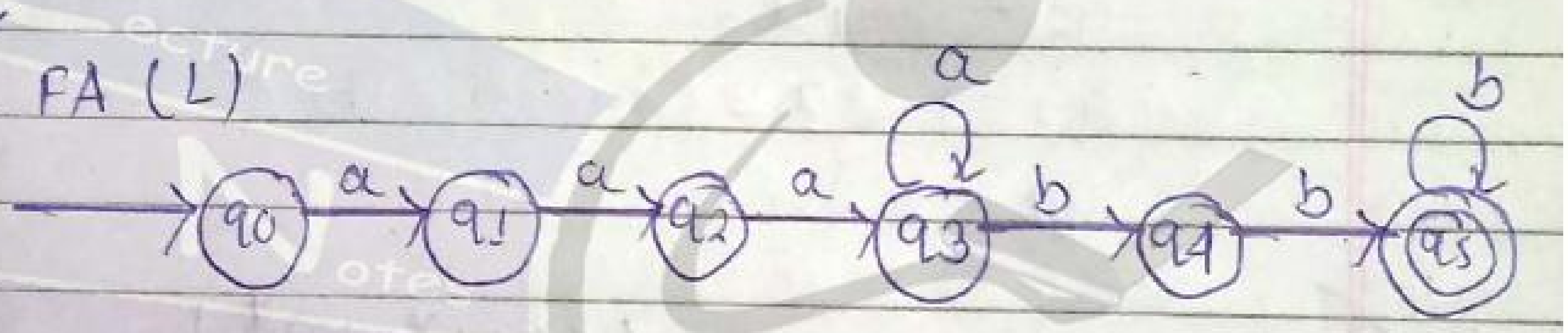
$q_0 \rightarrow 0q_1 | \lambda$
$q_1 \rightarrow 1q_0$
$q_2 \rightarrow 1q_2 | 0q_1 | \lambda q_0$

step 3: Find reverse of RLG i.e. LLG i.e. $(L^R)^R \equiv L$

$q_0 \rightarrow q_1 0 | \lambda$
$q_1 \rightarrow q_0 1$
$q_2 \rightarrow q_2 1 | q_1 0 | q_0 \lambda$

eg: construct right and left linear grammar for $L = \{a^n b^m : n \geq 3, m \geq 2\}$

step 1: • Find FA (L)

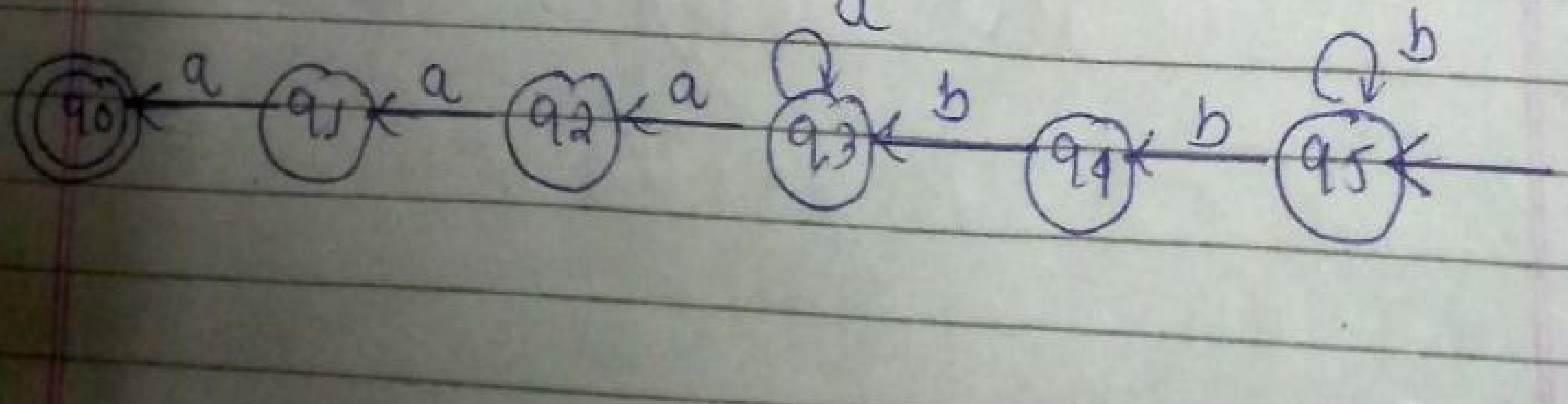


• Find RLG for L

$q_0 \rightarrow aq_1$
$q_1 \rightarrow aq_2$
$q_2 \rightarrow aq_3$
$q_3 \rightarrow aq_3 | bq_4$
$q_4 \rightarrow bq_5$
$q_5 \rightarrow bq_5 | \lambda$

now to find LLG

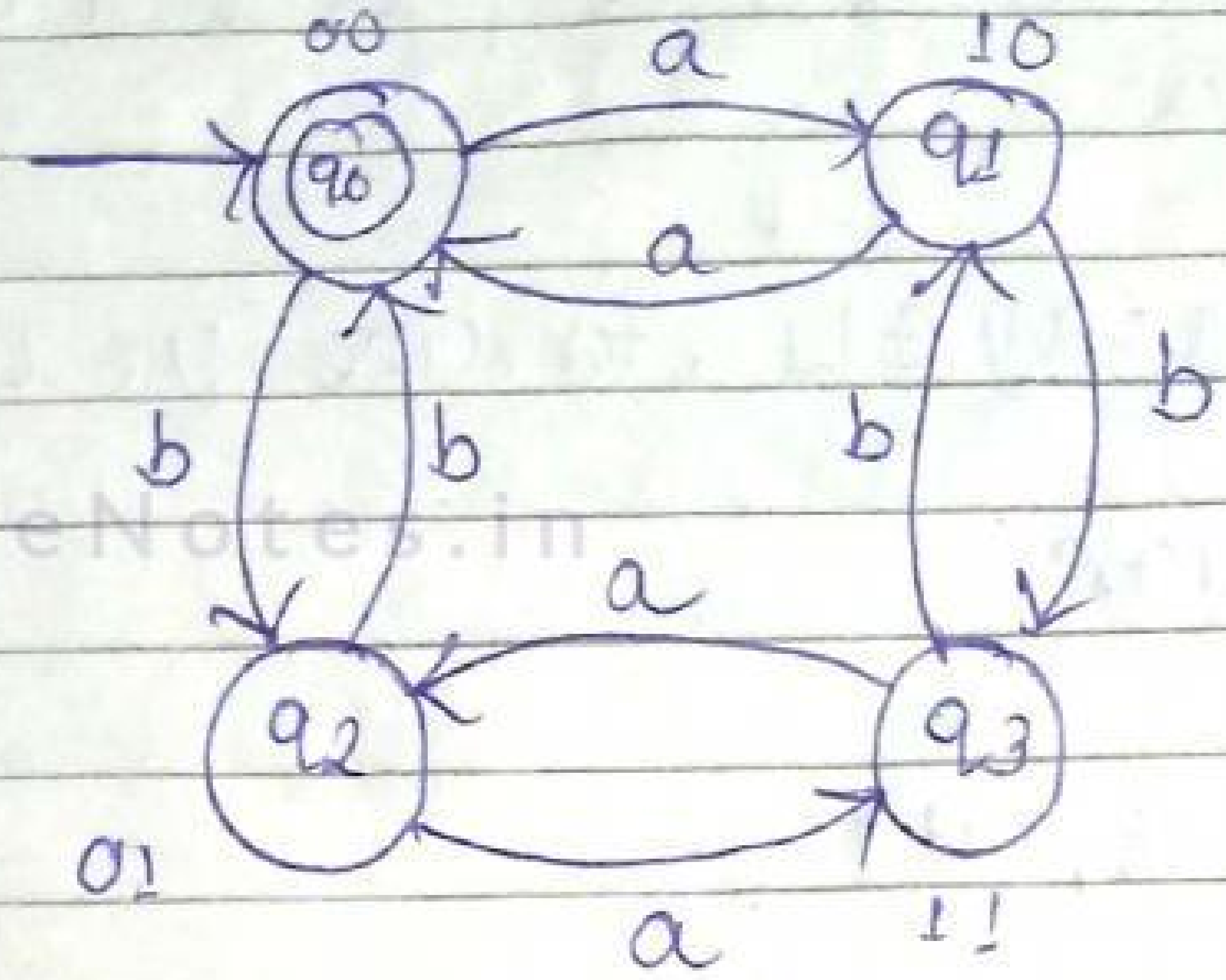
step 1: Find reverse of FA i.e. L^R



Scanned by CamScanner

eg. write a regular grammar for

$L = \{w : n_a(w) \text{ and } n_b(w) \text{ are even}\}$



RLG

$q_0 \to aq_1 \mid bq_2 \mid \lambda$
$q_1 \to aq_0 \mid bq_3$
$q_2 \to aq_3 \mid bq_0$
$q_3 \to aq_2 \mid bq_1$

(b) Type 2 (context free Grammars) or (CFL)

If all productions are in the form
$A \to \alpha$ where $A \in V$
$\alpha \in (V \cup T)^*$

N.B if grammar is regular $\longrightarrow$ context free grammar
but the
reverse may or may not be true

N.B PDA is the machine which is used to accept these languages

LectureNotes.in

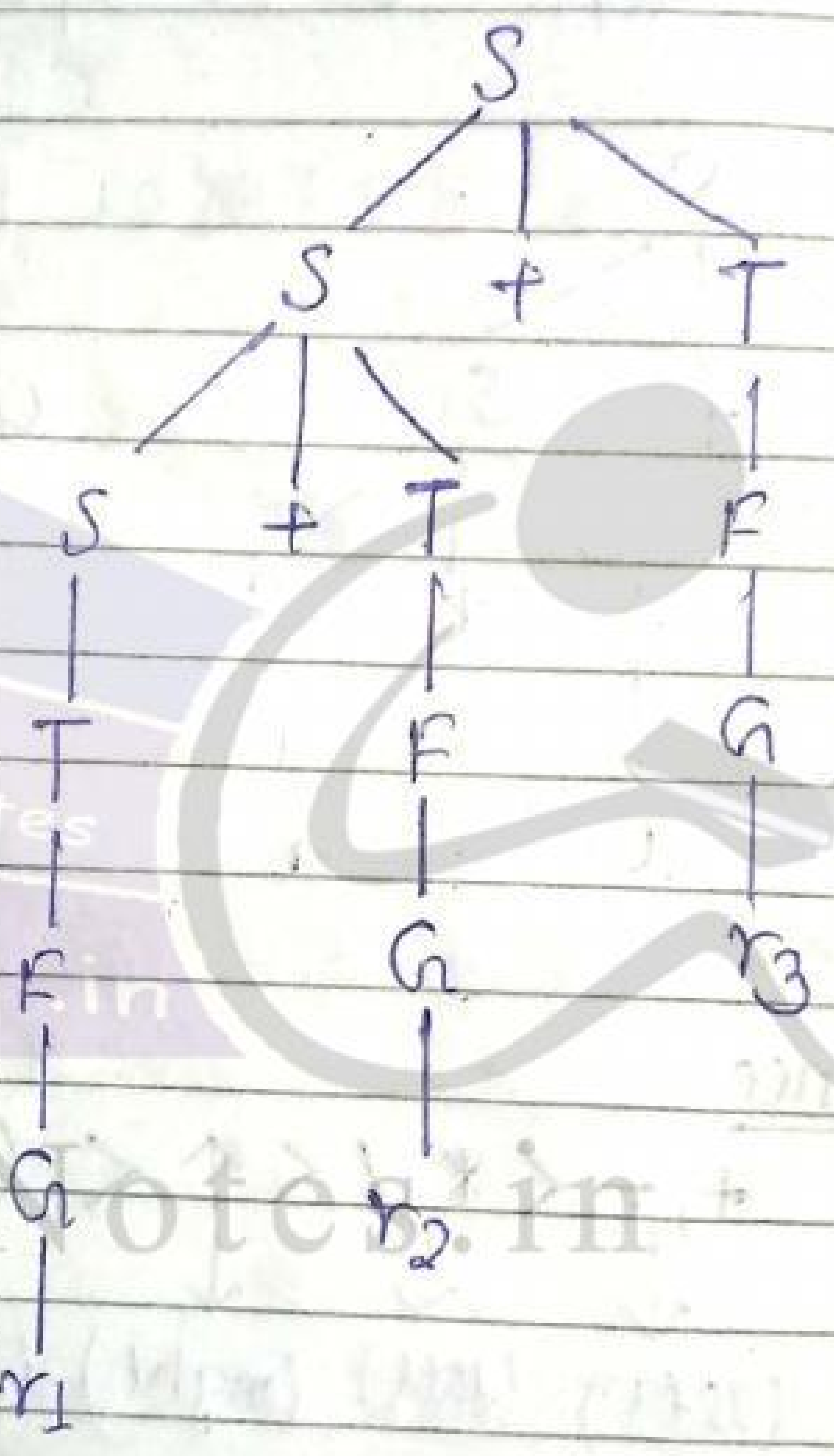
Scanned by CamScanner

eg: $S \to S+S \mid S^* \mid S.S \mid (S) \mid r_1 \mid r_2 \mid r_3$
convert to unambiguous.

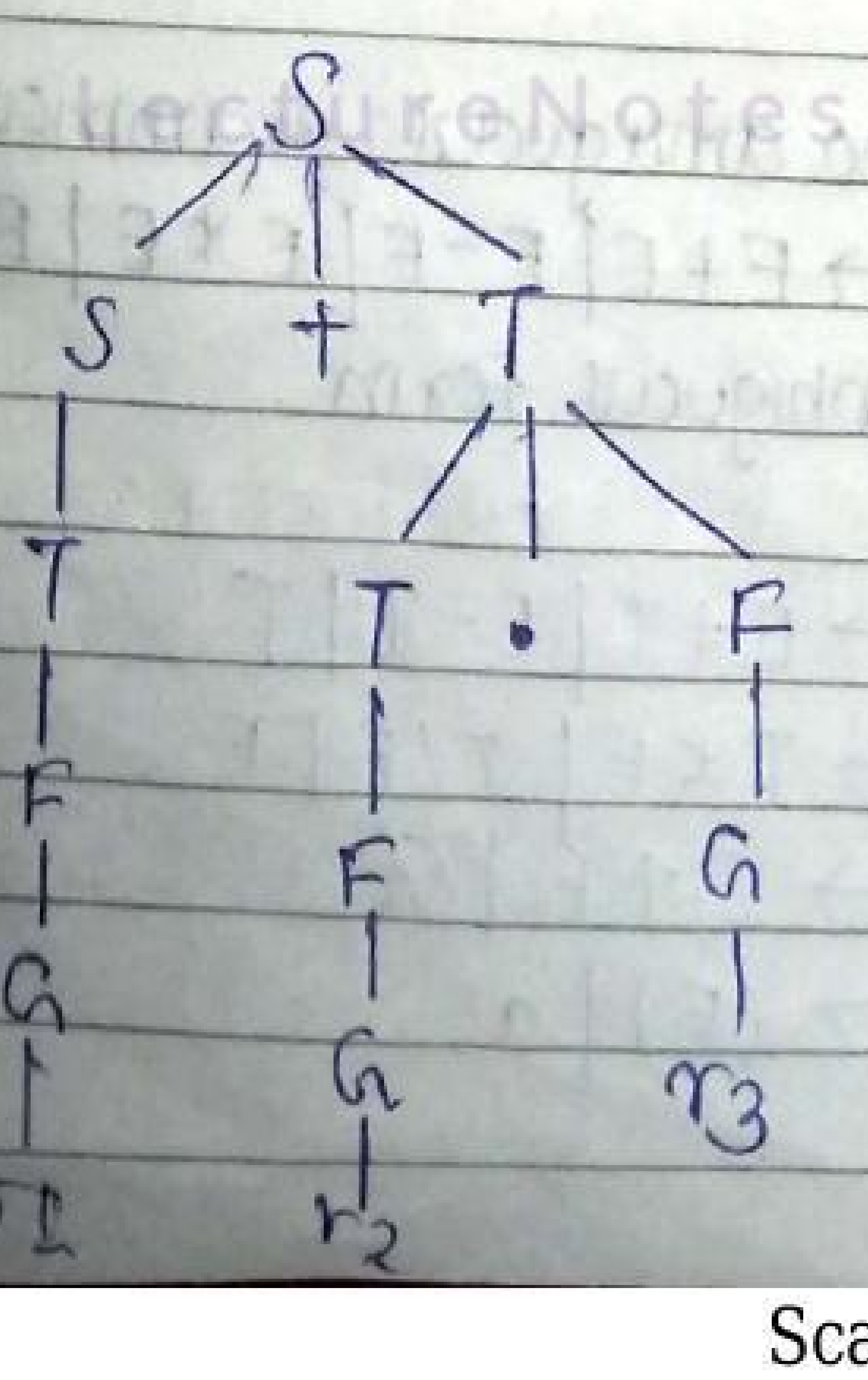
Precedency: $+ < . < * < (\,) < r_1, r_2, r_3$
associativity: left left left

$S \to S+T / T$
$T \to T.F / F$
$F \to F^* \mid G$
$G \to (S) \mid r_1 \mid r_2 \mid r_3$

say: $r_1+r_2+r_3$



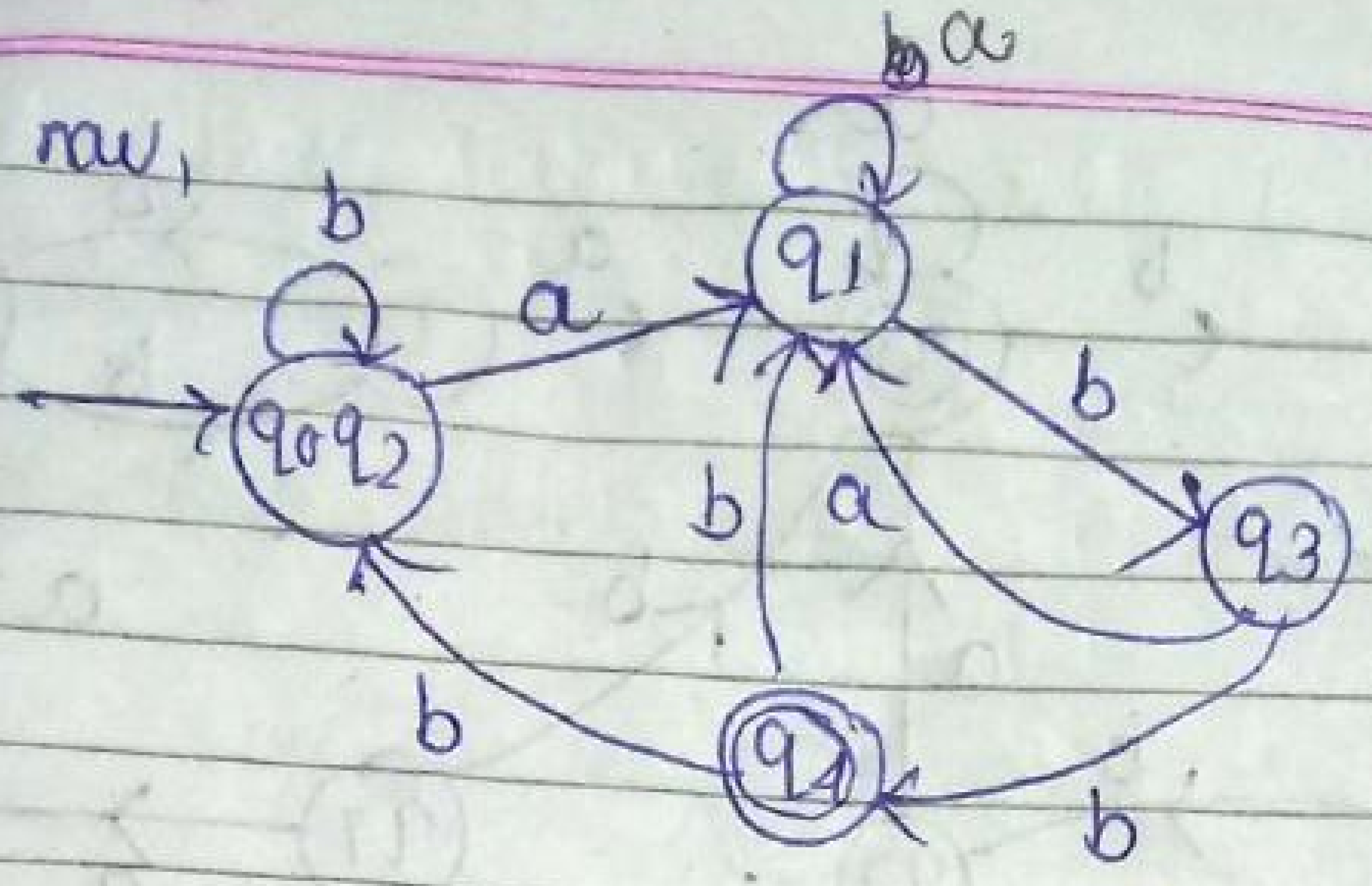
say: $(r_1+r_2r_3)^*$



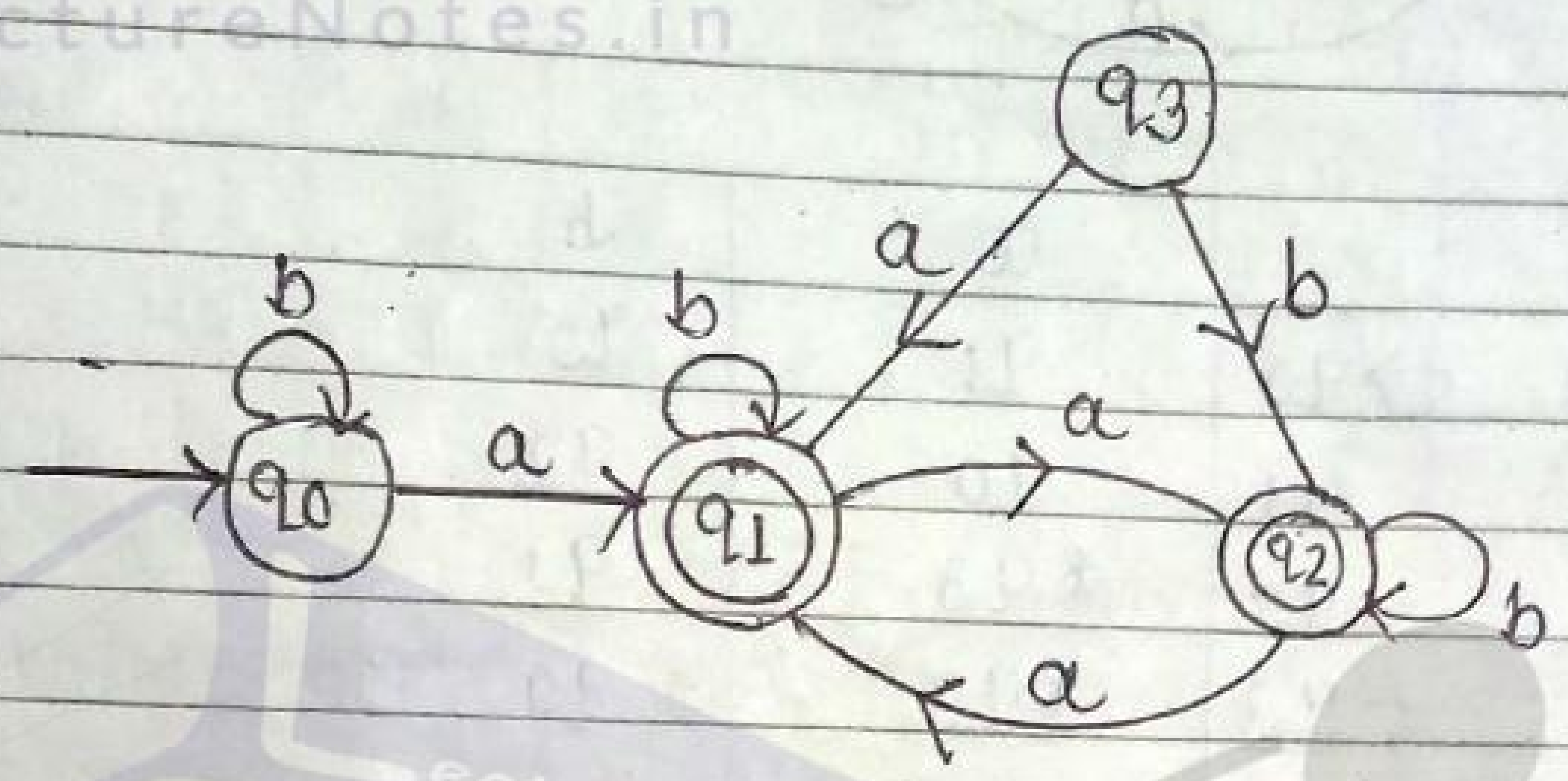
LectureNotes.in

Scanned by CamScanner

minimised so now, DFA



eg:



sol:

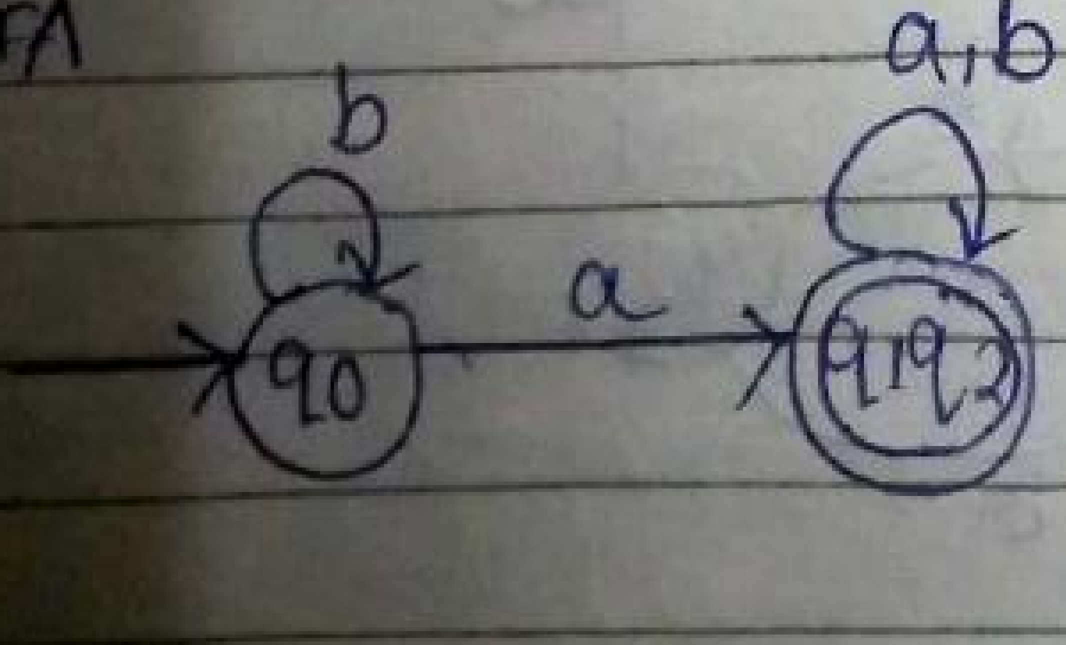
	a	b
→q0	*q1	q0
*q1	*q2	*q1
*q2	*q1	*q2
~~q3~~	~~*q1~~	~~*q2~~

q3 is not reachable so delete it

Step1: 0 equivalent states : [q0] [q1 q2]

Step2: 1 equivalent states : [q0] [q1 q2]

so minimised DFA

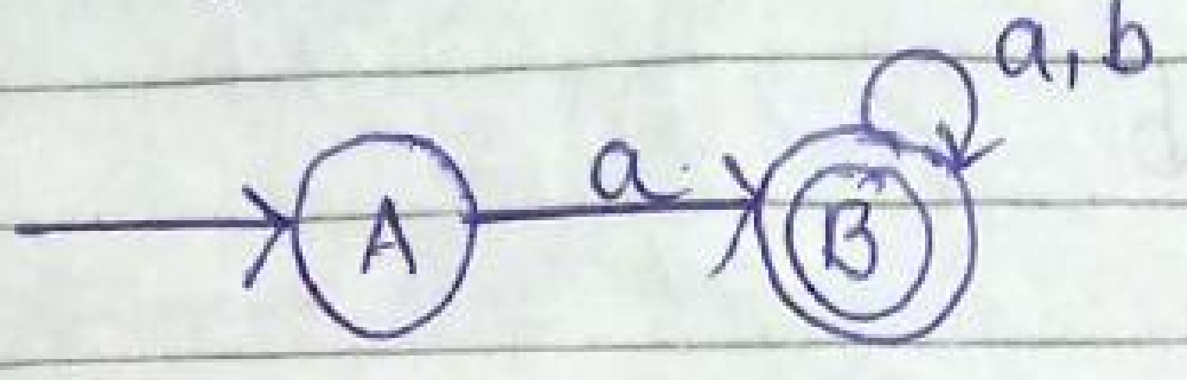


Scanned by CamScanner

NFA to DFA

eg: all strings starting with a

NFA :



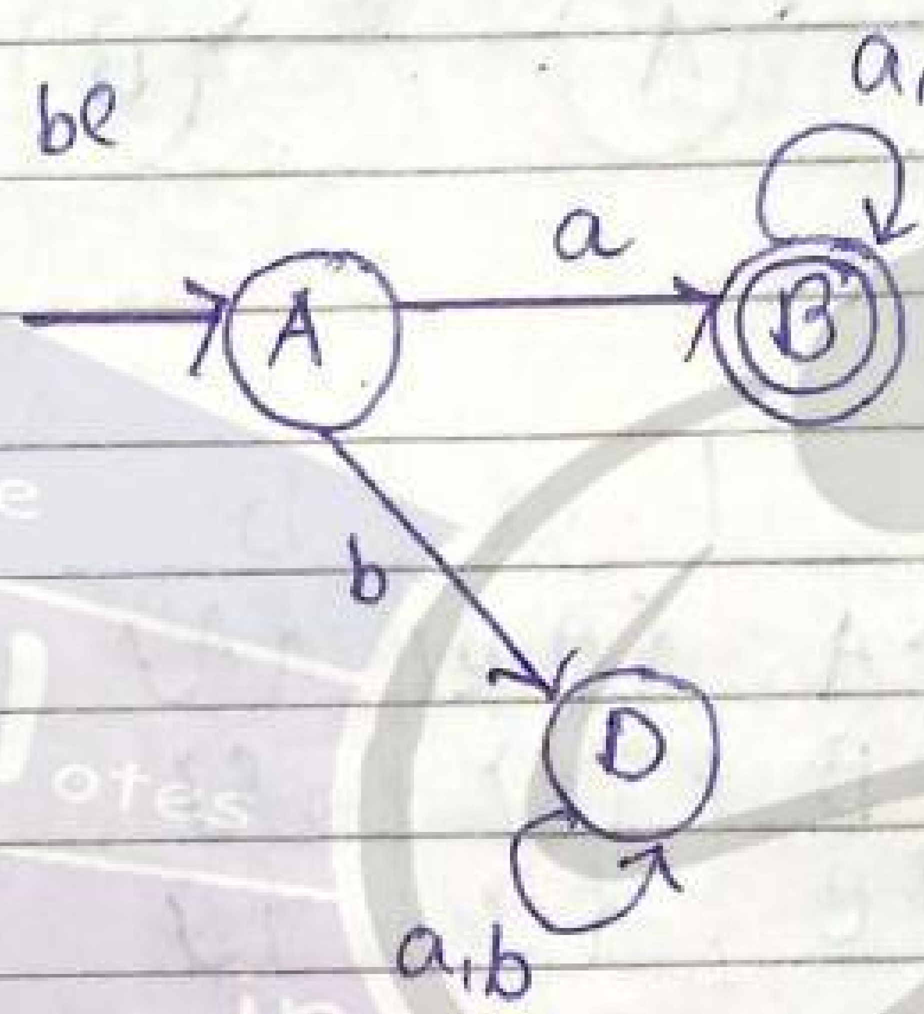
then

	a	b
→A	B	∅
*B	B	B

convert →

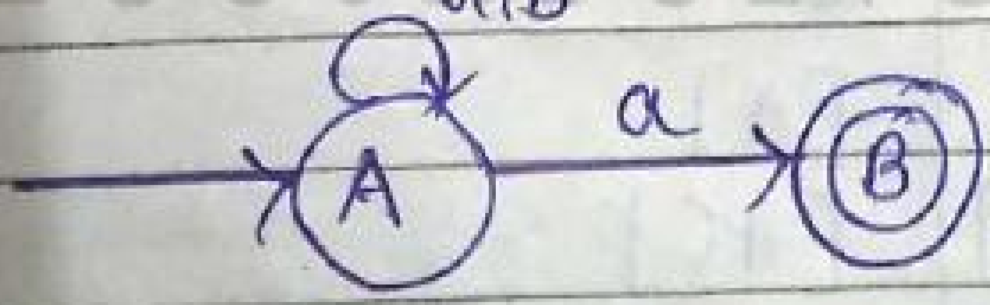
	a	b
A	B	D
B	B	B
D	D	D

so corresponding DFA will be



eg: strings ending with a

NFA :



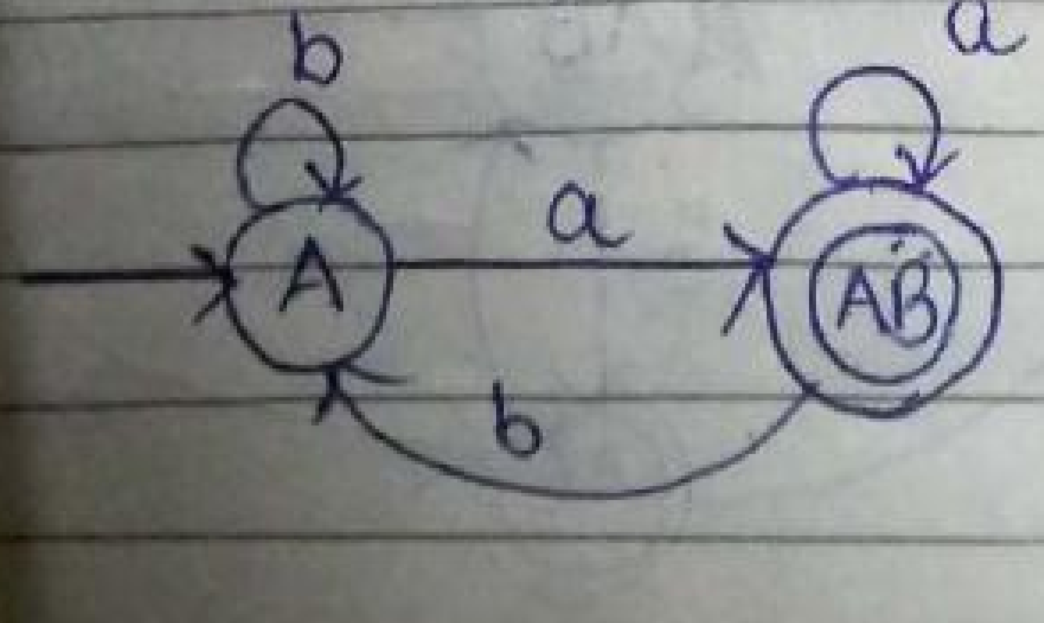
then

	a	b
→A	{A B}	{A}
*B	{ }	{ }

convert →

	a	b
→[A]	[AB]	[A]
[AB]	[AB]	[A]
[illegible]		

DFA

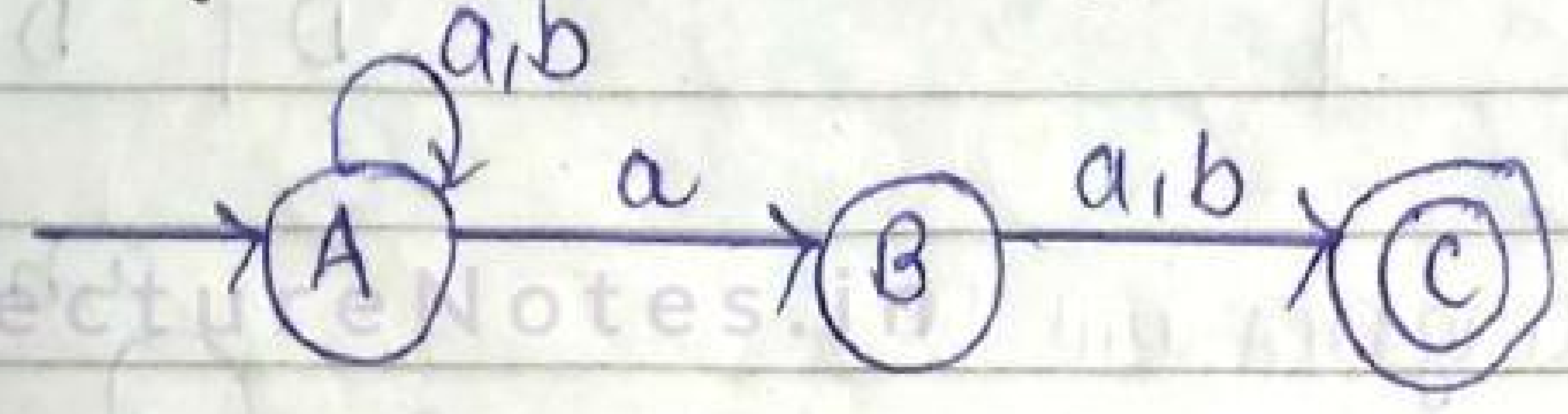


LectureNotes.in

Scanned by CamScanner

eg: all strings in which second symbol from RHS is "a"

