System Sfw - LL Bee , Compiler Design

LL Bee
↳ Derivation

Compiler Design : MCA C401

CLASSTIME Pg. No.
Date / /

→ compiler system sfw :

→ Compiler → source program in source lang trans to target program in target lang.

(LLL) HLL to LLL (machine) → optimised route to convert HLL to LLL (Goal)

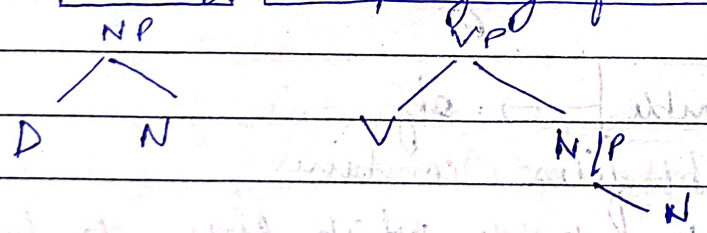
Compiler design → 1st Phase :- Lexical analyses (Lexical Analyzer)
(Convert character stream to token stream)
token will be predefined
↳ ~~lexeme~~ → user defined identifier
lexeme ↳ lexeme ko category dey detey hai
↳ (tokens) ↳ int , char
<token type>

Symbol table :-
entry of tokens in table . (So if old token ko identify krna check in Symbol table)

S → NP VP
NP → Art/D N
VP → V NP

NP: Noun Phrase
VP: Verb Phrase
Art: Article

The boy is playing football



→ 2nd Phase : Syntax Analysis (Syntax Analyzer)
tree (Parse tree)
↳ do not give meaning to statement ;
Coversion (implicit type casting) , explicit type casting

→ 3rd Phase : Meaning to sentence (Semantic Analysis)
Syntax Driven Translation

Compiler ka yahan front end khatam honey wala hai

4th Phase
* Intermediate Code (Not yet final / optimised)
* 3 address Code.

CLASSTIME Pg. No.
Date / /

Code optimisation

5th Phase: Code Optimisation
~~5th Phase~~ assembly Code generated, cost estimated kam hogi wah code lengy

Chapter 1 : Introduction

↳ Compiler is a program that translates a source program (a program written in a source language) and translate into the equivalent program called target program (the program in target language)

Source program (.cpp, .java) → [Compiler] → Target program
- .obj (Machine lang)
- .o
- .a / .asm (assembly)
- .exe

translating .asm to .obj → Assembler / Compilation

.asm → [Assembler] → .obj

Linker → multiple obj files combine
↳ dynamically (runtime) decide which func to link to program → .exe code
DLL (dynamic linked libraries)

- obj1
- obj2
- obj4

→ [Linker] → .exe / relocatable.

- exe is relocatable code (add base + logical) so that if lower add not available (as it is reserved for OS) so, .exe should get place in memory

Compiler / Translator

CLASSTIME Pg. No.
Date / /

* Loader
↳ brings .exe to memory.

.exe → layout how it is stored in physical memory
.exe stored on disk and double click it goes to memory

Linker (BSA)
↳ Branch & save address

* Assembler, Linker, Loader, Compiler all are needed

target i/p → [.exe/target] → o/p.

Compiler →

source code → compiler → target program (.exe)

(i/p to .exe) → .exe/target → o/p.

Interpreter (line by line) → ~~Adv.~~ Debugging (gdb) → gnum debugger C++

Source code → input → [Interpreter] → o/p

↳ Adv. of interpreter over compiler
• Debugging / Error Diagonstics

Interpreter → inbuilt facility of ~~Segmentation~~ debugging
↳ halt whenever error encountered

Compiler :→ fast, efficient, correct errors in one go
interpreter :- if error on 999 line then also ~~again & again~~ it runs complete program again

* the machine language program / target program by compiler is much faster than interpreter
* interpreter more good for error diagonistics (as line by line debugging)

JIT ★ (intermediate code)

- java
  ↓ javac
  Bytecode (intermediate code)
  → Input → JVM → o/p
  → JVM machine
  → JVM machine 2

(use jre → java runtime environment)

(to compile intermediate code once → we use just in time compiler) speed up execution

★ JIT (jisko run hona usi ko run krega already intermediate line ko nhi krega execute)

Read ★★

★ The compiler may produce an assembly lang. program as its o/p bcz assembly lang. is easier to produce as o/p and is easier to debug.

- The assembly lang. is then processed by a program called assembler that produces (.obj) relocatable machine code as o/p.

- Large programs are often compiled in pieces so the relocatable code may have to be linked together with other relocatable object files, library files into the code that actually runs on the machine. The Linker is responsible for this task.

  O/P → .exe file

(extern in C → external references to handle with linker)

- Loader – it puts the executable file into memory for execution.

CLASSTIME Pg. No.
Date / /

Platform Independent

Structure of Compiler

It divides into 2 ~~Phases~~ parts:

(1) Analysis / frontend. (2) Synthesis / back-end.

Analysis

- it breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation (IR) of the source program.
- the analysis also collects the info about source program and stores in a data structure called symbol table, which is passed along with IR to synthesis part.

Structure table made in lexical analysis

Synthesis

- It constructs the desired target program from IR and info in the symbol table.

Q/P: (make bidirectional arrows in fig 1.6) Diagram from symbol table.

↳ Phase of compiler

(machine dependent code optimizer)

↳ Optimal Phase.

↳ target program / .asm → used in book

Java: narrowing, explicitly, widening, implicitly

Phase I : Lexical Analysis

entity: Lexical Analyzer / Scanner

a) • first Phase known as Lexical Analysis / Scanning of source program

b) • It reads the stream of character and gro

[ ] // character class

[_a-zA-Z][a-zA-Z_0-9]* → identifier regular exp

any one some/more

(Pts) First check in reserved words then identifier list

CLASS TIME Pg. No. Date / /

the characters into meaningful sequences lexemes. (help of delimiters) → by default space

i) for each lexeme the lexical analyzer produces as o/p a token of the form:

< token_name, attribute value >

Eg. int a,b,c; → id (identifier)

↓ token stream ↓

<INTEGER><id,1><COMMA><id,2><COMMA><id,3><SEMICOLON>

symbol table mein store krty hai identifiers (pinkey corr. attribute values)

dynamic

Symbol table

1	a	int	4 (size)
⋮			

1 is attr value of a which is idx to symbol table

* position = initial + rate * 60

Imp * Diagram 2: Refer to Fig 1.7

(lexeme recognized by lexical analyser/identifier assigned token id)

(Syntax tree → operator as internal node)

(*) first →

$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to id \mid (E)$

left recursion

:lex → flex - fast lexical analyzer generator

bison package → yak - syntax analyzer

CLASSTIME Pg. No.
Date / /

LDF // Load Float from disk to register
MULF // Multiply float from reg to reg
#60.0 (immediate addressing)

int a,b,c;
↓
INTEGER (TOKEN)

↳ jo recognised hua → lexeme
jo category milli hai → token

★ Front end / Analysis
1. Lexical Analysis
2. Syntax Analysis
3. Semantic analysis
4. Intermediate Code generation

★ Back End / Synthesis
5. Machine Independent Code Optimization
6. Code generation
7. Machine Dependent Optimization

[reduce temporaries]

Rest theoretical part covered by yourself.
(Chapter 2, 1) → Automata.

★ CHAPTER: 3 LEXICAL ANALYSIS

1. find tokens and give categories to it (finite automata)
2. eat out all whitespace & combine multiple spaces to 1

prog1.l $\xrightarrow{\text{compiler}}$ (flex) → translator prog1.l
↳ lex.yy.c

↓

gcc lex.yy.c -o ankit

./ankit

CLASSTIME Pg. No.
Date / /

.l files contains some rules, definition and auxillary fns.

```
%%
definition
%%
rules
%%
auxillary fns
```

fix :- we specify ~~lexemes~~ pattern lexemes

table of rules

↓

pattern ..... {action}

↙

regular expression

(Keyword & identifier → so we put keyword first then identifier to differentiate both)

lex ka smallest program is %%

Lexical Analyser

1. Read the input characters
2. Group them into lexemes
3. Sequence of tokens for each lexeme

Lexeme : words derived from input character stream

token : lexeme mapped into token name and attribute value.

(*) Multiline comments ka regular exp?

[ ] → character class
↳ select any one from it

CLASSTIME Pg. No.
Date / /

Select [Ss][Ee][Ll][Ee][Cc][Tt]

X = a + b * 2

Lexemes = { X, =, a, +, b, *, 2 }

Tokens = { <Id, 1>, <=>, <Id, 2>, <+>, <Id, 3>, <*>, <num, 2> }

num = {digit}{digit}+

digit = 0 | 1 | 2 | 3 . . . 9 (terminal)

id [_ a-z A-Z][a-z A-Z _ 0-9]*

N.T

flex (Windows) lex (Linux)

num (non terminal)

(Grammar)

Source Program → [Lexical Analyzer] ⇄ token / getnextToken → [Syntax Analyzer]

(tab hoga jab lex and yak ~~combine~~) ko Interface kr Ahen

lex → lexical analyzer yak → syntax analyzer

Source program → [L.A] ⇄ token / getnextToken → [S.A] → Semantic Analyzer

[Symbol table]

gcc (yak) lex.yy.c (lex) y.tab.c

Syntax analysis (yacc) → CFG

CLASSTIME Pg. No.
Date / /

Ch3

Token ~~lexeme~~ : pair of token name and optional attr. value

Eg. <id, 1> example of token.

Lexeme : seq. of characters recognized by pattern for token.

pattern is a structure that is matched by multiple strings

Eg. printf("Total = %d \n", score);
string literal

~~token~~ printf

~~lexeme~~ <id, 1> <parenthesis> <string literal "Total = %d \n">

token <comma> <id, 2> <parenthesis> <semicolon>

~~pattern~~ Eg. if characters i and f — if lexeme

characters e, l, s, e — else lexeme

parenthesis / right parenthesis

comparison identifier <or>, <=, >=, ==, !=

user defined macros LE (Less Equal to) <=

LT (Less than) <

☆ We need to define one token for

each keyword and pattern for keyword is same as keyword itself

* One token represents all identifiers (id)

* Token for each punctuation symbol such as (,), , , ;

* The token name influences parsing (Syntax Analysis) actions.

* Handling Lexical Errors : (ERROR RECOVERY ACTION)

↳ 4 recovery actions that lexical analyzer takes:-

a.) Delete one character from remaining i/p.

eg: printf ("Hello \n"); $

b) Insert missing character in the remaining i/p.

eg: printf ("Hello \n") (Semi colon missing)

c.) Replace a character by another character

eg: elsp → else

d) transpose to adjacent characters.

eg: fi → if (swap f & i)

parsing table ↓ lexical analysis	☆ Alphabet :→ finite set of symbols. (punctuation, digits, characters, symbols etc)
tape & stack matching	☆ String : words made up of finite sequence of symbols drawn from alphabet.

$|s|$ = length of string

ϵ = string of length 0

CLASSTIME Pg. No.
Date / /

Language :- countable set of strings over some fixed alphabet.

eg. $L = \phi$ $\qquad$ $L = \{\epsilon\}$

☆ Prefix : By removing 0 or more symbols from end of s, we find prefix of string.

hello →
hello
hell
hel
he
h
ϵ

n length string, n+1 prefixes

☆ Suffix : By removing 0 or more symbols from beginning of s, we find suffix of string.

☆ Substring : By removing any suffix / prefix from s, we find substring

n length string → $\frac{n(n+1)}{2}$ → 1 → substring

eg $\sum_{i=1}^{n} i + 1$ $\qquad$ "ϵ"

☆ proper prefix : Prefixes of s that are not equal to 0 and not equal to itself.

n length string → n−1 proper prefixes.

★ <u>subsequence</u> : deleting 0/or more not necessarily consecutive pos of s.

CLASSTIME Pg. No.
Date / /

n length string → 2^n subsequence

↳ hello :- has alphabet key pass 2 possibilities
hai ya ehega ya nhi
$2^5 : \underline{32}$

L, M // L and M are 2 languages

★ $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$

★ $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$

★ $L^* = \bigcup_{i=0}^{\infty} L^i$ // +ve lang.

★ $L^+ = \bigcup_{i=1}^{\infty} L^i$ // L^i → Power alphabet ...

