"Compilers - Aho, Monica S. Lam, Ravi Sethi principles, Techniques & Tool".

CLASSMATE Date Page

Theory of computation - Theorems.

10M - Tutorial
10M - Technical paper or AI, ML, DL project.

Introduction to Compiler Design

Language processors : A software program that translates and executes high-level programming languages into machine language.

i) Preprocessor ← macro
ii) Compiler ← HLL (modified)
iii) Interpreter (or) and
iv) Assembler (machine dependent) ← Assembly language
v) Linker/Loader (for addn'al libs)

object code

↓

final ⇒ (relocatable object code).

RISC, CISC, 8086x (object code)

Compiler Design has 6 phases : (mainly 5)

Phase 1: Lexical Analysis (scanning) ← source program — Input:

↓

Output: divides the source program into set of tokens

↓ Input

Phase 2: (parsing) Syntax Analysis — checks whether it has any c tokens. All the rules are checked (meets the syntax.)

syntax tree/ parse tree ↓

phase 3: Semantic Analysis — checks if the source code makes sense, (type checking), undeclare vars etc

modified parse tree ↓

phase 4: Intermediate code generation — Translates the parse tree into machine code that can be executed by computer

3AC (3 Address code) ↓

library fn checking ↓

Phase 5: code optimization → code generation analysis

Code optimisation → gives modified 3Ac if optimization is done

Phase 6: code generation ← Architecture

(X) 3AC (Three address code):

2 operands & one result is stored using three

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

$t1 = \text{uminus } c$ (unary minus on c)

$t2 = b * t1$

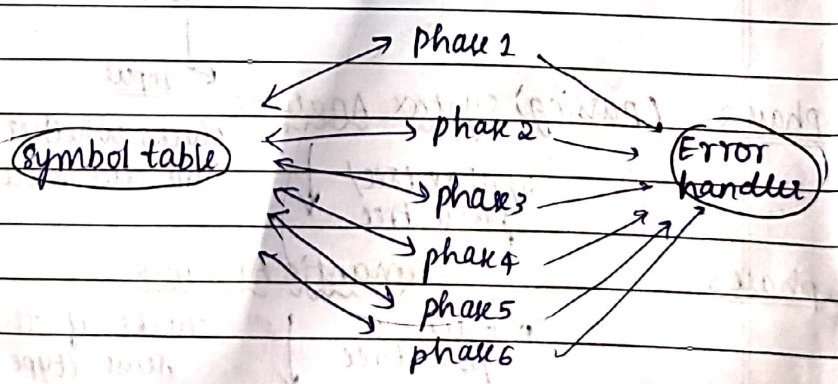
$t3 = \text{uminus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

#	op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a



Tool: 'Lex' to write Regular expression & Lexical analysis.

Example:

$S = a + b$

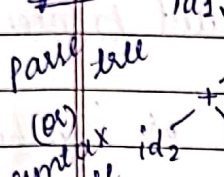
Scanner → Lexical Analysis

<id,1>, <=>, <id,2>, <+>, <id,3>

Parser → syntax Analyser

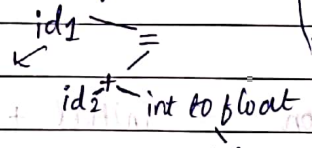
Syntax Analysis:

parse tree (or) syntax tree



Result

Semantic Analyzer



Intermediate code Generator

data structure used: 3AC

$t1 = int\ to\ float(id3)$

$t2 = t1 + id2$ let $id1 = t2$

code optimizer

$t1 = int\ to\ float(id3) + id2$

$id1 = t1 + id2$

Code Generator

```
LDF R1 id2
LDF R2 id3
ADDF R1, R2, R3
STF id1 R1
```

load floating point.

add floating point

Store floating point.

Assembly lang code → Assembler → Object code

Consider a, b ⇒ float; b = int

LA:

* Input scanned from Left to right character by character
* tokenizes.
* Regular expressions.

RE: $L\_(L\_|d)^*$

L → Letter

_ → underscore

d → digit.

$L\_ \rightarrow [a\text{-}z\ A\text{-}Z\ \_]$

$d - [0\text{-}9]$

* → 0 or more occurence of the content

id (identifier) → $L\_(L\_|d)^*$

var → starting should be a letter or underscore later can be anything.