NFA



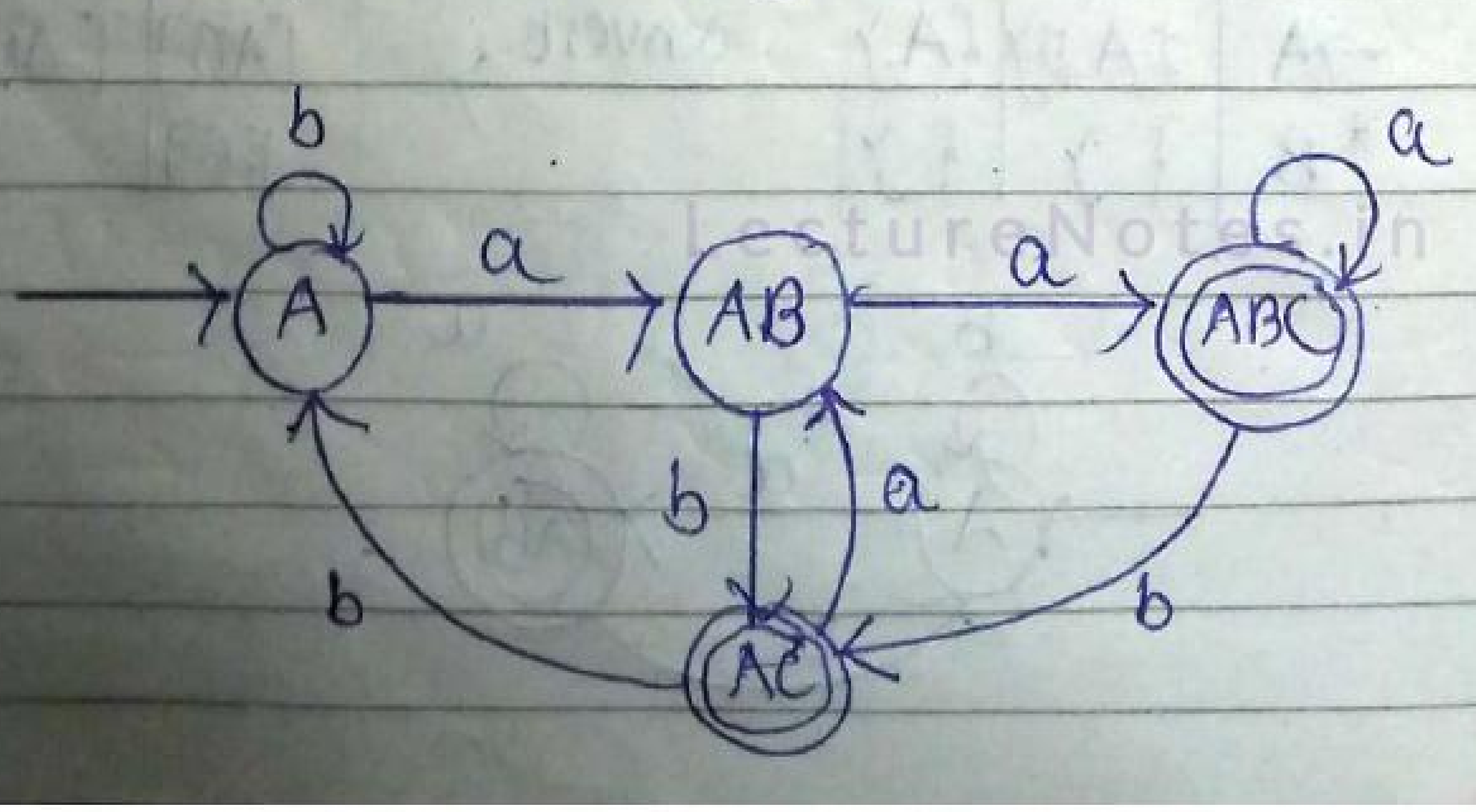
then

	a	b
→A	{A,B}	{A}
B	{C}	{C}
*C	{ }	{ }

after converting

	a	b
→[A]	[AB]	[A]
[AB]	[ABC]	[AC]
*[ABC]	[ABC]	[AC]
*[AC]	[AB]	[A]

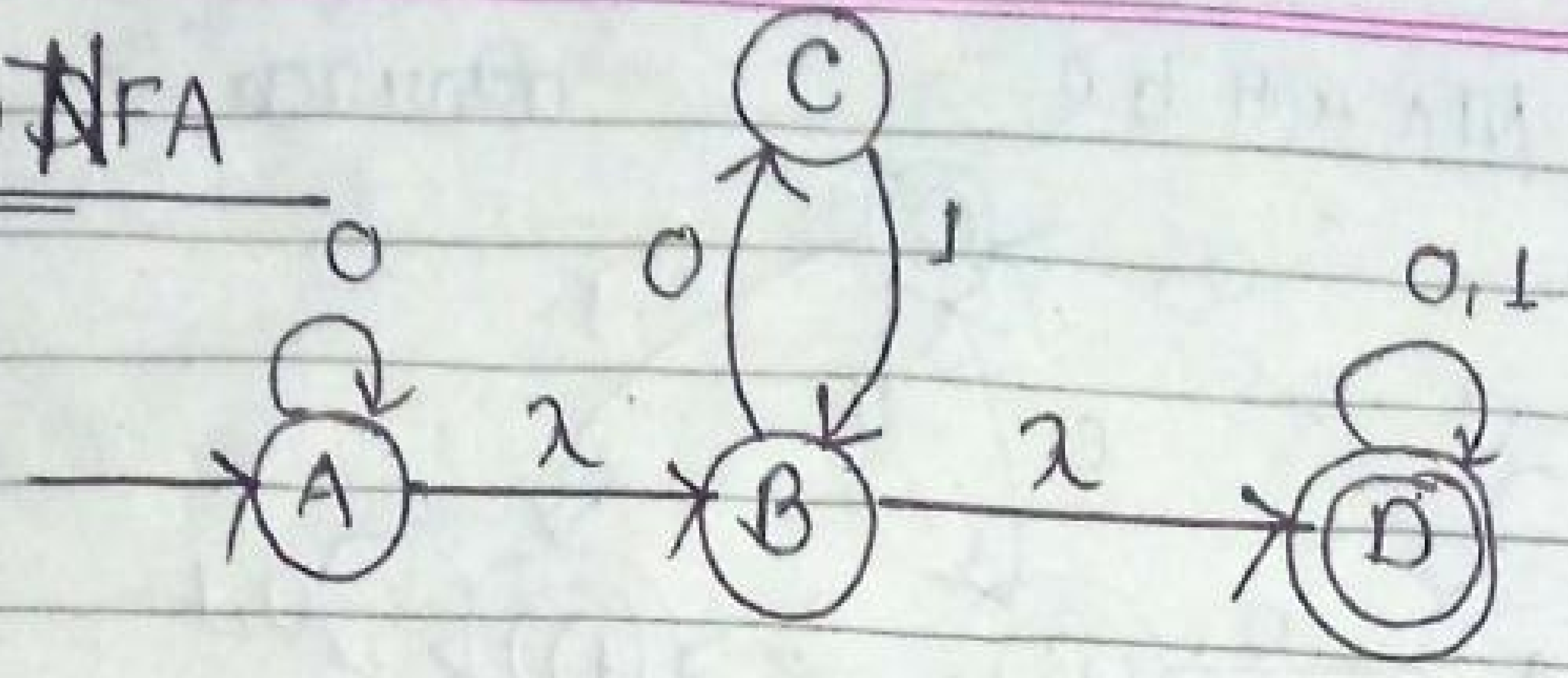
so DFA



LectureNotes.in

λNFA to NFA

eg:



sol: ~~λ closure of A = {A, B, D}~~
~~λ^* closure of A on 0 = {~~

	0	1
→ A	{A, B, C, D}	{D}
B	{C, D}	{D}
C	{∅}	{B, D}
D	{D}	{D}

$$\begin{tikzcd} A \arrow[r, "\lambda^*"] \arrow[dr] \arrow[ddr] & A \arrow[r, "0"] & A \arrow[r, "\lambda^*"] & A,B,D \\ & B \arrow[r] & C \arrow[r] & C \\ & D \arrow[r] & D \arrow[r] & D \end{tikzcd}$$

and again

$$\begin{tikzcd} A \arrow[r, "\lambda^*"] \arrow[dr] \arrow[ddr] & A \arrow[r, "1"] & \emptyset & \\ & B \arrow[r] & \emptyset & \\ & D \arrow[r] & D \arrow[r, "\lambda^*"] & D \end{tikzcd}$$

now

$$\begin{tikzcd} B \arrow[r, "\lambda^*"] \arrow[dr] & B \arrow[r, "0"] & C \arrow[r, "\lambda^*"] & C \\ & D \arrow[r] & D \arrow[r] & D \end{tikzcd}$$

now

$$\begin{tikzcd} B \arrow[r, "\lambda^*"] \arrow[dr] & B \arrow[r, "1"] & \emptyset & \\ & D \arrow[r] & D \arrow[r, "\lambda^*"] & D \end{tikzcd}$$

again

$$\begin{tikzcd} C \arrow[r, "\lambda^*"] & C \arrow[r, "0"] & \emptyset \arrow[r, "\lambda^*"] & \emptyset \end{tikzcd}$$

$$\begin{tikzcd} C \arrow[r, "\lambda^*"] & C \arrow[r, "1"] & B \arrow[r, "\lambda^*"] \arrow[dr] & B \\ & & & D \end{tikzcd}$$

again

$$\begin{tikzcd} D \arrow[r, "\lambda^*"] & D \arrow[r, "0"] & D \arrow[r, "\lambda^*"] & D \end{tikzcd}$$

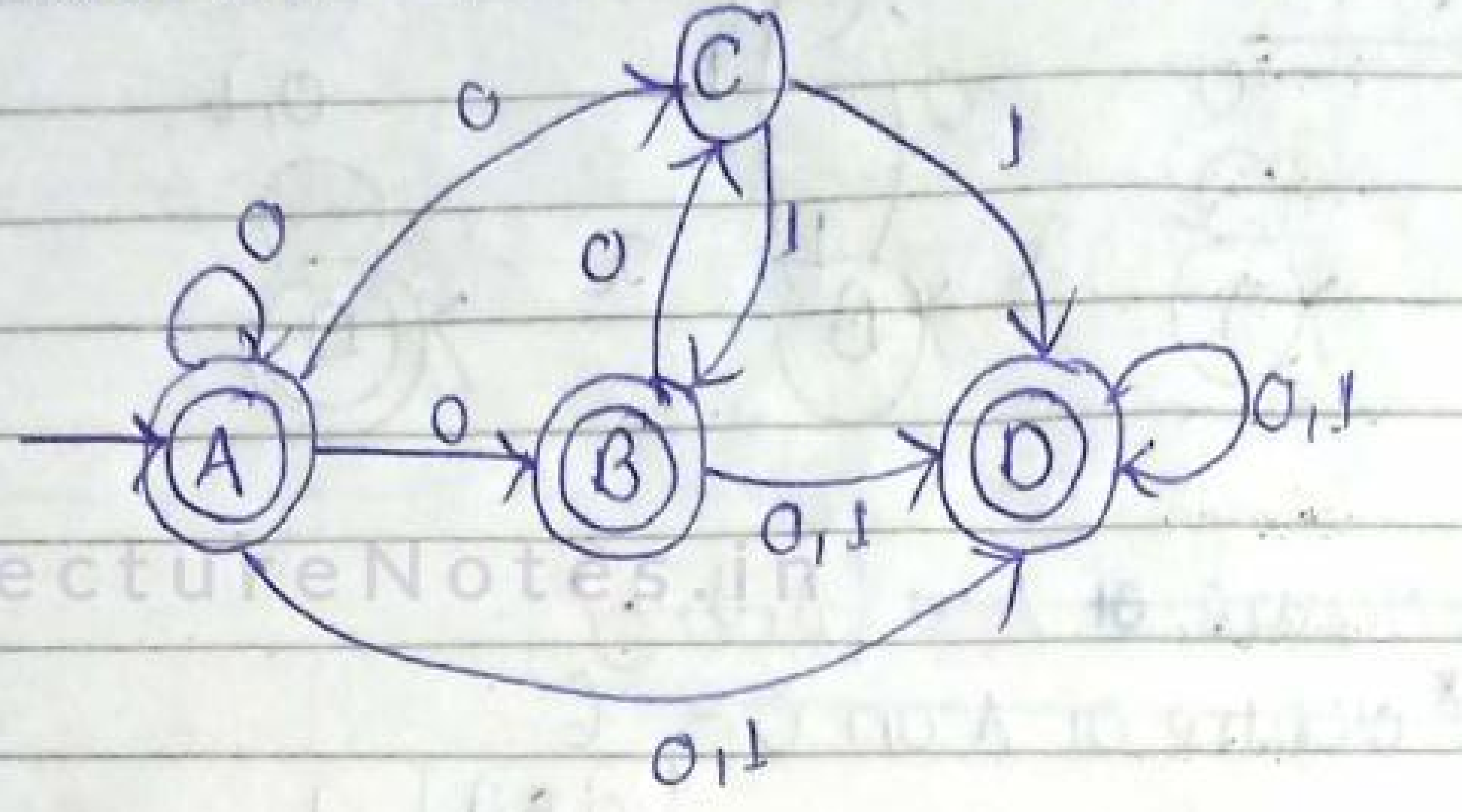
$$\begin{tikzcd} D \arrow[r, "\lambda^*"] & D \arrow[r, "1"] & D \arrow[r, "\lambda^*"] & D \end{tikzcd}$$

LectureNotes.in

Scanned by CamScanner

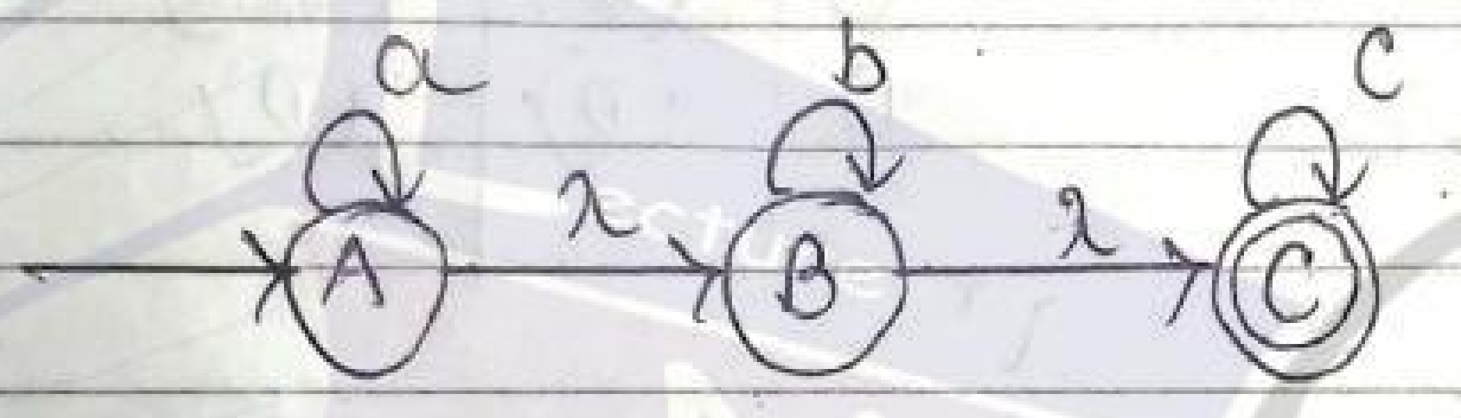
N.B
- no. of states are same
- initial state is same
- Final state can change if any state can reach to its final state through λ transition.

so now NFA will be.



λNFA to DFA

eg:



step 1 λNFA to NFA

	a	b	c
→ A	{A,B,C}	{B,C}	{C}
B	{ }	{B,C}	{C}
*C	{ }	{ }	{C}

now

λ^* a λ^*

$A \to A \to A \to A,B,C$

$\hookrightarrow B \to \emptyset$

$\hookrightarrow C \to \emptyset$

λ^* b λ^*

$A \to A \to \emptyset$

$\hookrightarrow B \to B \to B,C$

$\hookrightarrow C \to \emptyset$

λ^* c λ^*

$A \to A \to \emptyset$

$\hookrightarrow B \to \emptyset$

$\hookrightarrow C \to C \to C$

again

λ^* a λ^*

$B \to B \to \emptyset$

$\searrow C \to \emptyset$

λ^* b λ^*

$B \to B \to B \to B,C$

$\searrow C \to \emptyset$

λ^* c λ^*

$B \to B \to \emptyset$

$\searrow C \to C \to C$

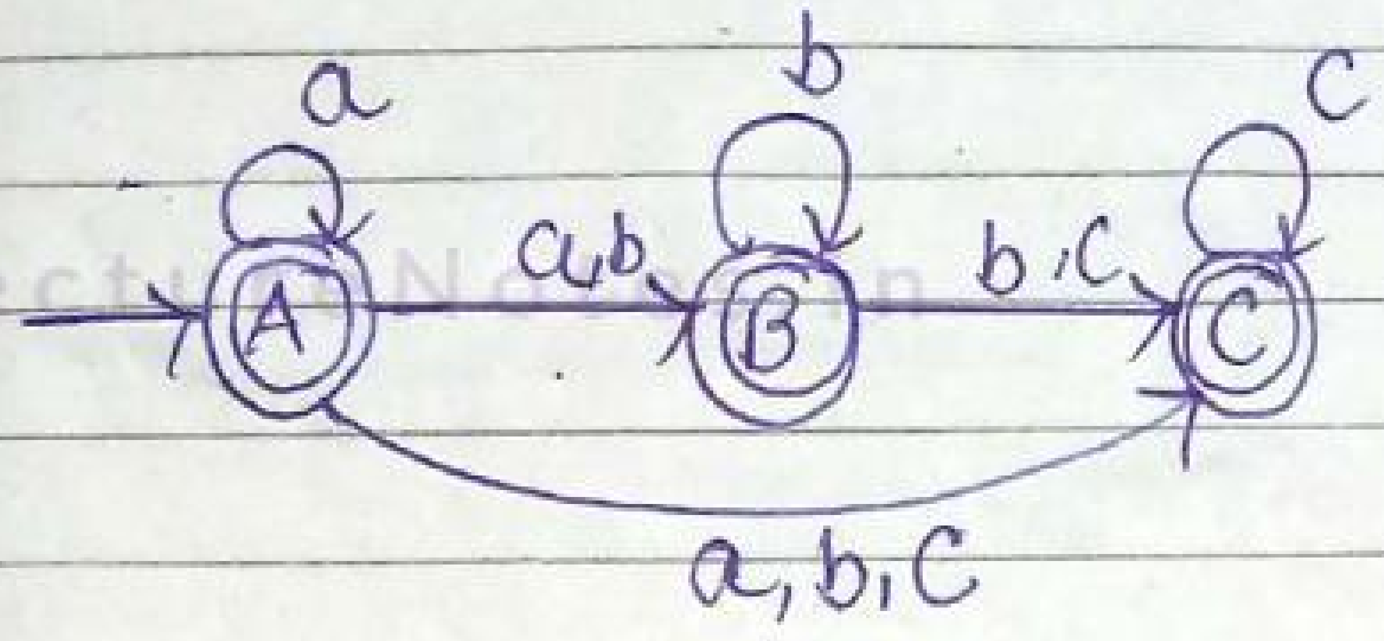
LectureNotes.in

Scanned by CamScanner

again,

$$\begin{array}{ccc} \lambda^* \; a \; \lambda^* & \lambda^* \; b \; \lambda^* & \lambda^* \; c \; \lambda^* \\ C \to C \to \emptyset & C \to C \to \emptyset & C \to C \to C \to C \end{array}$$

so NFA will look

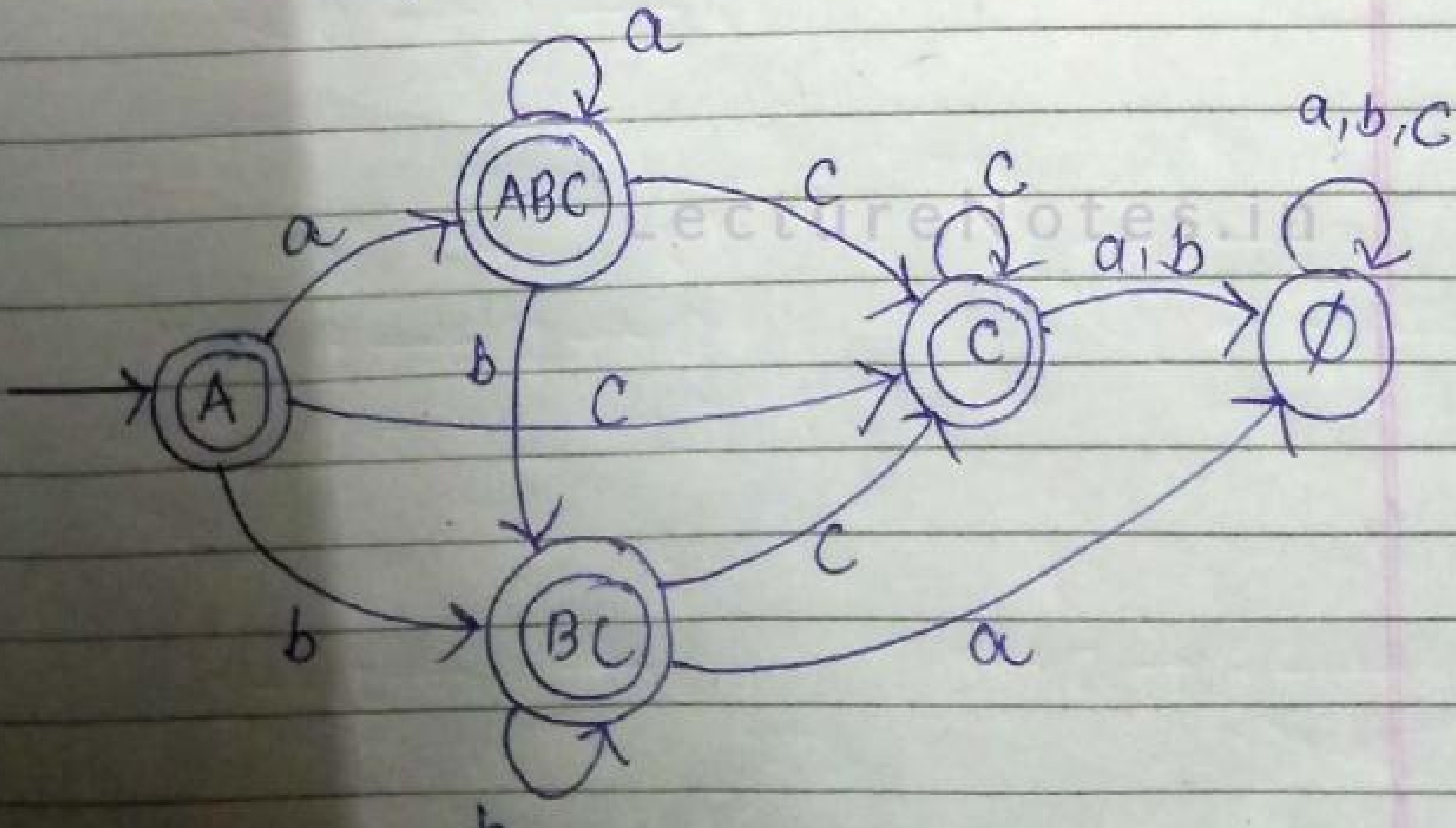


step2: λNFA to DFA

after converting

	a	b	c
→[A]	[ABC]	[BC]	[C]
[ABC]	[ABC]	[BC]	[C]
[BC]	[∅]	[BC]	[C]
[C]	[∅]	[∅]	[C]

so corresponding DFA will be



LectureNotes.in

Scanned by CamScanner

Compiler Design

LectureNotes.in

LectureNotes.in

LectureNotes.in

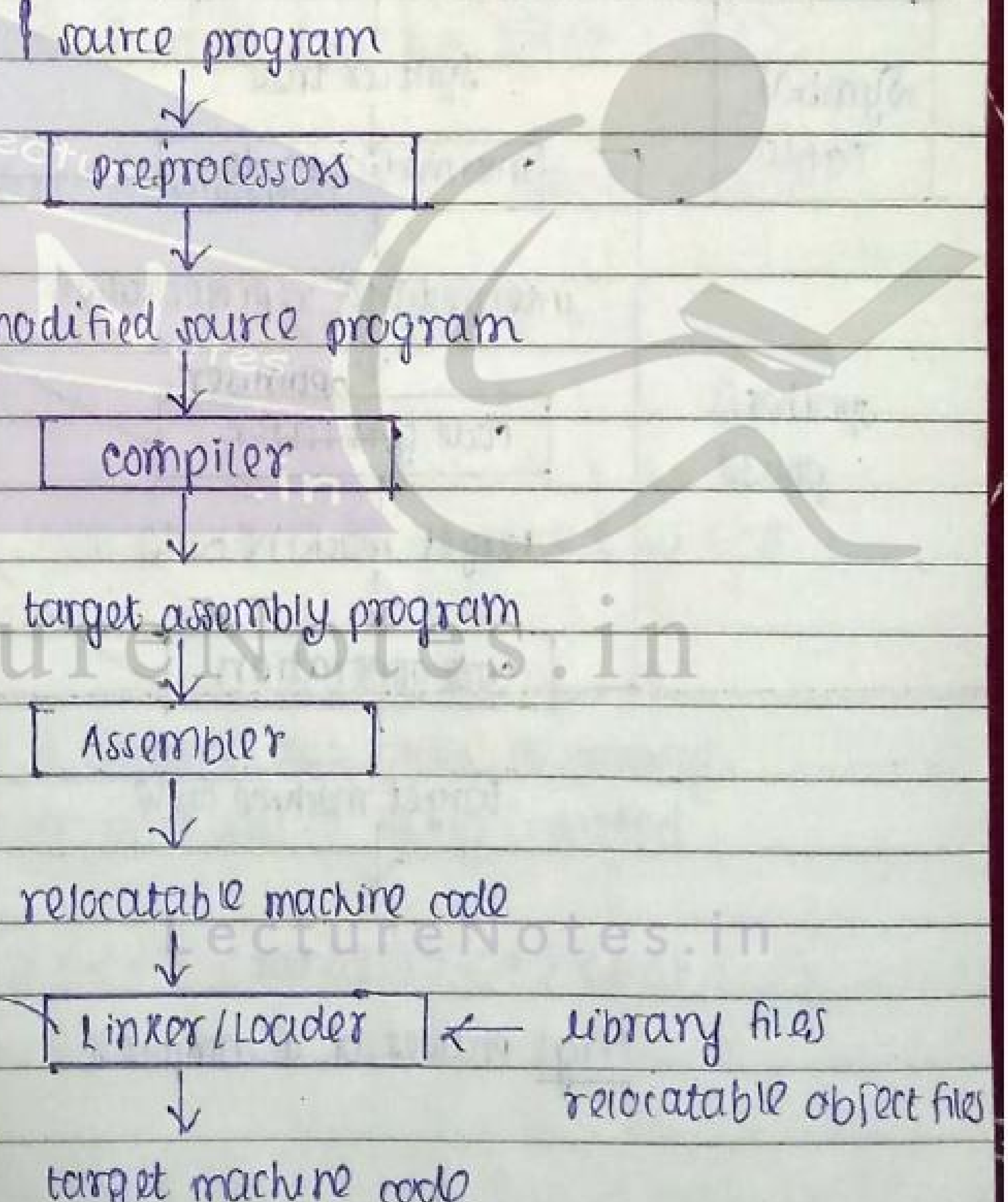
Scanned by CamScanner

Introduction

Compiler : software which converts source program into target program.

Interpreter : executes the program line by line, takes a lot of time but the chances of occurence of error is least when compared with compiler.

Java is a platform independent programming language, Java code or source code is converted to an intermediate form called byte code (done by the compiler). The byte code is interpreted by the virtual machine and this byte code can run on any machine.



resolves external memory addresses coded in one file refers to a location in another file. such kind of memory allocations are done by linker.

Fig: A language processing system

LectureNotes.in

Scanned by CamScanner

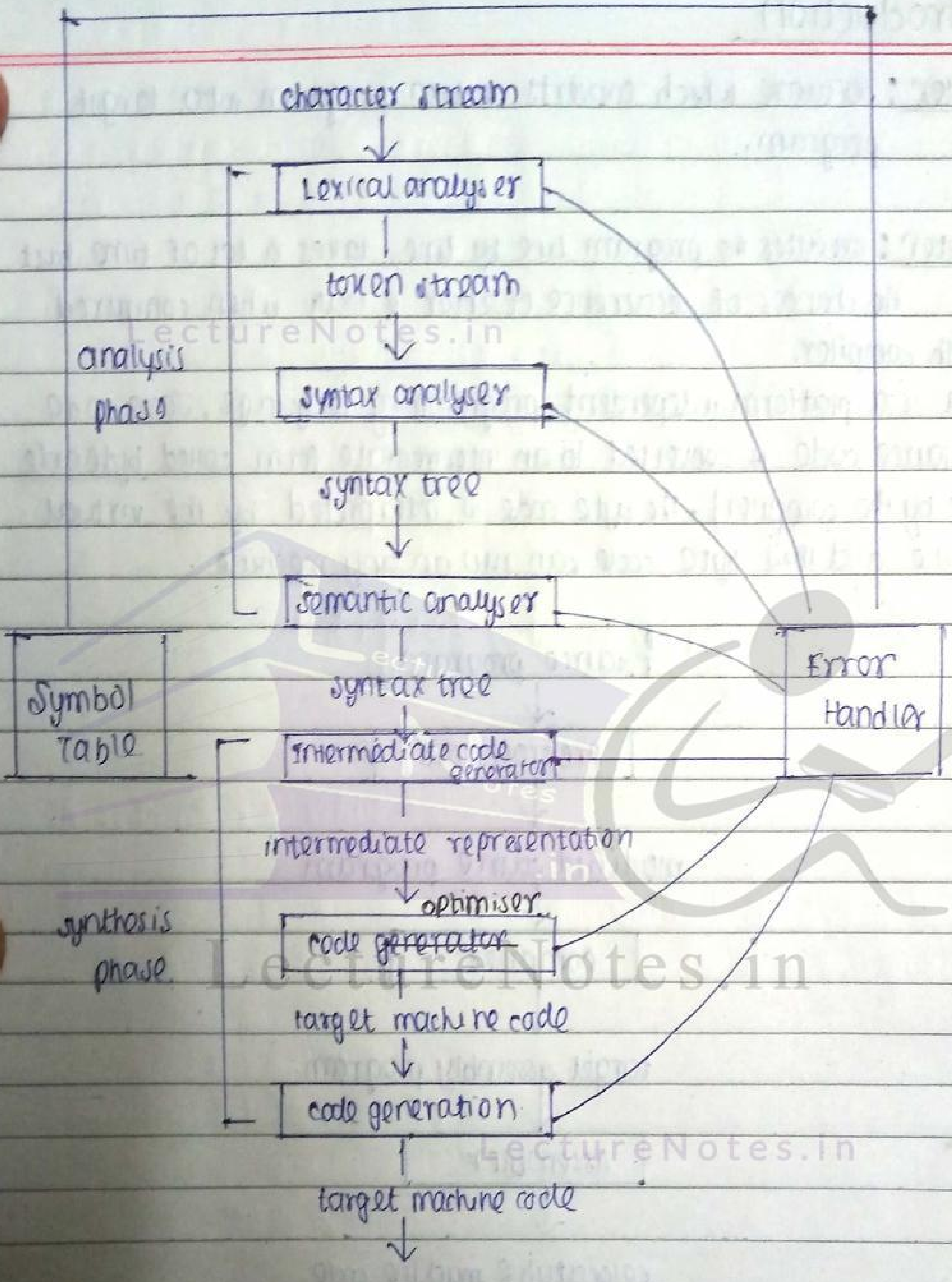


Fig: phases of a compiler

LectureNotes.in

- **Lexical** • 1st phase of a compiler, also called as scanning.
  • the lexical analyser reads the streams of characters making up the source program and groups the characters into meaningful sequences called lexemes.

• for each lexeme, the lexical analyser produces as output a token of the form

<token-name, attribute-value>

eg: position = initial + rate *60

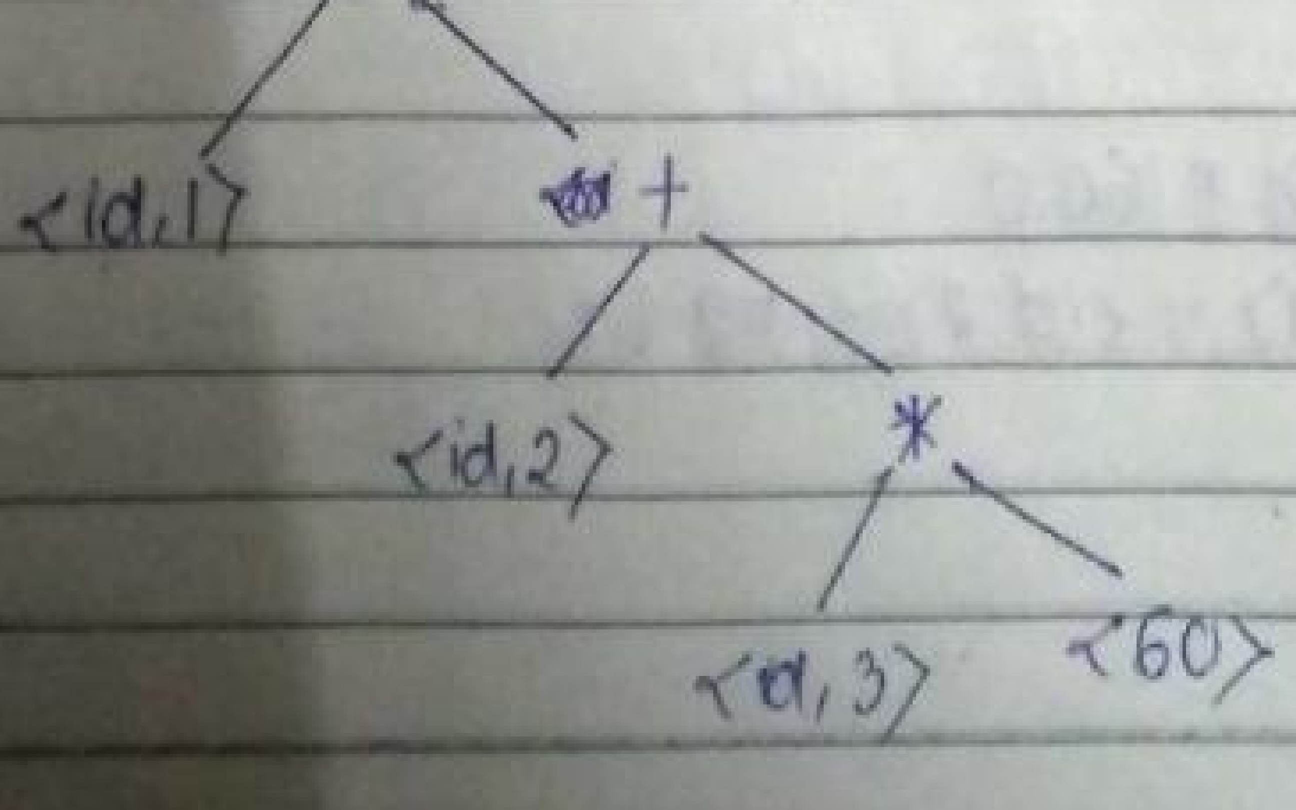
(1) position is a lexeme mapped into a token <id,1>
(2) <=>
(3) initial is a lexeme mapped into a token <id,2>
(4) <+>
(5) <id,3> total tokens = 7
(6) <*>
(7) <60>

Blanks separating the lexemes would be discarded by the lexical analyser.

LectureNotes.in

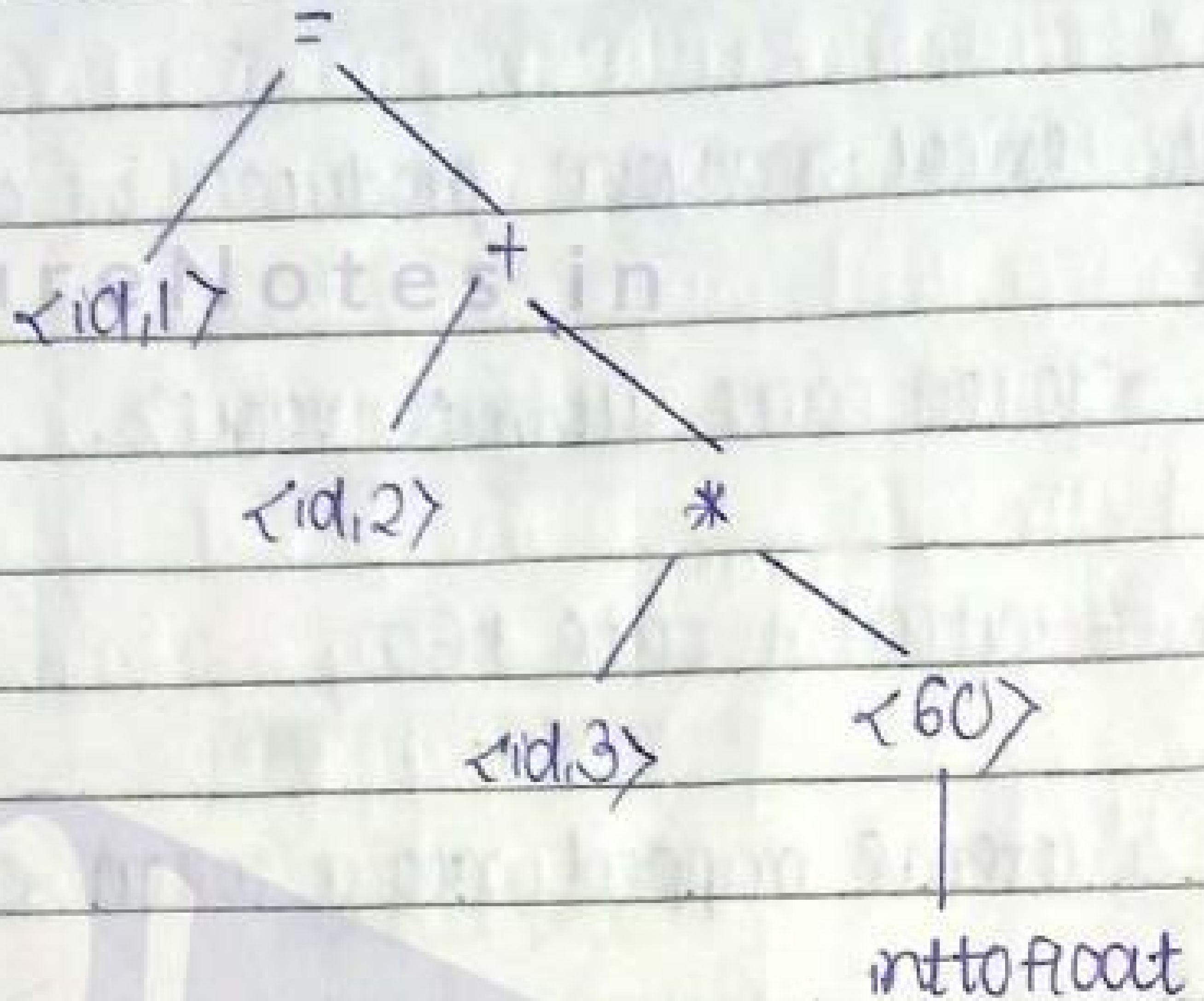
- **Syntax** • 2nd phase of a compiler also called as parsing.
  • syntax tree or parse tree is being created

<id,1> <=> <id,2> <+> <~~rat~~ id,3> <*> <60>



Scanned by CamScanner

- **semantic** • uses the syntax tree and the information in the symbol table to check the source program for semantic consistency.