$L^i = L^0 \cup L^1 \cup L^2 \ldots$
(0 length str) (1 leng str.)

Q. $L = \{a, b \ldots z, A, B \ldots Z\}$
$D = \{0, 1 \ldots 9\}$

Ans: $|L \cup D| = 62$ $(26 + 26 + 10)$
$|LD| = 520$ $((26+26) \times 10)$

$\{L(L \cup D)^*\}$ = set of letter and digits beginning with letter

★ ε is a reg. expression
→ $L(a) = \{a\}$
↳ regular set

reg exp.

a) ★ The unary operator * (kleen star) has highest precedence & is left associative

Rule ① Feist Paranthesize
Rule ② Then write meaning of regex } → Regular Expression

CLASSTIME Pg. No.
Date / /

b) Concatenation (.) has second precedence & left associative
c) union (|) has lowest precedence & left associative

Eg. $(a|((b)^* c))$

a or 0or more b followed by one c

★. $\epsilon r = r\epsilon = r$
$r^{**} = r^*$ (idempotent)

Regular

$r_1 \rightarrow$ pattern then regular
$r_1 \rightarrow$ rules then CFG

$d1 \longrightarrow r1$
$d2 \longrightarrow r2$
$d3 \longrightarrow r3$

→ di is symbol not in Σ (alphabet set)

★ letter_ → A|B|...|Z|a|b|...z|_

digit → 0|1|2|...|9

Id → letter_(letter_|digit)*
(shortand for id expression)

<u>number with fractions</u>

digits → digit digit*
optional Fraction → .digits|ϵ
optional Exponent → (E(+|-|ϵ)digits)|ϵ

CLASSTIME Pg. No.
Date / /

number → digits optional Fraction optional Exponent

☆ ~~ele~~ letter → [A-z a-z _]
digit → [0-9]
id → letter _

Phone No. with 10 digits and start with 6,7,8,9

[6789][0-9]⁹ → ([6789]([0-9]{9}))^for power

☆ → ~~number digits~~ digit → [0-9]
digits → digit+

? → optional

number → digits (.digits)? (E[+-]?digits)?

↓

Reduced version of creating +ve no.

ya contribute kregi ya nhi

meta / special characters

either a) "/*" → strip (usko phir int mein convert krna padega) Preferred

b) \/* (by escape sequence)

^[a-z] // beginning of line

[^a-z] // Negation of (a-z)
Sky alawa kuch bhi

[^abc] // abc ko chod kr kuch bhi

r1/r2 // r1 when followed by r2

"abc"/"123"
r1 r2

^[^ aeiou]* $ → kleen closure

CLASSTIME Pg. No.
Date / /

$ → end of line (koey agay kuch na ho)
→ starts with consonant (can have 0 or more consonants)
→ Any complete line that does not contain lower case vowel.

yytext → lexeme character array that contains accepted lexeme from pattern.

Flex - Lexical analysis

yylex() → call it in main()
↳ called when lexical analysis is done.

demo.l → flex → lex.yy.c

• cpp → counting multiline comments / single line / identifier
character stream

↳ remove single line / multiline comments
↳ Validity of mobile nos → token stream

to run any lex program

→ lex → stdio.h> stdlib.h> // exit()

... definitions ... → optional
%%
... rules ...
%%
... code ...

main
↳ auxillary fn.

definition → global variables

rules → pattern + action
(whitespaces ignored)

code → auxillary fns (main fn ....)

```
%{ #include ....
   int charcount = 0, linecount = 0;
%}
```

(optional) definitions global variables

```
%%
```

end of rules

```
.{charcount++;}   (. any character)
\n {linecount++; charcount++;}
```

```
%%
```

auxillary fn

```
int main()
{ yylex(); //Global fn
  printf("There were %d chars
          in %d lines\n",
  charcount, linecount);
  return 0;
}
```

(definition includes include directives exist # stdio.h default include in lex.yy.c)

. → ignores \n

pattern say policy no whitespace

. → pattern least

Store this code in count.l

lex -t count.l > count.c
cc -c -o count.o count.l
cc -o counter count.o -ll

[\n\t]+ one or more

(ek C file mein main mandatory hai)

(*) We can't have main both files → Linker Error.

; //empty action

Aditya

```
%{
    definition
%}
%%
.....rules.....
%%
.....code.....
```

→ .l file

```
int yywrap()
{ return 1;
}
```

marks end of .l file

```
%{
  #include <stdio.h>
%}

%%
"ankit" {printf("Hello\n");}
.
\n      return;
%%
int main()
{ yylex();
  return 0;
}
int yywrap()
{ return 1;
}
```

o/p

a.exe

Cmd

```
ls dir
flex first.l
dir
gcc lex.yy.c
dir
gcc lex.yy.c -o jan23
dir
jan23.exe
ankit ——→ Hello
```

GNU win32

Dev-cpp

for flex

Cmd

```
cd Desktop
ls dir
flex first.l
~~gcc~~ dir
gcc lex.yy.c
```

Make folders for diff programs

```
dir
gcc lex.yy.c
    -o jan23
(rename)
dir
jan23.exe
```

len · yy · cc → Override

↳ string made of any character

CLASSTIME Pg. No. Date / /

(1)

.* → complete character

. → single character

• ~~Ado~~ Aditya Chopra
↳ 11 time Hello] → .

Aditya
↳ Hello] → .*

↳ . → error (pattern say pehley no space

Q. Write regular definition for the following languages:

a.) All strings of lowercase ~~characters~~ letters that contain vowels in order

b.) All strings of lowercase letters in which letters are in ascending lexographic order

c.) string literal

Ans. a) ~~[a]* [e]* [i]* [o]* [u]*~~

b) ~~[a]* [b]* . . . . . [z]*~~ Str → (a)* (b)* (c)* . . (z)

a.)

~~str~~ consonant → [^aeiou] or [bcd . . .]

str → [^aeiou]* a* [^aeio]* e* [^aeiou]* i* [^aeiou] o* [^aeiou]* u* [^aeiou]*

Ans a.) consonant* a (consonant|a)* e (consonant|e)* i (consonant|i)* o (consonant|o)* u (consonant|u)*

b.) Str → (a)* (b)* . . . . . (z)*

bat

bcdfg . . . .

a.) consonant → [^aeiou]

str → (~~consonant~~)* (consonant | a)* (consonant | e)*
(consonant | i)* (consonant | o)* (consonant | u)*

b.) str → (a)* (b)* . . . . . (z)*

c.) string literal

cout << "Hello /* */ \' // \" \t World";

~~^["] [.] ["] $~~

~~^["] [.] ["] $~~

~~^["] [.\n]* ["] $~~

~~^["]~~

c.) \" ((\\\") | [^ "])* \"

string literal

" → meta character to make it pattern \"

single line comment

~~^~~ "\\" (.)* [\n]

single line comment

// - - - - //

/* . . . . */

★ single line comment

CLASSTIME Pg. No.
Date / /

Comments
↳ consisting of string surrounded by without an intervening */ unless it is inside double quotes.

↳ "//" or \/\/

next file where o/p write
↑
yyout
↓

also "//" • *

Remove ~~space~~ comments

• /\n → { fprintf (yyout, yytext) }; (ptr to file where o/p go)

yytext → character ptr to string

yyin (ptr to file where i/p go)

• /\n { fprintf (yyout, yytext); }

int main()
{ yyin = fopen ("cs.cpp", "r");
yyout = fopen ("b.cpp", "w");
yylex();
return;
}

★ Multiline comment

/* ---- */

(//) abcdefg //
(/*) abcdefg
(//) abcdefg
(//* * * * * /)
(//)
(//) abcdef

→ complete file
∃ single line
| multiline

* { cout << "Hello /* */ \" // \" \t world "; (//) Hello
(/*) abcdef (*/) cout << "hello";
cout << "abcdef" (/*) * * (*/) << "abcdef";

Single line :- 4 multiline 3

☆ %{

%}

2 Feb

CLASSTIME Pg. No.
Date / /

IDENTIFIER [a-zA-Z_][a-zA-Z0-9_]*

shorthand

%%

{IDENTIFIER}

Reserved words / Keywords → not part of identifier.

* exe // → last mein ene hai g+key word aa jaye

relop // relational operator

relop → < | > | <= | >= | = | <>

ws // whitespace

ws → (blank | tab | newline)+

(lexical analyzer ignores ws)

☆ ~~for~~ In the process of lexical analysis if it encounters ws it do not return to the parser rather it restarts its process.

↳ yylval // integer value → (attribute value)

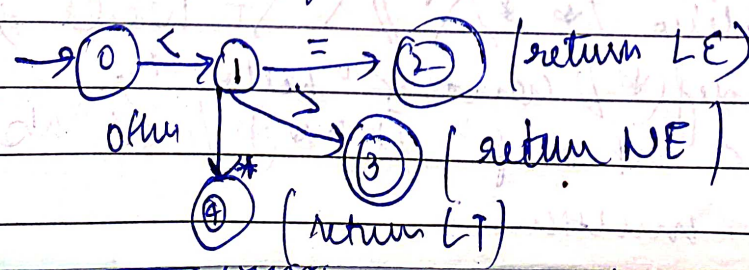
↓

global

~~yylen~~

yylen // ~~text~~ ~~lexeme~~ ka length

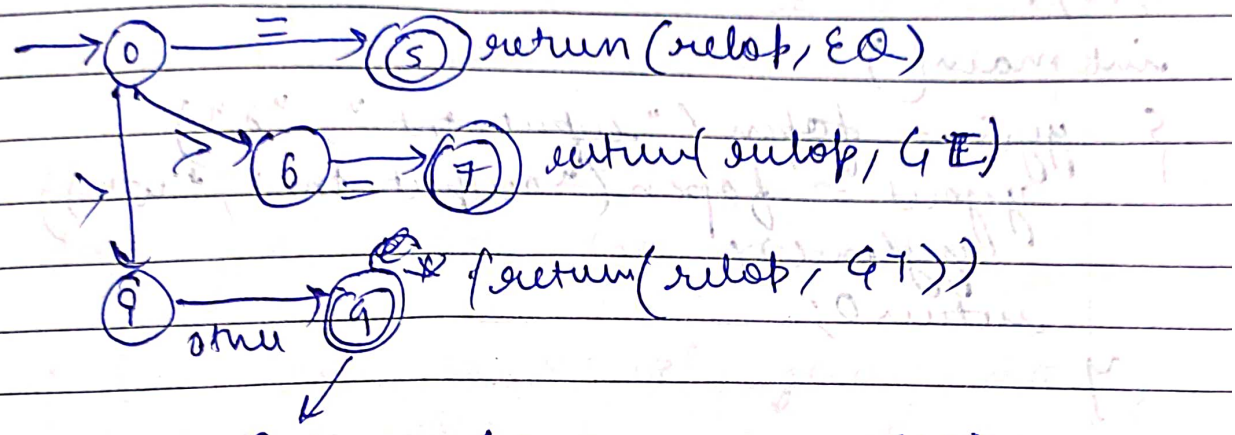
☆ Transition Diagram



relop (LE, LT, NE, ...)

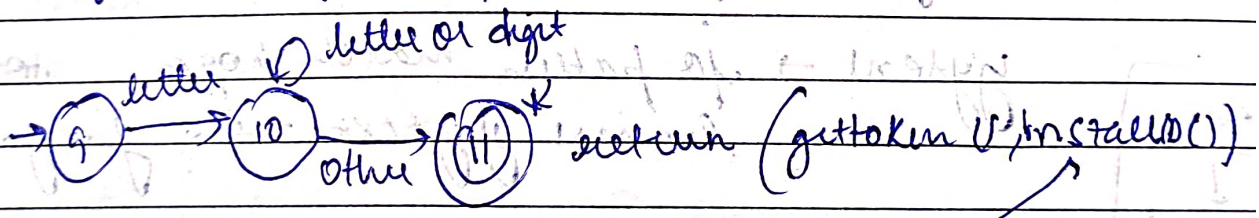
CLASSTIME Pg. No. Date / /

(0)* (Lexeme which has been recognized by lexical analyzer is less than only)
(whi lexeme thi)



Retract (Go back) → (*)

☆ T.D for identifier & Keyword



entering in symbol table *

☆ Conflict Resolution

↓
longer prefix is preferred

⇒ delim → (newline | tab | WS)*

Back Exercise / Lab Q imp of Compiler Design

(77) Q. replace each non empty seq of WS by single blank.

```
%}
%%
[ \t]*  {fprintf(yyout, " ");}
.|\n    {fprintf(yyout, "%s", yytext);}

%%
int main()
{   yyin = fopen("input.txt", "r")
    yyout = fopen("output.txt", "w");
    yylex();
    return 0;
}
```

(ignore \n multiple lines key gap ko)

★ ~~Arrow file~~ Pig Latin Q

yytext → jo pattern match hoga woh lenime yytext mein jaega

%%
↓
string format specifier

★ KNONSTAR [^*]*
PSTAR *+
NONSTARFWD [^*/]

★ SLL
→ "//".*

★ MLL
"/*" {KNONSTAR}{PSTAR}{NONSTARFWD}{KNONSTAR}{PSTAR} "*/"

12 2 2023

~~1234~~
abcd

CLASSTIME Pg. No.
Date / /

☆ DEC [0-9]* . [0-9]+

INT [0-9]+

☆ Pig latin 6/2

Q. Assume the file is a sequence of words (group of letters) separated by whitespace.

i) If first letter is consonant, move it to the right of the words and then add ay

ii) if first letter is vowel, just add ay to end of word.