* First give defn of keywords like (int, char, etc) then define the variables.

So,

$S = a + b$

$(id1 = id2 + id3)$

* scanning ends when ; or } is found.

Symbol table

1	s	identifier(var)
2	a	var "
3	b	var "

Literal table
↓
stores format specifiers

NOTE: * first four phases of the compiler design is called "frontend of the compiler or "Analysis phase"
* Last two phases is called the "Backend" or "synthesis phase"

Example: position := initial + rate * 60

(all the variables are integers)

↓

Lexical Analyzer

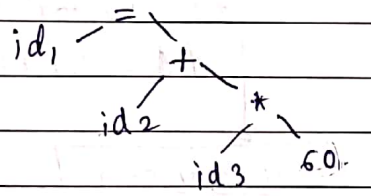
$id_1 = id_2 + id_3 * 60$

$<id_1, 1>, <=>, <id_2, 2>,$
$<id_3, 3><*><60>$

Symbol table
1 position
2 initial
3 rate

↓

Parser



↓

Semantic Analysis

↓

Intermediate code generator

$t1 = id_3 * 60$
$t2 = id_2 + t1$
$id_1 = t2$

Code optimizer

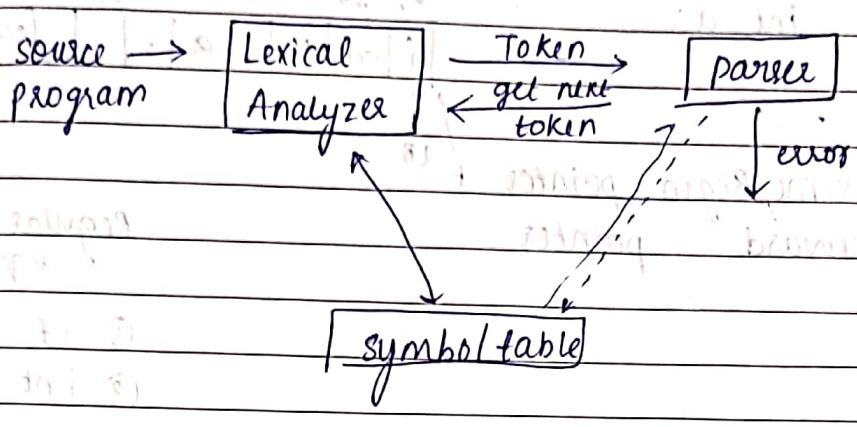
$t1 = id_3 * 60$
$id_1 = id_2 + t1$

Code generator

LD R2 id3
MUL R2 R2 60
LD R1 id2
ADD R1 R2
ST id1 R1

UNIT-II

Role of the Lexical Analyzer



Functions

1. Grouping input characters into tokens
2. Stripping out comments & white spaces
3. Handle include files & macros
4. Keep track of no. of newline characters

Lexemes:

Sequence of source code that matches one of the predefined patterns & thereby forms a valid token.

x + 5

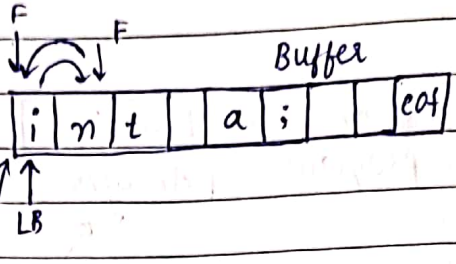
Lexemes correspond to certain tokens.

Ex: ① "main" is lexeme of type identifier (token)
② (,), {,} are lexemes of type punctuation (token)

Tokenisation:

Input Buffering Technique:

int a; ⇒



Lexeme Begin pointer
Forward pointer

Regular Expression sequence

① if
② int

Step 1: Forward pointer checks if 'i' is in any Regex sequence.

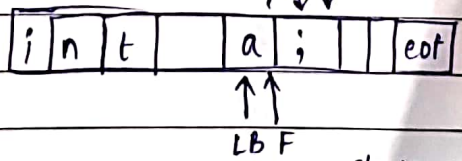
'i' is matching with ① so, increment forward pointer

Step 2: next it checks n ≠ f so, ① is not the regex. so, forward point decrement (retracts)

if there is a character mismatch, retract the forward pointer.