- **Intermediate code generator**

  • uses 3 address instruction generation to convert to the machine executable code.

  t1 = (inttofloat) 60
  t2 = <id,3> * t1
  t3 = <id,2> + t2
  <id,1> = t3

- **code optimisation**

  • attempts to improve the intermediate code so that better target code will result.

  t1 = <id,3> * 60.0
  ~~t2 =~~ <id,1> = <id,2> + ~~t2~~ t1

LectureNotes.in

Scanned by CamScanner

- code generation

  - takes as input an intermediate representation of the source program and maps it into the target language.

```
     (float)
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

- symbol table

  - records the variable names used in the source program
  - collects the information about various attributes of each name.

1	position	----
2	initial	----
3	rate	----

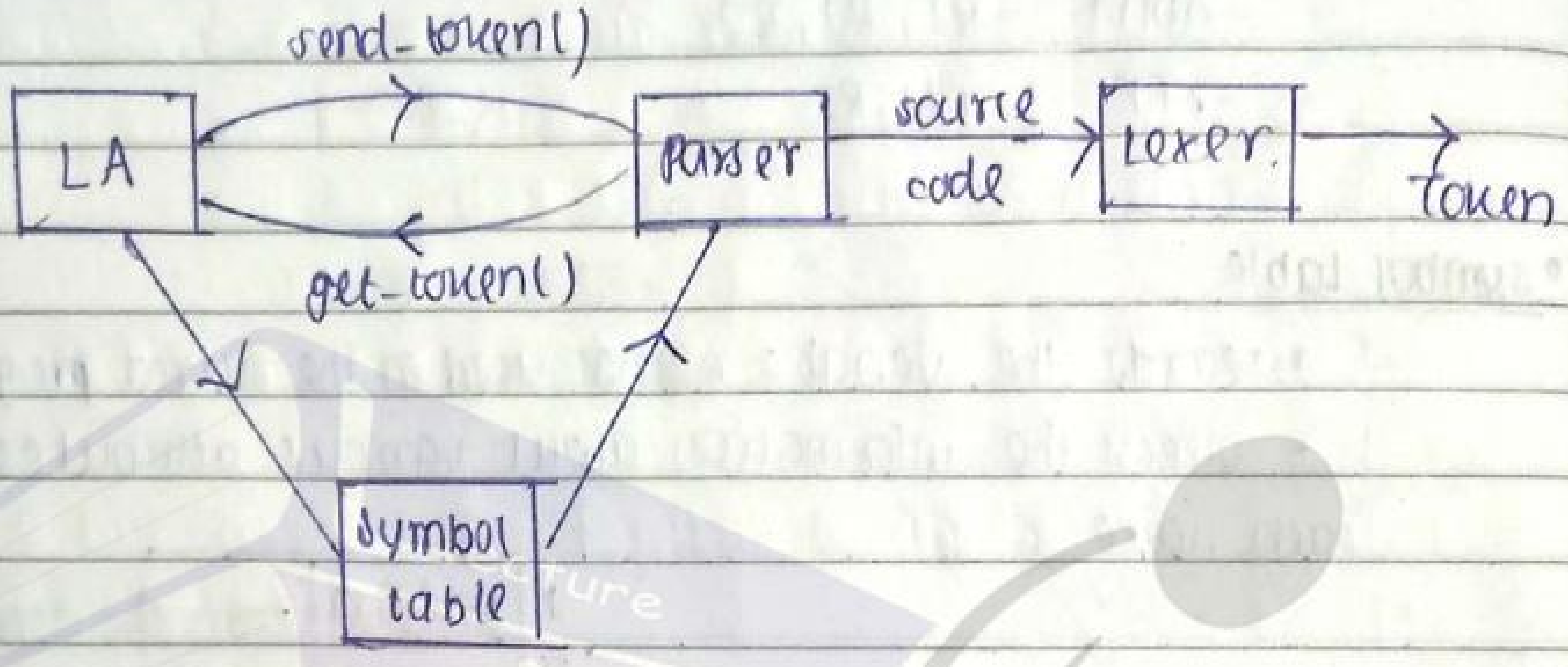
LectureNotes.in

Scanned by CamScanner

Lexical Analysis

- performed by lexical analyser.
- also called a lexer (reads the source code and divides the source code into tokens)
- lexeme is an actual representation of a stream of characters.
- lexeme is carried with the help of parser to syntax analysis.

LectureNotes.in



<u>construction</u>

- defines the rule based on I/P stream, these rules are pattern recognizing tools.
- constructs the regular expression.
- converts the RE to FA.

LectureNotes.in

2 ways to construct a lexer
(a) hardcode
(b) lex tool

lexical analyser is called scanner or tokenizer.

LectureNotes.in

eg: ① printf("TOC"); ⑦ tokens

② int add(int x, int y)
{ ⑯ tokens
return x+y;
}

Scanned by CamScanner

Ambiguaus & unambiguaus

eg1: $E \rightarrow E+E / E*E / id$

check whether the string $id+id*id$ derived from here is ambiguaus or not.

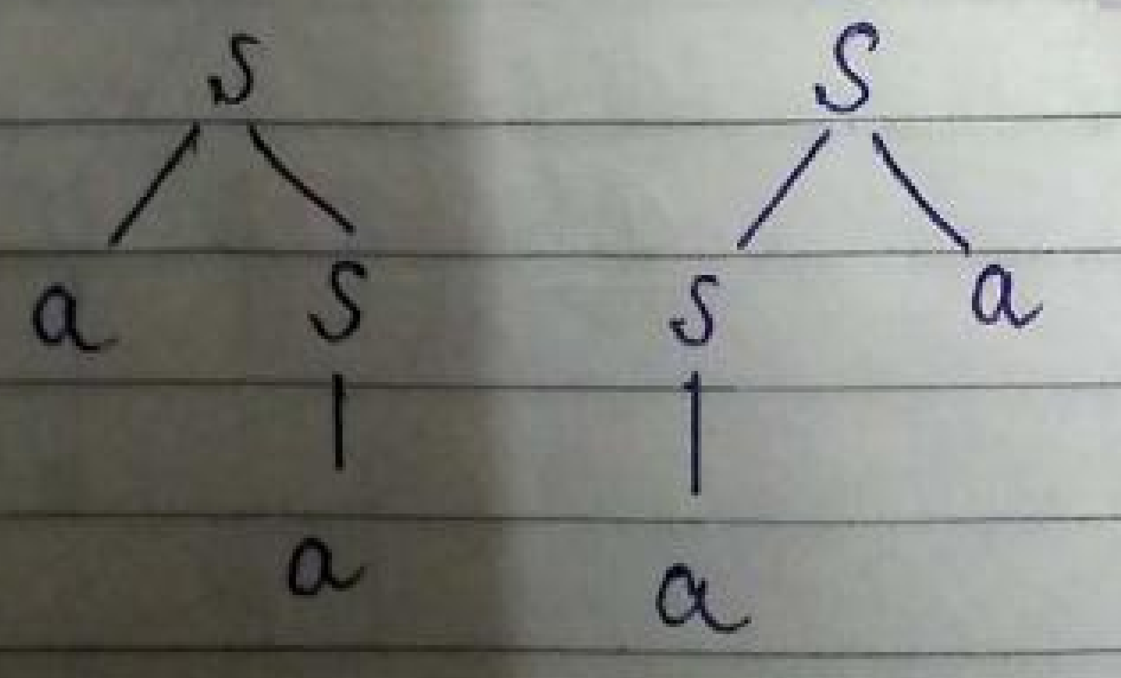
LMD	RMD
$E \Rightarrow E+E$	$E \Rightarrow E+E$
$\Rightarrow id+E$	$\Rightarrow E+E*E$
$\Rightarrow id+E*E$	$\Rightarrow E+E*id$
$\Rightarrow id+id*E$	$\Rightarrow E+id*id$
$\Rightarrow id+id*id$	$\Rightarrow id+id*id$

LMD	RMD
$E \Rightarrow E*E$	$E \Rightarrow E*E$
$\Rightarrow E+E*E$	$\Rightarrow E*id$
$\Rightarrow id+E*E$	$\Rightarrow E+E*id$
$\Rightarrow id+id*E$	$\Rightarrow E+id*id$
$\Rightarrow id+id*id$	$\Rightarrow id+id*id$

LectureNotes.in

(2) $S \rightarrow aS | Sa | a$

say $w = aa$

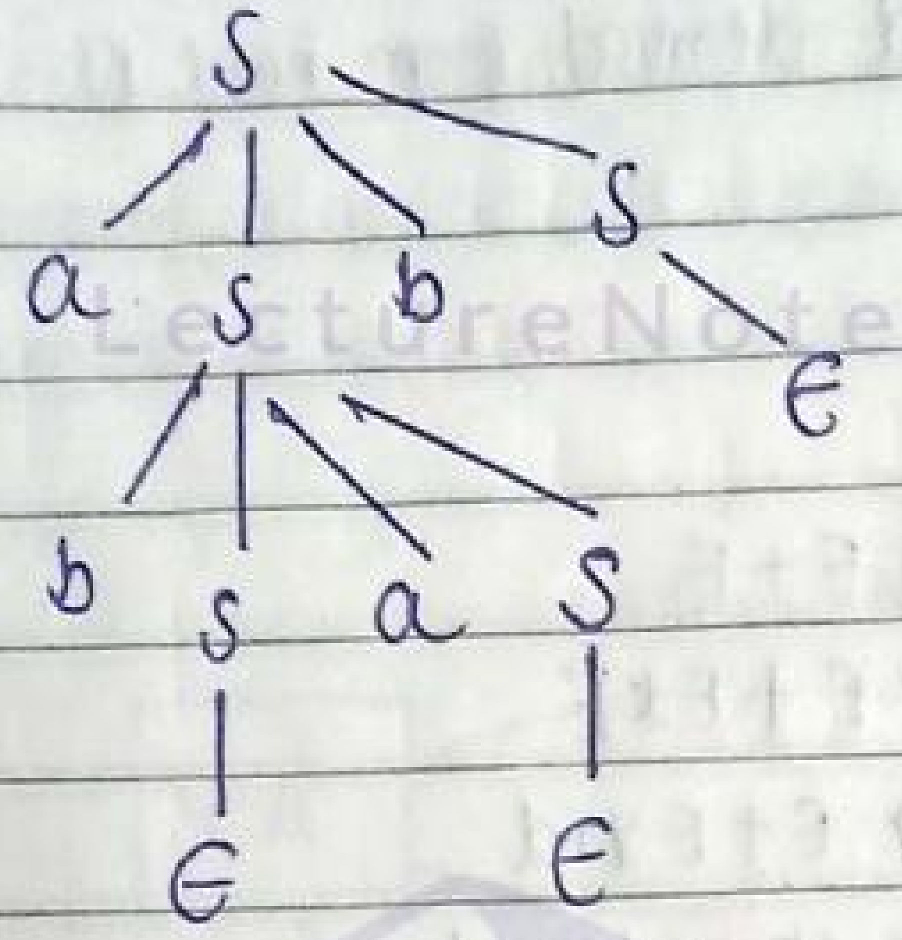


ambiguaus grammar

Scanned by CamScanner

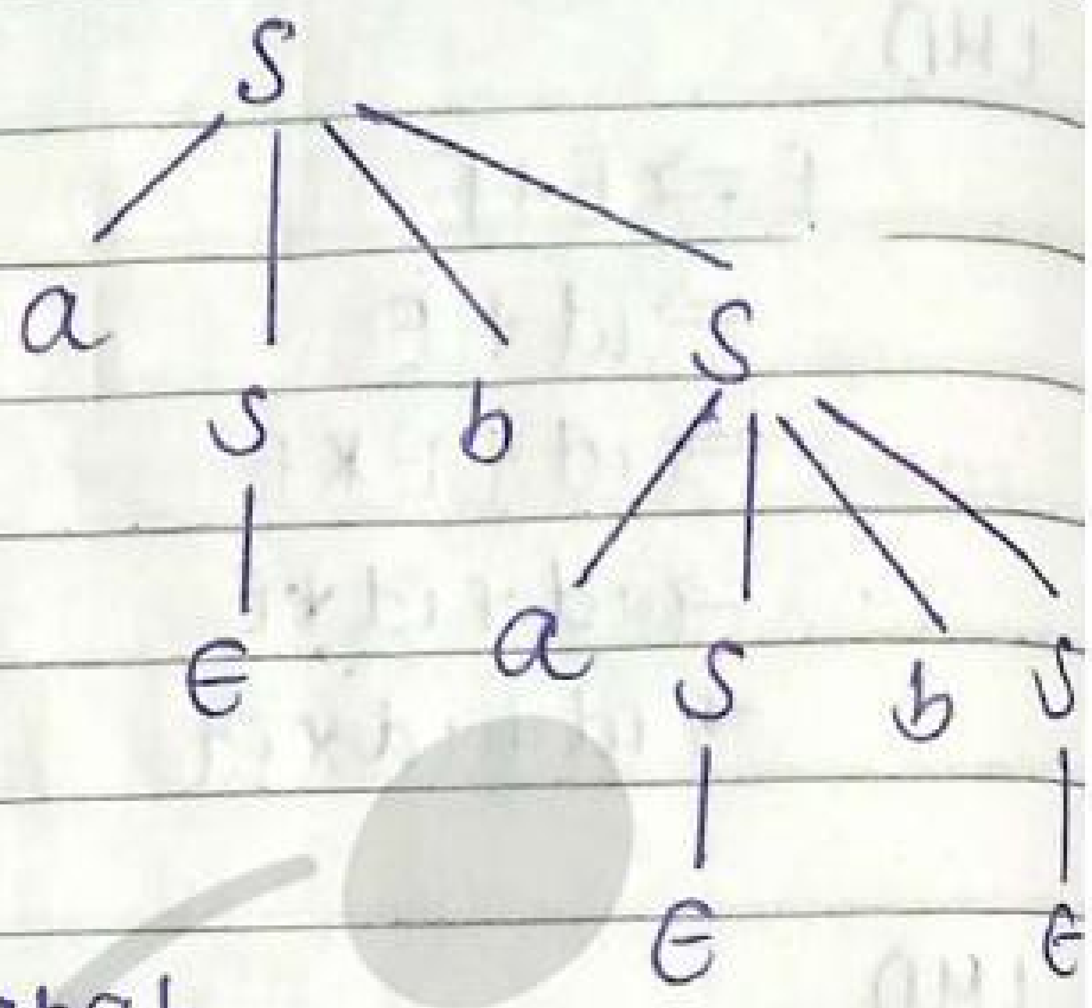
(3) $S \to aSbS \mid bSaS \mid \epsilon$

say, $w = abab$



abab

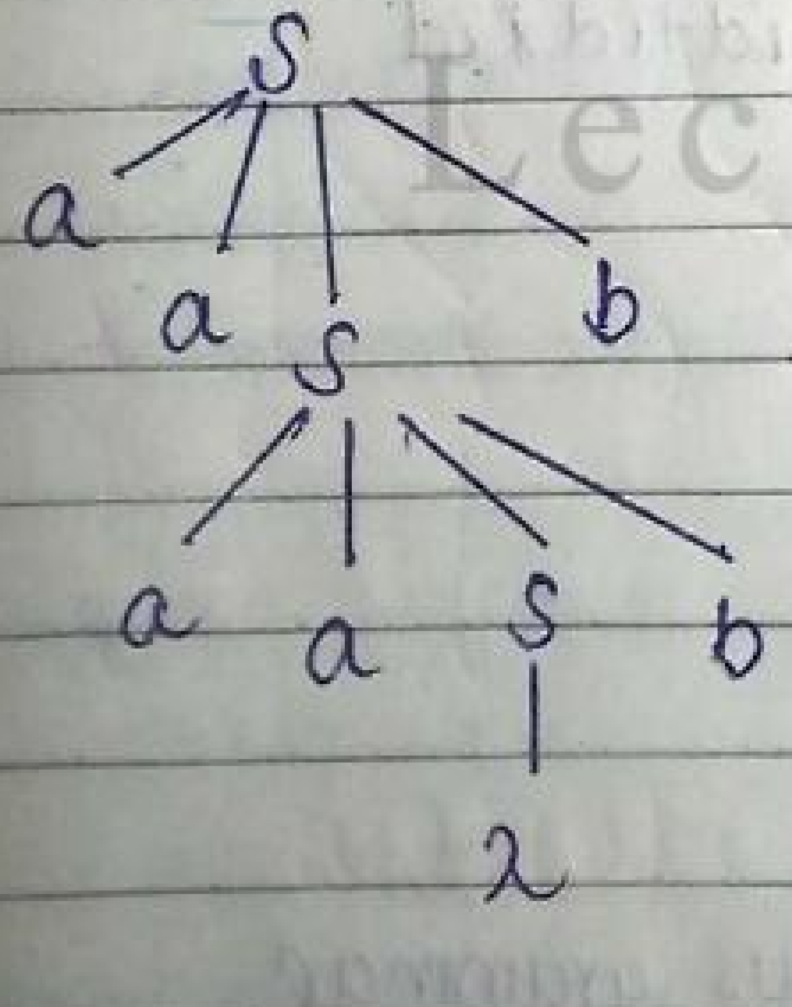
ambiguous



abab

(4) $S \to aaSb \mid \lambda$

say $w = aaaabb$



aabb

unambiguous

LectureNotes.in

Scanned by CamScanner

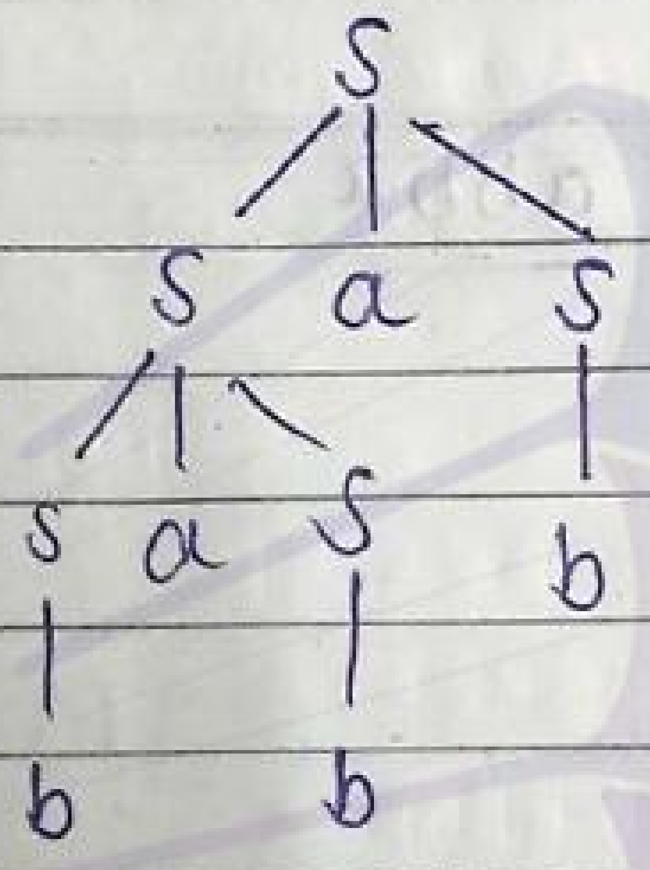
(5) $S \rightarrow A \mid a$

$A \rightarrow a$ w: aa

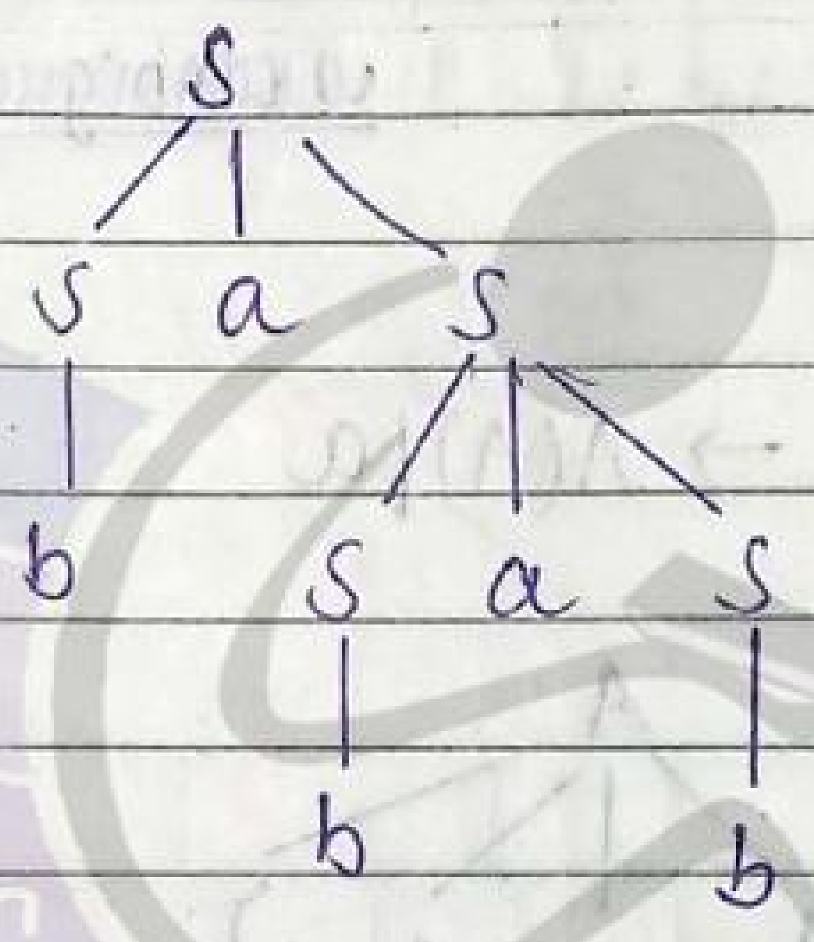
ambiguous

(6) $S \rightarrow SaS \mid b$

w: babab



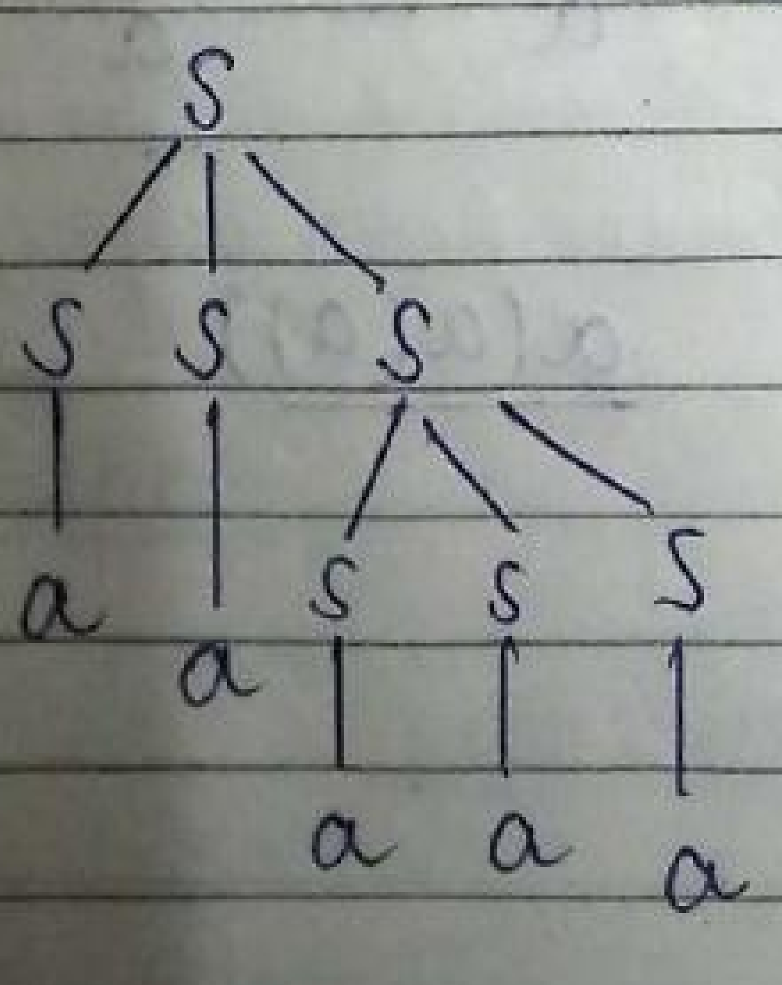
babab

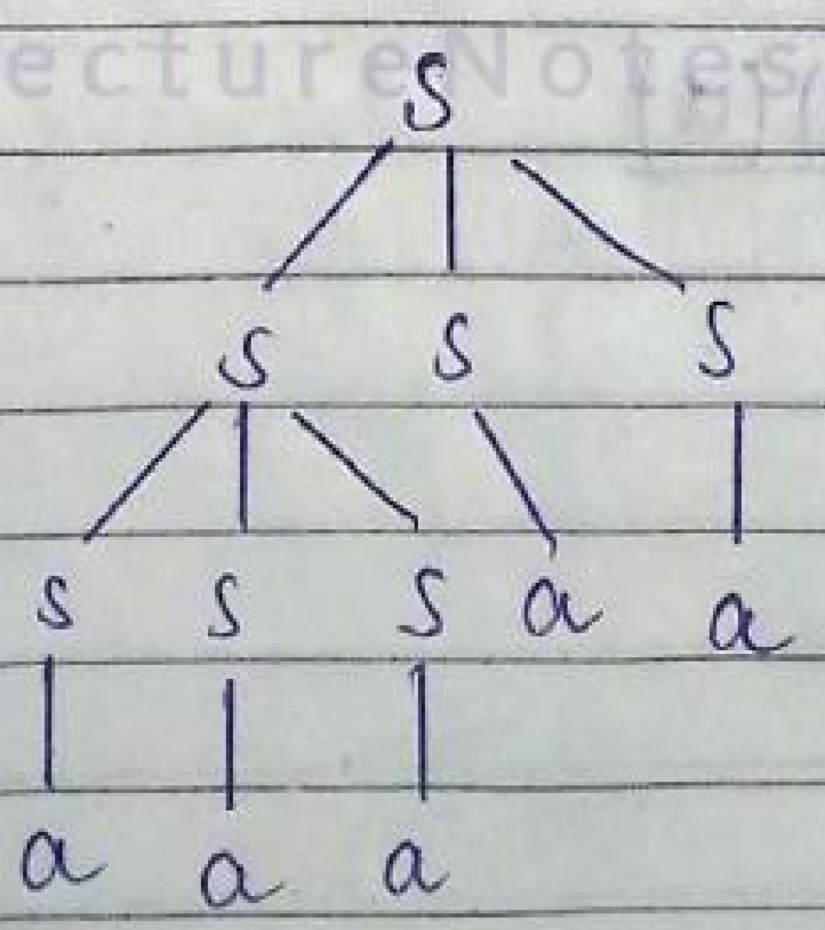


ambiguous babab

(7) $S \rightarrow SSS \mid a$

w: aaaaa





aaaaa aaaaa

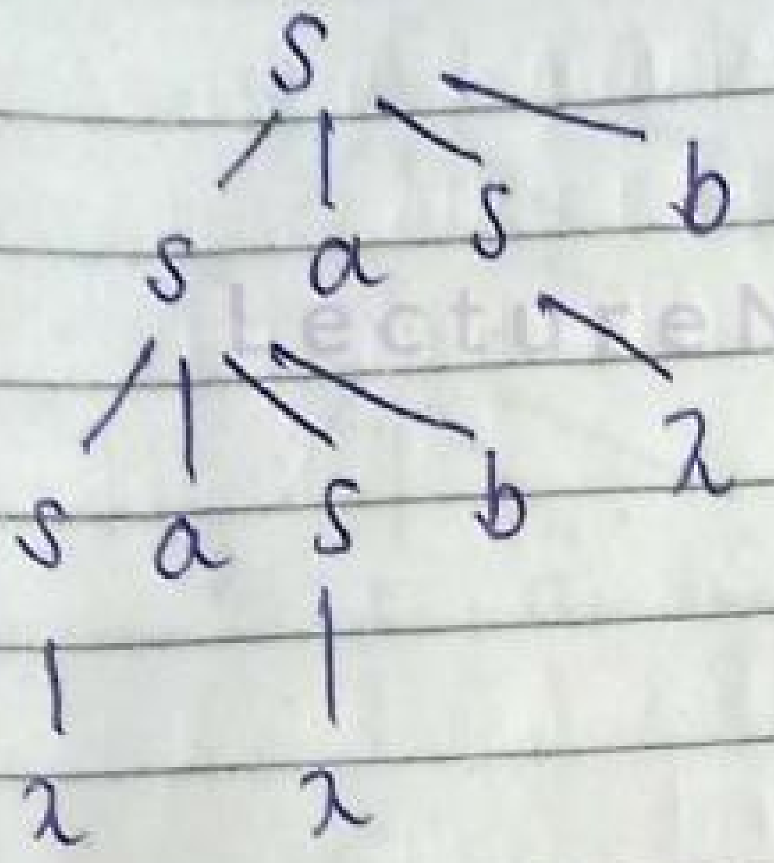
ambiguous

LectureNotes.in

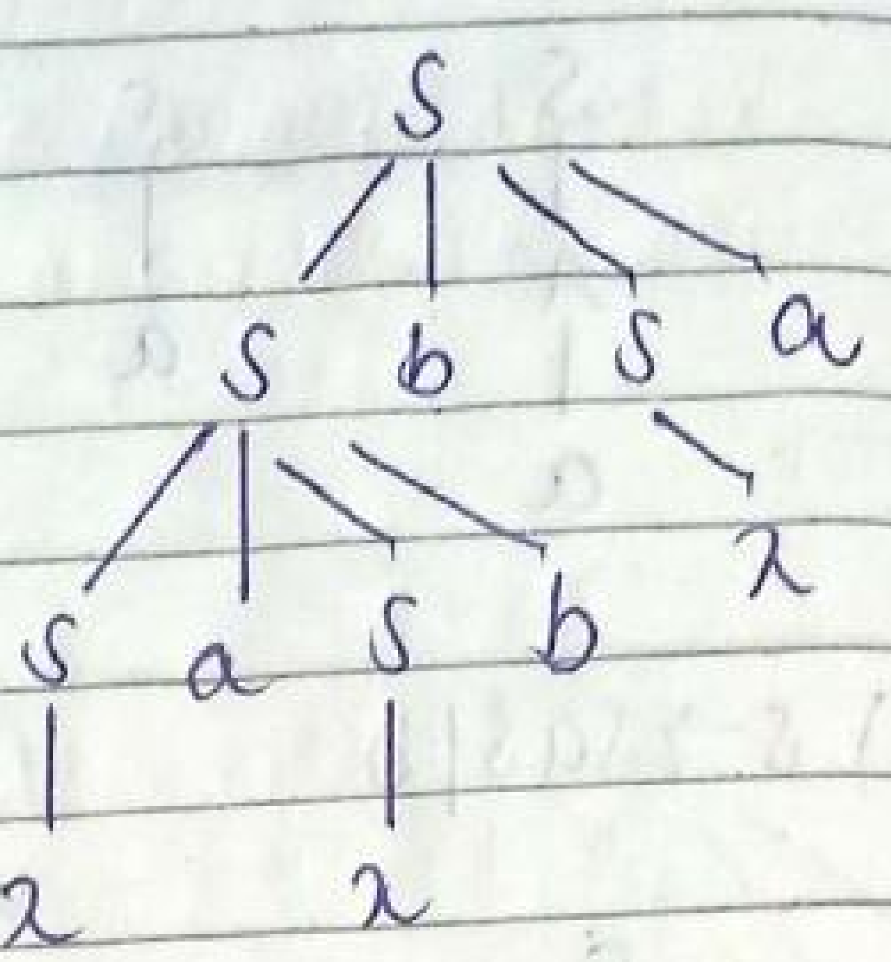
Scanned by CamScanner

(8) S → SaSb | SbSa | λ

w: abab



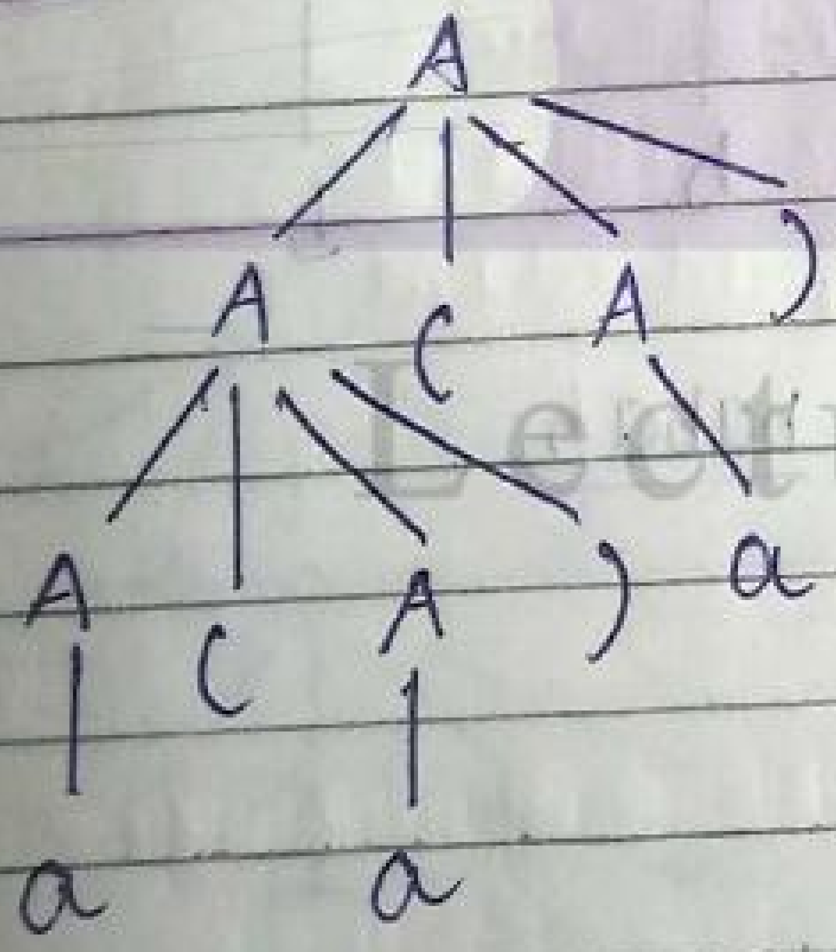
abab



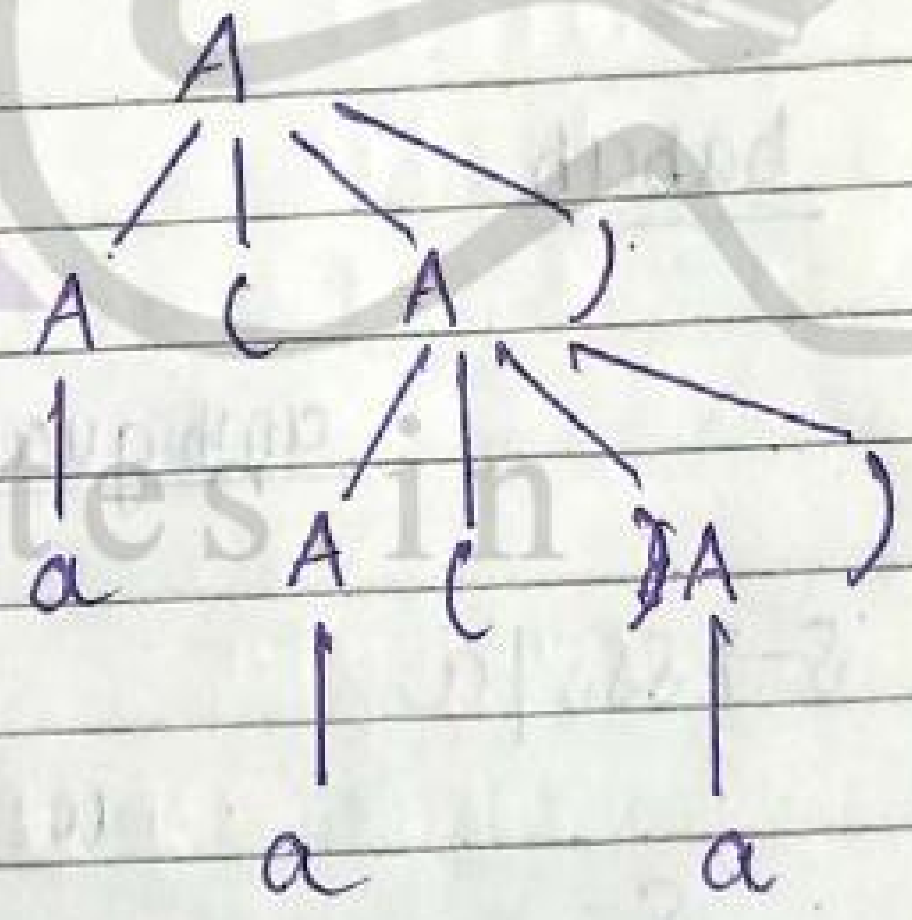
abba

unambiguous

(9) ~~S~~ A → A(A) | a



a(a)(a)

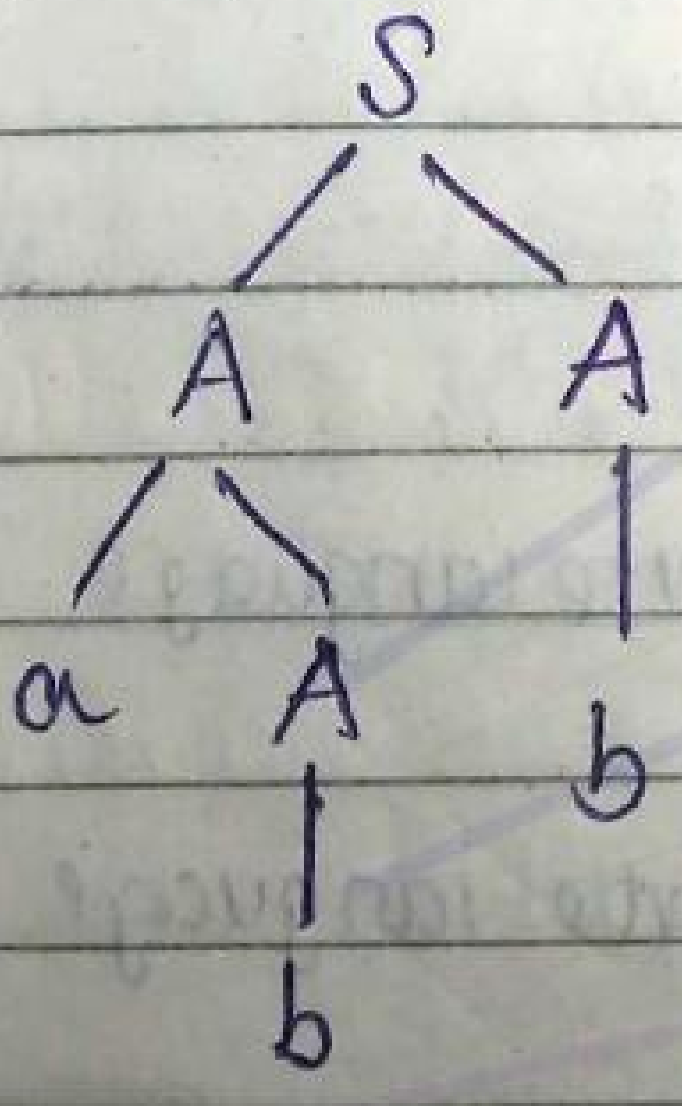


a(a(a))

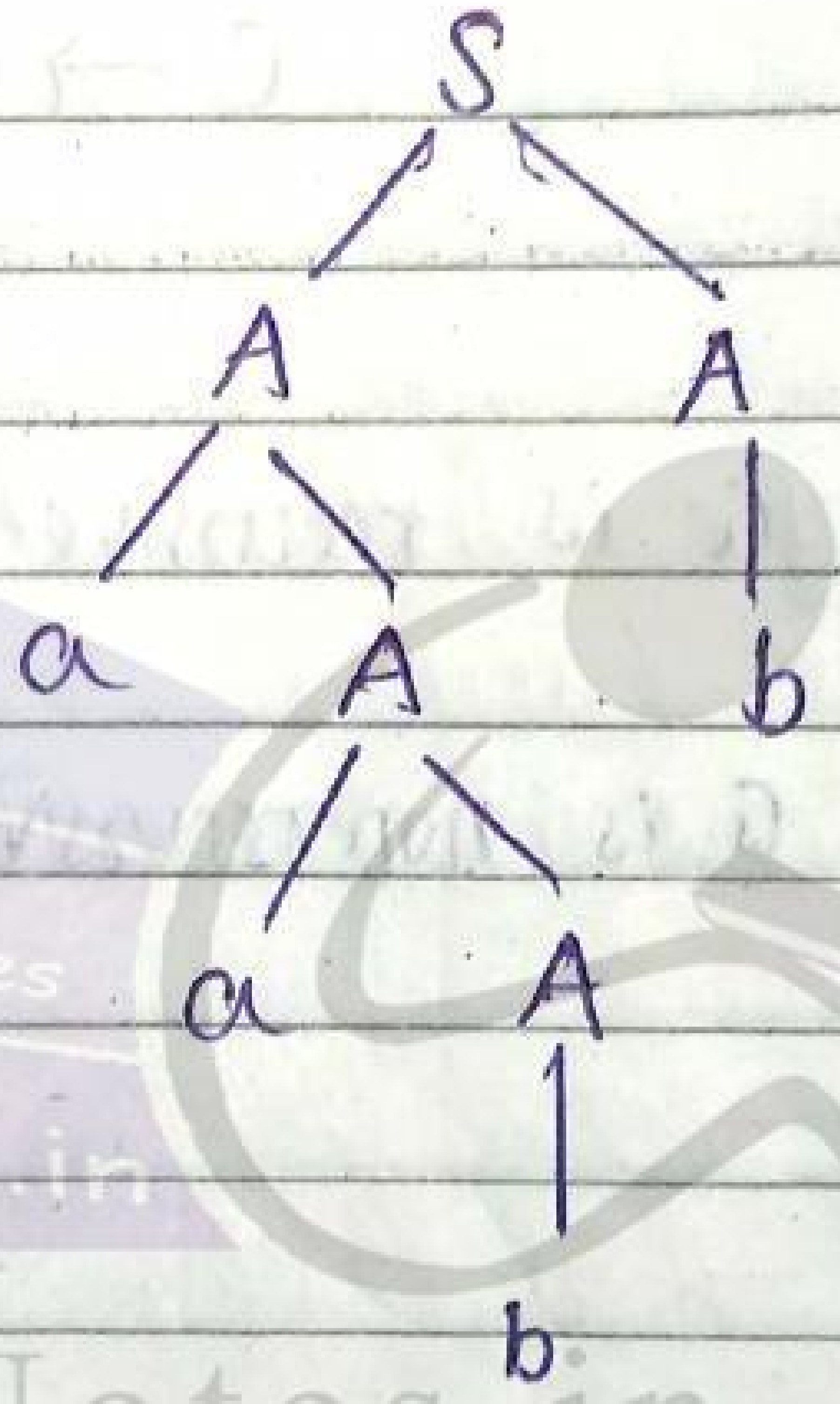
LectureNotes.in

Scanned by CamScanner

(ii) $S \to AA$
$A \to aA$
$A \to b$



abb



aabb

unambiguous

LectureNotes.in

Scanned by CamScanner

Classification II Rule

(Recursive and non recursive)

- **Recursive** : grammar G is said to be recursive if there exists atleast 1 production which has same variable both at LHS and RHS.

eg: ① $S \to Sa \mid b$

② $S \to aS \mid b$

③ $S \to aSb \mid \lambda$

- **non recursive** : grammar G is said to be non-recursive if no production contains same variable both at LHS and RHS.

eg: ① $S \to AB$

$A \to a$

$B \to b$

② $S \to A10B$

$A \to 01 \mid 10$

$B \to 10 \mid C$

$C \to 0 \mid 1$

N.B

- grammar G is recursive if it produces infinite language
- grammar G is non-recursive if it produces finite language.

LectureNotes.in

LectureNotes.in

Scanned by CamScanner

Left Recursion to Right Recursion

Recursion

Left Recursion

$A \rightarrow A\alpha | \beta$

here $A \rightarrow \beta\alpha^*$

Right Recursion

$A \rightarrow \alpha A | \beta$

to convert left recursive ~~$A \rightarrow \alpha A | \beta$~~ to right recursive so

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \epsilon \end{array}} \Longleftrightarrow A \rightarrow A\alpha | \beta$$

eg: $\underset{A}{E} \rightarrow \underset{A}{E}\,\underset{\alpha}{+T} / \underset{\beta}{T}$ convert into right recursive

so $E \rightarrow TE'$

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

eg: $\underset{A}{S} \rightarrow \underset{A}{S}\,\underset{\alpha}{0S1S} | \underset{\beta}{01}$

so

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' | \epsilon$

eg: $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

convert into deterministic grammar by using left factoring

sol: $S \rightarrow iEtSS'$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

eg: $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$

sol: $S \rightarrow aS' \mid b$
$S' \rightarrow SSbS \mid SaSb \mid bb$

still ~~not d~~ it is non deterministic.

$S \rightarrow aS'b$
$S' \rightarrow SS''$
$S'' \rightarrow SbS \mid aSb$

LectureNotes.in

LectureNotes.in

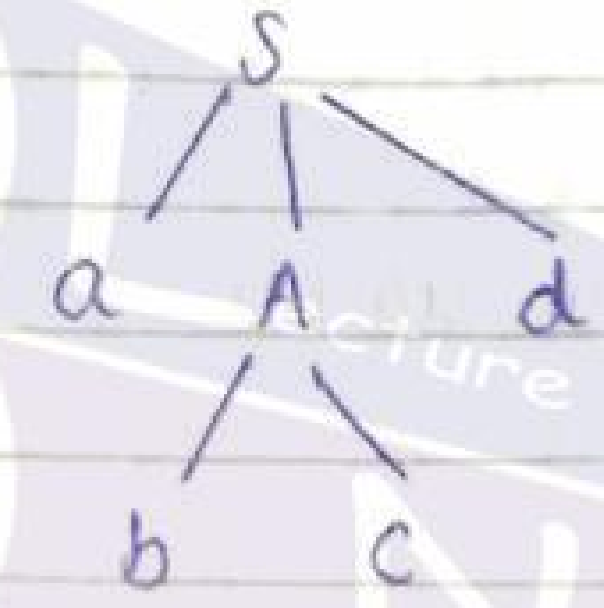
Scanned by CamScanner

(a) Brute Force

eg: $S \rightarrow aAd \mid aB$
$A \rightarrow b \mid c$
$B \rightarrow ccd \mid ddc$

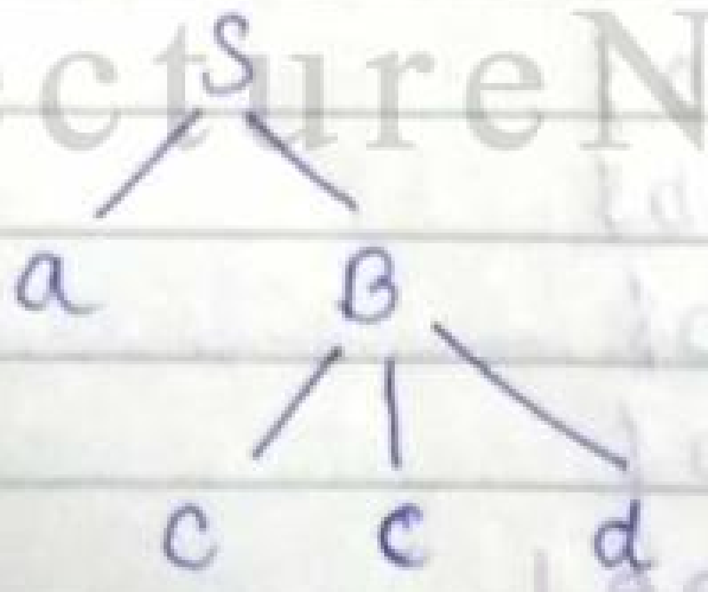
derive the string
$w = addc$

Sol: Initially it will start with $S \rightarrow aAd$ i.e.



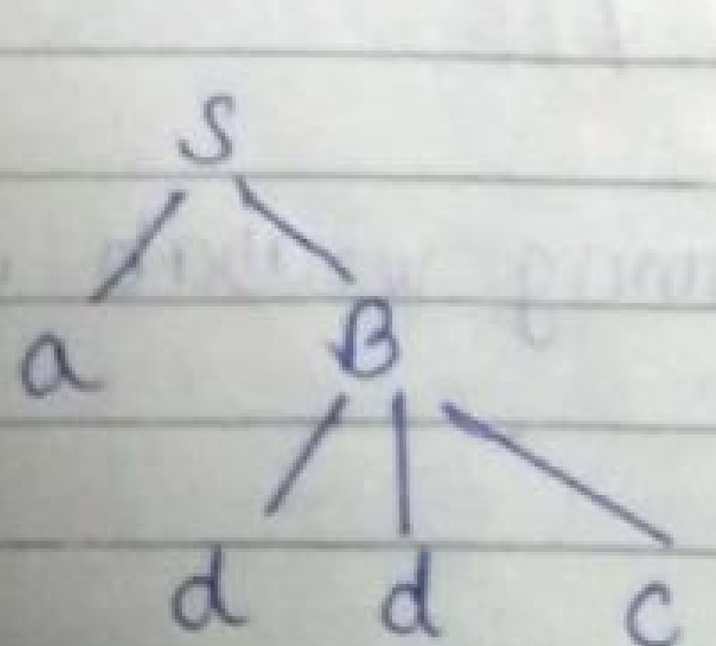
but that doesnt satisfy our desired string

Hence again it will backtrack and start with $S \rightarrow aB$



But this production of B will also not work

So again backtrack



now it satisfies with our desired string.

LectureNotes.in

LL(1) grammar

The grammar for which LL(1) parser can be constructed is known as LL(1) grammar

OR

The grammar whose parse table does not contain multiple entries is called LL(1) grammar.

Functions used —
- First(α) (determines first of value produ)
  ~~(can be terminal or non-terminal)~~
- Follow(A)
  (always non terminal)

Components

- I/P buffer : to store the I/P string
  eg: abab

a	b	a	b	$

- parse stack
- parse table : (which maintains or have production grammars)

NB LL(1) parser works on LL(1) parser algorithm

(a) I/P buffer:
- divided into cells
- each cell is capable of holding 1 symbol at a time.

(b) tape header / look ahead pointer:
- always points 1 symbol at a time
- symbol which is pointed by the tape header is called look ahead symbol.

LectureNotes.in

Scanned by CamScanner

algorithm

- first(α) : set of all terminals that may begin in sentential form which is derived from α