CNAN

Ans (whitespace cannot be considered as word)

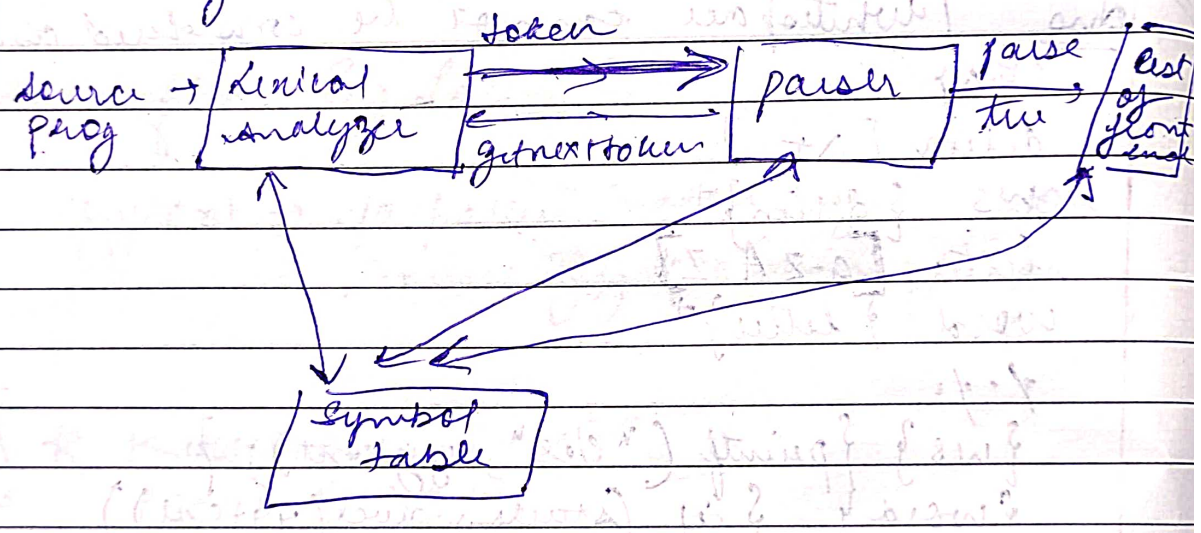
```
%{
delim [ \t ]
ws {delim}+
letter [a-z A-Z]
word {letter}+
%%
{ws} {printf("%s", yytext);}
{word} {if (starts_vowel(yytext))
          printf("%s ay", yytext);
        else
          printf("%s %c ay", yytext+1, yytext[0]);}
. printf("%s", yytext).
%%
```

```
int starts_vowel (char *s)
{ switch (s[0])
  {case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'u': return 1;
   default: return 0;
  }
```

☆ Syntax Analysis

2nd Phase of Compiler Design.

↳ Ask for tokens from lexical analyser has grammar trees to check validity of rules



$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$id \rightarrow [letter_][letter \mid digit]^*$

CFG → $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$

$-(id + id)$

$E \Rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

} left derivation

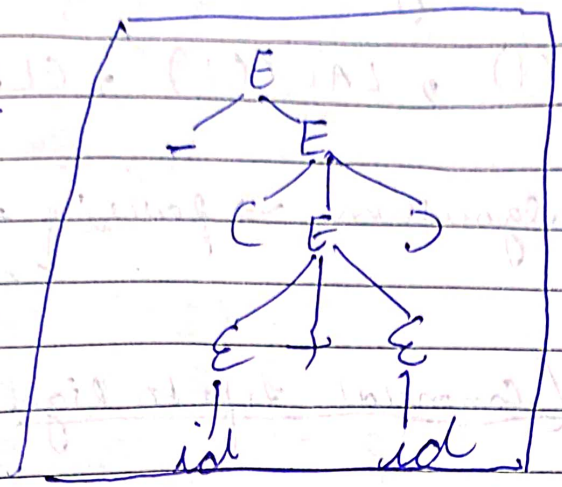
$\Rightarrow -(id + E)$

$\Rightarrow -(id + id)$

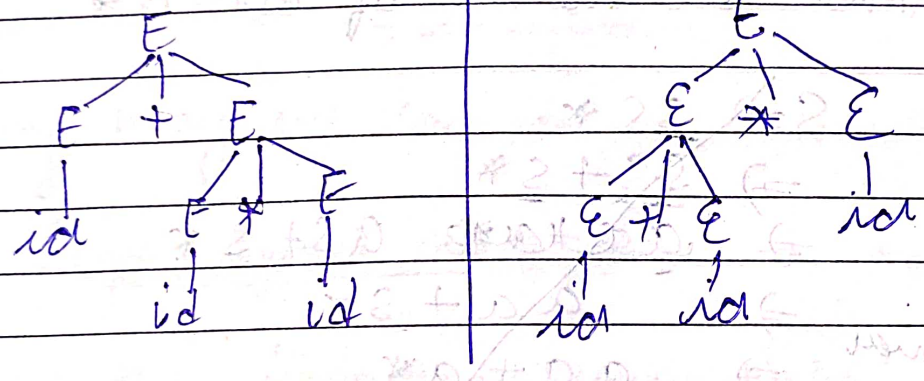
CLASSTIME Pg. No.
Date / /

(left most variable)

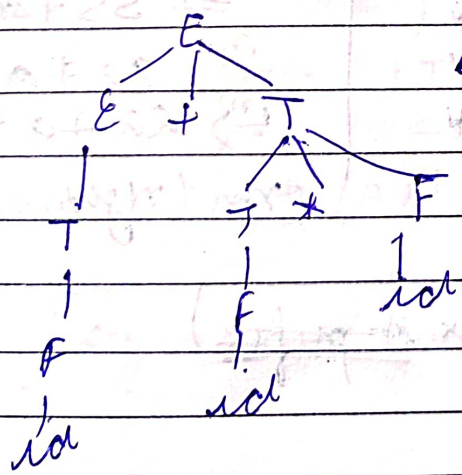
Parse Tree



Q. id + id * id → ambiguous



but $E \to E+T \mid T$
$T \to T*F \mid F$ → here only 1 tree possible
$F \to (E) \mid id$



Top down parser
↓
L-L parser
(left to right, leftmost derivation)

☆ Bottom up parse

LR left to right
↳ Rightmost derivation

(LR(0), SLR(1), LALR(1), CLR(1))

☆ (Parsing algorithm → parsing table → parse a expression)

☆ CLR (Canonical left to right

Q. $S \to SS+ \mid SS* \mid a$
leftmost derivation of $aa+a*$

$S \Rightarrow SS*$
$\Rightarrow SS+S*$
$\Rightarrow aS+S*$
$\Rightarrow aa+S*$
$\Rightarrow aa+a*$

Postfix grammar

$S \xRightarrow{lm} SS*$
$\xRightarrow{lm} SS+S*$
$\xRightarrow{lm} aS+S*$
$\xRightarrow{lm} aa+S*$
$\xRightarrow{lm} aa+a*$

lm (leftmost)

$S \xRightarrow{rm} SS*$
$\xRightarrow{rm} Sa*$
$\xRightarrow{rm} SS+a*$
$\xRightarrow{rm} Sa+a*$
$\xRightarrow{rm} aa+a*$

rm (rightmost

No ambiguity (postfix ~~& prefix~~)

Q. $S \to S(S)S \mid \epsilon$

string → (( ) ( ))

$S \overset{lm}{\Rightarrow} S(S)S$

$\Rightarrow S(S)S(S)$

$\Rightarrow \varepsilon(S)S(S)S$

$\Rightarrow \varepsilon(S(S)SS)S(S)S$

$\Rightarrow \varepsilon(\varepsilon(S)SS)S(S)S$

$\Rightarrow ()()$

$S \overset{lm}{\Rightarrow} S(S)S$

$\overset{lm}{\Rightarrow} \varepsilon(S)S$

$\overset{lm}{\Rightarrow} \varepsilon(S(S)S)S$

$\overset{lm}{\Rightarrow} \varepsilon(\varepsilon(S)S)S$

$\overset{lm}{\Rightarrow} ()(S)S)S$

$\overset{lm}{\Rightarrow} ()()$

→ Ambiguous (more than 1 tree)

any way we can derive

* Parser / Syntax Analyzer → Context free grammar

or ↓

Parse tree (governing some rules)

Expression Grammar

Version 1: $E \to E*E \mid E+E \mid (E) \mid id$

Version 2: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

V_2 is unambiguous version of V_1

* I say zyada derivation ya parse tree ↓ ambiguous

V_1 is ambiguous

V_2 is unambiguous

Rules to make grammar ambiguous to unambiguous :- Disambiguous Rule

CLASSTIME Pg. No. Date / /

V_2: 3 Non Terminals but Left Recursive

V_3:
$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

More non terminals but not left recursive

Top down / Recursive descend Parser

Left Recursive → disadvantage → left skewed hence backtract cost more hence avoid left recursive)

(Top down / Left Recursion)

If we get either 2 Leftmost or 2 rightmost
↓
then Ambiguous

* balancing height of tree ↑ } if we V_2 operation make changes as

$E \to T+E \mid T$
$T \to F*T \mid \epsilon$
$F \to (E) \mid id$

for id+id*id

* if else ( ), → terminals

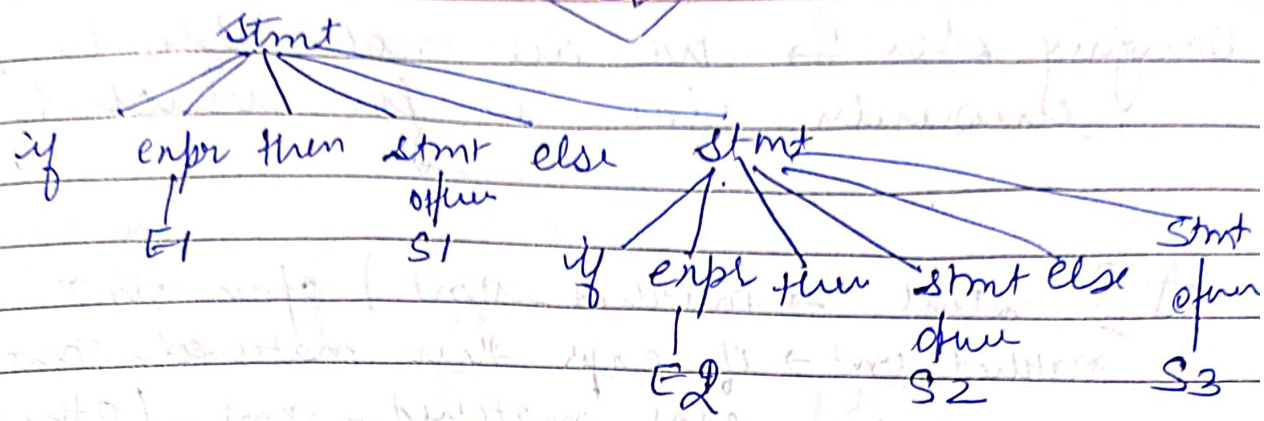
* Dangling else grammar (ambiguity)