Step 3: next it checks 'i' with ② nd regex.

it matches so, it sends 'int' to the parser. parser checks if it is a reserved word or not. it sends the 'int' back to Lexical analyser to increment the LB pointer to the end of this word. and masks it as token 1.



Step 4: next 'a' doesn't match with ① & ②, so, create another regex for id

NOTE: ① $a^* = \{\varepsilon_0, a, aa, aaa, aaaa, \ldots\}$

empty string.

② $a^+ = \{a, aa, aaa, \ldots\}$

Regex sequence

① if
② int
③ id → l_(l_/d)*
④ ;} → synchronisation token.

so, 'a' matches with ③rd regex

∴ 'a' is token 2.

steps: ; is a synchro token so, it marks end of a line or a statement.

∴

Tokens	Lexemes
int	i n t
id	a.

This Tokenization is also called "Sentinels" approach

eof - end of file (or) end of input.

Lex chars used to generate Regular Expression:

	Expression	Example:
	©/	a (any char)
[] ↓ character class	\c	* (char c literally)
	"s"	string "ab"
	.	any char other than new line
	^	^s, beginning of the line with s
	$	end of line search string.
	[s]	[abc], any of the char in string
	[^s]	[^abc], any one char not in string 'abc'.
Kleene closure ←	r*	Zero or more occurence of r
positive closure ←	r+	one or more occurence of r
	r?	zero or one occurrence of r.
	r{m,n}	b/w m and n occurence of r

$r_1 r_2$	ab
$r_1 \mid r_2$	$a \mid b \rightarrow$ (a or b)
(r)	$(ab) \rightarrow$ grouping
r_1 / r_2	r_1 when followed by r_2

Example:

① $(a \mid b)\,(a \mid b)$ (a or b, a or b)

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

so, $\Sigma = \{a, b\}$

$\Sigma \rightarrow$ set of alphabets.

Specification of Tokens:

Regular expressions → notation for specifying lexeme pattern

Alphabet – finite set of symbols

Language – countable set of strings over some fixed alphabet.

NOTE: $\{0, 1\}$ → binary alphabets.

Terminologies of strings (s):

① Prefix(s): removing 0 or more chars from the end of s.

② Suffix(s): removing 0 or more chars from the beginning of s.

③ Substring(s): obtained by deleting any prefix & suffix from s.

④ Proper prefix(s): suffix/prefix other than s, ε → epsilon (empty string itself)

⑤ Subsequence(s): obtained by deleting 0 or more chars not necessarily consecutive chars.

* In a string of length of 'n'

i) prefix: → n+1

ii) suffix: → n+1

iii) proper prefix: → n-1

iv) substring :→ $\frac{n(n+1)}{2}$

v) subsequence (NP hard) :→ 2^N

* $(a|b)^* \Rightarrow (a^* b^*)^*$ $\Sigma = \{a, b\}$

Language (L) = { ε_0, a, b, aa, ab, bb, aab, ... }

* Describe the languages denoted by the following regular expression

i) a $(a|b)^*$ a → all strings of 'a' & 'b' (all possible combinations) starting & ending with 'a'.

ii) $(a|b)^*$ → all possible combinatiorial strings of 'a' & 'b' including empty string (ε_0)

* Write regular definitions using character class

i) All lowercase consonants in order:

[^aeiou]

ii) comments (multiline)

Epsillon ε_0 L'Hospital

* write Regular expression to accept cse(aiml) students of 2022 batch msrit email id.

1MS22CI(d,d d) @ msrit.edu

$d_1 \rightarrow [0|4] \Rightarrow q_3|q_4$

$d \rightarrow [0-9]$

$r_1 \rightarrow$ "1MS22CI"

$r_2 \rightarrow$ "msrit.edu"

$r_1 (d, dd) @ r_2$

$d_2 \rightarrow (2|3)$

regex: "1MS2(d_2)CI((d_1)dd)@msrit.edu

Recognition of Tokens:

relop → <|>|<=|>=|=|<>

Transition Diagram: → deterministic

Transition of a high level language code to a low level language code is represented using Transition diagram.

start state: →○

* → backtracking.