Rules

① If α is a terminal then first(α) = α

② If α is a non-terminal defined by $\alpha \to \epsilon$ then first(α) = ϵ

③ If α is a non-terminal and α derives

$\alpha \to X_1 X_2 \ldots X_K$ for $K \geq 1$

place a in first(α) if for some i, a is in first(X_i) and ϵ is in all of first(X_1). first(X_2), ... first(X_n)

then first (E) = a

eg: $A \to BCDEa$

$B \to \epsilon$

$C \to \epsilon$

$D \to \epsilon$

$E \to \epsilon$

④ If a is a nonterminal and defined by non-null production $\alpha \to X_1 X_2 X_3$ then ·

first (α) = first(X_1) = first (X_1) ∪ first (X_2)

= first (X_1) ∪ first(X_2) ∪ first(X_3) ∪ {ϵ}

Follow Rules

① If follow(A) = \$ if A is starting symbol

② If $S \to \alpha AB$ and $B \neq \epsilon$

then follow (A) = first(B) without ϵ

③ (with ϵ) if $S \to \alpha AB$ and $S \to \alpha A$ and $B \xrightarrow{*} \epsilon$

then

follow(A) = follow(S)

LectureNotes.in

eg1:

	First	Follow
$S \rightarrow ABCDE$	{a,b,c}	{\$}
$A \rightarrow a \mid \epsilon$	{a,ϵ}	{b,c}
$B \rightarrow b \mid \epsilon$	{b,ϵ}	{c}
$C \rightarrow c$	{c}	{d,e,\$}
$D \rightarrow d \mid \epsilon$	{d,ϵ}	{e,\$}
$E \rightarrow e \mid \epsilon$	{e,ϵ}	{\$}

~~First(S) = {a,c}~~ Follow(E) = Follow(S)

First('

(if a variable is at the rightmost then follow of that variable is equal to follow of the variable which produces it)

eg 2 $S \rightarrow \epsilon$

First(S) = {ϵ}

Follow(S) = {\$}

(3) $S \rightarrow a \mid \epsilon$

First(S) = {a,ϵ}

Follow(S) = {\$,a}

(4) $S \rightarrow aA \mid \epsilon$

$A \rightarrow b$

First(S) = {a,ϵ}

Follow(S) = {\$,a}

First(A) = {b}

Follow(A) = {b}

LectureNotes.in

(5) S → aAa | ε
A → b

First(S) = {b, ε}
First(A) = {b}
Follow(S) = {$, a}
Follow(A) = {a, b}

(6) S → aAB
A → b | ε
B → c

First(S) = {a
First(A) = {b, ε}
First(B) = {c}

	First	Follow
(7) S → ABCDE	{a, b, c, d, e, ε}	~~$~~{$}
A → a | ε	{a, ε}	{b, c, d, e}
B → b | ε	{b, ε}	{c, d, e}
C → c | ε	{c, ε}	{d, e}
D → d | ε	{d, ε}	{e}
E → e | ε	{e, ε}	{$}

	First	Follow
(8) S → aABd	{a}	{$}
A → BAb | ε	{b, ε}	{d, b}
B → Adb | ε	{d, b, ε}	{d}

	First	Follow
(9) S → aA	{a}	{$}
A → b	{b}	{$}

LectureNotes.in

Scanned by CamScanner

	First	Follow
(10) S → aAb	{a}	{$}
A → Ba|b	{b, d}	{b}
B → d	{d}	{a}

	First	Follow
(11) S → S0S1|ε	{0, ~~1~~ ε}	{$, 0, 1}

	First	Follow
(12) S → AaAb|BaBb	{a}	{$}
A → ε	{ε}	{a, b}
B → ε	{ε}	{a, b}

	First	Follow
3) S → AB	{a, b, ε}	{$}
A → a|ε	{a, ε}	{b, $}
B → b|ε	{b, ε}	{$}

~~(14) S→~~

	First	Follow
(14) S → ABCDE	{a, b, c, d, e}	{$}
A → a|ε	{a, ε}	{b, c, d, e}
B → b|ε	{b, ε}	{c, d, e}
C → c|ε	{c, ε}	{d, e}
D → d|ε	{d, ε}	{e}
E → e	{e}	{$}

LectureNotes.in

Scanned by CamScanner

	First	Follow
(15) S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC | ε	{b, ε}	{g, f, h}
D → EF	{g, f, ε}	{h}
E → g | ε	{g, ε}	{f, h}
F → f | ε	{f, ε}	{h}

LectureNotes.in

	First	Follow
(16) S → Bb | Cd	{a, b, c, d}	{$}
B → aB | ε	{a, ε}	{b}
C → cC | ε	{c, ε}	{d}

	First	Follow
(17) E → TE'	{id, (}	{$,)}
E' → +TE' | ε	{+, ε}	{$,)}
T → FT'	{id, ε}	{+, $,)}
T' → *FT' | ε	{*, ε}	{+, $,)}
F → id | (E)	{id, (}	{*, +, $,)}

LectureNotes.in

	First	Follow
(18) S → ACB | CbB | Ba	{d, g, h, ε, b, a}	{$}
A → da | BC	{d, g, h, ε}	{h, g, $}
B → g | ε	{g, ε}	{$, a, h, g, $}
C → h | ε	{h, ε}	{g, $, b, $h}

Scanned by CamScanner

	First	Follow
(19) $S \to aABb$	{a}	{\$}
$A \to c \mid \epsilon$	{c, ε}	{d, b}
$B \to d \mid \epsilon$	{d, ε}	{b}

LectureNotes.in

Construct LL(1) parser

For every production $A \to \alpha$

repeat the steps

- add $A \to \alpha$ in $M[A, a]$ for every symbol a in First(A)
- if First(α) contains ε add $A \to \alpha$ in $M[A, b]$ for every symbol b in Follow(A)

> the grammar G is LL(1) if its parse table does not contain multiple entries.

eg:

(eg:) Check whether the grammars are LL(1) or not.

(1)	First	Follow
$E \to TE'$	{id, (}	{\$,)}
$E' \to +TE' \mid \epsilon$	{+, ε}	{\$,)}
$T \to FT'$	{id, (}	{+,), \$}
$T' \to *FT' \mid \epsilon$	{*, ε}	{+,), \$}
$F \to id \mid (E)$	{id, (}	{*, +,), \$}

LectureNotes.in

Scanned by CamScanner

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

LectureNotes.in

N.B If E' → +TE' | ⓔ then we do follow of the ~~previous~~ left hand side i.e. Follow (E')

same

T' → *FT' | ⓔ do follow (T')

so this is a LL(1) parser.

(2)

	First	Follow
S → iEtSS' | a	{i, a}	{$, e}
S' → eS | ε	{e, ε}	{$, e}
E → b	{b}	{t}

LectureNotes.in

	i	t	a	b	~~$~~e	$
S	S→iEtSS'		S→a			
S'					S'→e S'→ε	S'→ε
E				E→b		

Scanned by CamScanner

(3) $R \rightarrow \underbrace{R+R}_{\alpha_1} \mid \underbrace{RR}_{\alpha_2} \mid \underbrace{R^*}_{\alpha_3} \mid \underbrace{(R)}_{\beta_1} \mid \underbrace{a}_{\beta_2} \mid \underbrace{b}_{\beta_3}$

at first eliminate left recursion

$R \rightarrow (R)R' \mid aR' \mid bR'$

$R' \rightarrow +RR' \mid RR' \mid *R' \mid \epsilon$

First (R) = { (, a, b }

First (R') = { +, (, a, b, *, ϵ }

Follow (R) = {), (, a, b, +, *, $ }

Follow (R') = {), (, a, b, +, *, $ }

	(	)	+	*	a	b	$
R	$R\rightarrow(R)R'$				$R\rightarrow aR'$	$R\rightarrow bR'$	
R'	$R'\rightarrow RR'$ $R'\rightarrow\epsilon$	$R'\rightarrow\epsilon$	$R'\rightarrow +RR'$ $R'\rightarrow\epsilon$	$R'\rightarrow *R'$ $R'\rightarrow\epsilon$	$R'\rightarrow RR'$ $R'\rightarrow\epsilon$	$R'\rightarrow RR'$ $R'\rightarrow\epsilon$	$R'\rightarrow\epsilon$

LectureNotes.in

so this is not a LL(1) parser.

LectureNotes.in

Trick

- A grammar without ϵ rule in LL(1) for each production of the form $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_n$
then if
(a) First(α_1) ∩ First(α_2) …… ∩ First(α_n) = ϕ
then it is LL(1) grammar

- grammar with ϵ rule in LL(1) for each rule of the form $A \rightarrow \alpha | \epsilon$
  if (a) $First(\alpha) \cap Follow(A) = \phi$
  then it is an LL(1) grammar
- ambiguous grammar is not LL(1)
- left recursive " " " LL(1)
- non-left factor " " " LL(1)

- a grammar in which every production has only 1 alternative then right hand side is always LL(1)

(4) $S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

$First(S) = \{a, b\}$
$First(A) = \{a, b\}$

$First(S) \cap First(A) = \phi$

then LL(1) grammar

(5) $E \rightarrow E+T | T$
$T \rightarrow int$

first remove left recursion $E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow int$

$First(E) = \{int\}$
$First(E') = \{+, \epsilon\}$
$First(T) = \{int\}$

LL(1) grammar

LectureNotes.in

Scanned by CamScanner

	First	Follow
(6) $E \to TE'$	{id, (}	{\$,)}
$E' \to +TE' \mid \epsilon$	~~fom,~~ {+, ϵ}	{\$,)}
$T' \to FT'$	{id, (}	{+,), \$}
$T' \to *FT' \mid \epsilon$	{*, ϵ}	{+,), \$}
$F \to (E) \mid id$	{id, (}	{*, +,), \$}

LectureNotes.in

so here

~~First (+TE') = {+}~~
~~Follow (E') = {\$,)}~~

E

applying the rule
$First(\alpha) \cap Follow(A) = \phi$

we can see
this is an LL(1) grammar

LectureNotes.in

(7) $S \to aSbS \mid bSaS \mid \epsilon$

~~First(S)~~ = $S \to aSbS$ {a}
$\mid bSaS$ {b}
$\mid \epsilon$ {a, b, \$}

LectureNotes.in

not LL(1) and also ambiguous.

represents in
which column
we keep this