Stmt → if exp then stm |
if exp then stm else stm |

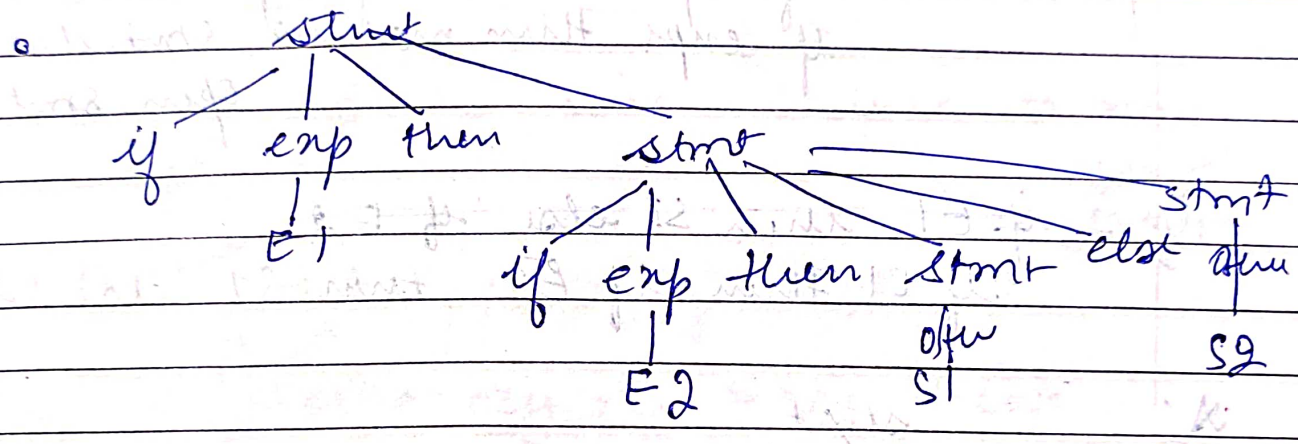
Q. if E1 then S1 else if E2 then S2 else S3

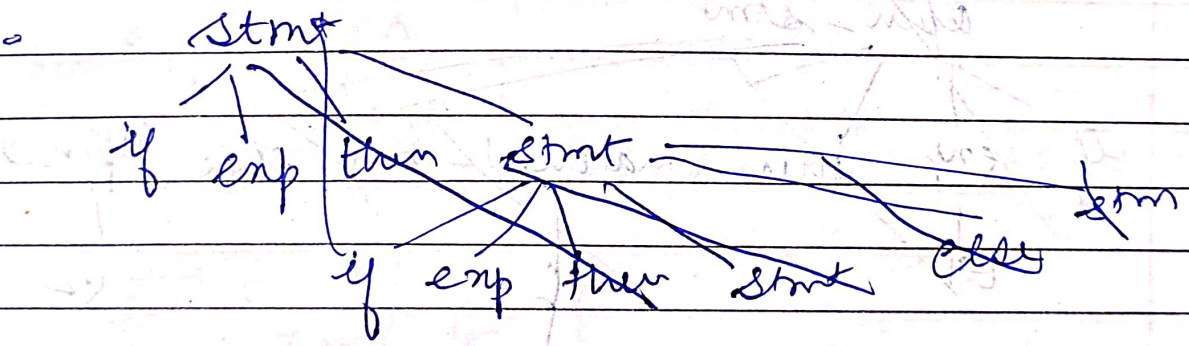
CLASSTIME Pg. No.
Date / /

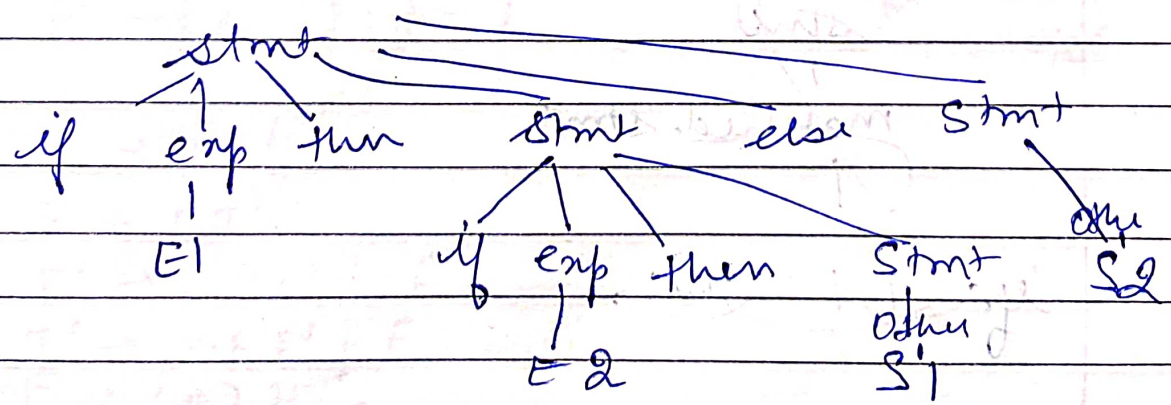
A



Q if E1 then if E2 then S1 else S2







2 leftmost → ambiguous.

Match each else with closest then

↓

disambiguity Rule

(Dangling else → we are not able to encounter else is for which if)

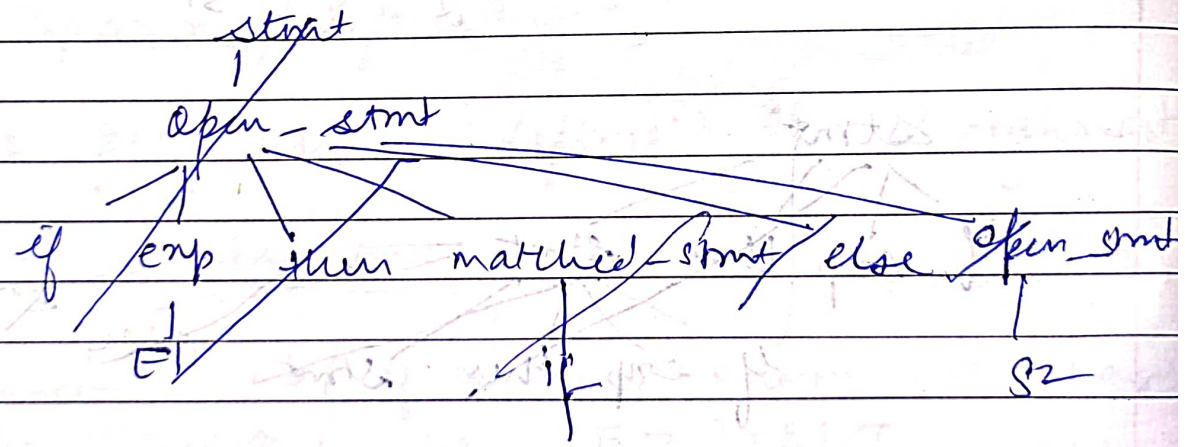
☆ Q stmt → matched_stmt | open_stmt

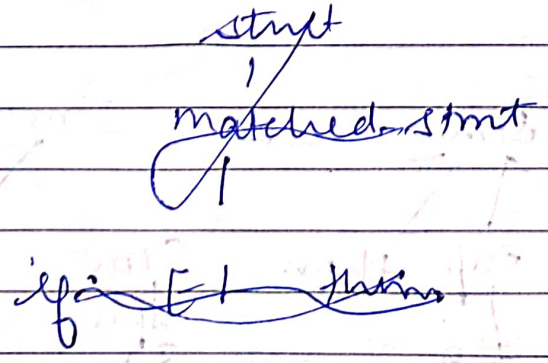
matched_stmt → if exp then matched_stmt
else matched_stmt | other

open_stmt → if exp then stmt |
if exp then matched_stmt else
open_stmt

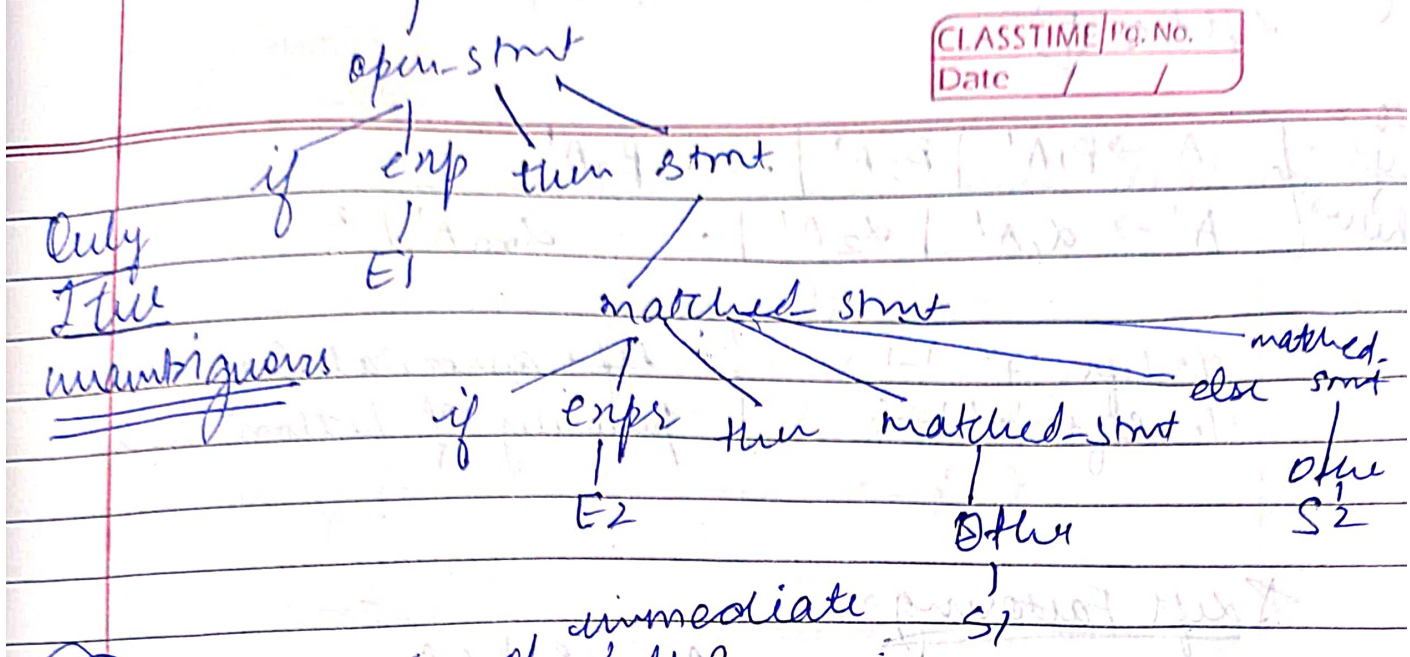
Q. ~~if E1 then S1 else if E2~~
if E1 then if E2 then S1 else S2

A:





Other → S1, S2



Only The unambiguous

Elimination of immediate Left Recursion

Must if left recursive convert to non left recursive

α // Terminal / Non terminal

Left Recursive $A \to A\alpha \mid \beta$ (Left Recursive)

jo direct dey rha make Alias

Non Left Recursive

$$\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}}$$

Q

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Left Recursive
$E \to E + T$
$T \to T * F$
(by head same)

A

$E \to TE'$
$T \to FT'$
$E' \to +TE' \mid \epsilon$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_m \mid \beta_1 \mid \beta_2 \cdots \mid \beta_n$

Left Recursive

CLASSTIME Pg. No.
Date / /

Non left Recursive

$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \alpha_m A' \mid \epsilon$

% left '+' '-'
% left '*' '/'

} left associativity + priority of bottom more

☆ Left Factoring

$A \to \alpha\beta_1 \mid \alpha\beta_2$

$\Downarrow$

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

Top down / Predictive

V.V. imp (Paper)

→ Convert left recursion to non left recursion

Q $S \to iEtS' \mid iEtSeS \mid a$

$E \to b$

Ans. Left factored common

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

} → Leftmost

Left right

☆ Q: Parser / Left first

→ Left Recursive → cause recursive descent parser even some with backtracking to go into a loop.

☆ → FIRST AND FOLLOW

⇒ find first of all non terminals.