(accepting) final state : ◎

NOTE: each state can scan only one character at a time.

deterministic means: ○——→○

one state can only have one edge connecting to another state.

accept cse(aiml) students
id.

msrit.edu
→ (0 4) ⇒ a|z
[0-9]

1"
edu"

r2

d2 → (2|3)

@ msrit.edu

nistic

ode to a low level
Transition diagram.

* → backtracking.

nly one character

nly have one edge
nother state.

* If while scanning, the characters are similar, to differentiate it from the original character, check the longest matching pattern.

suppose 'a <= b' is the goal.
it checks a <= b but there is no '=' in goal string.
a <> b but there is no '>' in goal string.

* Transition diagrams have a collection of circles or nodes called 'states'.
Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

it summarizes all we need to know about the characters between "Lexeme begin" & "Forward pointer".

* Edges are directed from one state to another
Each edge is labeled by a symbol or set of symbols.

* If we are in some state 's', the next input symbol is 'a', we look for an edge out of 's' labeled by 'a'. If we find such an edge, we advance the forward pointer & enter the state of the transition diagram to which that edge leads.

* If we want to retract forward pointer one position (i.e., lexeme doesn't include the symbol that got us to the accepting state), then we additionally place a * near that accepting state.

Example: consider the token "relop".

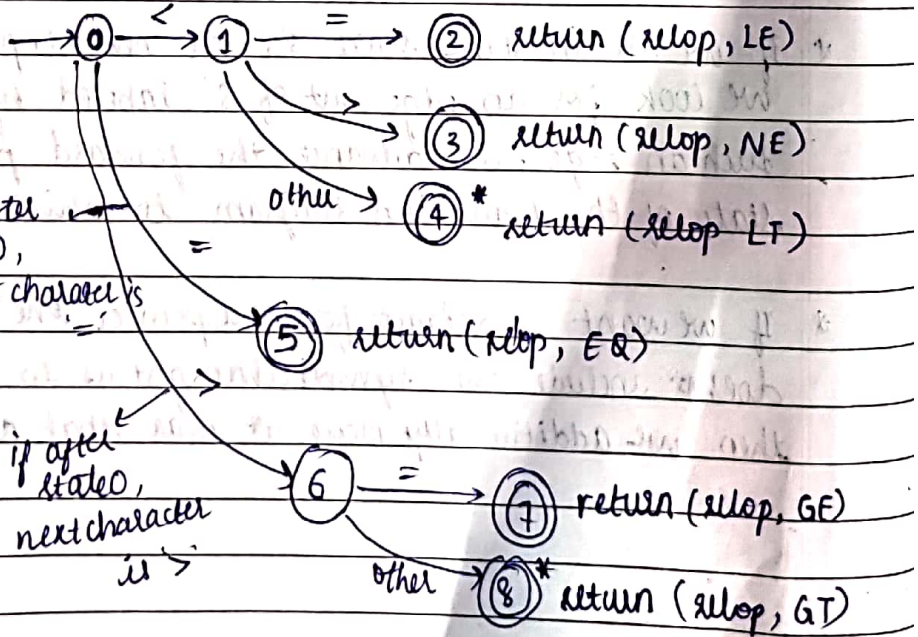
* we begin at state 0, the start state
* If we see < as the first input symbol, then among the lexemes that match the pattern for relop, we can be only looking at <, <> or <=.

∴ We go to state 1 & look at next character.

- If it is =, then we recognize lexeme <=, enter state 2. and return the token relop with attribute LE.

- If in state 1, the next character is >, then instead we have a lexeme <> and enter state 3 and return indication that not-equals operator has been found.

- on any other character, the lexeme is <, & we enter state 4 to return that information state 4 is a* to indicate, that we must retract the input one position.



classmate
Date
Page

UNIT-3

context-free grammars: (CFG)

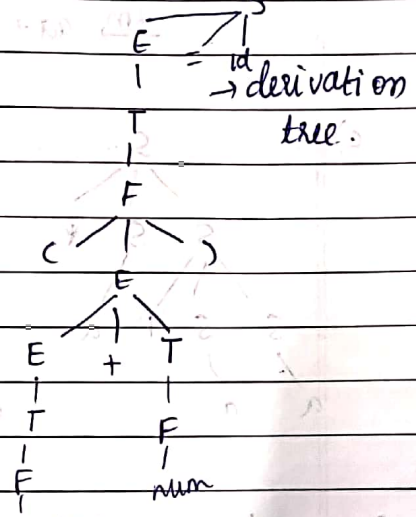
$G = (V, T, P, S)$

V = set of non-terminals
T = terminals/tokens
P = set of grammar production
S = start symbol.