productions

(8) $S \to aABb$

$A \to c \mid \epsilon$ $\quad \{c\} \quad \{d, b\}$

$B \to d \mid \epsilon$ $\quad \{d\} \quad \{b\}$

so this is LL(1) since all productions will be present in different cells,

(9) $S \to A \mid a$ $\quad \{a\} \cap \{a\}$ ×

$A \to a$

so it is not LL(1) because both of them are going in different cells.

(10) $S \to aB \mid \epsilon$ $\quad \{a\}$ $\quad \text{Follow}(S) = \text{Follow}(C) = \text{Follow}(B) = \text{Follow}(S) = \{\$\}$

$B \to \boxed{bC} \mid \epsilon$ $\quad \{b\}$ $\quad \text{Follow}(C) = \text{Follow}(B) = \text{Follow}(S) = \{\$\}$

$C \to cS \mid \epsilon$ $\quad \{c\}$ $\quad \text{Follow}(S) = \text{Follow}(C) = \text{Follow}(B) = \text{Follow}(S) = \{\$\}$

so this is LL(1) grammar.

(11) $S \to AB$

$A \to a \mid \epsilon$ $\quad \{a\}$ { $\quad \text{Follow}(A) = \text{Follow}(B) = \{b, \$\}$

$B \to b \mid \epsilon$ $\quad \{b\}$ $\quad \text{Follow}(B) = \text{Follow}(S) = \{\$\}$

so this is LL(1) grammar

(12) $S \to aSA \mid \epsilon$ $\quad \{a\}$ $\quad \text{Follow}(S) = \text{First}(A) = \{c, \$\}$

$A \to c \mid \epsilon$ $\quad \{c\}$ $\quad \text{Follow}(A) = \text{Follow}(S) = \text{First}(A) = \{c\}$

not LL(1) due to multiple entries for same production

LectureNotes.in

Scanned by CamScanner

(13) $S \to A$

$A \to Bb \mid Cd$ $\{a, b\} \cap \{c, d\}$

$B \to aB \mid \epsilon$ $\{a\} \cap \text{Follow}(B) = \{b\}$

$C \to cC \mid \epsilon$ $\{c\} \cap \text{Follow}(C) = \{d\}$

so this is LL(1) grammar.

(14) $S \to aAa \mid \epsilon$ $\{a\} \to \text{Follow}(S) = \{\$, a\}$ X

$A \to abS \mid \epsilon$

multiple entries for a single production so not LL(1)

(15) $S \to iEtSS' \mid a$ $\{i\} \cap \{a\}$

$S' \to eS \mid \epsilon$ $\{e\} \cap \to \text{Follow}(S') = \text{Follow}(S)$

X $= \text{First}(S') = \{e\}$

$E \to b$

so not LL(1) grammar.

(16) $S \to ACB \mid CbB \mid Bd$ $\{d\} \cap \{b\} \cap \{g, d\}$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to b \mid \epsilon$ $\{b\}$ $\text{Follow}(C) = \text{Follow}(A) = \text{First}(C)$

X $= \{b, \$\}$

so not LL(1) grammar

LectureNotes.in

Scanned by CamScanner

Bottom up Parser

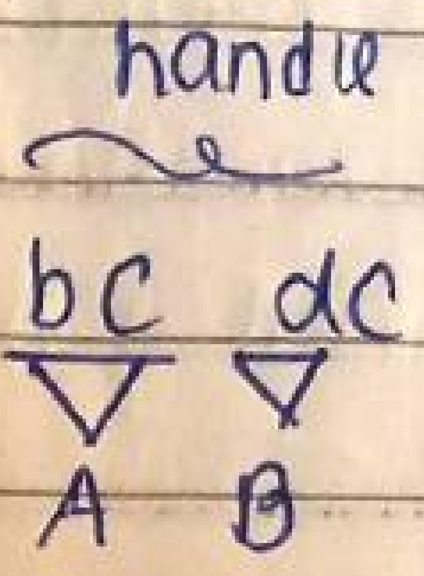
- The process of construction of the parse tree in the bottom-up-manner that starts with children and proceeds to the root is called bottom-up-parser

eg: $S \rightarrow aABC$

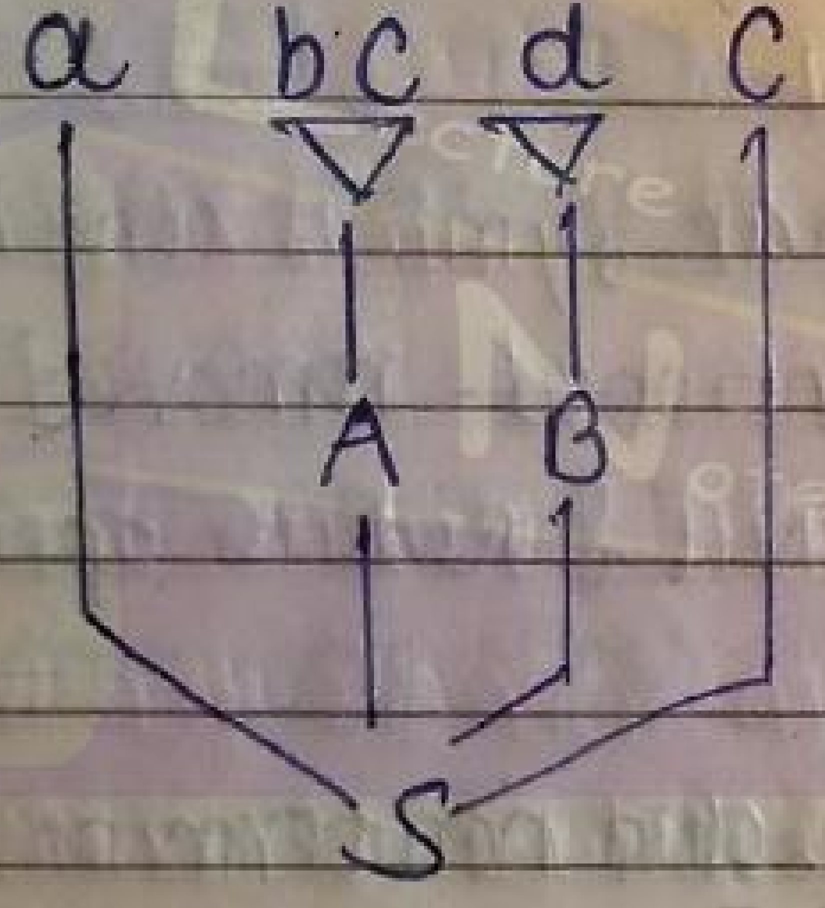
$A \rightarrow bc \mid aB$

$B \rightarrow d$

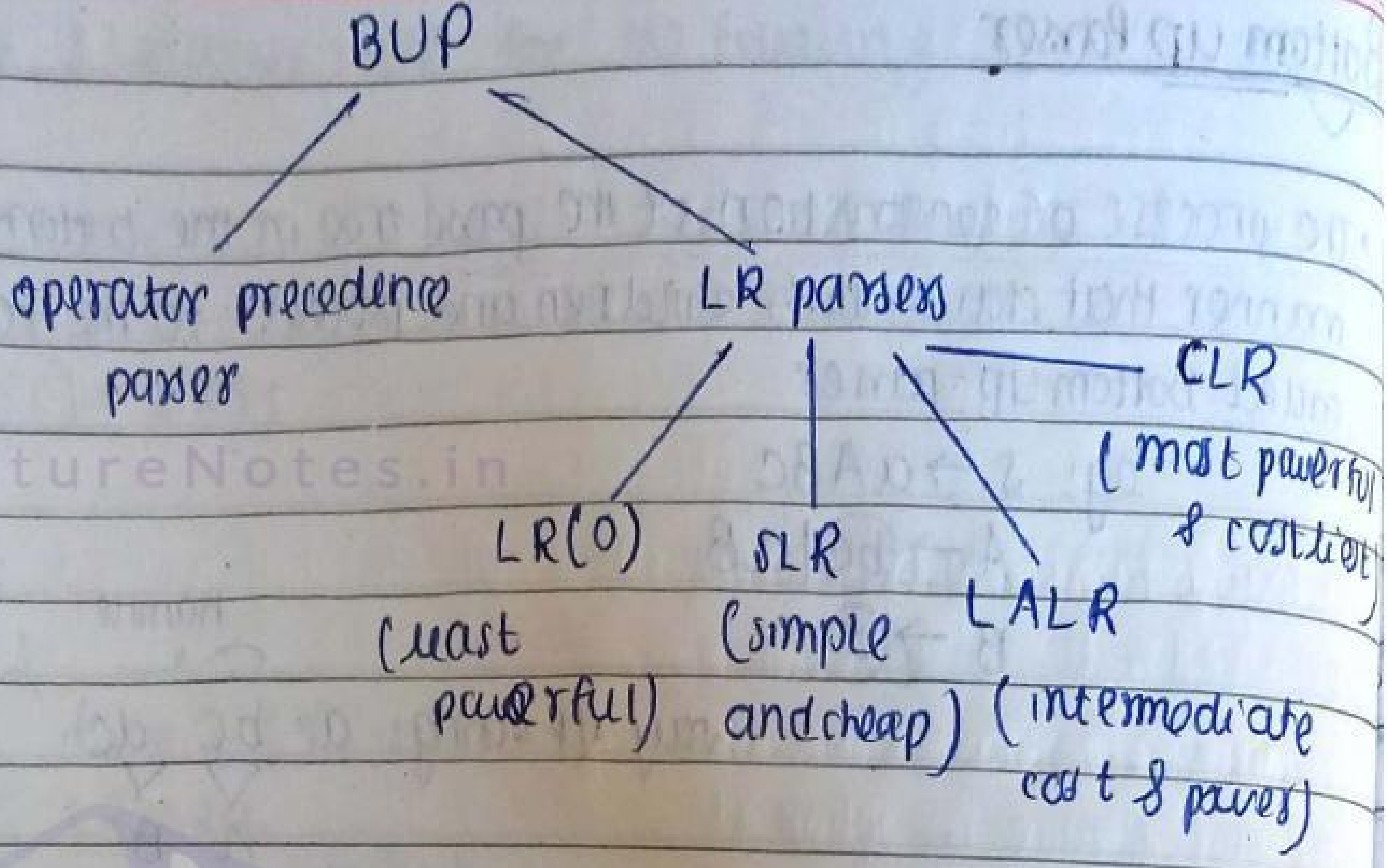
say i/p string: a bc dc



<u>handle</u>: substring which is getting matched with anything in the right handside.

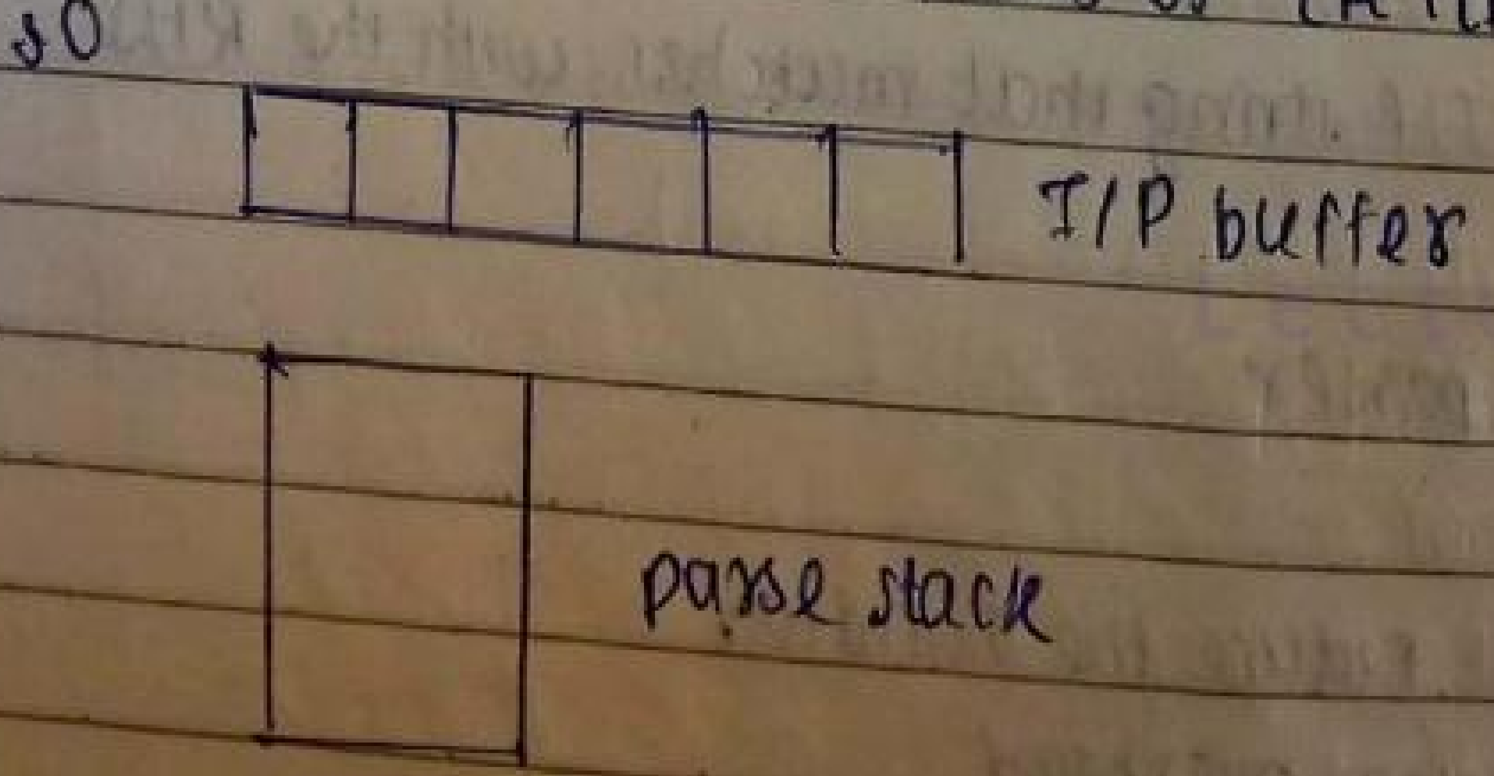


- handle is a substring of given I/P string that matches with the RHS of any production
- also called shift reduce (SR) parser
- uses the reverse of RMD
- has to detect the handle and reduce the handle.
- detecting the handle is the main overhead
- BUP is constructed for unambiguous grammar except for operator precedence parser
- can be constructed for the grammar which has more complexity
- parsing mechanism is faster than topdown parser with higher performance



Components

- I/P buffer ——— consists of I/P string to be parsed and I/P string ends with the endmaker symbol.
- parse stack ——— grammar symbols which are inserted or ~~mentioned~~ removed from stack using Shift & Reduce operations.
- parse table ——— terminals and non-terminals and compiler items or LR items



SR algorithm

	a	b	c	A	B	C	D
I0							
I1	S/R						
In							

- action part
  - shift and reduce operations
  - will be implied on terminal elements
- goto part
  - only shift operation
  - performs shift operation only on non-terminal elements

Operations

- shift operation can be applied whenever the handle does not occur from the top symbol of the stack.
- reduce operation can be used whenever handle occurs from the top symbol of the stack.
- accept operation is used after passing the complete I/P string, if the stack consists only of the start symbol then the I/P string and the parse is successful.
- error operation proceeding the I/P if the stack does not contain the start symbol at the end then the I/P string is not parsed that is the parse tree is not constructed. so the result is error.

eg: $S \to AA$
$A \to aA$
$A \to b$
say $w = abab$

stack	I/P	Action
$	abab$	shift
$a	bab$	shift
$ab	ab$	reduce
$aA	ab$	reduce
$A	ab$	shift
$Aa	b$	shift
$Aab	$	reduce
$AaA	$	reduce
$AA	$	reduce
$ ~~A~~S	$	~~reduce~~ accept

Classification of LR parser:

Based on the construction of table is divided into 4 types
LR(0), SLR(1), CLR(1), LALR(1)

Procedure

(a) obtain the argumented grammar for the given grammar
(b) create the canonical collection of LR items
(c) grammar DFA is created and prepare the table based on LR item.

argumented grammar

- grammar which is obtained by adding one more production that derives the start symbol is called argumented grammar

$S' \to .S$
$S \to AB$
$A \to a$
$B \to b$

- helps in separating the final item from the non-final item
- the need for argumented grammar is when you have multiple productions in start symbol then we can decide what is the final string.

eg: a production '.' at any point in the RHS is called LR(0) item

$A \to .abc$
$A \to a.bc$
$A \to ab.c$
$A \to abc.$ → final item

LectureNotes.in

Scanned by CamScanner

canonical collection

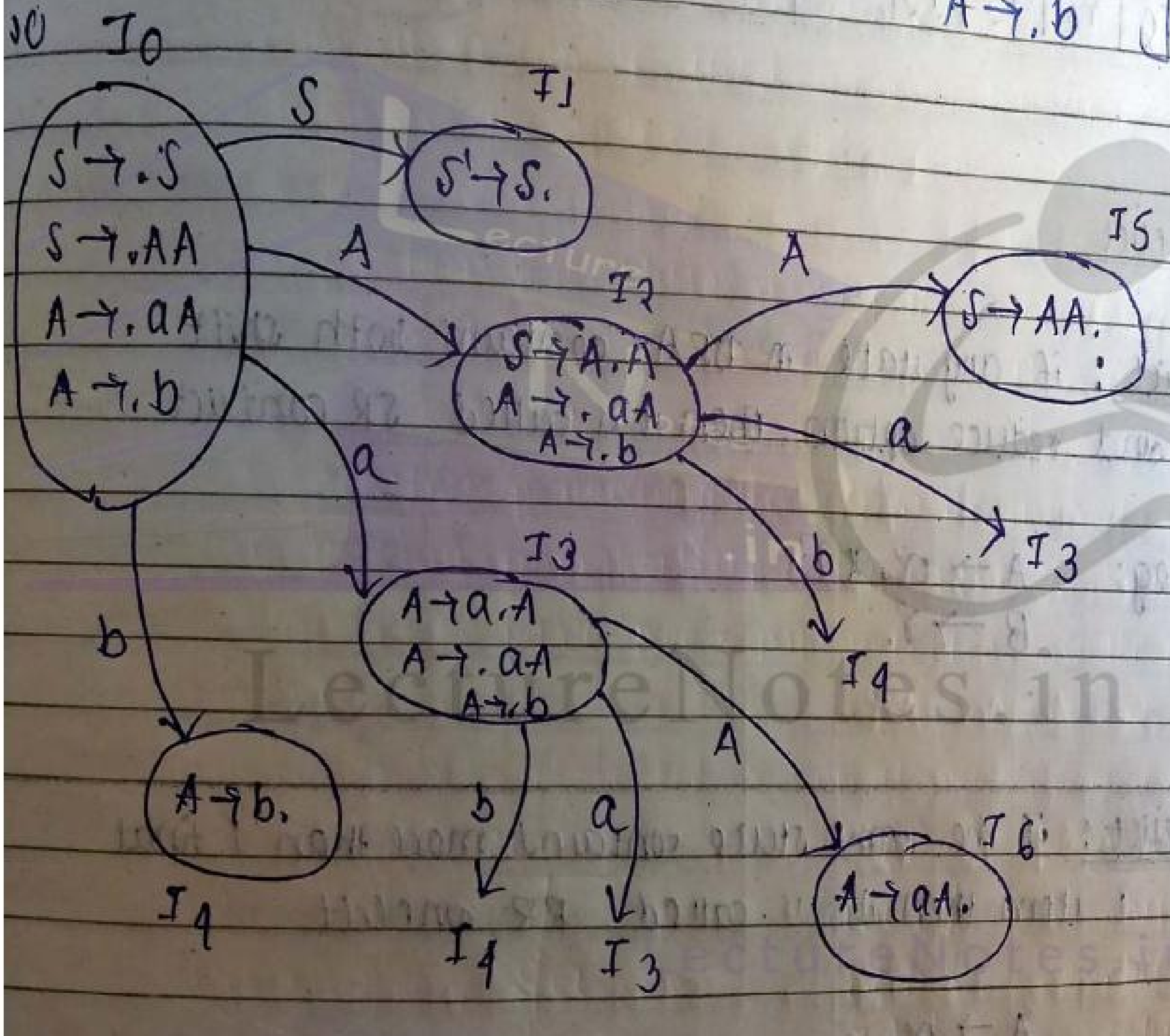
LR(0)

Q $S \to AA$ r_1
$A \to aA$ r_2
$A \to b$ r_3

sol: Step 1: create argumented grammar

$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$
Ist compiler I_0

so I_0



construction of LR(0) parse table.

2 parts — action part
— goto part

Rules

(a) $Goto(I_i, X) = I_j$ where $X \to$ terminal elements
(b) $Goto(I_i, X) = J$ where $X \to$ non-terminal "
(c) If I_i is a final item that represents the rule r_i

Scanned by CamScanner

LR(0) parse table

	action			goto	
	a	b	$	S	A
I0	S3	S4		1	2
I1	Acc	Acc	Acc		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

no conflict so LR(0) grammar

Types of conflict

(a) **SR conflict** : if any state in DFA contains both shift and reduce option then it is called SR conflict.

eg: $A \rightarrow \alpha . xB$
$B \rightarrow \gamma .$

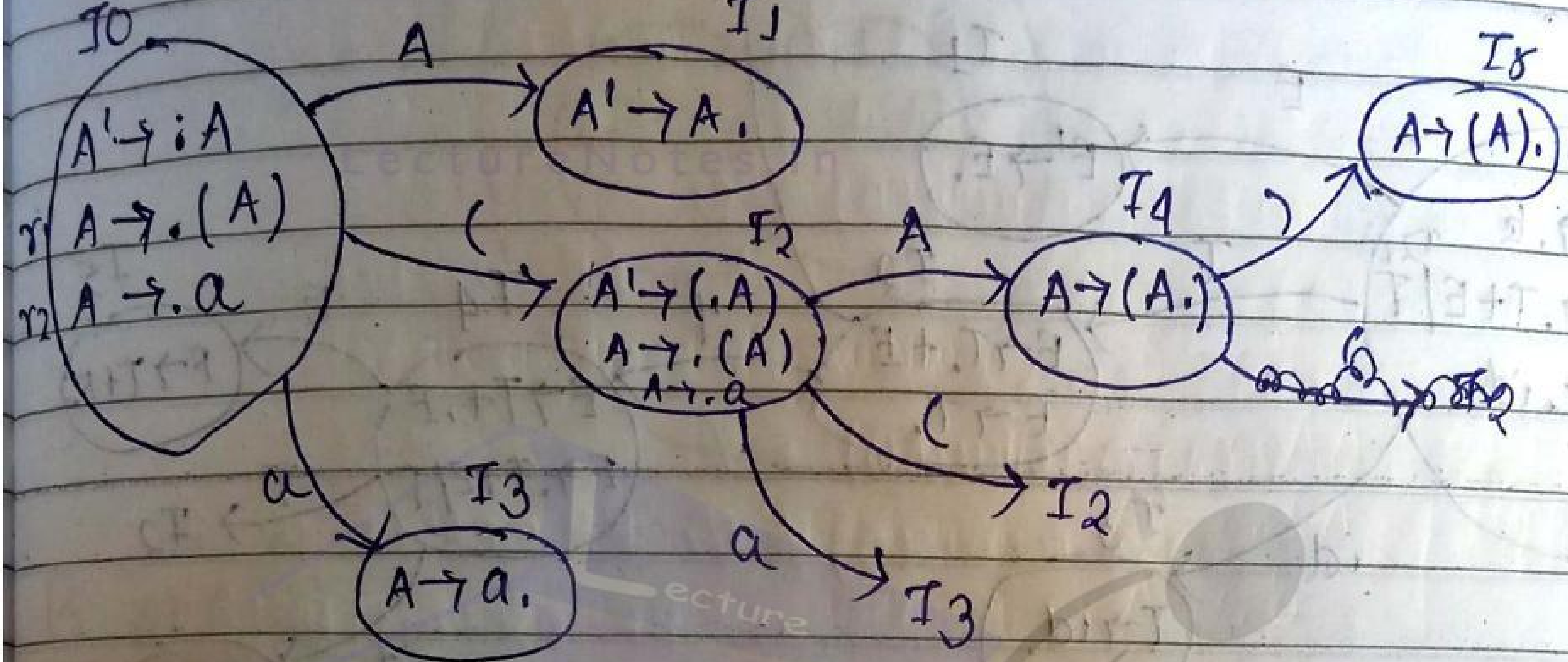
(b) **RR conflict** : if the same state contains more than 1 final item then it is called RR conflict

$A \rightarrow \alpha .$
$B \rightarrow \gamma .$

LectureNotes.in

Scanned by CamScanner

Q A → (A) | a check for LR(0)



so
parse table is

	a	(	)	$	A
I0	S3	S2			1
I1	ACC	ACC	ACC	ACC	
I2	S3	S2			4
I3	r2	r2	r2	r2	
I4	r1	r1	r1	r1	
I5	r1	r1	r1	r2	

no conflicts so this is an LR(0) grammar.

Scanned by CamScanner

Q $E \to T+E \mid T$
$T \to id$

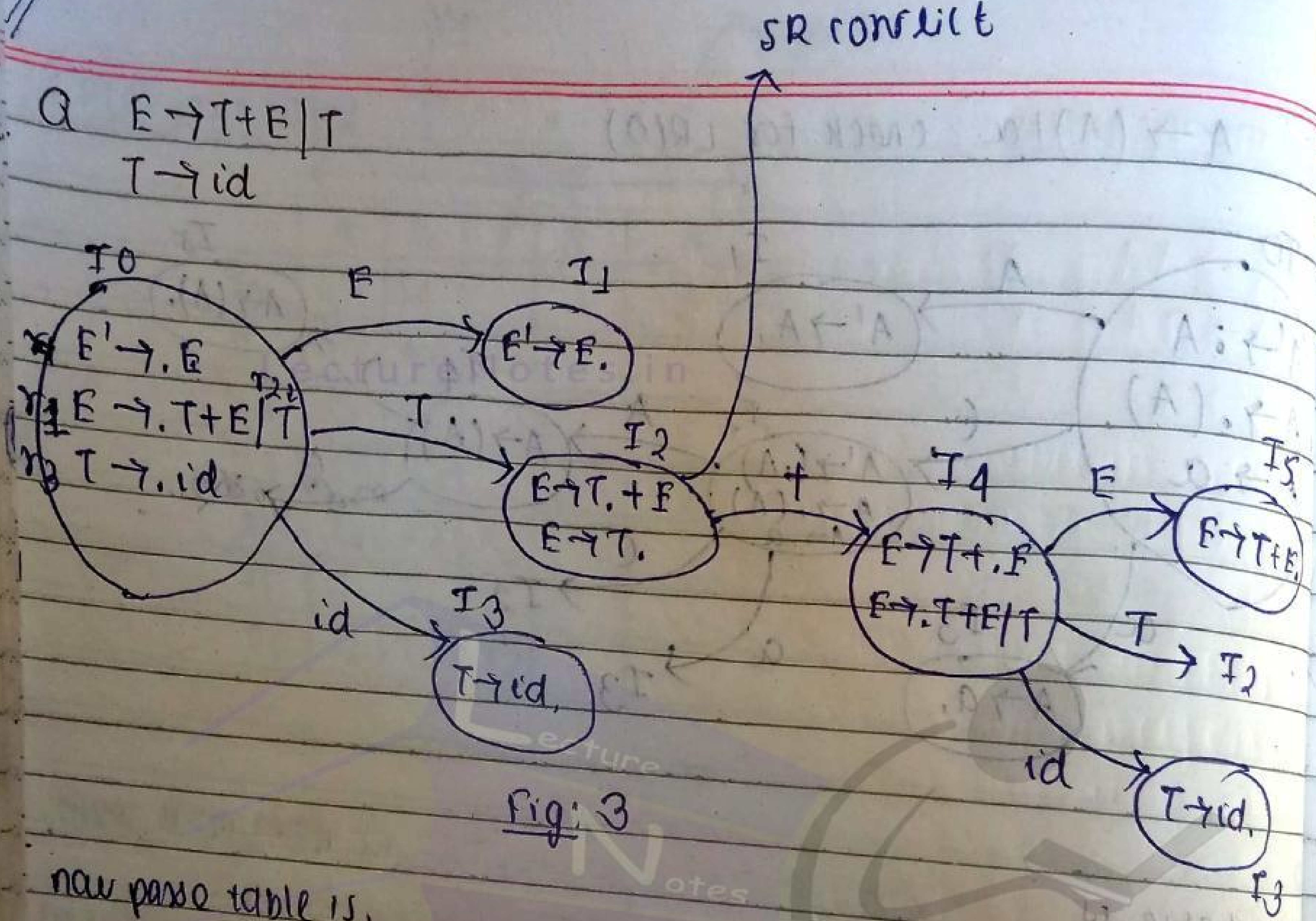
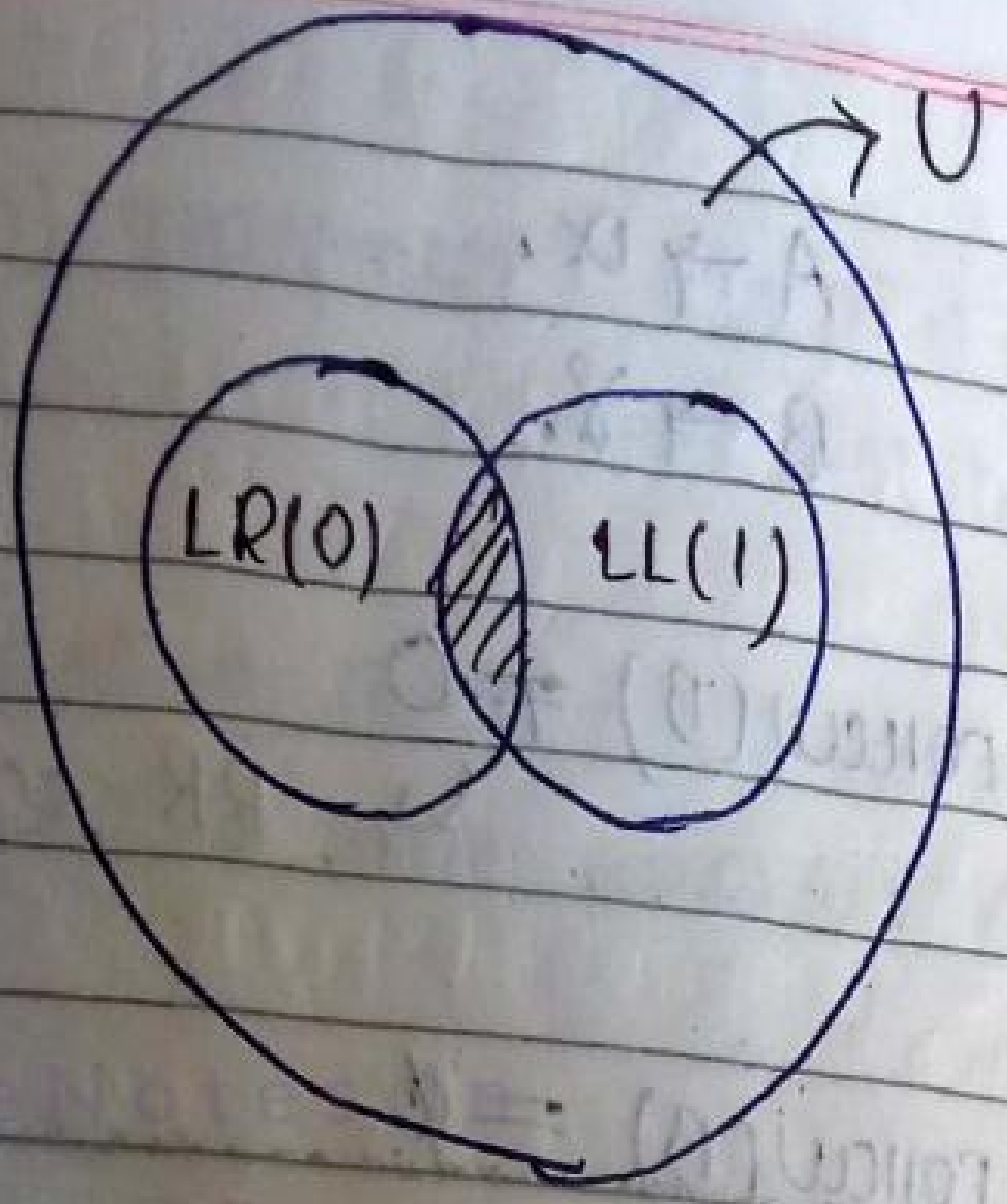


Fig: 3

now parse table is,

	+	id	$	E	T
I0		S3		1	2
I1	Acc	Acc	Acc		
I2	S4 r2	r2	r2		
I3	r3	r3	r3		
I4				5	2
I5	r1	r1	r1		

Scanned by CamScanner



B SLR(1)