Id + Id * Id

CLASSTIME Pg. No.
Date / /

↳ left key jitney non terminal → uska first nikalna

1) If X is terminal, first(X) = {X}
2) ——— non terminal and X → Y₁ ... Yₖ, if a (a) in First(Y) then it will be in X
if all first(Yᵢ) ε then X = {ε}

Q
S → Bb | Cd
B → aB | ε
C → cC | ε

First

1) find first of S, B, C

FIRST(S) = ? , FIRST(B) = ? , FIRST(C) = ?

FIRST(S) = FIRST(Bb) ∪ FIRST(Cd)
= FIRST(aBb) ∪ FIRST(b) ∪ FIRST(Cd)

(terminal key start ka terminal)

= {a} ∪ {b} ∪ FIRST(cCd) ∪ FIRST(d)

= {a} ∪ {b} ∪ {c} ∪ {d}
= {a, b, c, d}

FIRST(B) = FIRST(aB) ∪ FIRST(ε)
= {a} ∪ {ε} = {a, ε}

First(C) = FIRST(cC) ∪ FIRST(ε)
= {c, ε}.

Q

$S \to ABCDE$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$C \to c$
$D \to d \mid \epsilon$
$E \to e \mid \epsilon$

CLASSTIME Pg. No.
Date / /

A: FIRST(S) = FIRST(ABCDE)
~~= CEABD~~

FIRST(E) = $\{e, \epsilon\}$
FIRST(D) = $\{d, \epsilon\}$
FIRST(C) = $\{c\}$
FIRST(B) = $\{b, \epsilon\}$
FIRST(A) = $\{a, \epsilon\}$

FIRST(S) = $\{a\} \cup \{b\} \cup \{c\}$ = $\{a, b, c\}$

(aega nhi jaega c sey bcz c degr nhi bnega c hamesha aega)

Q If $C \to c \mid \epsilon$
then FIRST(S) = $\{a, b, c, d, e, \epsilon\}$

→ Top down

FIRST(S) = FIRST(ABCDE) ∪ FIRST(BCDE)
= a ∪ FIRST(BCDE)

Exams imp

CLASSTIME Pg. No.
Date / /

Q. FOLLOW

1.) $ // end of string for which parsing is done (end marker)

2.) S is start symbol

$A \to \alpha B \beta$

FOLLOW(B) = FIRST(β)

3.) $A \to \alpha B$ FOLLOW(A) = FOLLOW(B)

Q. Non left recursive

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

A.

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

FIRST(T') = FIRST(FT')
= FIRST(F)
= {(, id}

FIRST(E) = FIRST(TE')
= FIRST(T)
= {(, id}

FIRST(E')
= FIRST(+TE')
= {+, ε}

Non terminal right part dhoondho

E ko dhoondo starting non terminal

E ko RHS mein follow

FOLLOW(E) = {$} ∪ FIRST(')

= {$,)}

FOLLOW(E') = (when β is empty part ko follow)

= FOLLOW(E) (∵ E → TE', E' → ε)

(E') dhund rhi hoyegi

= {$,)}

FOLLOW(T) = FIRST(E')

= {+} ∪ FOLLOW(E')

= {+, $,)}

Follow mein kbhi ε rhi hoyegi

FOLLOW(T') = FOLLOW(T)

FOLLOW(F) = FIRST(T') ∪ FOLLOW(T)

= {+, *, $,)}

Q S → ABCDE

A → a | ε

B → b | ε

C → c

D → d | ε

E → e | ε

	FIRST(·)	FOLLOW(·)
S	{a, b, c}	{$}
A	{a, ε}	{b, c}
B	{b, ε}	{c}
* C	{c}	{d, e, $, $}
D	{d, ε}	{e, $, $}
E	{e, ε}	{$}

FOLLOW(A) = FIRST(BCDE)

$= \{b\} \cup FIRST(CDE)$

$= \{b, c\}$

FOLLOW(C) = FIRST(DE)

$= \{d, e, \$\}$

Q. $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

	FIRST(·)	FOLLOW(·)
S	$\{a, b, d, g, h, \epsilon\}$	$\{\$\}$
A	$\{d, g, h, \epsilon\}$	$\{\$, h, g\}$
B	$\{g, \epsilon\}$	$\{\$, h, g, a\}$
C	$\{h, \epsilon\}$	$\{\$, b, g, h\}$

FIRST(C) = $\{h, \epsilon\}$

FIRST(B) = $\{g, \epsilon\}$

~~FIRST(A) = {d, g, ε}~~

FIRST(S) = $\{d, g, h, \epsilon\}$

FIRST(A) = $\{d\} \cup FIRST(BC)$

$= \{d\} \cup \{FIRST(B) - \epsilon\} \cup FIRST(C)$

$= \{d\} \cup \{g\} \cup \{h, \epsilon\}$

$= \{d, g, h, \epsilon\}$

H

Stop at terminal term

CLASSTIME Pg. No.
Date

FIRST(S) = FIRST(ACb) ∪ FIRST(CbB) ∪ FIRST(Ba)

= FIRST(A) {d, g, h, ε} ∪ {h, b} ∪
{g, a}

= {a, b, d, g, h, ε}

☆ FIRST(S) = FIRST(ACB) ∪ FIRST(CbB) ∪ FIRST(Ba)

FIRST(ACB)
= FIRST(A − {ε}) ∪ FIRST(B − {ε}) ∪
FIRST(C − {ε}) ∪ ε

FIRST (ACB) = {d, g, h} ∪ {h} ∪ {g} ∪ ε

FIRST(CbB)
= FIRST(C − {ε}) ∪ {b}
= {h, b}

FIRST(Ba) = FIRST(B − {ε}) ∪ {a}
= {g, a}

S = {d, g, h, ε, b, a}

FOLLOW(A) = FIRST(CB)
= FIRST(C − {ε}) ∪
FIRST(B − {ε}) ∪ ε
= {a, g, h, ε}

FIRST(C − {ε}) ∪
FIRST(B − {ε}) ∪
= {$, h, g}

FOLLOW(S) → ε

FOLLOW (B) = FIRST (A) ∪ FOLLOW (A)
FOLLOW(S) ∪ FIRST(C) ∪

CLASSTIME Pg. No.
Date / /

$= \{\$, a, \}$

FOLLOW (C) = b ~~∪ FOLLOW (A)~~

$= b \cup \{a, g, h, \epsilon\}$

$= \{b, a, g, h, \epsilon\}$

$= b \cup$ ~~FOLLOW~~ FIRST (B) $\cup$ ~~FOLLOW~~ FOLLOW (A)

FOLLOW (C)

$C - \{\epsilon\}$ $\{\epsilon\}$ FOLLOW (A)

FOLLOW(C) = (FIRST (B) $- \{\epsilon\}$) ∪ FOLLOW(S) ∪ $\{b\}$ ∪ FOLLOW(A)

$= \{\$, b, g, h, \}$

(★) $LL^{(1)}$ Grammar (Top Down Parser (LL' Parser))

Left to Right — Leftmost derivation

(L) Look ahead

(★) LL' never works on left recursion
ambiguous grammar, LL' fails
(nonterminal, terminal → multiple entries
↓
ambiguity)

Q Check grammar is in $LL^{(1)}$ Construct its parsing if in $LL^{(1)}$

Pg. No.
Date / /

If

1) For no terminal a do both α and β derive strings beginning with a

2) At most one of α & β can derive empty string.

3) If $\beta \overset{*}{\Rightarrow} \epsilon$ then α does not derive any string beginning with terminal in FOLLOW(A)

Likewise $\alpha \overset{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with terminal in FOLLOW(A)

(1) $A \to \alpha_1 | \alpha_2 | \alpha_3 \ldots \alpha_n$

If any FIRST(α_1), FIRST(α_2), FIRST(α_3) ... FIRST(α_n) contains same element / terminal

then grammar is not in $LL^{(1)}$

(2) $A \to \alpha | \epsilon$

if FIRST(α) $\cap$ FOLLOW(A) $\neq \phi$

grammar is not in $LL^{(1)}$

Q $S \to (L) | a$ → // terminal

$L \to SL'$ a // terminal

$L' \to \epsilon | , SL'$

Check whether in $LL^{(1)}$. If yes construct parsing table.

A:

	FIRST(·)	FOLLOW(·)
S	{(, a}	{$, , ,)}
L	{(, a}	{)}
L'	{ε, ,}	{)}

FOLLOW (L) → {)}

Thank You FOR BEING WITH ME ☺

FOLLOW (L') → FOLLOW (L)

= {)}

FOLLOW(S) = FIRST(L') U $ U FOLLOW(L)

= {$, , ,) }

S → (L) (1) | a (2)

L → SL' (3)

L' → ε (4) | , SL' (5)

FIRST

S → (L)

☆ Parsing table

(ε) → nhi aega khi

Non terminals \ Terminals	a	(	)	,	$
S	S→a	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ε	L'→,SL'	

→ first of table ka pdh kr bau sha hai

→ ε par follow par rana, a sha wahan likhengay.

No intersection

(1) A → α | ε

FIRST (α) ∩ FOLLOW(A) ≠ ϕ (not in 2 (1))

L' → ε | ,SL' (2 (1))

(FIRST (,)) ∩ FOLLOW(L')

= {,} ∩ {)} = ϕ

Q. $S \to aSbS \;(1) \mid bSaS \;(2) \mid \epsilon \;(3)$

	FIRST	FOLLOW
S	$\{a, b, \epsilon\}$	$\{b, a, \$\}$

	a	b	$
S	$S \to aSbS$, $S \to \epsilon$	$S \to bSaS$, $S \to \epsilon$	$S \to \epsilon$

intersection

Problem (this grammar not in LL(1))

(ambiguous, left recursive)

Q. $S \to A\,a\,A\,b \;(1) \mid B\,b\,B\,a \;(2)$

$A \to \epsilon \;(3)$

$B \to \epsilon \;(4)$

	FIRST(·)	FOLLOW(·)
S	$\{a, b\}$	$\{\$\}$
A	$\{\epsilon\}$	$\{a, b\}$
B	$\{\epsilon\}$	$\{a, b\}$

Non T \ Term	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

unambiguous

~~ambiguity~~ + non left recursive :→ LL(1)

Q. $S \to aSA \;(1) \mid \epsilon \;(2)$

$A \to c \;(3) \mid \epsilon \;(4)$

	FIRST(·)	FOLLOW
S	$\{a, \epsilon\}$	$\{\$, c\}$
A	$\{c, \epsilon\}$	$\{\$, c\}$

cost of production

CLASSTIME Pg. No.
Date / /

	a	c	$
S	$S \to aSA$	$S \to \epsilon$	$S \to \epsilon$
A		$A \to c$, $A \to \epsilon$	$A \to \epsilon$

$\frac{1}{1+e^{-z}}$

Not in $LL^{(1)}$

Ambiguous grammar → not in $LL^{(1)}$

unambiguous → to check ($LL^{(1)}$)
↓
solve

$LL^{(1)}$
- Unamb
- Non left recursive

0
↓
iteration
new
change
in the

Sigmoid
last

Initialization proper → converge fast
X.shape[0] // features

(vanishing / exploding gradients) Read

Initialisa
- He
- Xavier
- Random

☆ Gaussian (He Initialization)

deep learning. ai

~~DS ki PDF~~

CLASSTIME Pg. No.
Date / /

Q.

$S \to iEtSS' / a$

$S' \to es / \epsilon$

$E \to b$

	FIRST(·)	FOLLOW(·)
S	{i, a}	{e, $}
S'	{e, ε}	{e, $}
E	{b}	{t}

Parsing Table

NT \ Input Symbol	i	t	a	e	b	$
S	S→iEtSS'		S→a			
S'				S'→es, S'→ε		S'→ε
E					E→b	

S'→es, S'→ε → Hence grammar not in LL.