$S \rightarrow id = E$
$E \rightarrow E+T \mid E-T \mid T$
$T \rightarrow T*F \mid T/F \mid T\%F \mid F$
$F \rightarrow (E) \mid id \mid num$

non-terminal → terminals

$S = 3 + 4$



→ derivation tree.

Leftmost derivation → expansion
Rightmost derivation → reduction

stmt → E;
non terminal
| if (E) stmt
| for (optexpr, optexpr, optexpr) stmt
| while (E) stmt
| other

optexpr → ε | E
non-terminal

Grammars for some statements in C and Java

① consider the CFG
$S \rightarrow SS+ \mid SS* \mid a$ and the string $aa+a*$

a) Give a LMD for the string
b) Give a RMD for the string
c) Give a parse / derivation tree for the string.

SS+

SS+

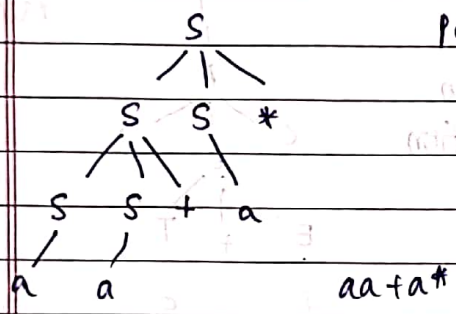
SS*

classmate Date Page

1. a) $S \xrightarrow{lm} SS^*$ ⇒ LMD
$\xrightarrow{lm} SS+S^*$
$\xrightarrow{lm} aS+S^*$
$\xrightarrow{lm} aa+S^*$
$\xrightarrow{lm} aa+a^*$

b) $S \xrightarrow{rm} SS^*$
$\xrightarrow{rm} Sa^*$
$\xrightarrow{rm} aSS+a^*$
$\xrightarrow{rm} Sa+a^*$
$\xrightarrow{rm} aa+a^*$

c) Parse tree



aa+a*

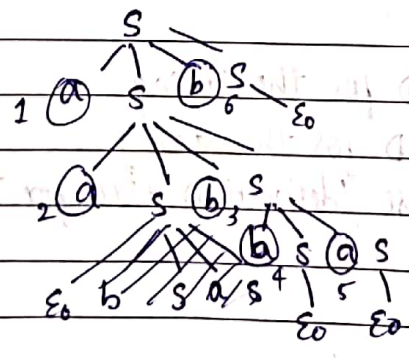
2. Write CFG for accepting strings 0's and 1's, where equal number of 0's followed by same number of 1's.

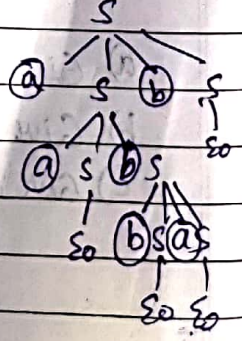
$S \rightarrow 0S1 \mid 01$

0 0 S 1 1
0 0 0 S 1 1 1
0 0 1 1

3. $S \rightarrow aSbS \mid bSaS \mid \varepsilon_0$ with string aabbab. Draw parse tree.

0 S 1
0 0 1 1





4. $E \to E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid id \mid num$

For string id + id * id. check the CFG is ambiguous

$E \xrightarrow{lm} E*E$
$\xrightarrow{lm} E+E*E$
$\xrightarrow{lm} id+E*E$
$\xrightarrow{lm} id+id*E$
$\xrightarrow{lm} id+id*id$

$E \xrightarrow{rm} E+E$
$\xrightarrow{rm} E+E*E$
$\xrightarrow{rm} E+E*id$
$\xrightarrow{rm} E+id*id$
$\xrightarrow{rm} id+id*id$

ambiguous because it has multiple LMDs & RMDs.