- Procedure for constructing the parse table is similar to LR(0) parser, but there is one restriction on reducing the entries, whenever there is a final item then place the reduce entries under the Follow symbol of LHS variable.
- if the SLR parse table does not contain multiple entries i.e. it is free from conflicts then it is SLR(1) grammar

(a) SR conflict

$A \to \alpha . x B$
$B \to \gamma .$

shortcut

- if Follow(B) $\cap \{x\} \neq \phi$
  ↳ SR conflict in SLR(1)

- if Follow(B) $\cap \{x\} = \phi$
  ↳ SR conflict in LR(0) but not in SLR(1)

(b) **RR conflict**

$A \to \alpha.$
$B \to \gamma.$

Shortcut

- if $Follow(A) \cap Follow(B) \neq 0$
  $\hookrightarrow$ RR conflict in SLR(1)

- if $Follow(A) \cap Follow(B) = \phi$
  $\hookrightarrow$ RR conflict in LR(0) not in SLR(1)

eg: $E \to T + E \mid T$
$T \to id$

N.B TURN BACK TO FIG(3)

then, we can see: I_2 has SR conflict

$E \to T. + E$
$E \to T.$

$Follow(E) \cap \{+\}$
$= \{\$\} \cap \{+\}$
$= \phi$

so the grammar has SR(0) conflict in LR(0) but not in SLR(1)

so this is a SLR(1) grammar.

LectureNotes.in

Scanned by CamScanner

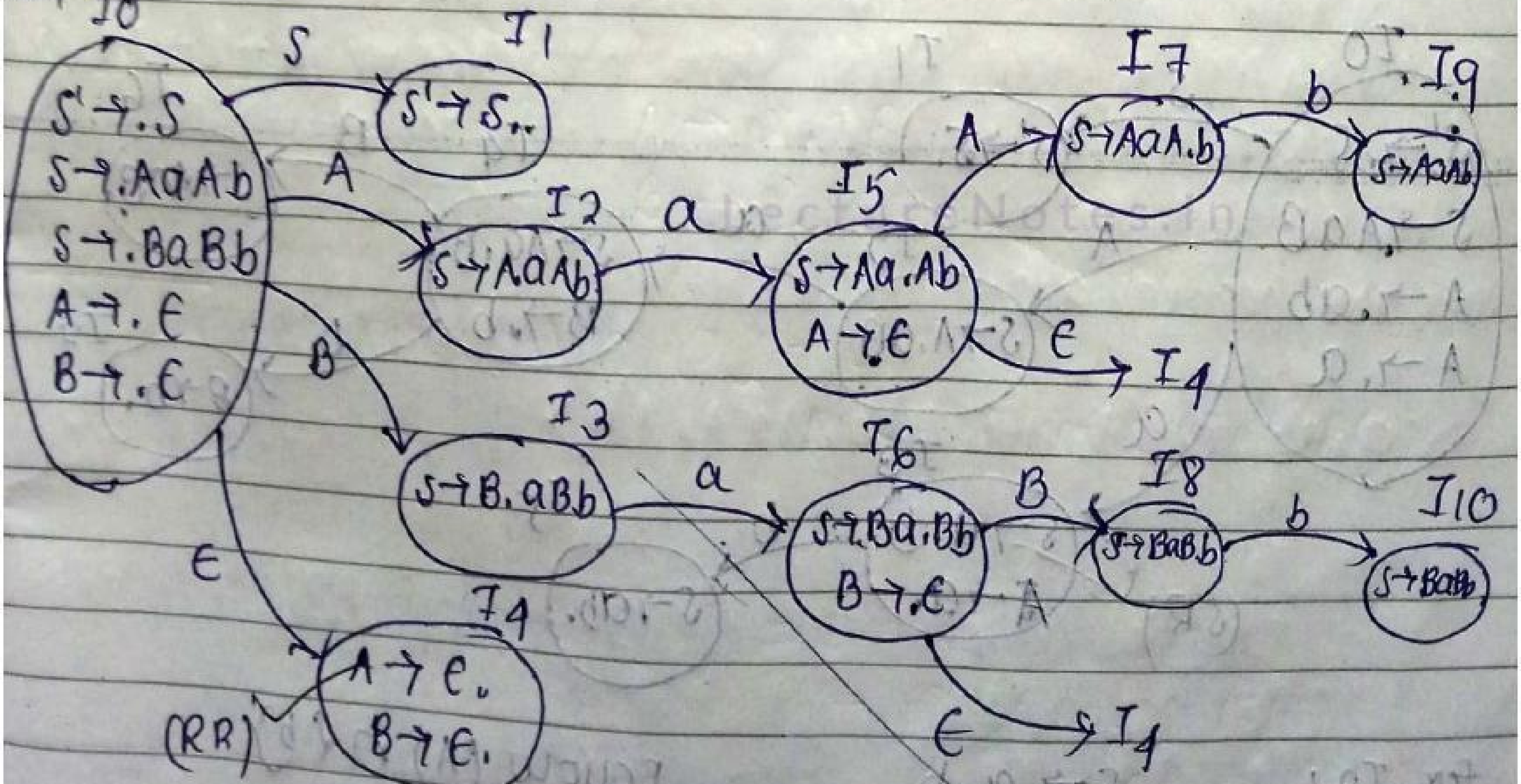
Properties

- Every LR(0) grammar is SLR(1) grammar
- Every SLR(1) " need not be LR(0)

- no. of entries in SLR(1) <= no. of entries in LR(0) parse table
- SLR(1) parser is more powerful than LR(0).

eg:
S → AaAb r_1
S → BaBb r_2
A → ε r_3
B → ε r_4

so,



LectureNotes.in

Scanned by CamScanner

now using parse table,

	a	b	$	S	A	B
I4	r3 r4	r3 r4				

OR

$Follow(A) = \{a, b\}$

$Follow(B) = \{a, b\}$

so $Follow(A) \cap Follow(B) \neq \phi$

↳ so RR conflict in SLR(1) and in LR(0) also

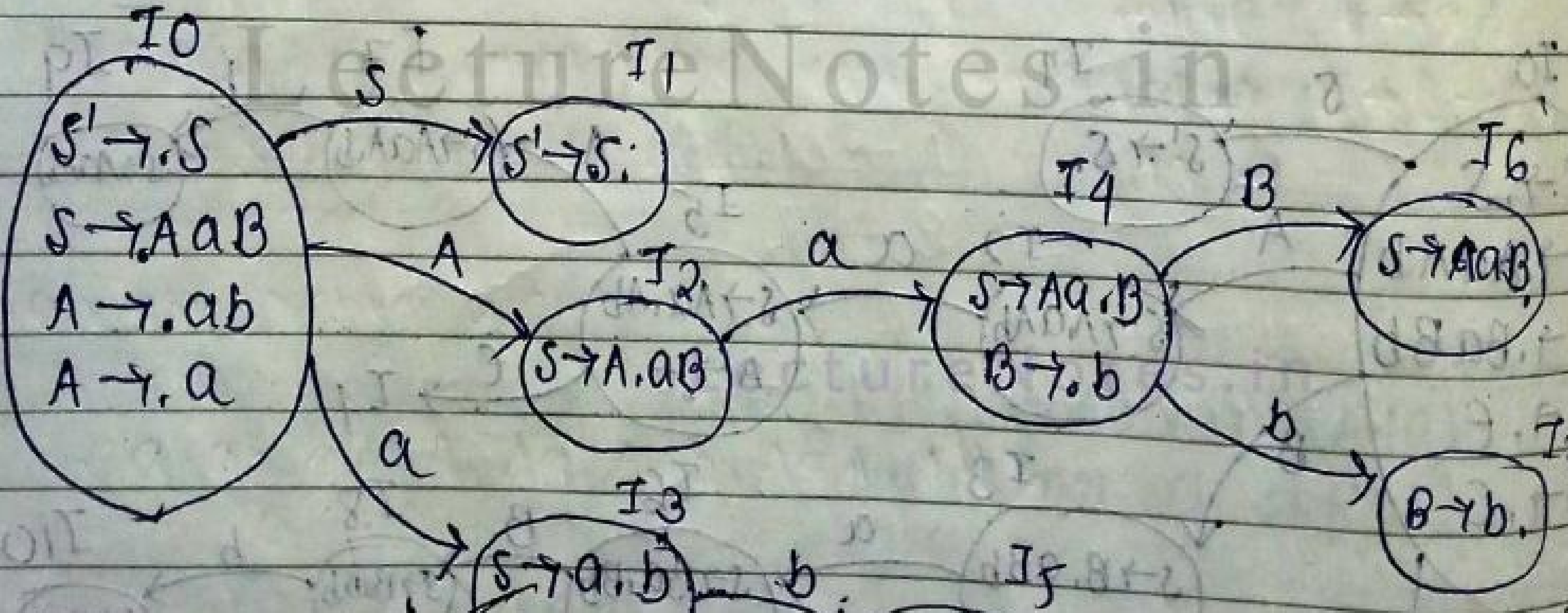
so this grammar is <u>not</u> a <u>SLR(1) grammar</u>.

eg:

$S \rightarrow AaB \quad r_1$

$A \rightarrow ab \mid a \quad r_2$

$B \rightarrow b \quad r_3$



for I3: $S \rightarrow a.b$

$A \rightarrow a.$

$Follow(A) \cap \{b\}$

$= \{a\} \cap \{b\}$

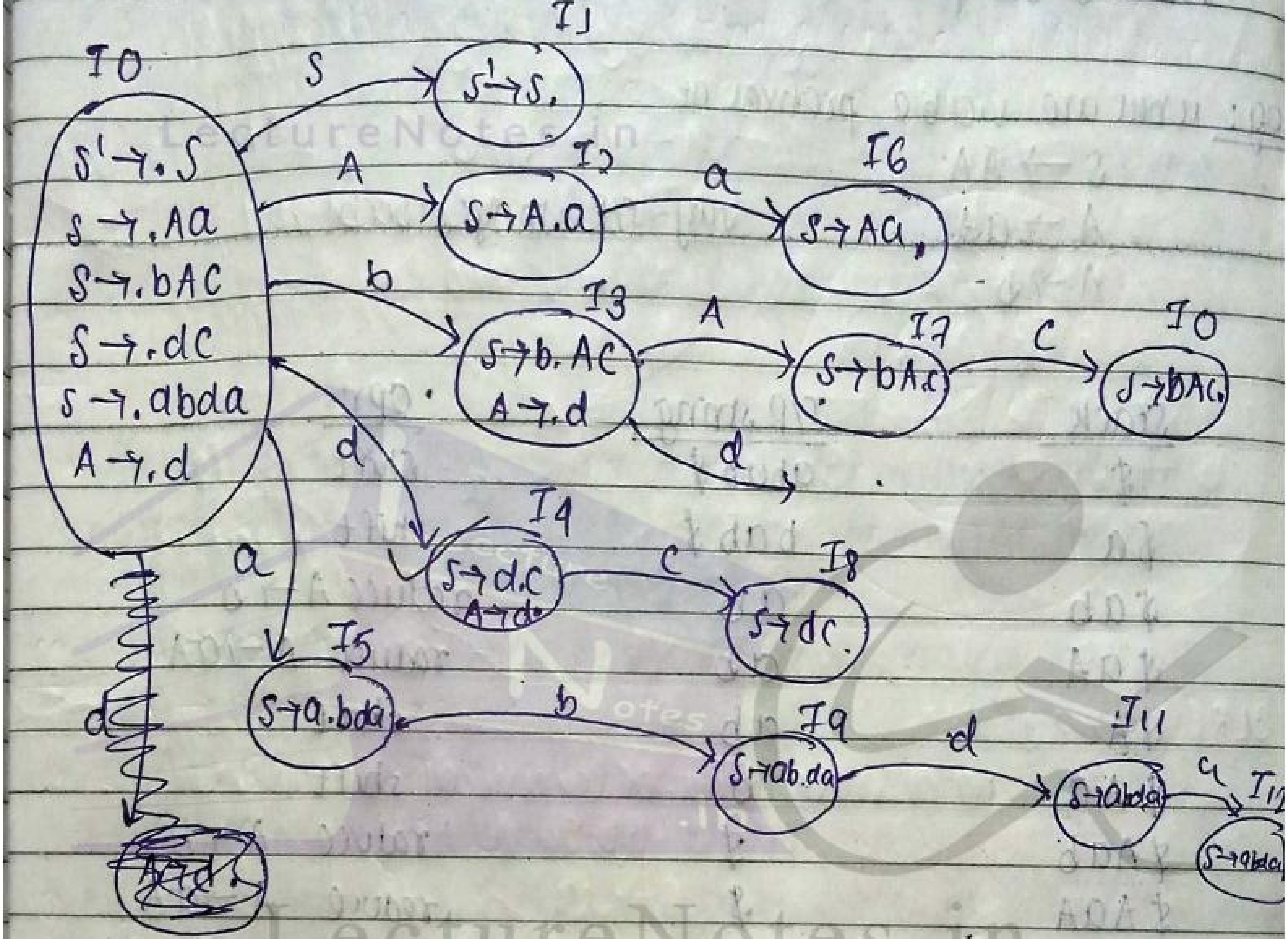
$= \phi$

so SR conflict in LR(0) not in SLR(1)

Scanned by CamScanner

so this is a SLR(1) grammar.

eg: $S \to Aa \mid bAC \mid dc \mid abda$
$A \to d$



so now for I4 (SR) $S \to d.c$
$A \to d.$

$\text{Follow}(A) \cap \{c\}$
$= \{a, c\} \cap \{c\}$
$\neq \emptyset$

so this an SR conflict in both SLR(1) and LR(0)

so this is not SLR(1) grammar

Viable prefix

- The prefixes of right sentential form that can appear on the stack of SR parser are called <u>viable prefixes.</u>

eg: what are viable prefixes of

$S \rightarrow AA$
$A \rightarrow aA$ say I/P string: abab
$A \rightarrow b$

Stack	I/P string	opr
\$	abab\$	shift
\$a	bab\$	shift
\$ab	ab	reduce $A \rightarrow b$
\$aA	ab	reduce $A \rightarrow aA$
\$A	ab	shift
\$Aa	b	shift
\$Aab	\$	reduce $A \rightarrow b$
\$AaA	\$	reduce $A \rightarrow aA$
\$AA	\$	reduce ~~A →~~ $S \rightarrow AA$
\$S	\$	accept

so viable prefixes are

<a, ab, aA; A, Aa, Aab, AaA, AA>

LectureNotes.in

Scanned by CamScanner

C LR(1)

Types — CLR(1) → canonical LR(1) parser

Types — LALR(1) → Look Ahead LR(1) parser

- LR(1) parser or LR(1) grammar depends on 1 lookahead symbol
- LR(0) " or LR(0) " doesn't " " any " "
- CLR(1) most powerful and mostly considered as LR(1)

- the grammar is LR(1) if it is free from multiple entries
- for every grammar if SLR(1) parser is constructed then CLR(1) parser is also constructed.
  But if CLR(1) parser is constructed we may or may not get construct SLR(1)

- CLR(1) is more powerful than SLR(1)
- no. of entries in SLR(1) <= no. of entries in CLR(1) parse table.

CLR(1)

eg: $S \rightarrow AA$
$A \rightarrow aA/b$

$S' \rightarrow .S, \$$
$S \rightarrow .AA, \$$
} augmented grammar

OR

$A \rightarrow \alpha.B, a/b$
$B \rightarrow .\gamma, a/b$

Rule

if $A \rightarrow \alpha.B\beta, a/b$

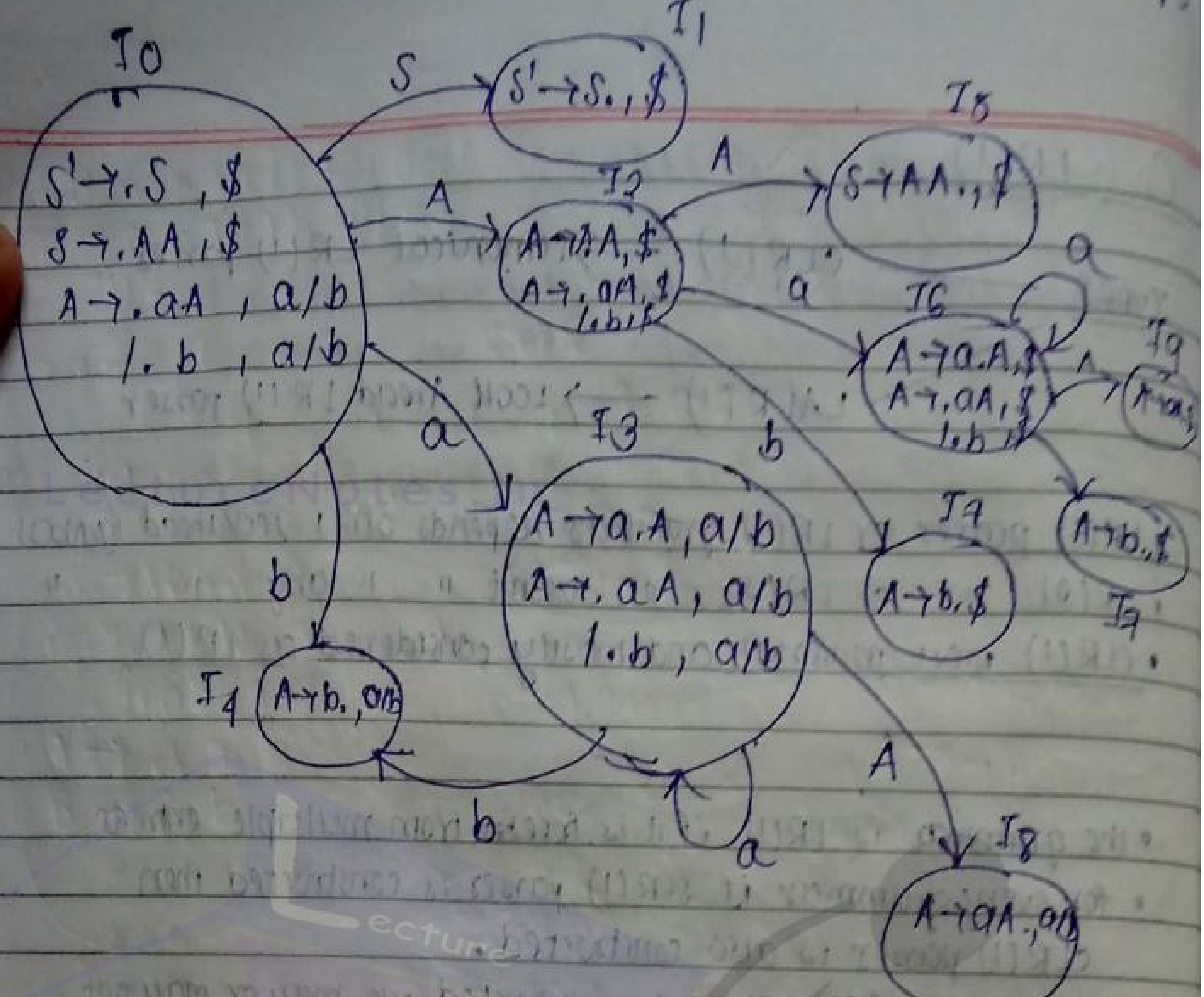
then $B \rightarrow .\gamma, c/d$

First of

LectureNotes.in

Scanned by CamScanner

whenever we do transition lookahead never change



CLR(1) parsing table,

no. of states = no. of canonical connection

Here I3 and I6 same LR(0) items but diff. lookahead symbol
I4 and I7 same LR(0) " " " " "
I8 and I9 " " " " " " "

	a	b	$	S	A
I0	S3	S4			
I1					
I2	S6	S7			5
I3	S3	S4			
I4	r3	r3			
I5			r1		
I6	S6	S7			9
I7			r3		
I8	r2	r2			
			r2		

I3, I6 → I36

I4, I7 → I47

I8, I9 → I89

no. of states in $CLR(1) \geq LR(0) = SLR(1)$
$= LALR(1)$

now reducing no. of states and this is LALR(1) parsing table

	a	b	\$	S	A
I0	S36	S47			2
I1					
I2	S36	S47			5
I36	S36	S47			89
I47	r3	r3	r3		
I5			r1		
I89	r2	r2	r2		

Conflicts

N.B if these 2 are same then SR conflict

~~shift~~

SR conflict

RR conflict

LR(1) items

$A \to \alpha.\boxed{a}\beta, c/d$
$B \to \gamma., \boxed{a}/\$$

$A \to \alpha., a$
$B \to \alpha., a$

conclusions

① The grammar is not CLR(1)
if it is not CLR(1) definitely not LALR(1)

② The grammar is CLR(1)
then it may or may not be LALR(1)

LectureNotes.in

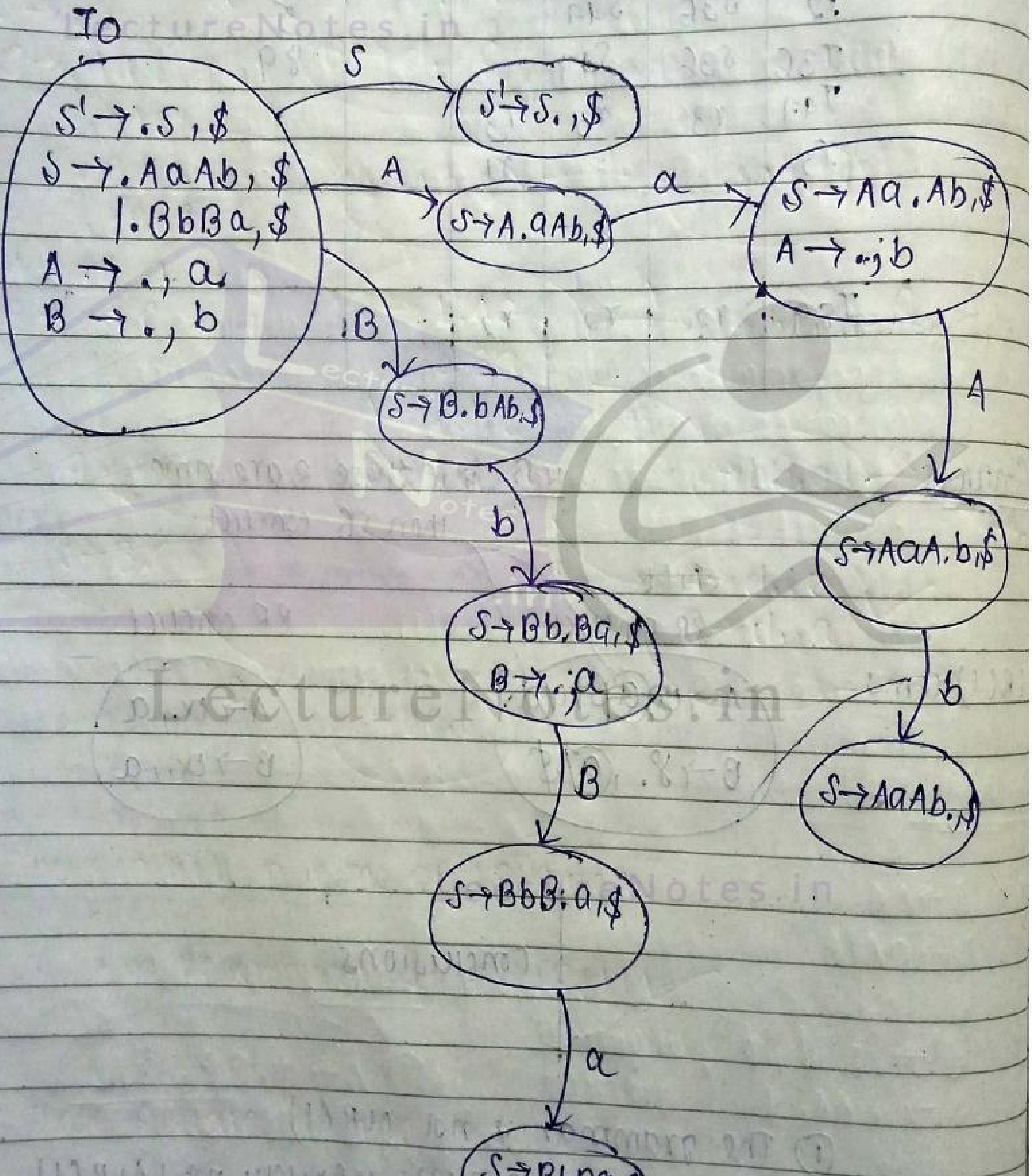
Scanned by CamScanner

eg: $S \to AaAb \mid BbBa$ LL(1) ✓

$A \to \epsilon$ LR(0) X

$B \to \epsilon$ SLR(1) X

check (a) CLR(1) (b) LALR(1)



Conclusion: No conflict

so CLR(1) ✓

Scanned by CamScanner

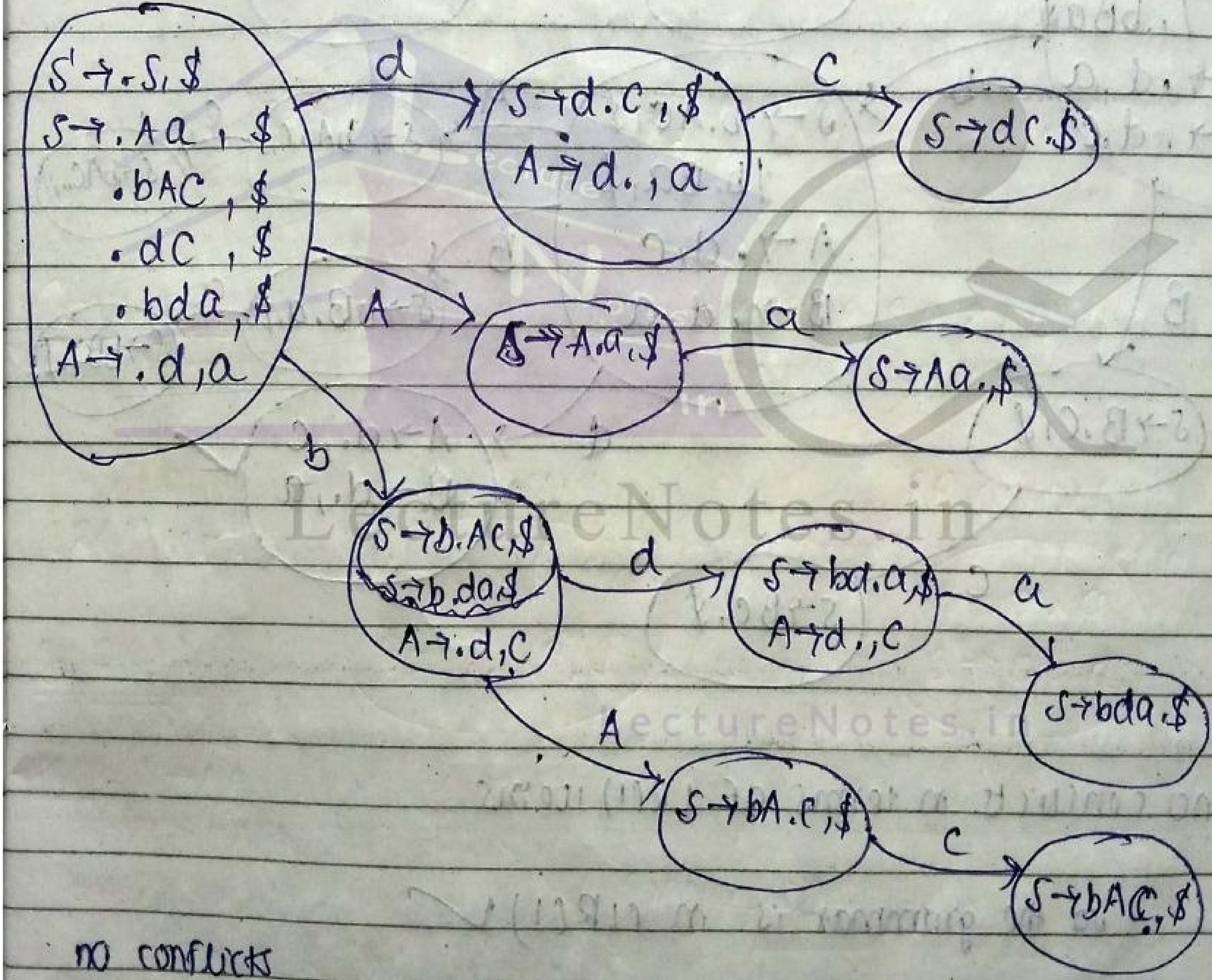
there cannot be any merging

bcs no 2 LR(0) items are same

so LALR(1) ✓

eg: $S \to Aa / bAC / dC / bda$

$A \to d$



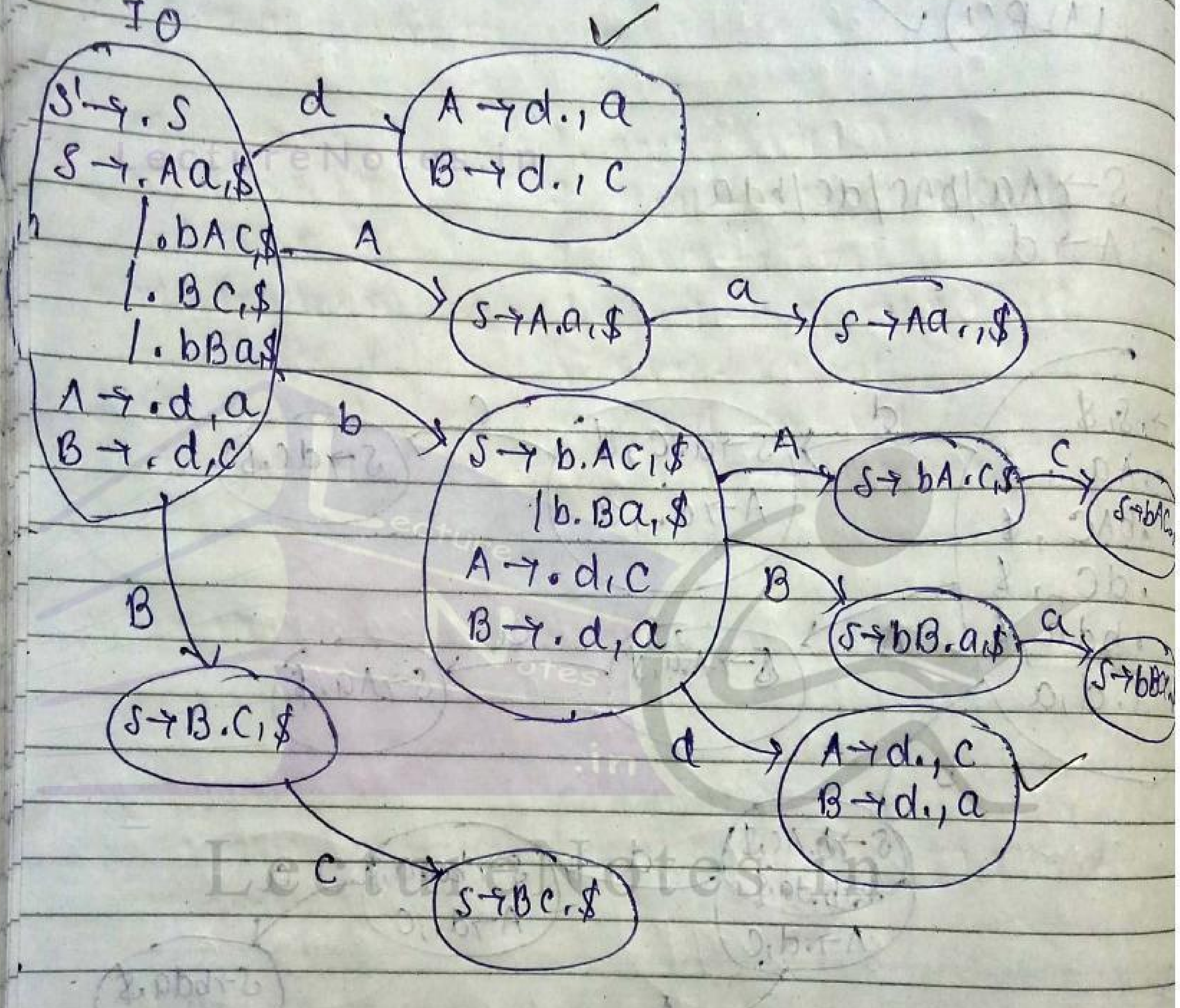
no conflicts

so it is CLR(1)

and also LALR(1)

LectureNotes.in

eg: $S \to Aa \mid bAC \mid Bc \mid bBa$
$A \to d$
$B \to d$



no conflicts in terms of LR(1) items

∴ so the grammar is in CLR(1) ✓

These 2 states we have to merge
we get

$A \to d.,\ a/c$
$B \to d.,\ c/a$

so not LALR(1) ✗

LectureNotes.in

~~Sen~~ sorting ~~ee~~ accdng to power

CLR(1) > ~~SLR(1)~~ LALR(1) > SLR(1) > LR(0) = LL(1)

LectureNotes.in

LectureNotes.in

Scanned by CamScanner

Semantic Analysis.

- Semantic analyser verifies each and every sentence of the source code. syntax analyser/parser takes care of the operator working on the wide number of operands and considers the type of operands.

- Semantic analysis can be implemented by passing along ~~KA~~ with the semantic tools; by attaching the semantic rules for each and every grammar,

Syntax vs Semantics

- syntax determines valid form of program.
- semantics " behavior of valid program.

- syntax is what can be specified by CFG -does not match intuition
  → some things that seem to be syntax are not definable in CFG.

  eg: number of arguments in function call.
- anything that requires compiler to compare constructs separated with other code, or to count items, or nested structures are semantics

SDT (Syntax Directed Translation)

- The grammar with semantic rules is known as SDT.
- models for translation from parse trees into assembly/machine code.

- These rules in effect specify the flow of information between different points in the parse tree for a program.

Scanned by CamScanner

~~eg: E → E+E~~
~~E → id~~

attribute
value

eg: Rules of grammar

1. L → E	1 print (E.val)