★ push → shift pop → reduce

Top down Check whether id+id*id$ accept or not

Q. $E \to TE'$ (1)

$E' \to +TE'$ (2) | ϵ (3)

$T \to FT'$ (4)

$T' \to *FT'$ (5) | ϵ (6)

$F \to (E)$ (7) | id (8)

id+id * id$
↓
I/P

Matched	Stack	input	Action
	E$	id+id* id$	
	TE'$	id+id*id$	E→TE'
	FT'E'$	id+id*id$	T→FT'
	idT'E'$	id+id*id$	F→id
		~~id+id*id$~~	

Left most derivation

E$ in stack
↳ pop E and push E → (TE')

(pop top of stack & add its production)

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

id+ id * id $ ⟶ accept

Initial config of stack → starting Nonterminal + $
Final config of stack → $

Bottom Up Parsing (Shift Reduce) (LR parser)

LR → Left to Right + Rightmost derivation in reverse

SLR uses o/p of LR(0) + 1 addition

LR
- LR(0) (weak parser) (K=0 Look Ahead)
- SLR(1) Simple LR (1 Look Ahead)
- LALR(1) Look Ahead LR
- CLR(1) Canonical LR

LR(0) – ~~less~~ more conflict
SLR(1) – less ——

CLASSTIME Pg. No.
Date / /

yacc → LALR(1) parser / has redundant state

CLR(1) overcome LALR(1) → CLR(1) reduce no of states in LALR(1)

no of states CLR(1) ≤ no of states LALR(1)
(redundant states of LALR(1) merge / remove)

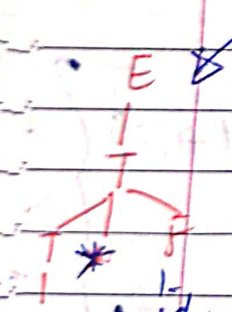
Q: $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

id*id

A- $E \Rightarrow T$
$\underset{rm}{\Rightarrow} T*F$
$\underset{rm}{\Rightarrow} T*id$
$\underset{rm}{\Rightarrow} F*id$
$\underset{rm}{\Rightarrow} id*id$

handle

Bottom to up
↓
reduce → ~~non~~ terminal
non terminal



E, T, T*F, T*id, F*id, id*id
↓
Right Sentential Form (RSF)
(jo bhi right side par occur kr sakta)

* Handle Pruning

Clarity of id rename to id1 id2

RSF	Handle	Reducing Production
id1 * id2	id1	F → id
F * id2	F	T → F
T * id2	id2	F → id
T * F	T * F	T → T*F
T	T	E → T

Q $S \to 0S1 \mid 01$ indicate handle in RSF

CLASSTIME Pg. No.
Date / /

i) 000111

$S \overset{rm}{\Rightarrow} 0S1$ $(S \to 0S1)$

$\overset{rm}{\Rightarrow} 00S11$ $(S \to 0S1)$

$\overset{rm}{\Rightarrow} 000111$ $(S \to 01)$

(Practice 0's from book)

LR parser ~~(Shift Reduce Parser)~~ (Bottom Up) Shift Reduce Parsing)

Stack → contains Grammar symbol | Input Buffer contains rest of the string to be parsed

Initial Configuration

STACK $

Input Buffer w$

↓

(terminal string to be parsed)

Q/P. STACK | O/P

$ | $

S

(starting Non Terminal)

(match ↔ shift, output ↔ reduce)

↓

(terminology in Bottom Up)

Q1 Same (LR(0))

Stack	INPUT	Action
$	id1 * id2 $	Shift
$ id1	* id2 $	reduce by F → id
$F	* id2 $	reduce by T → F
$T	* id2 $	shift
$T*	id2 $	shift
$T* id2	$	Reduce by F → id
$T*F	$	Reduce by T → T*F
$T	$	Reduce by E → T
$E	$	accept

4 Steps :- Shift, Reduce, Accept, Reject

CLASSTIME Pg. No.
Date / /

(*) LR parser (2 Conflicts) :-
Shift Reduce Conflict — id + id * id
Reduce Reduce Conflict — id + id * id

(1) SR Conflict
↳ overcome in yak
- Parenthesis
- Assume Left Associative / Right Ass.

(2) Reduce - Reduce
↳ overcome in yak

Associativity

% left '+' '-'
% left '*' '/'

(*) LR(1) / CLR(1) — $k=1$ LR(0) $k=0$

LR(K) restrict parser $K \leq 1$
↳ $K = 0/1$

When K omitted, K assumed to be 1

$\boxed{LR \rightarrow LR(1) \rightarrow CLR(1)}$

(*) LR(0) Automaton / Goto Graph

- Construct Augmented Grammar (extra ~~productions~~ rule of with alias of start symbol) $E' \rightarrow E$

Q: same as above

$E \rightarrow E+T$	put .	$E \rightarrow . T$
$E \rightarrow T$	in	$E \rightarrow . E+T$
$T \rightarrow T*F$	front	$T \rightarrow . T*F$
$T \rightarrow F$	of	$T \rightarrow . F$
$F \rightarrow (E)$	every	$F \rightarrow . (E)$
$F \rightarrow id$	right	$F \rightarrow . id$
	Production	

(stops ~~[illegible]~~ [illegible])

Closure property

Arleen Kaur Bakshi

CLASSTIME Pg. No.
Date / /

Mehak

* $E' \to \cdot E$ (0th production)
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot F$
$T \to \cdot T*F$
$F \to \cdot (E)$
$F \to \cdot id$

→ · left hand side symbols are passed

→ $A \to XYZ\cdot$ final item

→ $A \to \epsilon$ generates one item $A \to \cdot$

· first item (I_0) · Key wagy non terminal aa jaye so ~~non terminal~~ production usky corresponding copy

LR(0) [K=0, K=1 (Look ahead)]

SLR(1)/SLR
LR(1)/CLR(1)
look ahead K=1

(#) Non terminal ↑, Automaton ↑

(#) Canonical LR(0) collection ($I_0, I_1 - - - - I_n$)

(☆) goto

Q: # Construct LR(0) automaton/Goto Graph

(· Key bad non terminal ko closure
~~($E' \to E$)~~

I_0
$E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

E → I_1: $E' \to E\cdot$, $E \to E\cdot+T$

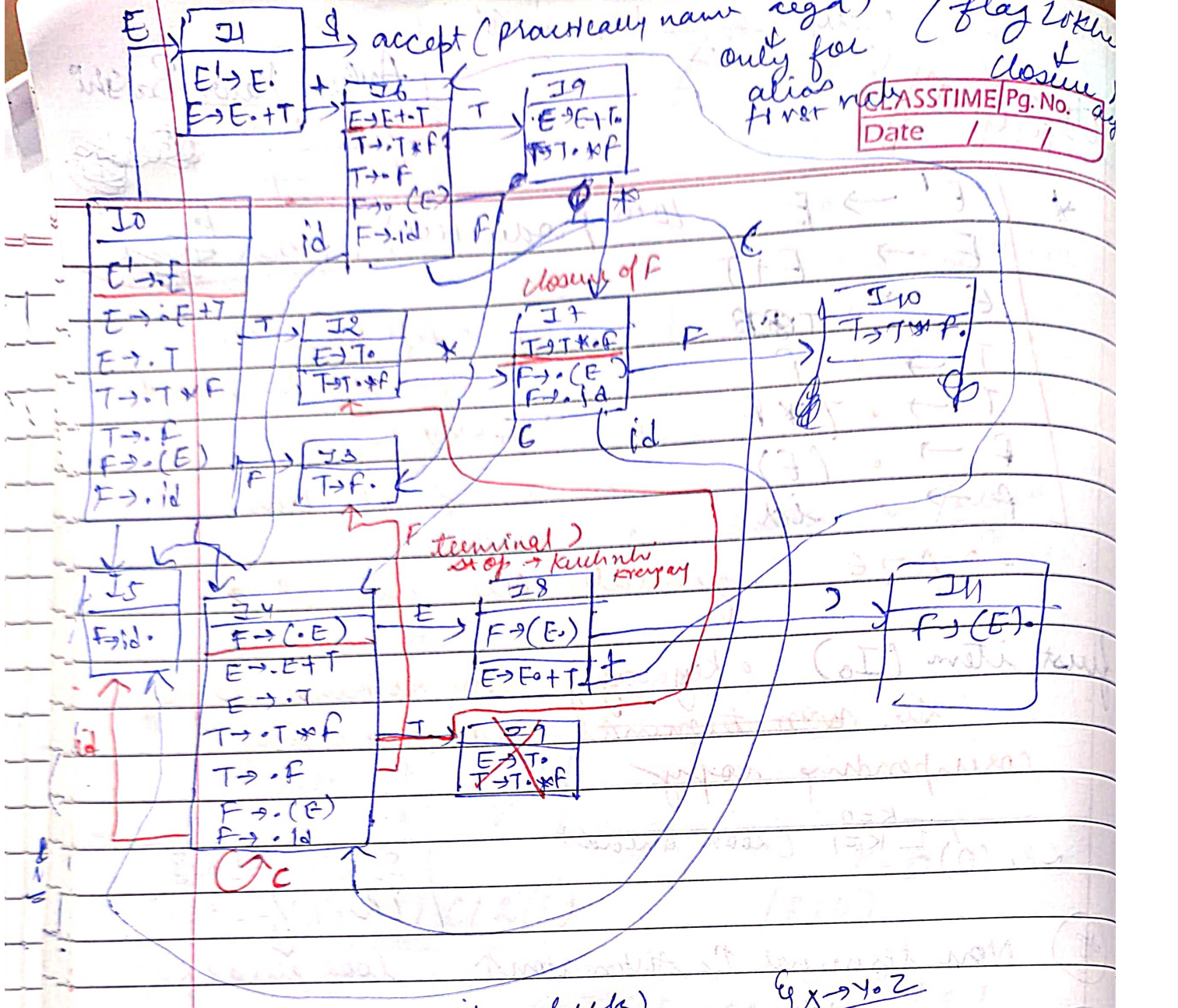
T → I_2: $E \to T\cdot$, $T \to T\cdot*F$

F → I_3: $T \to F\cdot$

(→ I_4: $F \to (\cdot E)$, $E \to \cdot E+T$, $E \to \cdot T$, $T \to \cdot T*F$, $T \to \cdot F$, $F \to \cdot (E)$, $F \to \cdot id$

→ Non terminal closure

id → I_5: $F \to id\cdot$


I1
E'→E.
E→E.+T
accept
I6
E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id
I9
E→E+T.
T→T.*F
closure of F
I7
T→T*.F
F→.(E)
F→.id
I10
T→T*F.
I0
E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id
I2
E→T.
T→T.*F
I3
T→F.
I5
F→id.
I4
F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id
I8
F→(E.)
E→E.+T
I11
F→(E).
I7
E→T.
T→T.*F
terminal
stop → kuch nahi karnaya
id
id
id

(final item → so reduce to parent)

(jab bhi reduce hota hai, RHS main jo Prev item hai pop from stack state & check parent us item par kaha ja rha hai)

Reduce by F→id (pop one state (5) and shift (push) to state where 'O' is going by reading F)

	02 $T	$	reduce by E→T
S/r	01 $E	$	accept
confid			

Parsing with help of automaton

ACTION[i, a]

(☆) Shift Reduce / Bottom Up

Shift
Reduce
Accept (acc)
Reject/Error

row i (state) coln (a terminal)

GOTO (2d array) → GOTO[Ii, A] = Ij

state sy non terminal pardh key → Ij state

Action // Terminals GOTO (NON TERMINALS)

r6 // Reduce (Rule No 6)

s6 // shift to 6

2	
5	
7	

☆ Parsing table for exp. grammar (SLR(1))

Fig 4.37 → Pg 277

(☆) If LR(0) → r2 in complete row

LR(0)

↓ improvement

SLR(1) (Follow up jahan S2/r7)

Follow of (E)

final item ka head

		State
(1)	$E \to E+T$	0
(2)	$E \to T$	1
(3)	$T \to T*F$	2
(4)	$T \to F$	3
(5)	$F \to (E)$	4
(6)	$F \to id$	5

Regularization

23 Feb:

Q. Or

$S \to AA$
$A \to aA$
$A \to b$
(same Q)

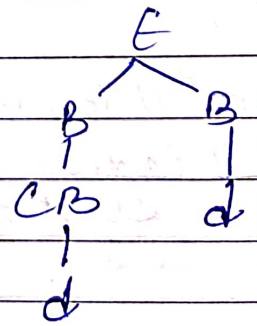
Q. Check whether grammar is LR(0) or not