2. E → E₁+T	2 E.val → E₁.val + T.val
3. E → T	3. E.val → T.val
4. T → T₁ * F	4. T.val → T₁.val * F.val
5. T → F	5. T.val → F.val
6. F → (E)	6. F.val → E.val
7. F → id	7. F.val → id.lexval

Types of attributes

(a) synthesized

- value is only computed when symbol is on LHS of the production.

- attributes can be computed independently of context.

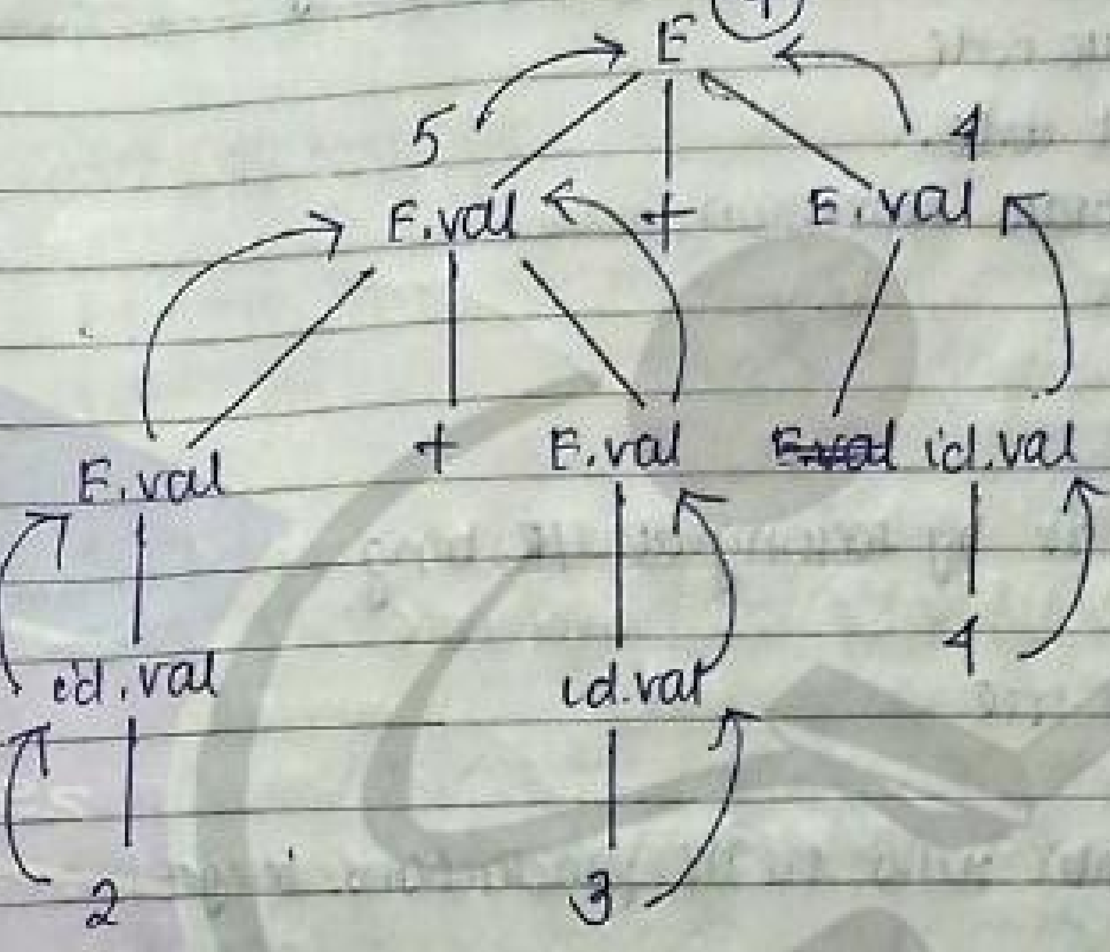
(b) Inherited

- value computed in productions where symbol is on RHS
- attributes computed using context.
- cannot avoid those in semantic analysis.

eg: E → E+E given $w = 2+3+4$
E → id

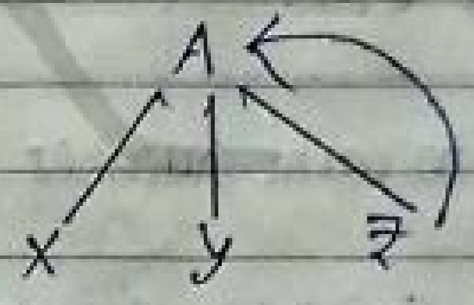
so, E.val = E.val + E.val
E.val = id.val

then the grammar will be $w = id+id+id$



decorated OR annotated parse tree (that shows the attribute value at each and every node)

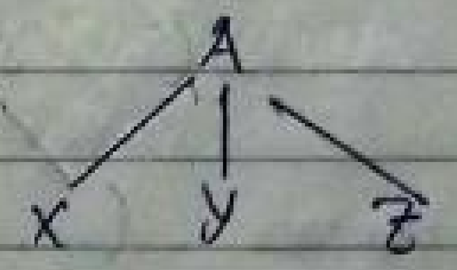
eg: A → XYZ



attribute value is evaluated by the attribute value of its children is called synthesized attribute

$A.S = f(X.x \mid Y.y \mid Z.z)$

eg: A → XYZ



attribute whose value is evaluated with help of attribute value of its parent or its siblings is called inherited attribute

$Y.y = f(A.S \mid X.x \mid Z.z)$

Scanned by CamScanner

N.B

- apart from parsing, the SDT can be used to store the typed information into symbol table.
- to build the syntax tree or DAG (Directed Acyclic Graph)
- to verify consistency check —— type checking
  parameter checking
- to generate intermediate code
- to generate the target code.
- to evaluate the algebraic expression.

Rules to construct

Step 1: define the grammar by looking at I/P string

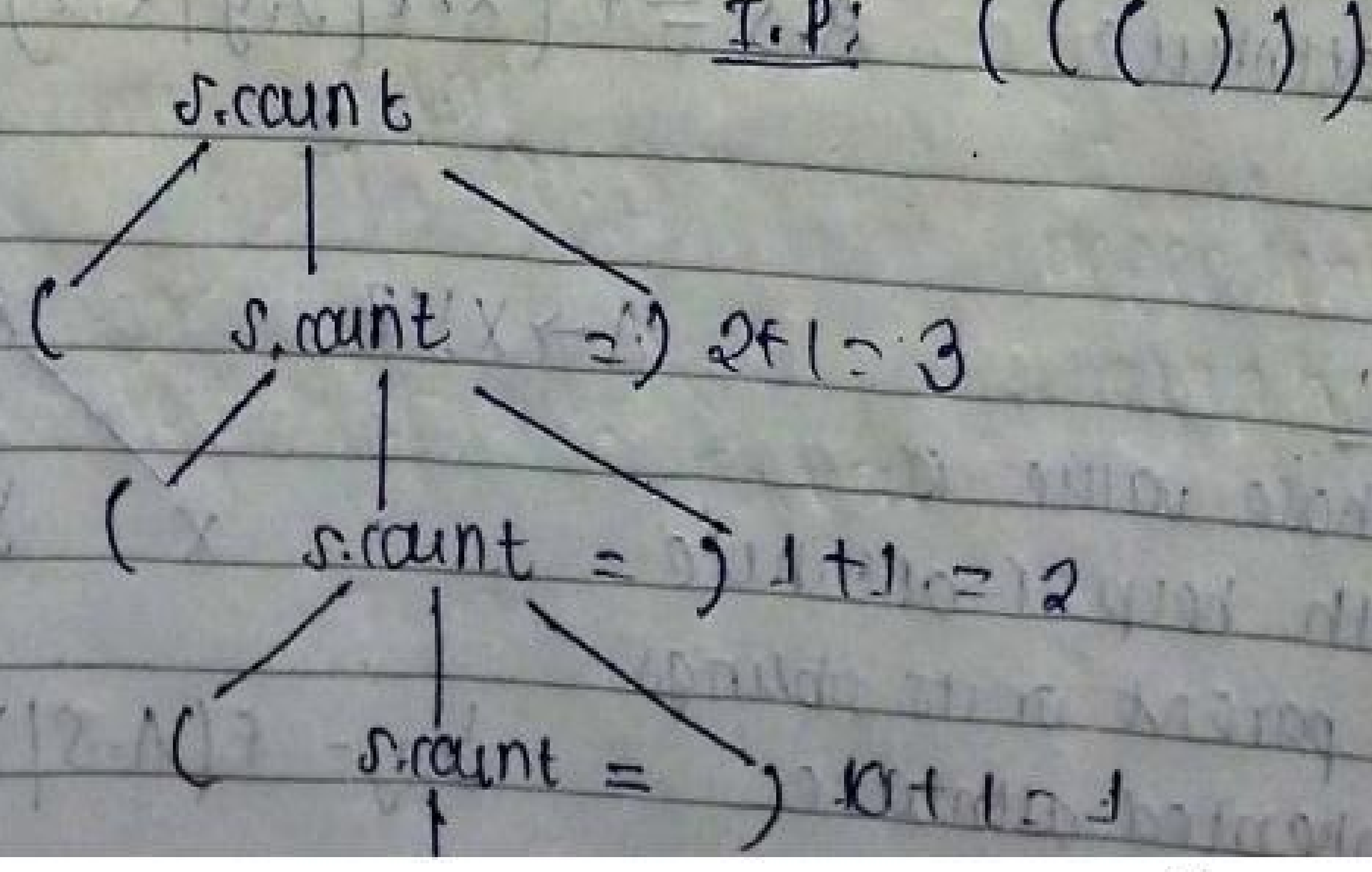
Step 2: construct the parse tree

Step 3: attach the semantic rules to the productions to get the desired O/P.

eg: draw SDT to count number of paranthesis in a statement

$S \rightarrow (S)$ // S.count = S.count + 1
$S \rightarrow \epsilon$ // S.count = 0

I.P: ((()))



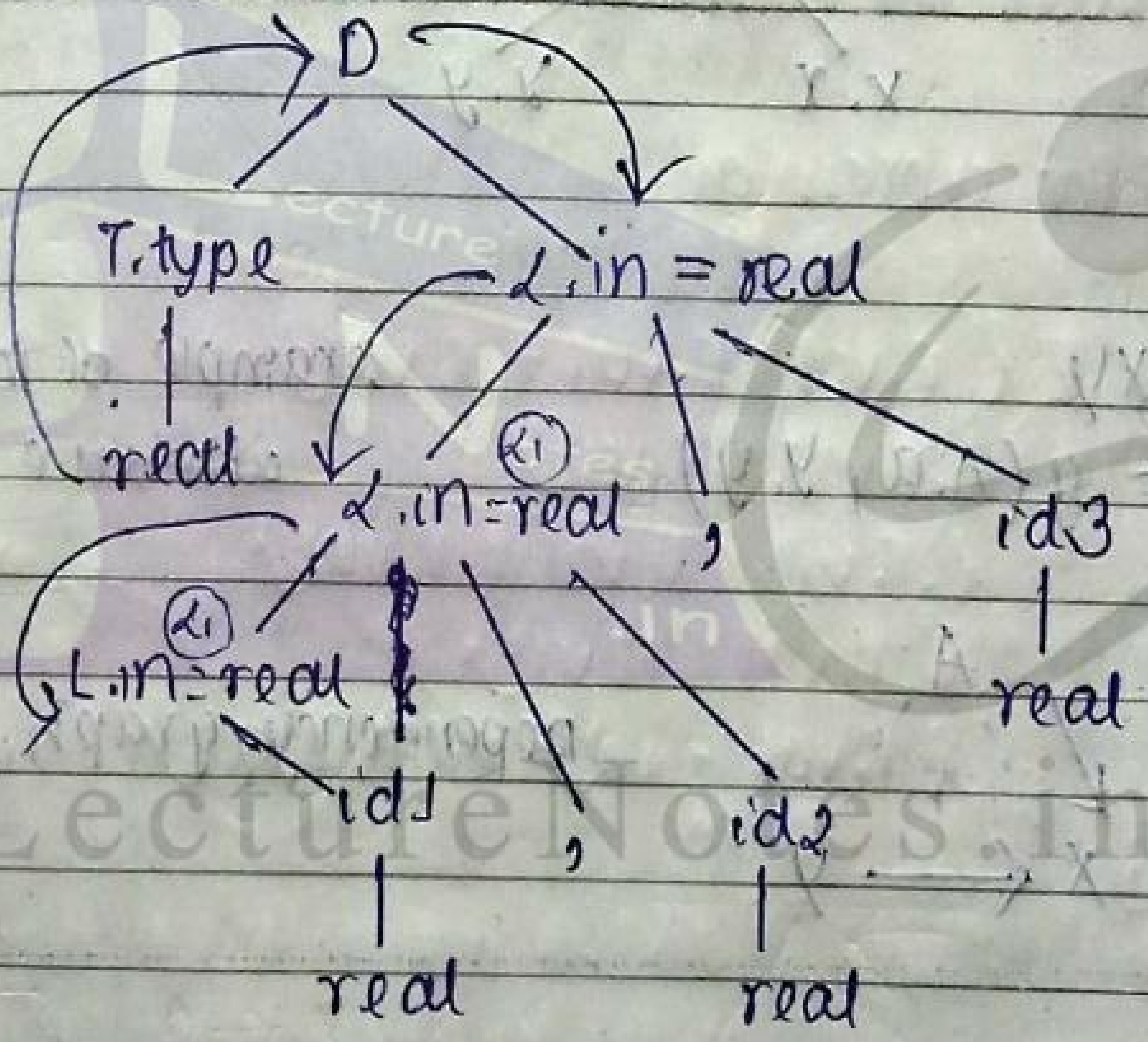
LectureNotes.in

Scanned by CamScanner

eg: draw SDT for the following

Inherited attributes	Semantic rules
D → TL	L.in = T.type
T → int	T.type = int
T → real	T.type = real
L → L1, id	L1.in = L.in
L → id	add.type (id.entry, L.in)

I/P : real id1, id2, id3



symbol table

	Type
id3	real
id2	real
id1	real

LectureNotes.in

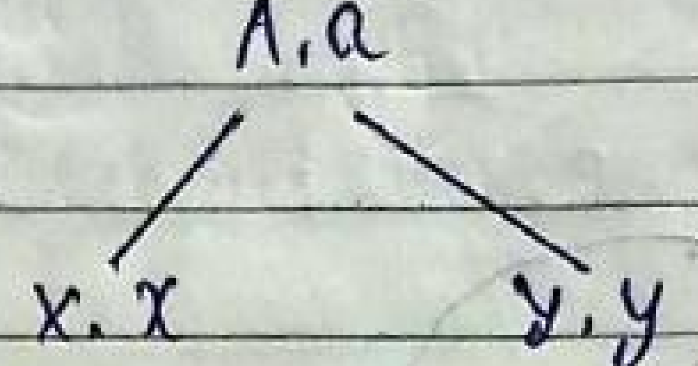
Scanned by CamScanner

Dependency Graph

The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be shown by a directed graph called, dependency graph.

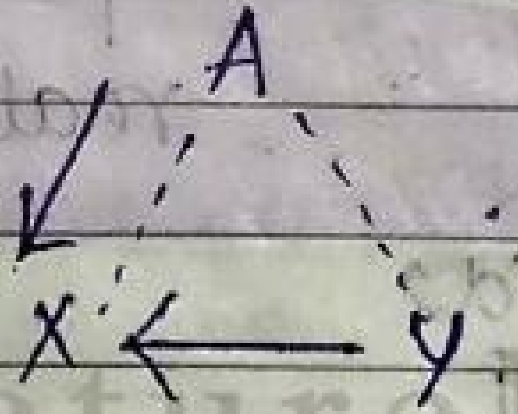
eg: $A \rightarrow XY$ example of synthesized attribute

$A.a = f(X.x, Y.y)$



eg: $A \rightarrow XY$ example of inherited attribute

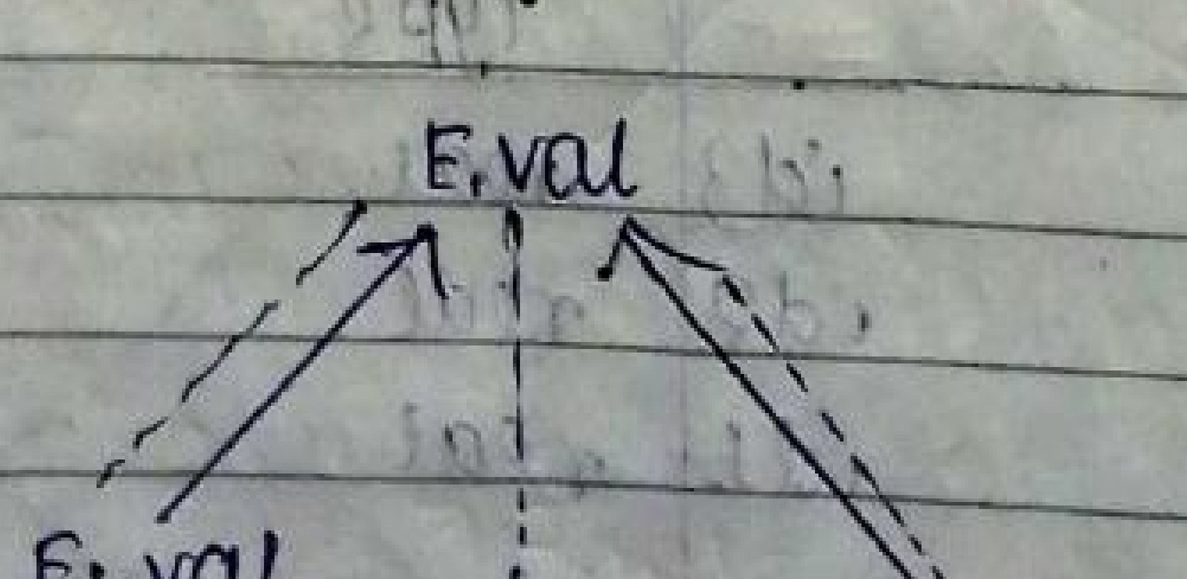
$X.i = g(A.a, Y.y)$



Dependency graph

eg: $E \rightarrow E_1 + E_2$

semantic rule: $E.val = E_1.val + E_2.val$

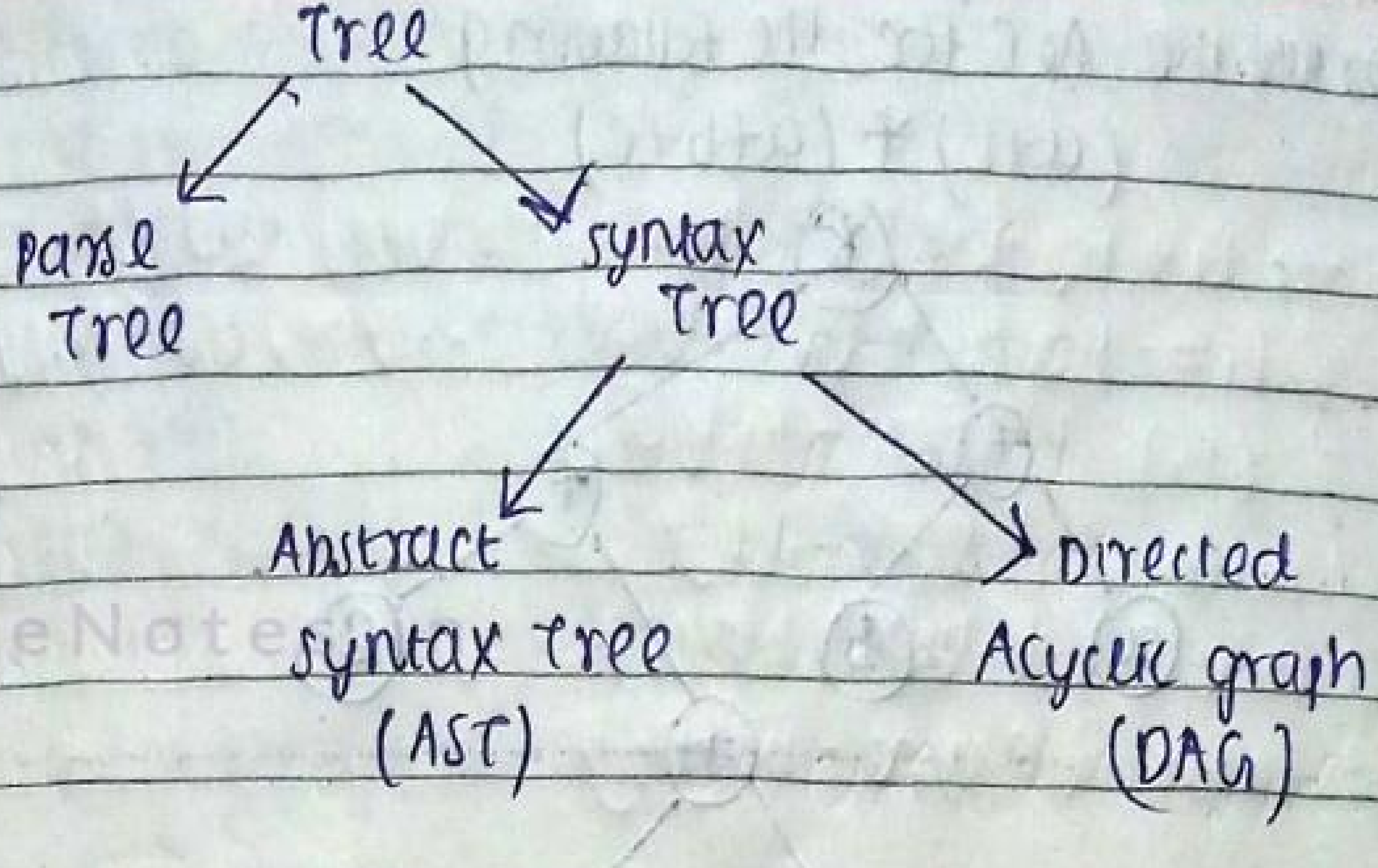


Dependency graph

LectureNotes.in

Scanned by CamScanner

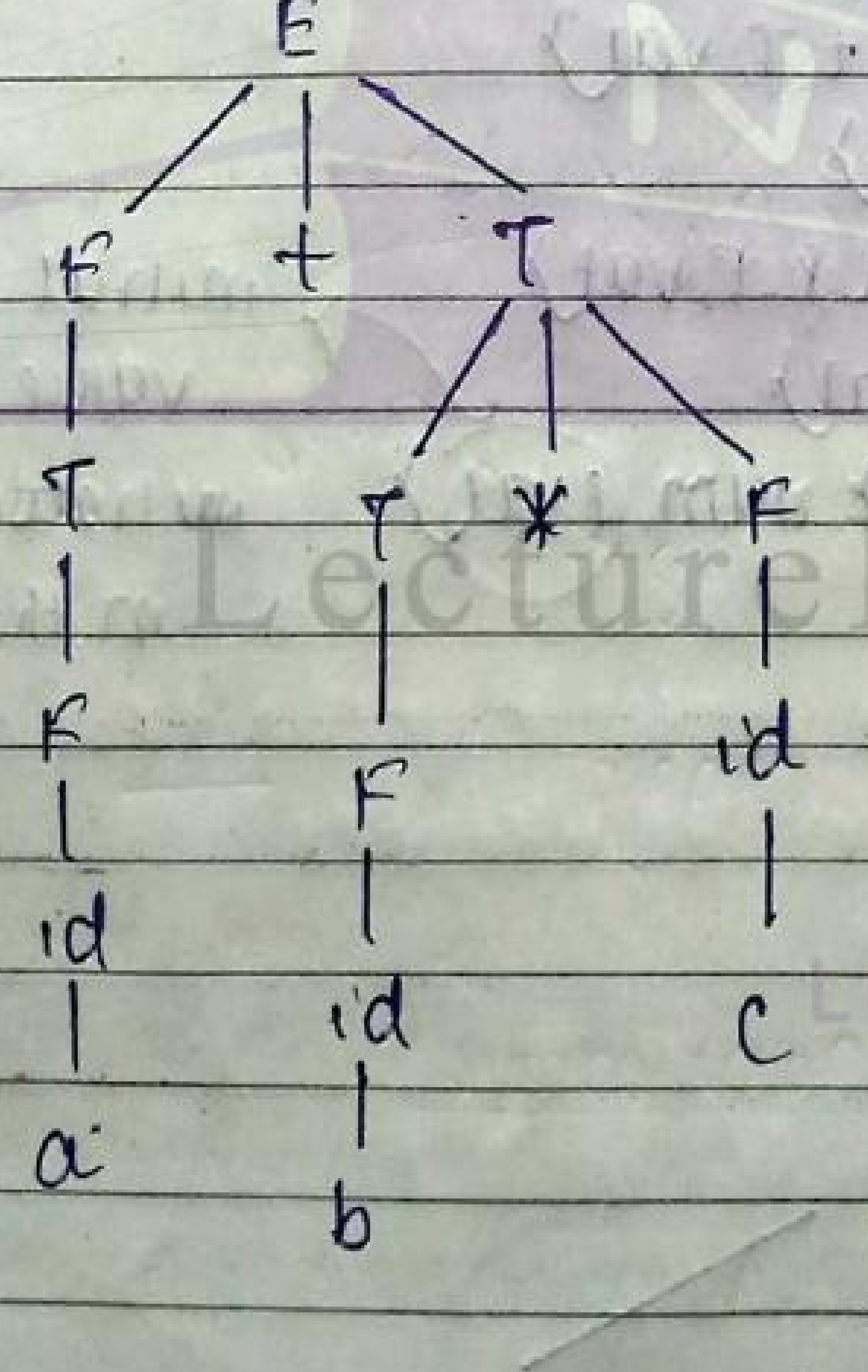
Tree



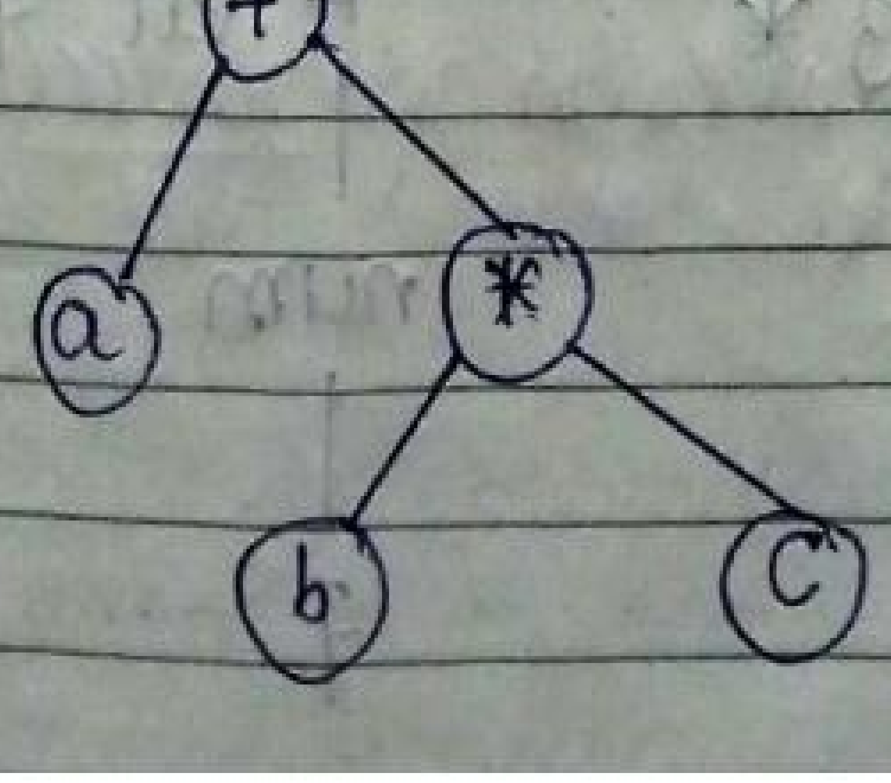
eg: Draw AST for the following SDT

E → E+T|T
T → T*F|F   I/P: a+b*c
F → id

parse



AST

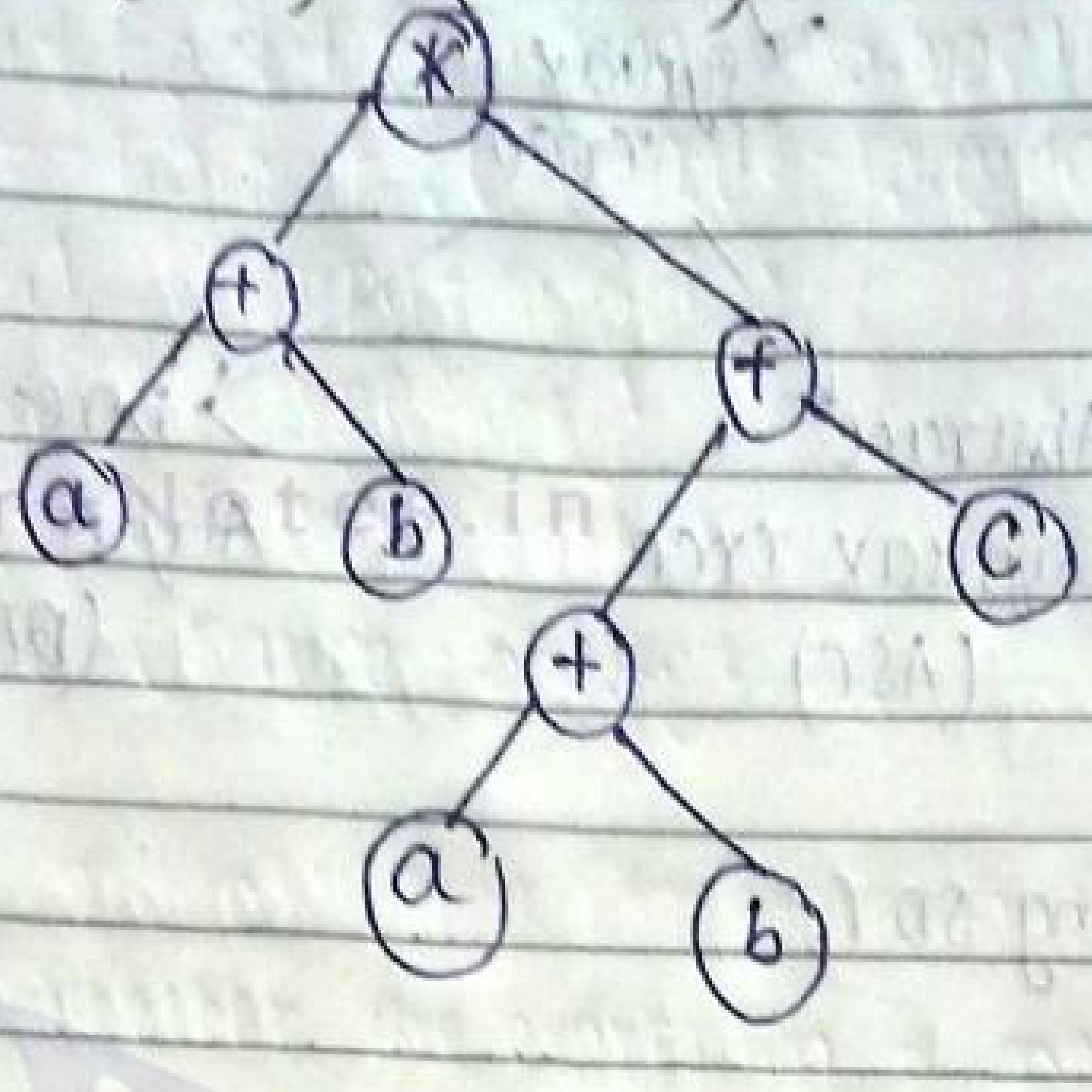


LectureNotes.in

Scanned by CamScanner

eg: Draw the AST for the following

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

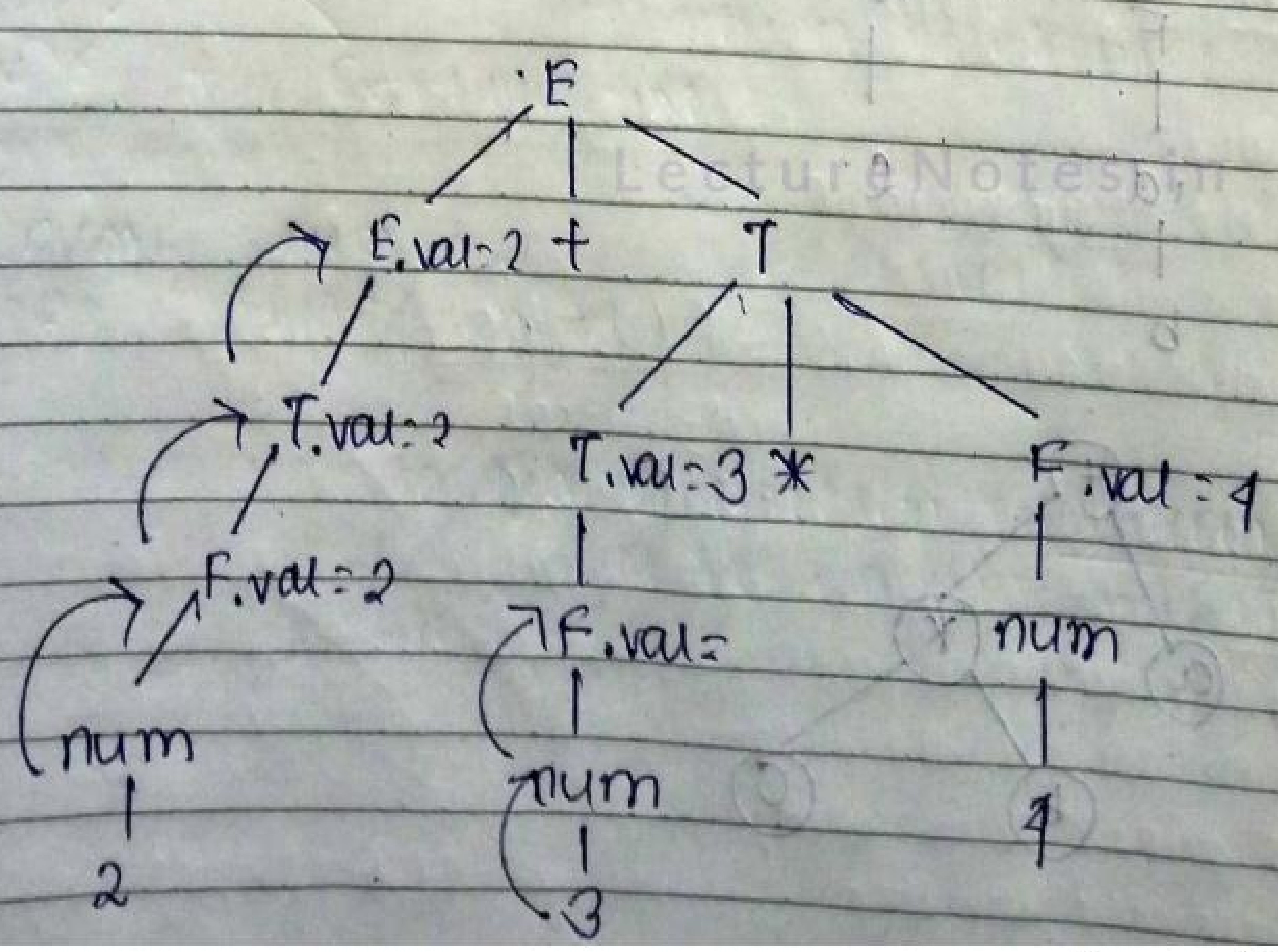


eg: draw SDT for expression

E → E+T	{ E.val = E.val + T.val }
/T	{ E.val = T.val }
T → T*F	{ T.val = T.val * F.val }
/F	{ T.val = F.val }
F → id	{ F.val = ~~id~~ num.(val) }

initial value always initialised in this way

I/P: 2+3*4



LectureNotes.in

Scanned by CamScanner

eg: draw SDT for expression

E → E+T {printf("+");} ①

 |T {} ②

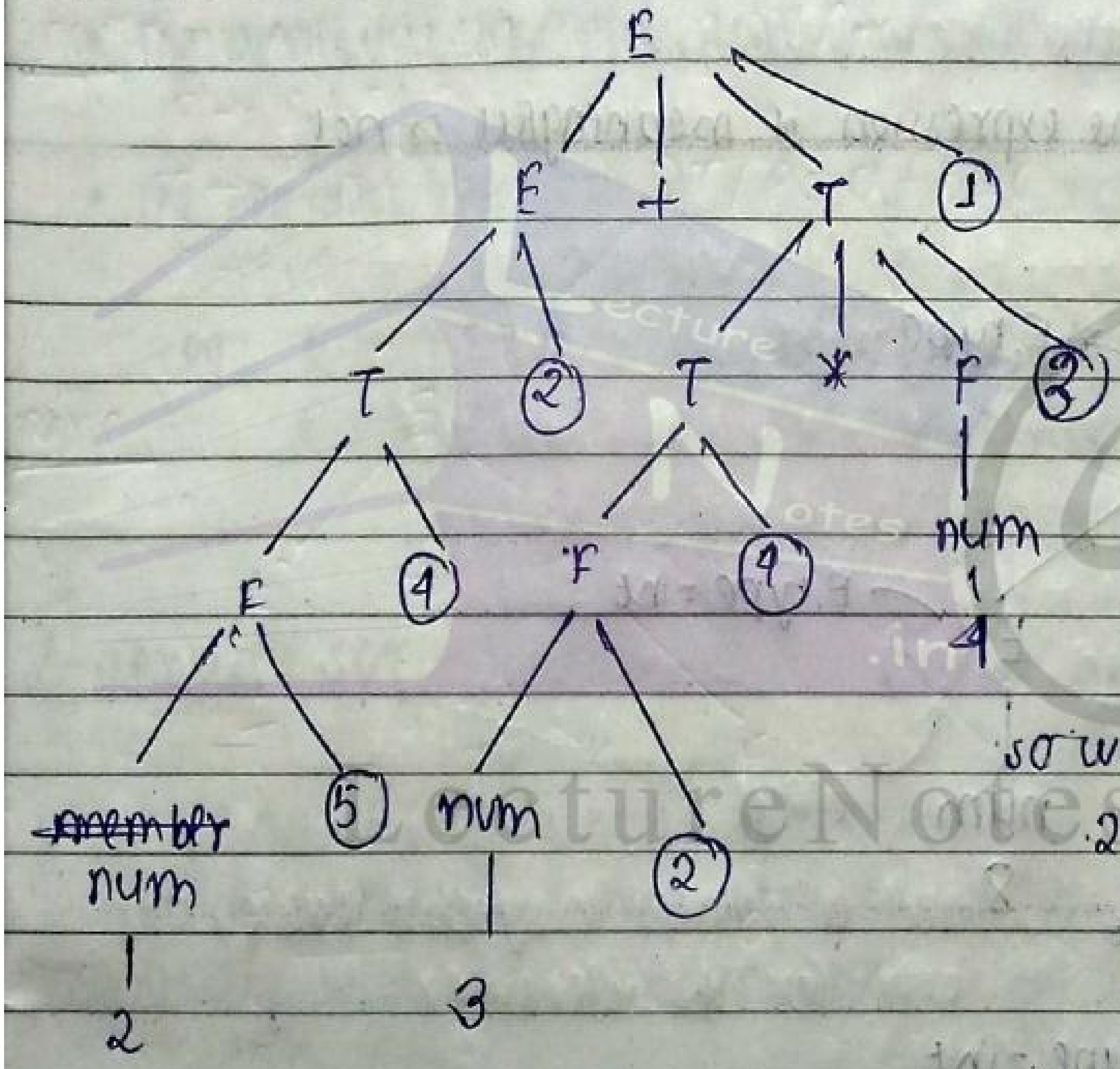
T → T*F {printf("*");} ③

 |F {} ④

F → num {printf(num.lval);} ⑤

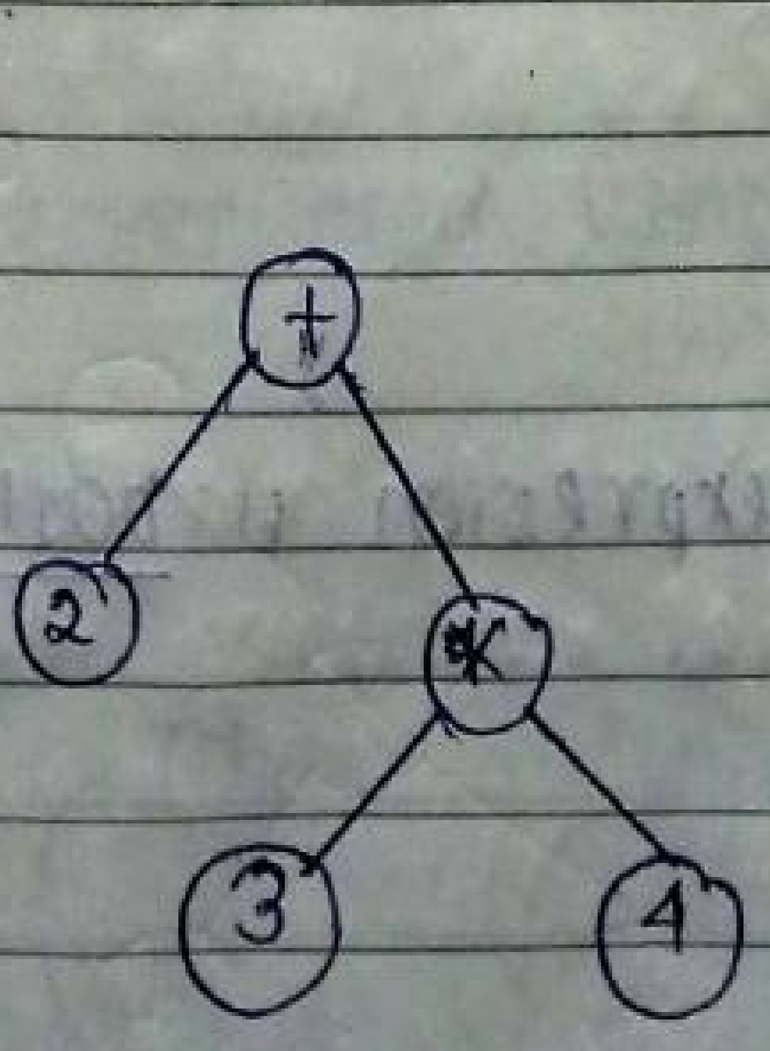
Infix to postfix

I/P: 2+3*4 find out postfix expression



so we get
234*+

AST:



LectureNotes.in

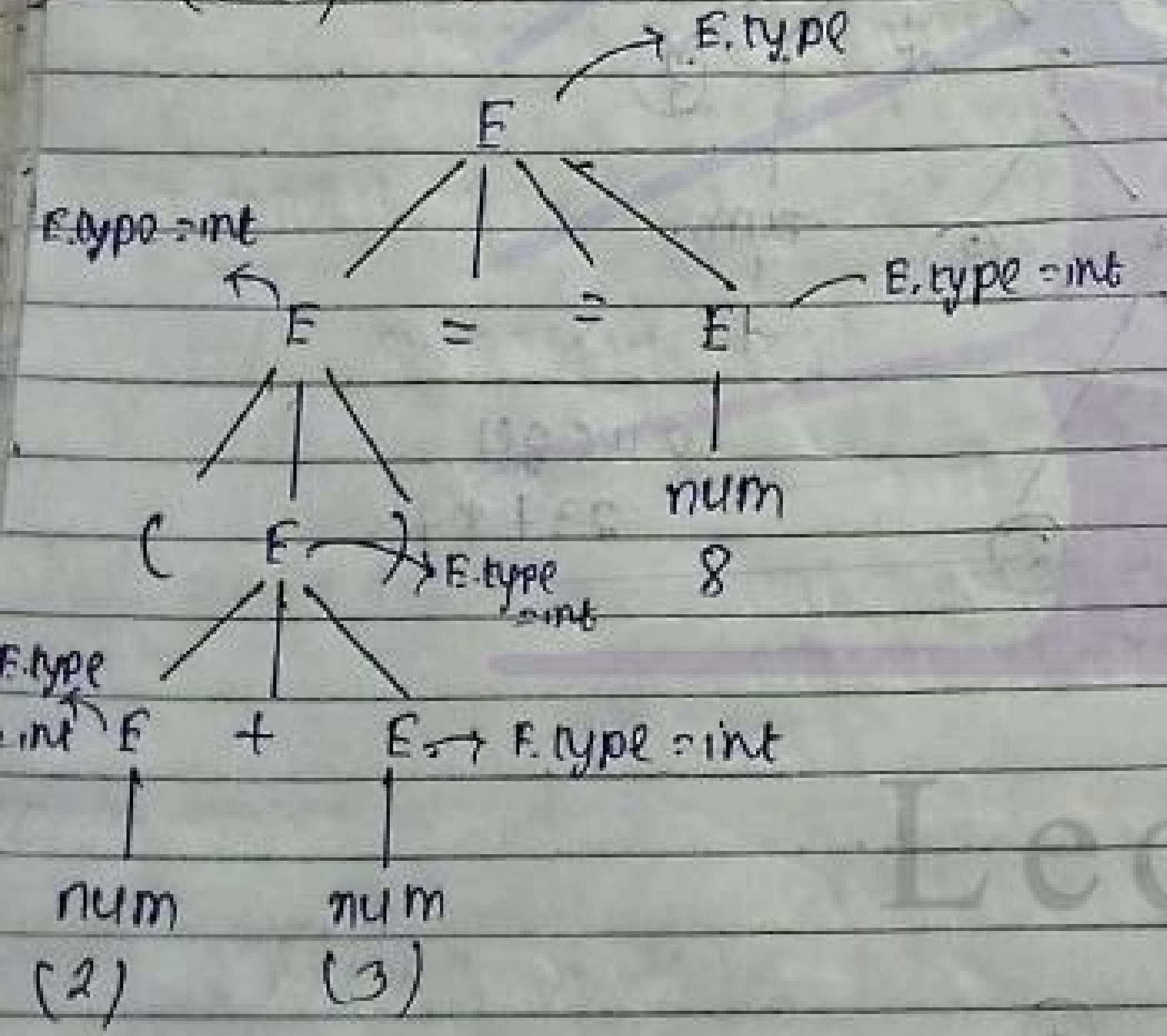
Scanned by CamScanner

Type check

① E → E_1 + E_2 {if ((E_1.type == E_2.type) && (E_1.type = int)) then E.type = int else error;}
/ E_1 == E_2 {if ((E_1.type == E_2.type) && (E_1.type = int/boolean)) then E.type = boolean else error;}
/ (E_1) {E.type = E_1.type;}
/ num {E.type = int}
/ true {E.type = bool;}
/ false {E.type = bool;}

eg: check whether the expression is meaningful or not.

(2+3) == 8



so type of expression is <u>boolean</u>.

S-attribute SDT

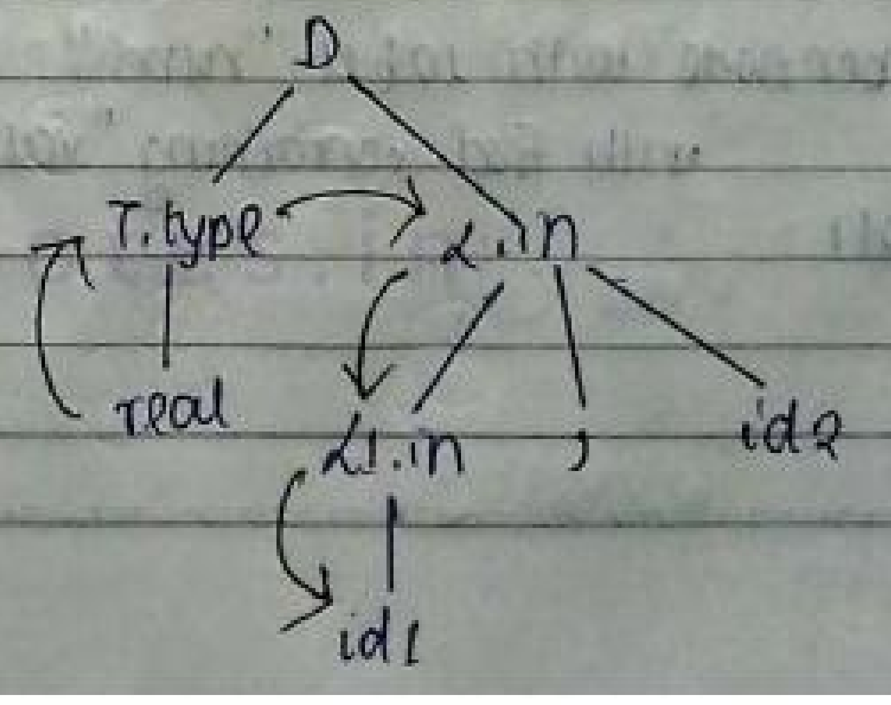
- An SDT is S-attributed if every attribute is synthesized
- The attributes are evaluated using Bottom up parsing
- The semantic rules can be placed at the right end of production

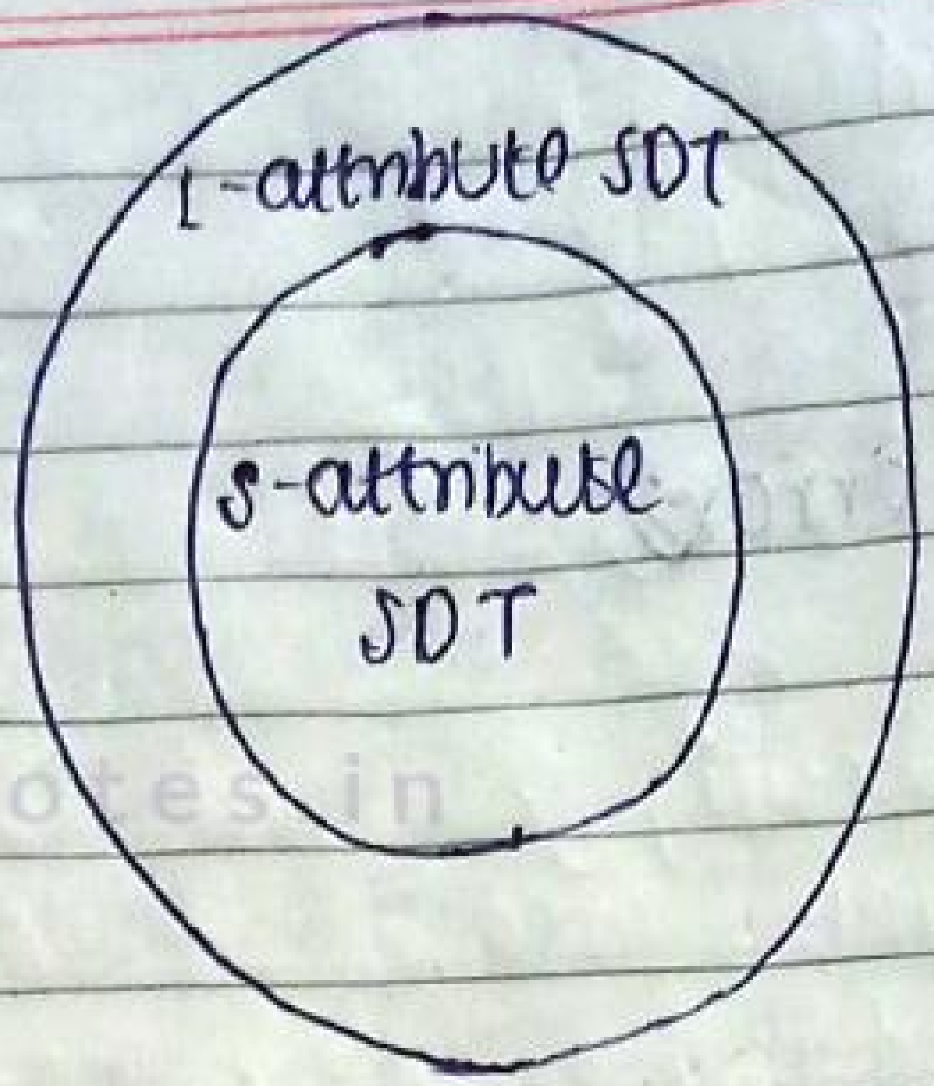
eg: E → E+E {E.val = E.val + E.val}
E → id {E.val = id.lval}

L-attribute SDT

- If an SDT uses both synthesised and inherited attributes with a <u>restriction</u> that inherited attributes can inherit attributes from <u>left siblings</u> only is called L-attribute SDT.
- semantic rules can be placed anywhere in RHS.
- attributes are evaluated using DFS from Left to Right direction.

eg:





LectureNotes.in

Construction of syntax tree (for expressions)

(a) for nodes

mknode (op, left, right)

(b) for operand field

mkleaf(id, entry)

label field

and a pointer to symbol table entry for the identifier

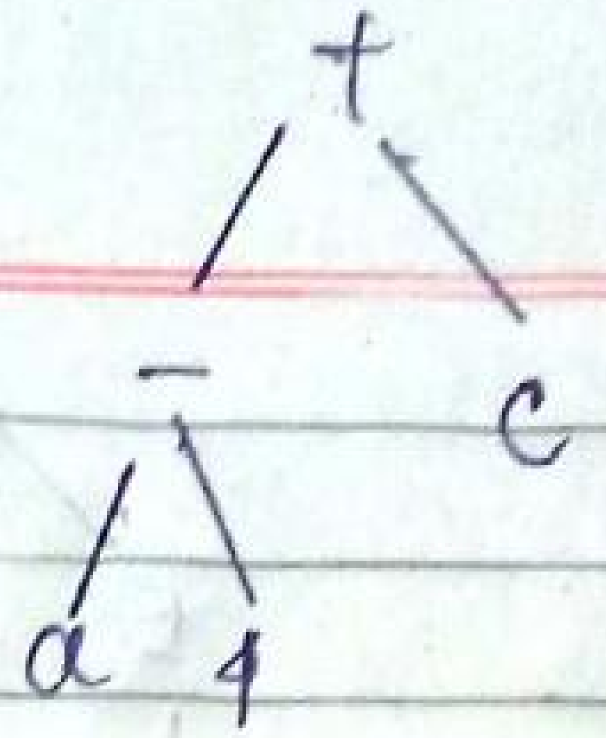
(c) for numbers

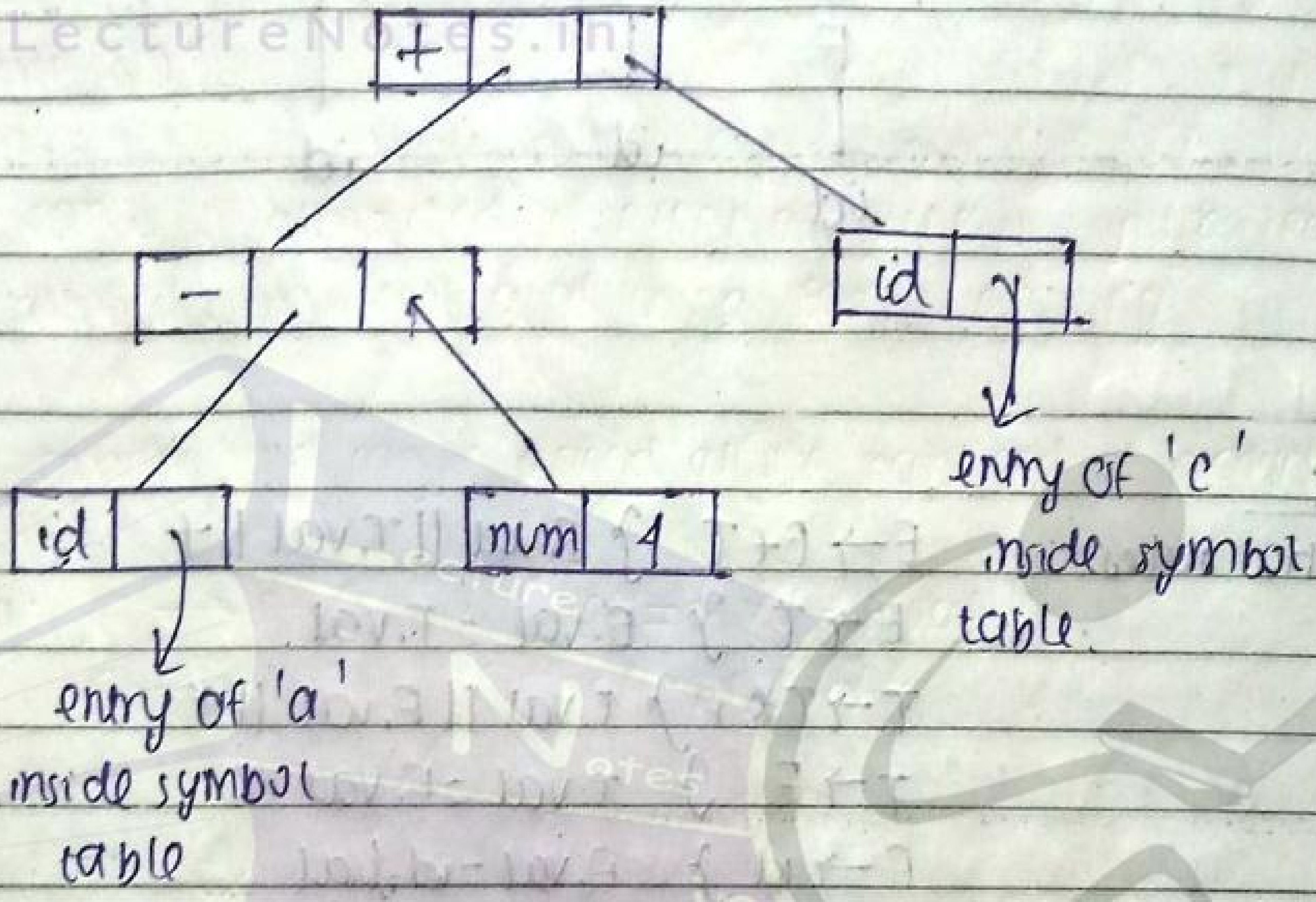
mkleaf(num, val)

makes a number node with label "num" with field containing "val" of the number.

Scanned by CamScanner

eg: expr: a-1+c





eg: write the SDT to convert infix to postfix

Infix: 2+3*5

Postfix: 235*+

so grammar is,

step1: $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

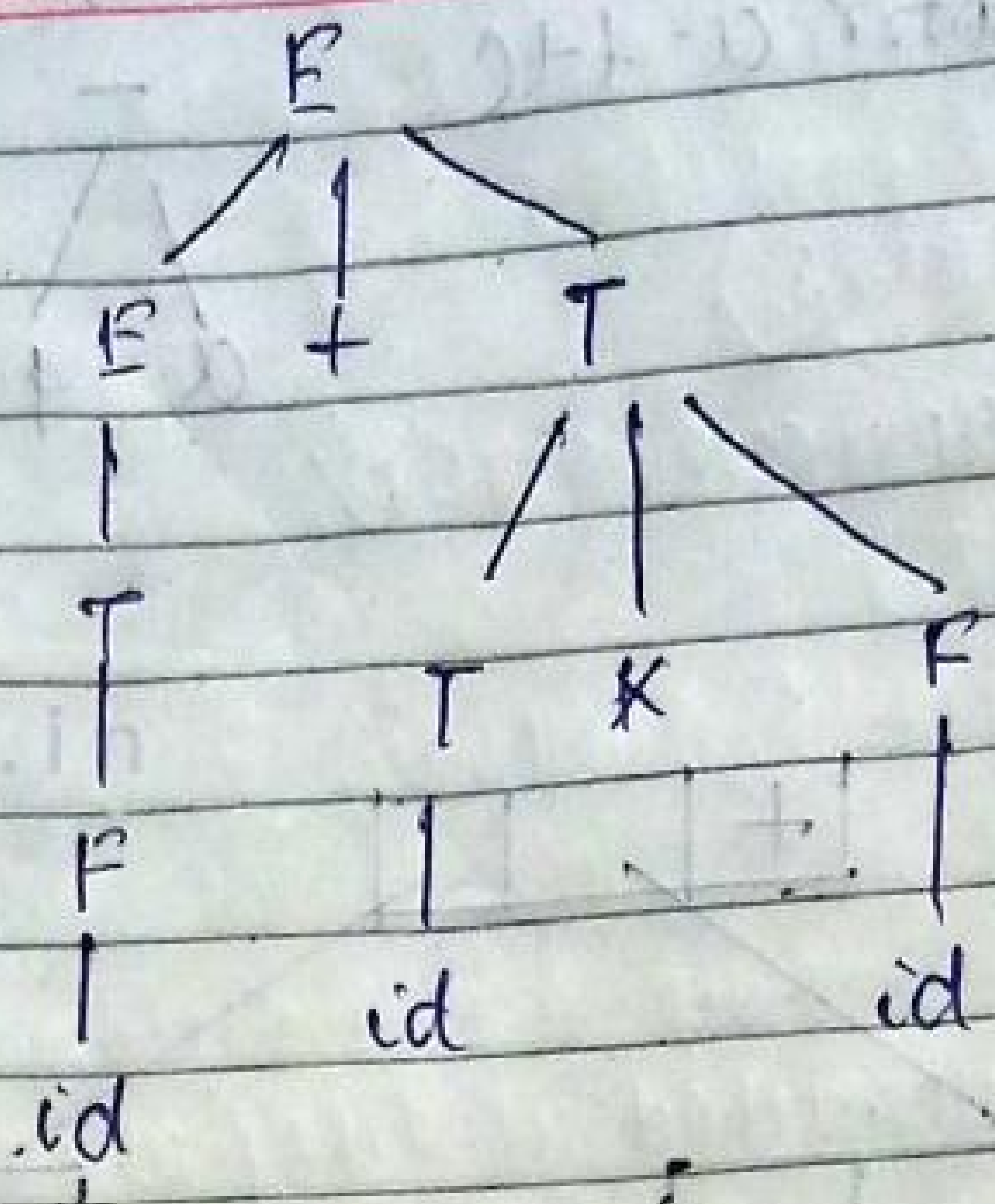
$F \rightarrow id$

I/P: id+id*id

LectureNotes.in

Scanned by CamScanner

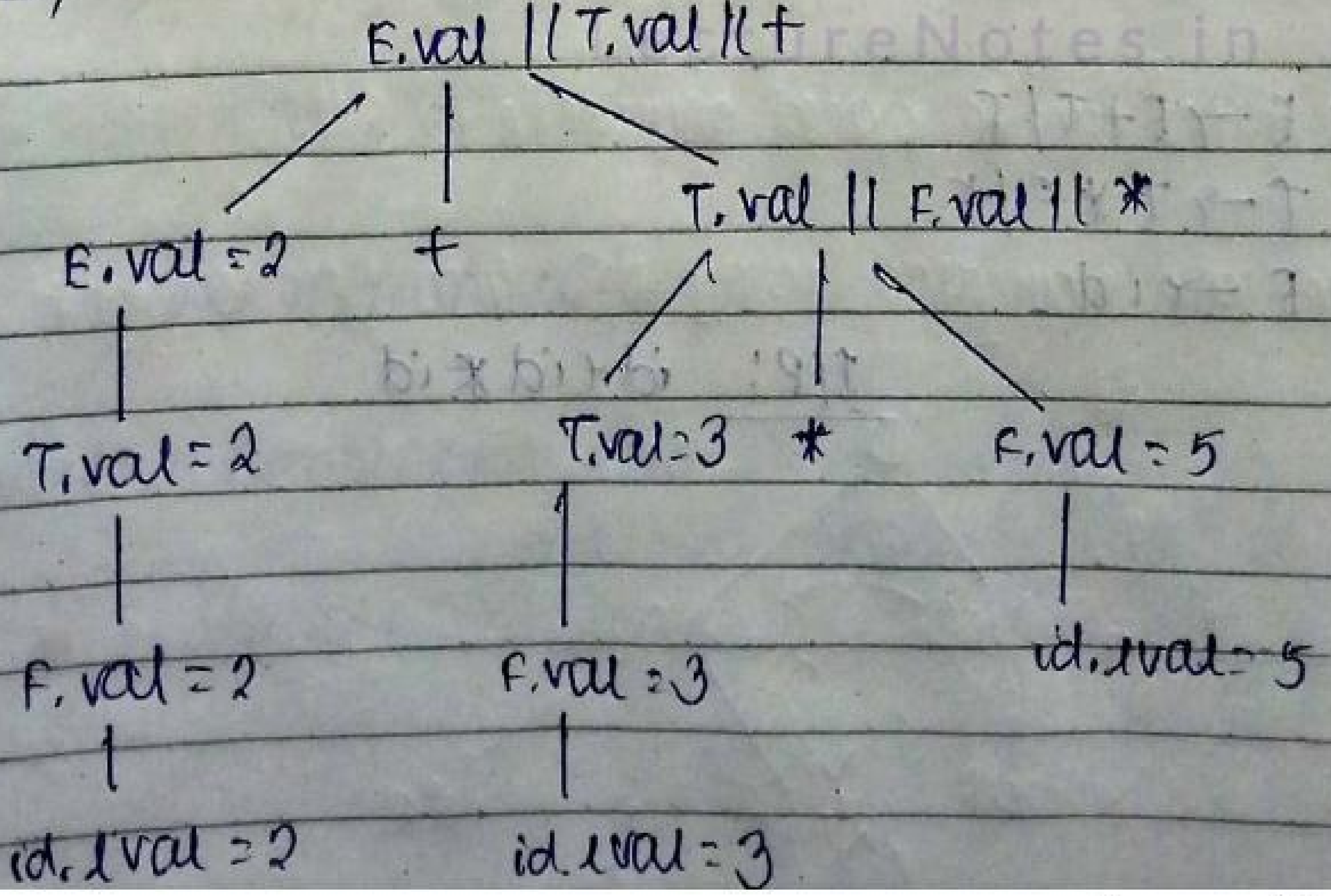
step2: Parse tree :



step 3:
semantic rules.

E → E+T } E.val || T.val || +
E → T } E.val = T.val
T → T*F } T.val || F.val || *
T → F } T.val = F.val
F → id } F.val = id.lval
→ value what is used is initialised in the lexical analyser

so now,

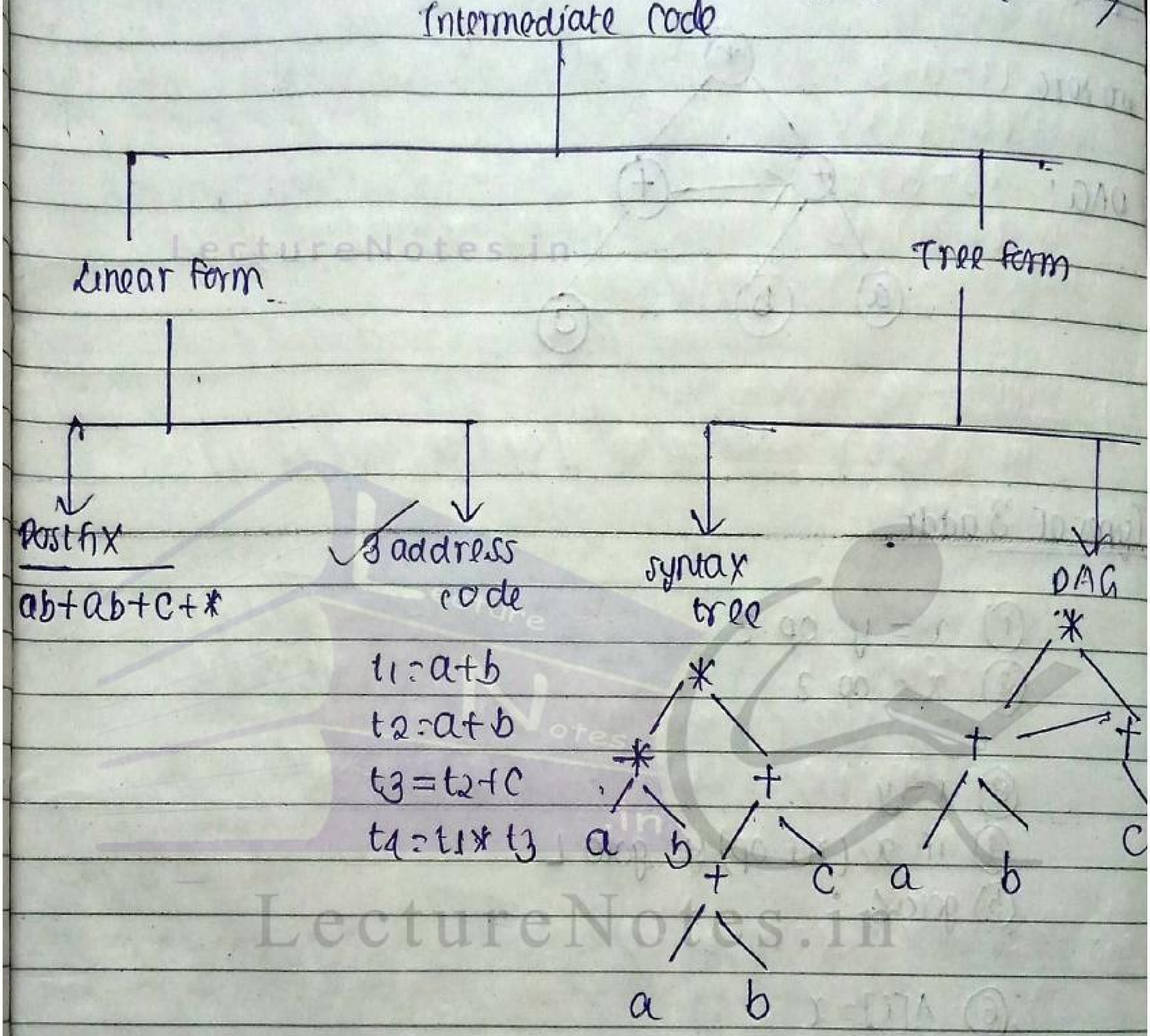


LectureNotes.in

Scanned by CamScanner

Intermediate Code Generation

eg: (a+b)*(a+b+c)



Postfix: ab+ab+c+*

3 address code:

t1=a+b
t2=a+b
t3=t2+c
t4=t1*t3

DAG (Directed Acyclic Graph)

- used to eliminate the common subexpression
- procedure to construct the SDT for DAG is similar to AST. except one restriction i.e.

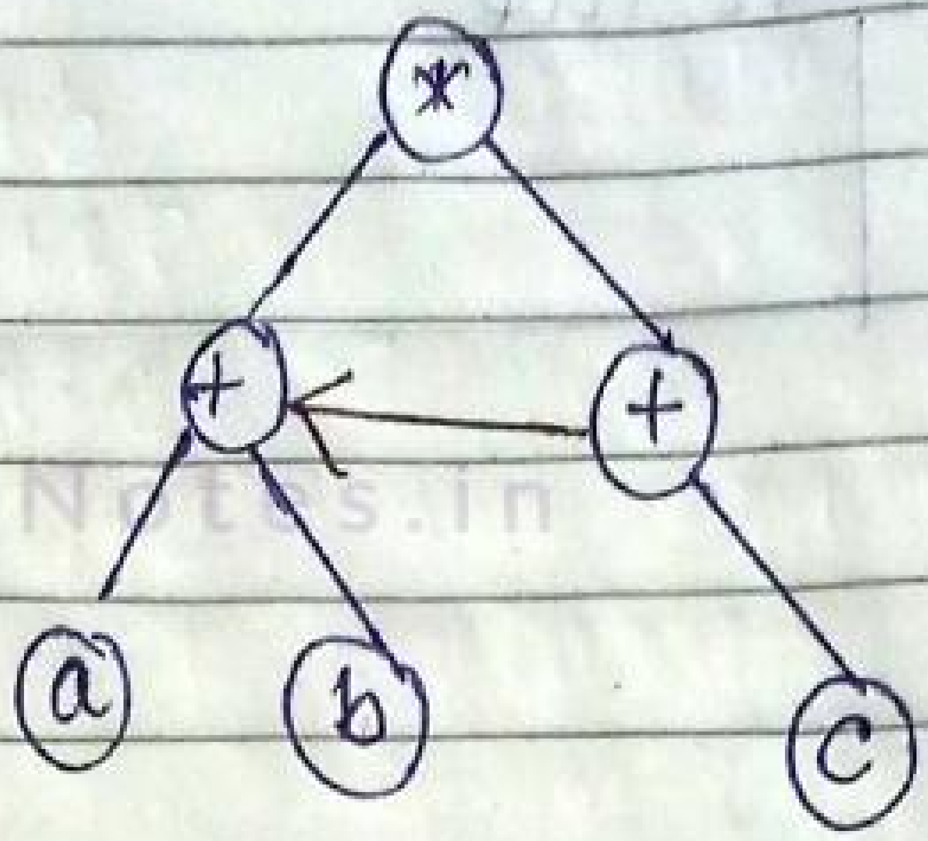
if any node is already created then make use of that node instead of going for new ~~creation~~ creation of same node.

Scanned by CamScanner

eg: $(a+b) * (a+b+c)$.

no here

DAG:



Types of 3 addr

① $x = y \text{ op } z$

② $x = \text{op } z$

③ $x = y$

④ if x (rel op) y goto L

⑤ goto L

⑥ $A[i] = x$

$y = A[i]$

⑦ $x = *p$

$y = \&x$

LectureNotes.in

Scanned by CamScanner

Implementation

eg: write 3 address code for $a = b * -c + b * -c$

so 3 address code:

```
t1 = -c
t2 = b*t1
t3 = -c
t4 = b*t3
t5 = t2+t4
ta= t5
```

eg:

```
if(c==0){
    while(c<20){
        c+2;
    }
}
else
    c = n*n+2;
```

```
(1) if c==0 goto 3
(2) goto 7
(3) if(c<20) goto 5
(4) goto 9
(5) c=c+2
(6) goto 3
(7) t2= n*n
(8) c = t2+2
(9)
```

eg:

```
do
  i=i+1;
while(a[i]<v);
```

```
(1) t1 =i+1
(2) i = t1
(3) t2=i* 8
(4) t3 = a[t2]
(5) if t3 <v goto 1
(6)
```

LectureNotes.in

Scanned by CamScanner

eg: (a+b)*c/(d+e)

```
t1 = a+b
t2 = t1*c
t3 = d+e
t4 = t1/t3
```

eg:
```
int x=10;
int y=20;
SOP(x+y)
```

```
x=10
y=20
t1=x+y
L: SOP(t1)
goto(L)
```

eg:
```
for(i=0; i<=10; i++)
{
    x=i;
    a[i]=x;
}
```

```
i=0
if i>0 goto L
x=i
a[i]=x
i=i+1
L:
```

LectureNotes.in

Scanned by CamScanner

eg: a*b+c|d

```
t1=a*b
t2=t1+c
t3=d
t4=t2|t3
```

eg:
```
int x=10
int y=20
  if(x>y)
     L1: SOP("FLA");
  else
     L2: SOP("CD");
```

```
x=10
y=20
if(x<=y) goto L2
  L1: SOP ("FLA")
goto L3
  L2: SOP("CD")
L3: Last
```

eg:
```
fact(x)
{
   int i=1;
   int f=1;
 for(i=1; i<=x; i++)
      f=f*i;
  x=f
}
```

```
leader  1. i=1
        2. f=1
leader  3. if(i>x)
leader  4. t1=f*i
        5. x=f
        6. t=i+1
        7. i=t
        8. goto(3)
leader  9. L: calling program
```

LectureNotes.in

Scanned by CamScanner

~~Interme~~

Implementation

Quadraple

A quadraple is:

$x = y \text{ op } z$

where x, y and z names, constants or compiler generated temporaries; op is any operator.

Syntax: op y, z, x

apply operator op to y and z, store the result.

(a) binary operater

eg: add a, b, c

gt a, b, c

result = y op z

Syntax: op y, z, result

(b) unary operater

result = op y

Syntax: op, y, , result

eg: uminus a, , c

not a, , c

inttoreal a, , c

(c) move operator

result = y

Syntax: mov y, , result

Scanned by CamScanner

```
eg: mov a, , c
    movi a, , c
    movr a, , c
```

(d) unconditional jumps

goto L

syntax: jmp , , L

```
eg: jmp , , L1      // jump to L1
    jmp , , 7       // jump to statement 7
```

(e) conditional jumps

if y relop z goto L

syntax: jmprelop y, z, L

```
eg: jmpgt   y, z, L1     // jump to L1 if y>z
    jmpgte  y, z, L1     // jump to L1 if y>=z
    jmpe    y, z, L1     // jump to L1 if y==z
    jmpne   y, z, L1     // jump to L1 if y!=z
```

```
eg: jmpnz  y, , L1      // jump to L1 if y is not zero
    jmpz   y, , L1      // jump to L1 if y is zero
    jmpt   y, , L1      // jump to L1 if y is true
    jmpf   y, , L1      // jump to L1 if y is false
```

LectureNotes.in

Scanned by CamScanner

Procedure parameters : param x, , or param x ✓

Procedure calls : call p, n, or call p, n ✓

where x is an actual parameter
invoke the procedure p with n parameters.

eg: f(x+1, y) ⟶ add x, 1, t1
param t1
param y
call f, 2

(f) indexed assignments

x = y[i] //move y[i], , x
y[i] = x // move x, , y[i]

(g) address and pointer assignments

x = &y // moveaddr y, , x
x = *y // movecont y, , x

disadv:

- wastage of memory because of use of temporary variable

adv:

- direct access to the location for temporaries
- easier for optimisation
- statements can be moved around

LectureNotes.in

eg: $a = b * -c + b * -c$

$t1 = -c$
$t2 = b * t1$
$t3 = -c$
$t4 = b * t3$
$t5 = t2 + t4$
$a = t5$

LectureNotes.in

	op	arg1	arg2	result	code
0	-	c		t1	t1 = -c
1	*	b	t1	t2	t2 = b*t1
2	-	c		t3	t3 = -c
3	*	b	t3	t4	t4 = b*t3
4	+	t2	t4	t5	t5 = t2+t4
5	=	t5		a	a = t5

Triples

Syntax: (op) arg1 arg2

operand

instead of result field we use pointers to the triple structure itself

LectureNotes.in

Index tells us the result

	op	arg1	arg2
0	-	c	
1	*	b	(0)
2	-	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Scanned by CamScanner

disadv:

- space efficiency (adv)
- code movement not possible.

eg: For the given expression show
(a) quadruple
(b) triple
(c) indirect triple

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

(1) t1 = a+b
(2) t2 = -t1
(3) t3 = c+d
(4) t4 = t2 * t3
(5) t5 = a+b
(6) t6 = t5+c
(7) t7 = t4+t6

Quadruples

	op	op1	op2	result
(1)	+	a	b	t1
(2)	-	t1	-	t2
(3)	+	c	d	t3
(4)	*	t2	t3	t4
(5)	+	a	b	t5
(6)	+	t5	c	t6
(7)	+	t4	t6	t7

LectureNotes.in

Scanned by CamScanner

Triples

	op	op1	op2
1	+	a	b
2	−	(1)	
3	+	c	d
4	*	(2)	(3)
5	+	a	b
6	+	(5)	c
7	+	(4)	(6)

Indirect triples

triples are separated in exacution order and use it when they are being called.

(1)
(2)
(3)
(4)
(5)
(6)
(7)

~~ao~~ adv

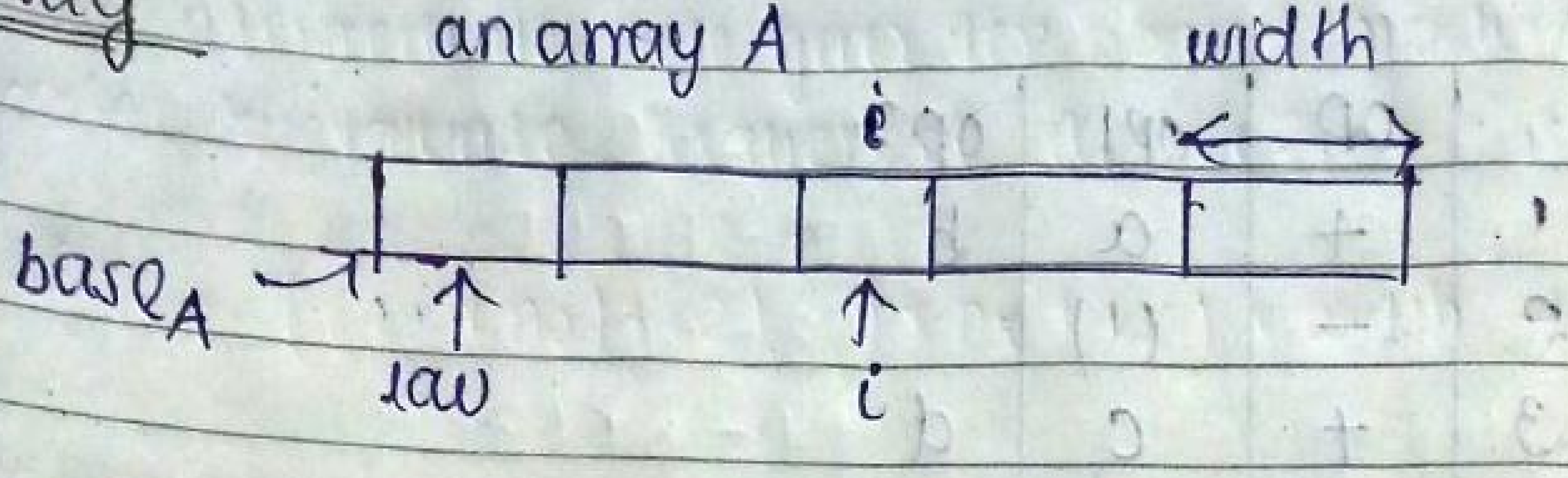
- statements can be moved

disadv

- 2 memory acces

LectureNotes.in

Array



then

$$\text{location of } A[i] = base_A + (i - low) * width$$

OR

$$\text{location of } A[i] = \underbrace{i * width}_{\text{should be compiled at run time}} + \underbrace{(base_A - low * width)}_{\text{should be compiled at compile time}}$$

LectureNotes.in

eg: write the expression in 3 address format

$$x = f(a[i]);$$

(1) $t1 = i * 4$
(2) $t2 = a[t1]$
(3) param $t2$
(4) $t3 = $ call $f, 1$
(5) $x = t3$

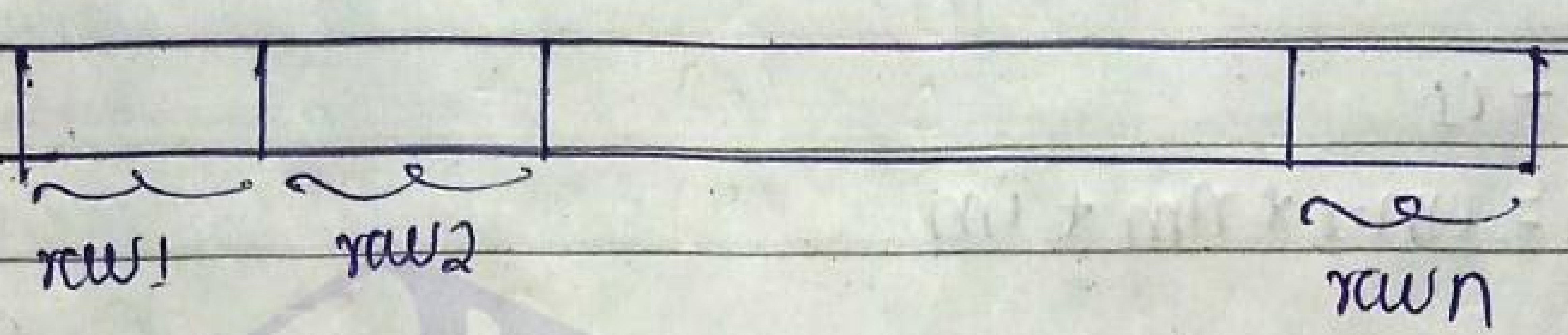
Scanned by CamScanner

2D Arrays

- can be stored in either raw-major (raw by raw)

OR

- in either column-major (column-by-column)



then

location of $A[i_1, i_2] = base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * width$

where $base_A$ = location of array A

low_1 = index of 1st raw

low_2 = index of ~~2nd raw~~ 1st column

n_2 = no. of elements in each raw

width = width of each array element

so

$$A[i_1, i_2] = \underbrace{((i_1 * n_2) + i_2) * width)}_{\text{can be computed at run time}} + \underbrace{(base_A - ((low_1 * n_2) + low_2) * width)}_{\text{can be computed at compile time}}$$

LectureNotes.in

Scanned by CamScanner

So the intermediate code generator should produce the codes to evaluate location of A[i1, i2, ----, ik]

$$((\ldots((i_1 * n_2) + i_2)\ldots) * n_k + i_k) * width + c$$

we can also recurrence equation to reduce the j

$e_1 = i_1$

$e_m = e_{m-1} * n_m + i_m$

eg: a 1D ~~array~~ double array A: 5....100
where $n_1 = 96$, width = 8 $low_1 = 5$

what will be the intermediate code generating x = A[y]

```
mov c, t1                //where c = baseA - (5) * 8
mult y, 8, t2
mov t1[t2], , t3
mov t3, , x
```

eg: a 2D int array A: 1..10 × 1..20
where $n_1 = 10$, $n_2 = 20$, width = 4, $low_1 = 1$, $low_2 = 1$

x = A[y, z]

```
mult y, 20, t1
add t1, z, t1
mov c, , t2              //where c = baseA - (1*20+1) * 4
mult t1, 4, t3
mov t2[t3], , t4
mov t4, , x
```

LectureNotes.in

Scanned by CamScanner

LectureNotes.in

eg: a 3D array A: 0...9 X 0...19 X .... 0..29
where $n_1 = 10$, $n_2 = 20$, $n_3 = 30$, width = 4, $low_1 = 0$, $low_2 = 0$, $low_3 = 0$

x = A[w, y, z]

```
mult  w, 20, t1
add   t1, y, t1
mult  t1, 30, t2
add   t2, z, t2
mov   c, , t3        //where c = base A − ((0*20+0)*30+0)*4
mult  t2, 4, t4
mov   t3[t4], , t5
mov   t5, , x
```

eg: x = A[y, z] where A: 10X20
write down 3 address code generation

```
t1 = y*20
t2 = t1+z
t3 = t2 * 4          // offset 11X4 = 44
t4 = base address of A
x = t4[t3]
```

↓

base[offset]
↓
base + [offset]

say base address = 100
then
required element
location = 144

LectureNotes.in

Scanned by CamScanner

Code Generation

- to generate the machine code it considers each 3 address instruction.
- also keeps tracks of what values are in which registers so that it can avoid generating unnecessary loads and stores
- decides how to use registers to best advantages

Four principle uses (of registers)

(a) In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.

(b) Registers make good temporaries to hold the result of a subexpression or a variable that is used only within a single basic block.

(c) Registers are used to hold global values that are computed ~~once~~ in one basic block and used in other blocks.

(d) Registers are often used to help with run-time storage management.

Code Generation Algorithm

- some set of registers is available to hold the values that are used within the block.

- for each operator there is exactly one machine instruction, that takes the necessary operands in registers and performs that operation leaving the result in a register.
  - LD reg, mem
  - ST mem, reg
  - OP reg, reg, reg

LectureNotes.in

Scanned by CamScanner

Register Descriptor

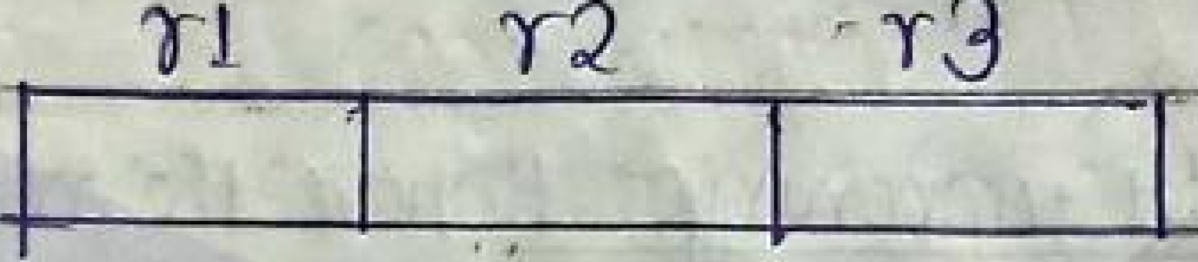
Descriptors are needed for variable store and decision.

Register descriptor:

— for each available ~~descriptor~~ register,

— keeps tracks of the variable names whose current value is in that register.

— initially, all register descriptors are empty.



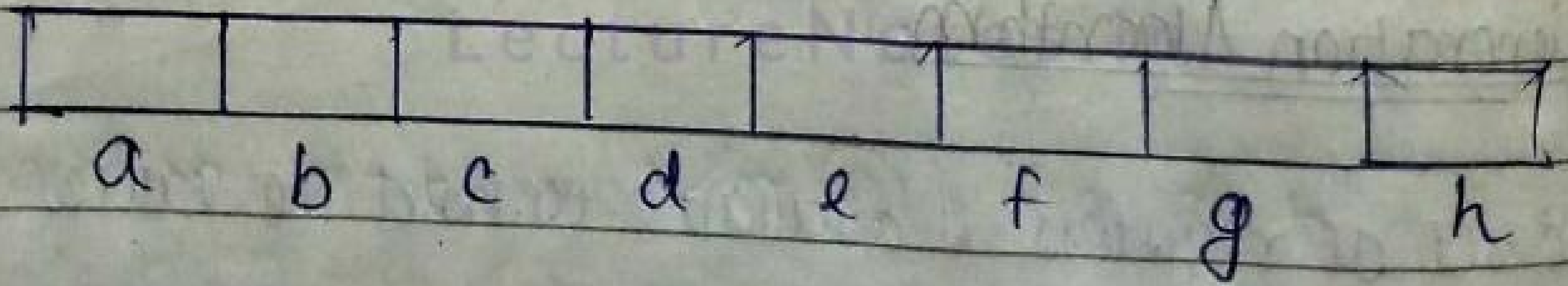
Register Descriptor

Address Descriptor

— for each program variable

— keeping ~~the~~ track of the locations where the current value of that variable can be found.

— stored in the symbol ~~entry~~ table entry for that variable name.



Algorithm

Function getReg(I):

- selecting registers for each memory location associated with the 3 address instruction I.

Scanned by CamScanner

Machine Instructions

for a 3 addr. instruction such as $x = y + z$ do the following:

Step 1: use getReg($x = y + z$) to select registers for x, y and z. call these as R_x, R_y and R_z

Step 2: If y is not in R_y (according to the register descriptor for R_y), then issue an instruction

LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y)

Step 3: Similarly if z is not in R_z issue an instruction

LD R_z, z' where z' is location for z.

Step 4: Issue the instruction ADD R_x, R_y, R_z

so the 3 address code for $x = y + z$ is

① $t = y + z$
② $x = t$

for ①

r1	r2	r3

y	z	t	x	e	f	g	h
y	z						

Scanned by CamScanner

set of instructions

LD r1, y
LD r2, z
ADD r2, r1, r2

r1	r2	r3
y	~~z~~	

y	z	t	x	e	f	g	h
y, r1	z	r2					

Machine Instructions (for copy statements)

→ For x = y, getReg will always choose the same register for both x and y

→ if y is not in that register Ry
generate an instruction
LD Ry, y

→ if y was in Ry, do nothing

→ need to adjust the register description for Ry so that it includes x as one of the values.

eg: for expression x = t   for ② i.e. x = t

r1	r2	r3
y	z	

y	z	t	x	e	f	g	h
y, r1	z	r2					

Scanned by CamScanner

LD r1, t

ie.

r1	r2	r3
x, y	z	

y	z	t	x	e	f	g	h
y, r1	z	r1					

Ending basic block

- Generate instruction

ST x, R where R is register in which x's value exists at the end of the block if x is live on exit from the block.

r1	r2	r3
x, y	z	

y	z	t	x	e	f	g	h
y, r1	z	r1					

For exiting from the basic block we need an instruction

ST x, r1

r1	r2	r3
x, y	x	

y	z	t	x	e	f	g	h
y, r1	z	r1	r1				

Scanned by CamScanner

Managing Register & Address (Descriptors)

(1) For the instruction LD R, x

(a) change register descriptor for register R so it holds only x
(b) change address " " x by adding register R as an additional location.

(2) For the instruction ST x, R

(a) change the address descriptor for x ~~by adding register R as an additional~~ to include its own location.

(3) For an operation such as ADD Rx, Ry, Rz for x = y + z
(a) change the register descriptor for Rx so that it holds only x.
(b) change the address " " x so that its only location is Rx

(c) Remove Rx from the address descriptor of any variable other than x.

(4) when process a copy statement say x = y after generating load for y into Ry

(a) add x to the register descriptor for Ry
(b) change the address descriptor for x so that its only location is Ry.

Scanned by CamScanner

eg: t = a-b
u = a-c
v = t+u
a = d
d = v+u

Initially

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

now t = a-b

LD R1, a
LD R2, b
SUB R2, R1, R2

R1	R2	R3
a	t	

a	b	c	d	t	u	v
a, R1	b	c	d	R2		

again,

u = a-c
LD R3, c
SUB R1, R1, R3

R1	R2	R3
u	t	c

a	b	c	d	t	u	v
a	b	c, R3	d	R2	R1	

again,

r = t+u
ADD R3, R2, R1

R1	R2	R3
u	t	v

a	b	c	d	t	u	v
a	b	c	d	R2	R1	R3

LectureNotes.in

Scanned by CamScanner

again,

a = d

LD R2, d

R1	R2	R3
u	a, d	v

a	b	c	d	t	u	v
R2	b	c	d, R2		R1	R3

again,

d = v + u

ADD R1, R3, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	R1	·		R3

exit,

ST a, R2

ST d, R1

d	a	v

a	b	c	d	t	u	v
a, R2	b	c	d, R1			R3

LectureNotes.in

Scanned by CamScanner

Code Optimisation

Optimisation

Machine Independent

- ✓(1) loop optimisations
  - (a) code motion(s) frequency reduction
  - (b) loop unrolling
  - (c) loop jamming
- ✓(2) folding
  - — constant propagation
- (3) redundancy elimination
- (4) strength reduction

Machine dependent

- (1) Register allocation
- (2) use of addressing modes.
- ✓(3) peephole optimisation
  - (a) redundant load/store
  - (b) flow of contro ~~options~~ optimisation
  - (c) strength reduction
  - (d) use of machine idioms

A Loop optimisations

(a) to apply loop optimisations we must first detect loops

(b) for detecting loops we use Control Flow Analysis (CFA) using Program Flow Graph (PFG)

(c) To find PFG we need to find basic blocks

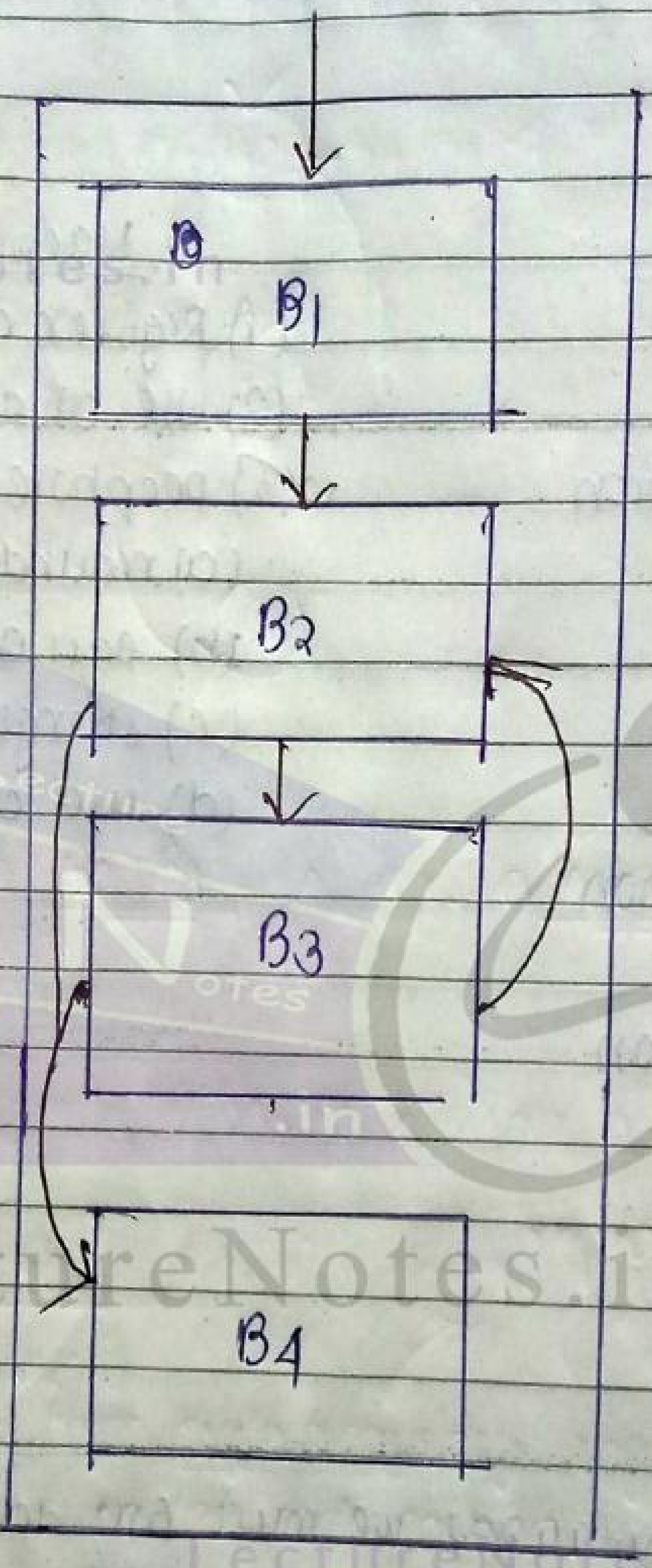
(d) a basic block is a sequence of 3 address statements where control ~~e~~ enters at the beginning and leaves only at the end without any jumps and halts.

LectureNotes.in

Scanned by CamScanner

optimisation ⟶ loops ⟶ CFA (PFG) ⟶ Basic blocks ⟶ leaders

say,



cycle: indicates loop then apply loop optimisation.

Scanned by CamScanner

Finding the basic blocks

In order to find the basic blocks we find the leaders in the program.
Then a ~~program~~ basic block will start from one leader to the next leader but not including the next leader.

identifying leaders

(1) I statement is a leader i.e. 1st statement
(2) statement that is a target of conditional or unconditional statement is a leader.
eg: if ( ) goto 200
then line 200 is leader
(3) statement that follows immediately a conditional or unconditional statement is a leader.

eg:
```
fact(x)
{
  int f=1;
  for(i=2; i<=x; i++)
      f=f*i;
  return f;
}
```

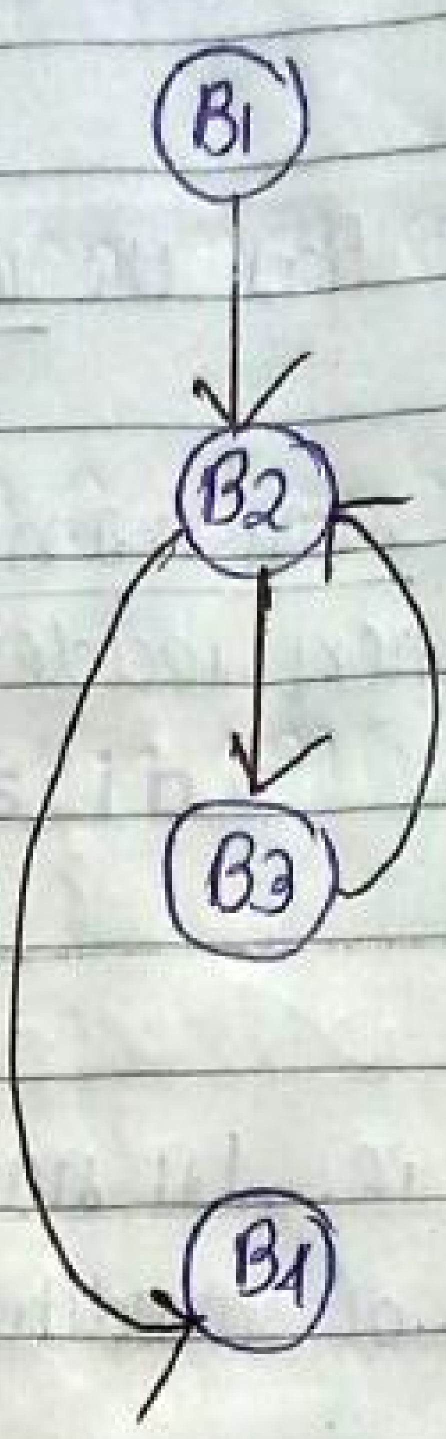
3address code

```
① f=1;                      B1
② i=2;
③ if(i>x) goto 9            B2
④ t1 = f*i;                 B3
⑤ f= t1;
⑥ t2 = i+1;
⑦ i=t2;
⑧ goto(3);
⑨ goto calling function     B4
```

leaders = {1, 3, 4, 9}

LectureNotes.in

so now PFG



we find cycle $B_2 - B_3$ i.e. it is a loop.

Types

① Frequency reduction

region which is called many times

moving the code from high frequency region to low frequency region is called code motion.

```
eg: while (i<5000)   i=0;
  {
      A = sin(x)/cos(x) * i;
      i++;
  }
```

⇓

```
t = sin(x)/cos(x)
while (i<5000)
{   A = t*i
     i++;
}
```

LectureNotes.in

Scanned by CamScanner

② loop unrolling

```
while ( i< 10)
{
    x[i] =0
    i++
}
```

⇓

```
while ( i<10)
{
    x[i] = 0 ;
    i++;
    x[i] = 0;
    i++;
}
```

while ( ) loop will work here 5 times so reduced

③ loop jamming

combines the bodies of the 2 loops.

```
for( i=0 ; i< 10 ; i++ )
    for( j=0 ; j< 10 ; j++ )
        x[i][j] = 0;
for ( i=0 ; i< 10 ; i++ )
        x[i][i] =0
```

↘

```
for (i=0; i<10; i++)
{
    for( j=0 ; j<10 ; j++ )
    {
        x[i][j] = 0;
        x[i] [i] = 0;
}
```

LectureNotes.in

Scanned by CamScanner

④ Folding:

Replacing an expression that can be computed at compile time by its values.

eg: $2+3+C+B = 5+C+B$

⑤ Redundancy elimination (DAG)

$$A = B+C$$
$$D = 2+B+3+C$$

$$D = 2+3+A$$

⑥ Strength reduction

Replacing a costly operation by cheaper one

eg: $B = A*2$ generally *, / is costly

$$B = A<<1$$

⑦ Algebraic Simplification

$$\left.\begin{array}{l} A = A+0 \\ x = 1*1 \end{array}\right\}$$ eliminate such statements.

LectureNotes.in

Scanned by CamScanner

B Machine Dependent

① Register allocation
- local allocation
- global allocation

} purely depends on the architecture

② use of addressing modes

③ peephole optimisation

(a) Redundant load and store elimination

x = y+z
- mov y, R0
- add z, R0
- mov R0, x

```
a = b+c  |  mov  b, R0
d = a+e  |  add  c, R0
         |  mov  R0, a
         |  mov  a, R0
         |  add  e, R0
         |  mov  R0, d
```

X eliminate

LectureNotes.in

Scanned by CamScanner

(b) Flow of control optimisation

- avoid jumps on jumps
- eliminate dead code

eg: L1 : jmp ~~L2~~ L4 ← after optimising

```
L1 : jmp L4
   :
   :
L2 : jmp L3
   :
   :
L3 : jmp L4
```

eg: #define x 0

```
if (x)
{
  ---
  ---
}
```

dead code elim x

(c) Use of machine idioms

```
i = i+1 | mov R0, i
        | add R0, 1    } inc i
        | mov i, R0
```

N.B

"90% of the time is spent executing 10% of the code". The frequently used parts are called hotspots, and are usually within loops, especially the inner loops.

LectureNotes.in

Scanned by CamScanner