① $E \to BB$
② $B \to CB \mid d$ ③

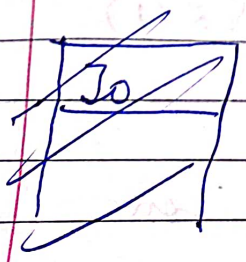
cdd ✓
cd ✗

A. (1/3) Augmented Grammar

$E' \to \cdot E$
$E \to \cdot BB$
$B \to \cdot CB$
$B \to \cdot d$



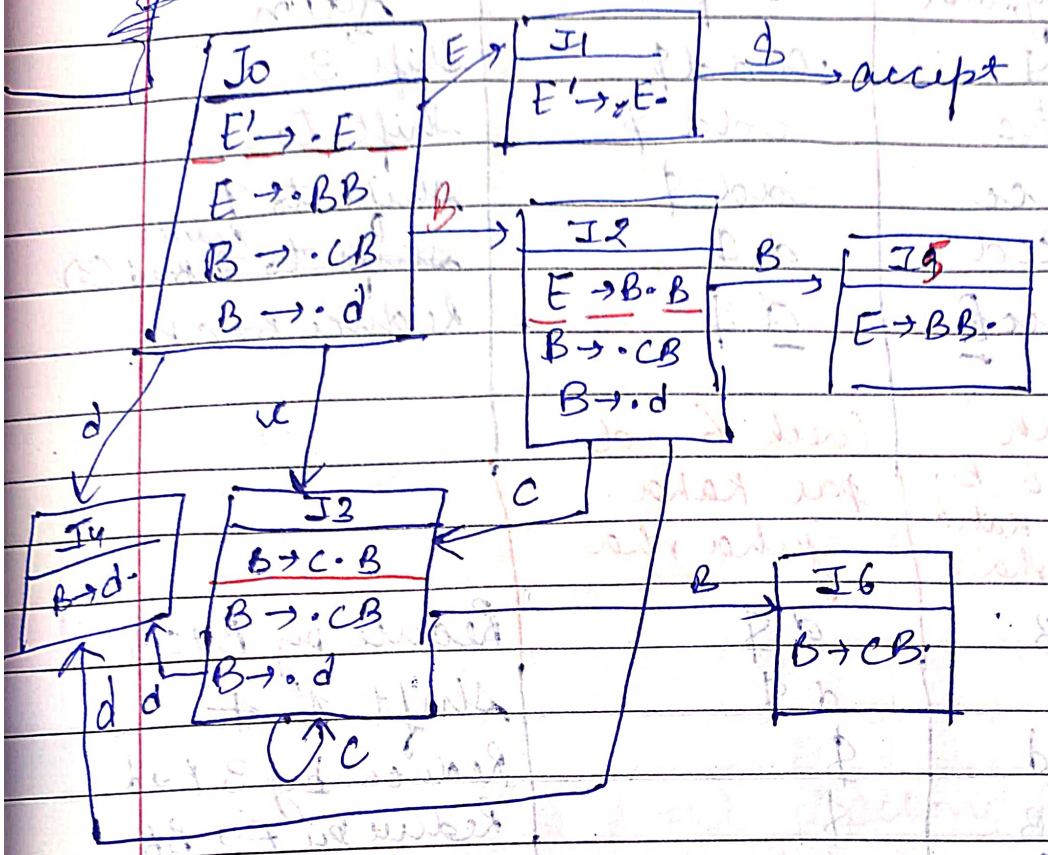
Constructing Automaton



$I_0, I_1, I_2, I_3, I_4, I_5, I_6$
7 states

CLASSTIME Pg. No.
Date / /

(4 marks)



Parsing table

State	Action	GOTO
0		
1		
2		
3		
4		
5		
6		

reduce
r 3
↓
rule No B→d ③

r1 → rule NO 1
E→BB

r 2
↳ rule NO 2 B→CB

(3.5 marks)

State	Action c	d	$	GOTO E	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
		s3	r2		

Parsing of ccdd (Help of Parsing table)

CLASSTIME Pg. No.
Date / /

Stack State	Symbol	Input	Action
0	$	ccdd$	Shift to 3
03	$c	cdd $	Shift to 3
033	$cc	dd $	Shift to 4
0334	$ccd	d $	~~Shift to 4~~ Reduce B→d
0336	$ccB	d$	Reduce B→cB
(pop 4, then 3 B par kaha jaana tha) 6	Check ki 6 B par kaha ja rha	Check 6 d par kaha jaha rha	
036	$cB	d $	Reduce by B→cB
02	$B	d $	Shift to 4
024	$Bd	$	Reduce by 3 B→d
025	$BB	$	Reduce by E→BB
01	$E	$	accept

★ SLR(1) (just change parsing table)

~~State~~

E → BB ①
B → cB ② | d ③

	FIRST	FOLLOW
E	{c, d}	{$}
B	{c, d}	{c, d, $}

[E→BB]
↓
FOLLOW(B) = FOLLOW(E)

If kaha conflict aega ACTION [4, c]
4 state par c par kya conflict aega

→ if input symbol dono par hai with If parsing table mein then go

State	action c	action d	action $	GOTO E	GOTO B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Every LR(0), without conflict is always in grammar

SLR(1) true

4 state (B → a) Follow of B dekha c, d, $ so sb par aega

5 state (E → BB) Follow of E dekhe ($ sirf us par aega)

6th state (B → CB) (Follow of B dekhe (c, d, $ sb par aega)

Q. Show that is LL(1) but not SLR(1)

Sol.

S → AaAb (1) | BbBa (2)

A → ε (3)

B → ε (4)

LL(1) ↓ (already kr skha hai)

Augmented Grammar

S' → · S

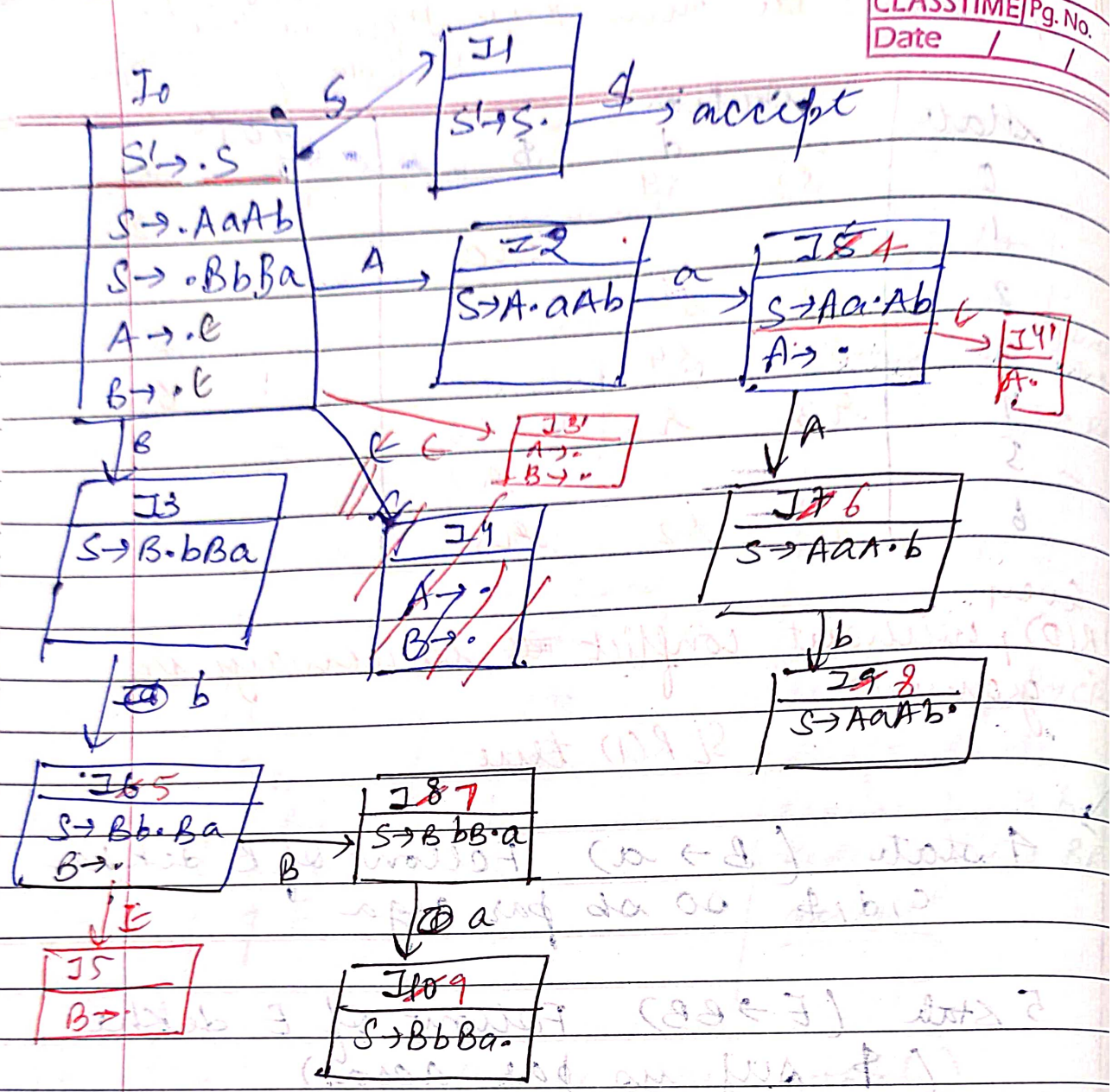
S → · AaAb

S → · BbBa

A → ·

B → ·

Automata



	first	follow
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{a, b}

State	ACTION a	b	$	GOTO A	B	S
0				2	3	1
1			acc			
2	s4					
3		s5				
4	r3	r3		6		
5	r4	r4			7	
6		s8				

Construct complete Parsing Table :-

	a	b	$	A	B	S
7	S9					
8			r1			
9			r2			

This Question Postponed Later ★★ (for conflict)

Reduce - Reduce conflict on (I3')

~~Q S → SA~~
Fig 4.6-6.

→ find shift/reduce conflict

CLASSMATE Date / /

Eg. 448 $S \to L = R \mid R$
$L \to *R \mid id$
$R \to L$

(Automato, parsing)

first 3 states conflict

Viable prefixes

(Prefixes of right sentential forms that can appear on stack of a shift reduce parser)

(symbol under stack in table)

Q. ~~id+id*id~~

Stack	Input	Action
$	id+id$	Shift id
$id	+id$	Reduce by F→id
$F	+id$	Reduce by T→F
$T	+id$	Shift +
$E+	id$	Shift id
$E+id	$	Reduce by T→id
$E+F	$	Reduce by T→F
$E+T	$	Reduce by E→E+T
$E	$	accept

$E \to E+T$
$E \to T$
$T \to T*F$
$T \to F$
$F \to id$

Viable prefix → (id, F, T, E+, E+id, E+F, E+T, E)

jo stack pr gaya

Rightmost derivative :

$E \overset{rm}{\Rightarrow} E+T \Rightarrow E+F \Rightarrow E+id \Rightarrow T+id \Rightarrow F+id \Rightarrow id+id$

$E \to E+T$
$\Rightarrow E+F$
$\Rightarrow E+id$
$\Rightarrow T+id$
$\Rightarrow F+id$
$\Rightarrow id+id$

(right sentential mein sey jitna stack mein pahunchey)

E+id is viable prefix (unique values)
E* id not viable prefix
(first check stack column & write all forms)

CLASSTIME Pg. No.
Date

More powerful LR parsers

LR(1) = LR(0) + lookahead

(Reduced version of LALR)

(1) Canonical LR / LR(1) items / CLR(1)

2) Look ahead LR / LALR / LR(0) based on states of items

- CLR(1) less/equal states than LALR
- CLR(1) not more states than LALR

CLR(1) and LALR(1)

LR(1) item = LR(0) item, Lookahead

LR(0) → Put reduce in full row
SLR(1) → " " follow coln only
CLR(1) → " " " Lookahead coln only
& LALR(1)

Q. E → BB
B → CB
B → d

Check LALR(1) or not

(B par ~~done~~ terminal mil gaya so ~~$~~ $ nhi liyay)

A. Augmented Grammar

E' → • E , $ (Look ahead)
E → • BB , $ (Lookahead of(E) = First($) = $ copy kr do necessary)
B → • CB , c/d
B → • d , c/d

($ nhi aega)
B ka lookahead aega

Case I : $A \to \alpha \cdot B\beta, c/d$

$B \to L, a$

CLASSTIME Pg. No.
Date / /

Look ahead (B) = First (β, c/d) = a

27/12 Check whether the following grammar is in LR(1) or not?

$E \to BB$

$B \to cB \mid d$

Augmented Grammar for LR(1)

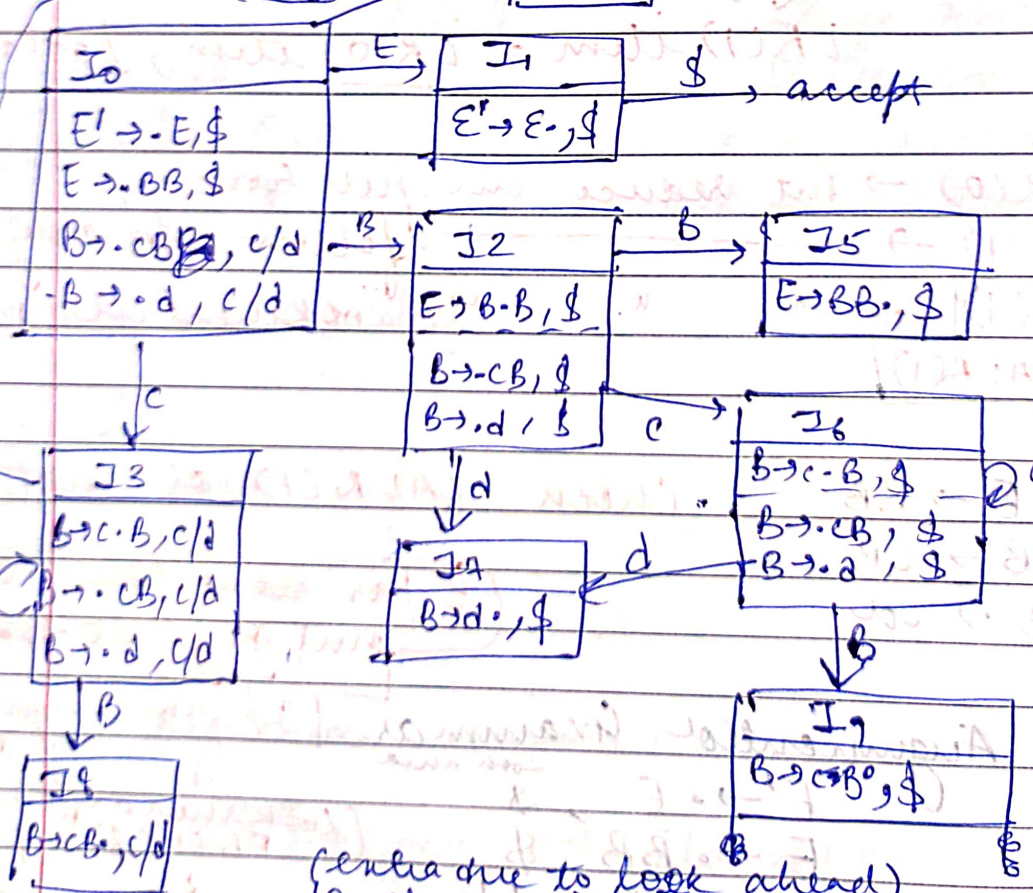
$E' \to \cdot E, \$$ (always \$ for first n

(FIRST(\$) = \$) $\to$ $E \to \cdot BB, \$$

$B \to \cdot cB \mid \cdot d, c/d$

FIRST(B \$) = {c, d}

first of (c/d) = c/d



(extra due to look ahead)
→ 10 states in LR(1) but it was 7 states in LR(0)

for product $E \to .BB$, check lft side Non terminal in grammar.

look ahead for $E \to .BB$ = First($\$$)

look ahead for $B \to .CB | d$ = First($B\$$) = $\{c, d\}$

(similar LR(0) but diff lookahead so LALR keep both states diff)

① $E \to BB$
② $B \to CB$
③ $B \to d$

left par Non terminal ko dekho .NMT right par kaho kaha aa raha uskey agay ka first nikalo

Parsing table [LR(1)]

State	Action c	Action d	Action $	GOTO E	GOTO B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

reduce wahan aega uskey lookahead par

LALR(1)

I_8, I_9
I_3, I_6
I_4, I_7

} 3 pairs of states with same LR(0) items, but diff with respect to look ahead

3 pair of states which are similar in LR(0) items but diff with respect to look ahead Hence they can be merged.

LALR
~~CLR~~(1) → Merging
(just change parsing table)

CLASSTIME Pg. No.
Date / /

$I_3, I_6 \rightarrow I_{36}$ | $I_8, I_9 \rightarrow I_{89}$
$I_4, I_7 \rightarrow I_{47}$

n (no of states)

Imp

$n(LR(0)) = n(SLR(1)) = n(LALR(1)) \leq n(CLR(1))$

State	ACTION c	d	$	GOTO E	B
0	s36	s47		1	2
1			acc		
2	s36	s47			5
3 6	s36	s47			89
4 7	r3	r3	r3		
5			r1		
~~3 6~~ merge	~~s36~~	~~s47~~			~~89~~
~~4 7~~			~~r3~~		
8 9	r2	r2	r2		
~~8 9~~			~~r2~~		

r sey panga nhi lena # only with s

Make new parsing table for merging

State	ACTION c	d	$	GOTO E	B
0	s36	s47		1	2
1			acc		
2	s36	s47			5
3 6	s36	s47			89
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

☆ Conflict can occur while merging in CLR(1)
So hence that would be ~~more~~ CLASSTIME Pg. No. Date / /
LR(1) but not in LALR(1) - by conflict occurred in LALR(1)

☆ Parsing table chota states kam → LR(0)
Parser [Conflict is trade of with error handle krney ki reqt.]

Max powerful Parser :- SLR(1), LALR(1)
less chances of conflict, but more spaces in parsing table - problematic for parser.

GOTO / LR(1) Automaton

- CLR(1) / LR(1)

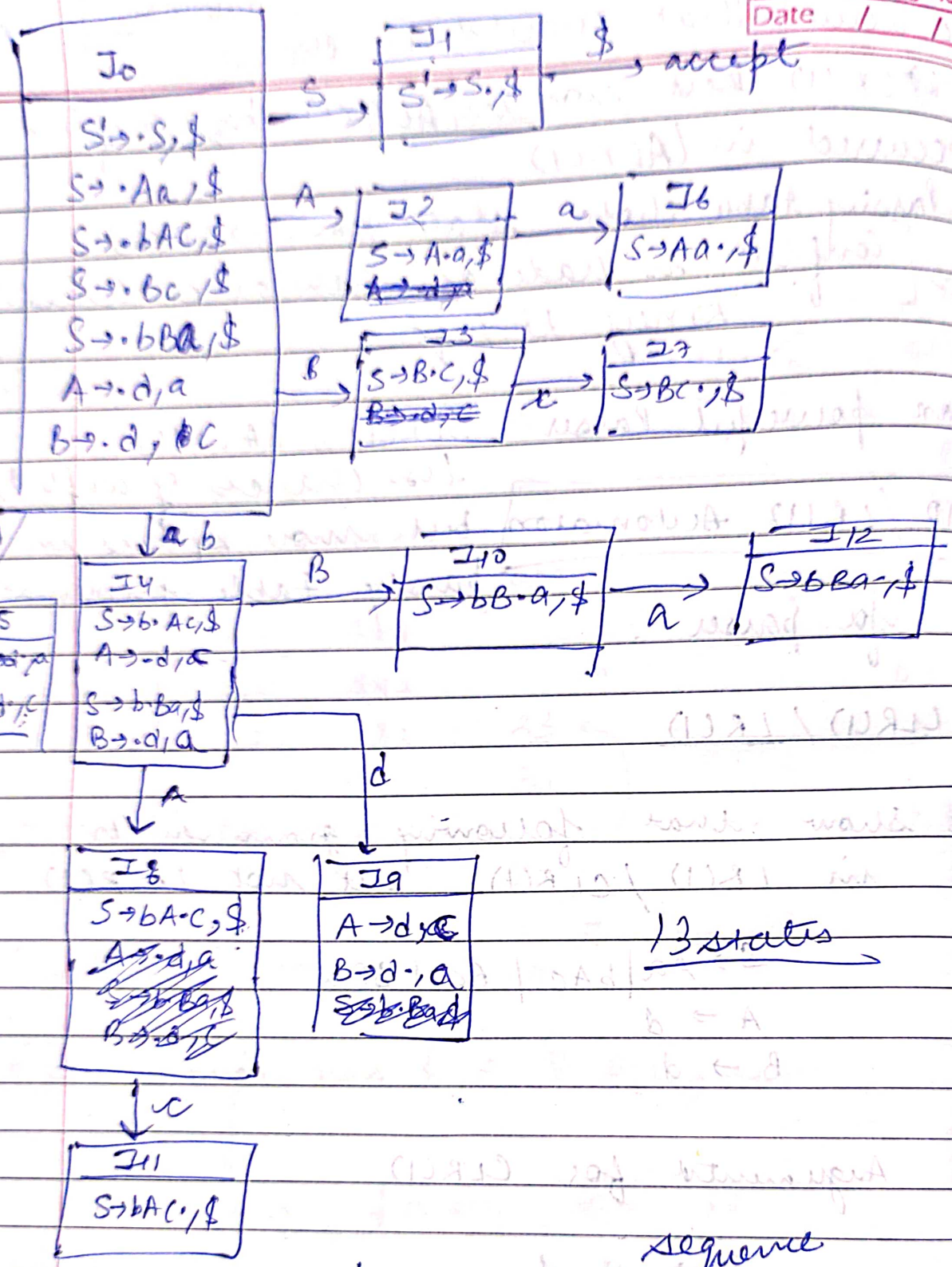
Q. Show that following grammar is in LR(1) / CLR(1) but not LALR(1).

$S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

A. Augmented for CLR(1)

$S' \to .S, \$$
$S \to .Aa \mid .bAc \mid .Bc \mid .bBa, \$$
$A \to .d, a$
$B \to .d, c$

~~S → Aa~~



13 states

sequence follow krn

S→ Aa 1
S→ bAC 2
S→ BC 3
S→ bBA 4
A→ d 5
B→ d 6

Parsing table of CLR(1)

CLASSTIME Pg. No.
Date 16/10/20

State	Action a	b	c	d	$	S	A	B
0		s4		s5		1	2	3
1					acc			
2	s6							
3			s7					
4				s9			8	10
5	r5		r6					
6					r1			
7					r3			
8			s11					
conflict 9	r8		r5					
10	s12							
11					r2			
12					r4			

5, 9 merging (conflict)

↓

LALR(1)

LALR(1) Merging

State	a	b	c	d	$	S	A	B
0		s4		s59		1	2	3
1					acc			
2	s6							
3			s7					
4				s59			8	10
5 9	r59		r6					
6					r1			
7					r3			
8			s11					
10	s12							
11					r2			
12					r4			

Yacc Practical

CLASSTIME Pg. No.
Date / /

→ yet another compiler compiler

inp: set of grammar rules → [yacc (bison)] → y.tab.h, y.tab.c } default
fn (yyparse())
o/p (C code)

bison (windows)
yacc (unix)

-yvd (flags)
(____.y) - extension of yak

C Linker → lex. yy.cc. & y.tab.h
(only main) main fn here only
↳ No main

- Scanner (Lexical Analyzer)
  Parser (Syntax Analyzer)

yytext, yylval : yacc -d
lexeme token value ↳ y.tab.h

- yacc -v (describes states, transitions/parser)
  (useful for debugging)
  ↓
  y.output (automata of grammar)
  ↳ (graphviz ...)

Window → bison -yvd
bison -yv --report=all test.y
(closure)

↳ yyparse automatically calls yylex

def
%%
rules (grammar semantic action)
%%
code

%% | smallest yacc

CLASSTIME Pg. No.
Date / /

%token
%start | y.tab.h

lex.l key andar dena hoga y.tab.h

→ lex.l mein y.tab.h na bhi ho.
toh chal rha bcz sb copy ho rha. (C program abhi bhi chl rha)

• stuff to be copied as it is into o/p.
(declaration)

A : $B_1 B_2 \ldots B_3$
| $C_1 C_2 \ldots C_3$
| $D_1 D_2 \ldots D_3$
;

Using Ambiguous Grammar (Continue ....)

Precedence & Associativity to Resolve Conflicts

LR(0) Automaton

Automata for Expression Grammar V1

$E' \to \cdot E$

$E \to \cdot E + E$ (1) | $\cdot E * E$ (2) | $\cdot (E)$ (3) | $\cdot id$ (4)