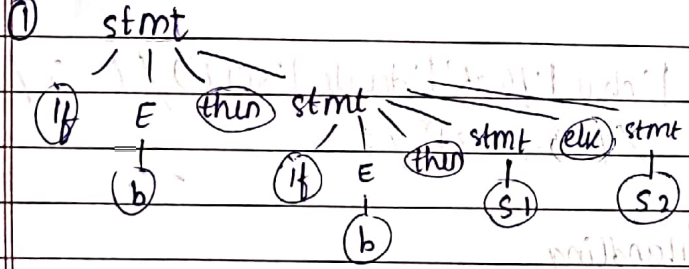
Eliminating Ambiguity in Dangling Else grammar:

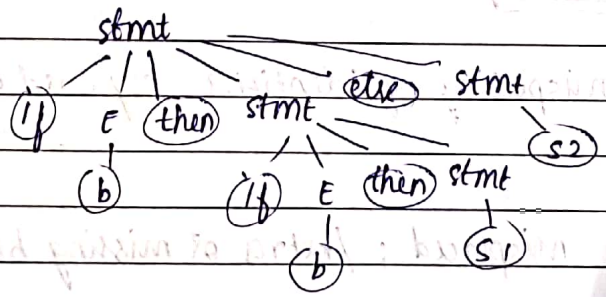
stmt → if E then stmt
| if E then stmt else stmt
| other

E → b

I/p :- if b then if b then s_1 else s_2

dangling else, because parser doesn't know if this 'else' belongs to the first 'if' or the latter 'if'.

① 

② 

it is ambiguous.

So, we modify the definition of 'stmt' to fix the error of dangling else

Modified:

stmt → matched_stmt
| open_stmt

matched_stmt → if E then matched_stmt else matched_stmt.
| other

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

E → b

every if is attached to an else.

* Write a CFG to accept a simple type declaration in C

type declaration → <type specifier> <identifier>

type specifier → 'int' | 'char' | 'float' | 'double' | 'void'

identifier1 → <identifier> | '*' <identifier>

identifier → ([a-z][A-Z] | [_][0-9][a-z][A-Z] | [a-z][0-9][A-Z]) id

identifier → id, identifier | id.

D → T L
T → 'int' | 'char' | 'float'
L → L, id | id.

int a;
int *a;

* Regular expression:

('int' | 'char' | 'float' | 'double' | 'void') / ( ) / ('*') / [_a-zA-Z]* / ;

Syntax Error Handling

* Lexical errors - misspelling of identifier, keyword or operators

* Syntactic errors - misplaced ; / extra or missing braces

* Semantic errors - mismatches b/w operators and operand typechecking.

* Logical errors : incorrect reasoning.

Error Recovery Strategy:

Panic Mode:- If discovered any error, parser discards I/P symbol one at a time until one of a designated set of synchronizing tokens is found.
↳ (;,(),...)

Phase level :- local correction

Error production:- G for possible common error routing

Global correction: General deterministic algorithm

Syntax Analysis/parsing

Start (S) ← String

Top down approach

predictive parsing (PP)

Recursive Descent parsing (RDP)

String (S) → Bottom up approach

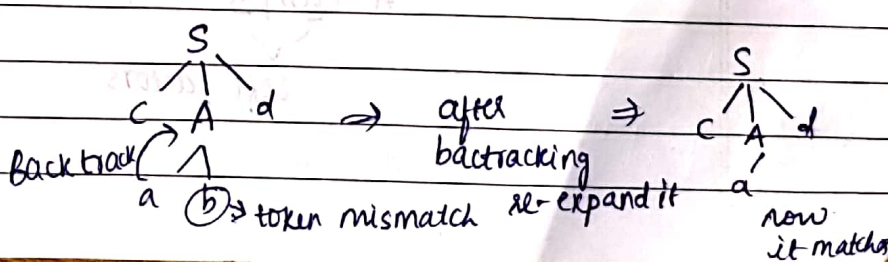
LR parsers

SLR CLR LALR

* In RDP → backtracking algorithms are used when a mismatched token is found but difficult to find an efficiend bt algo

* uses CFG as input

Ex: S → cAd
A → ab | a
i/p :- cad $



* PP is efficient parser but difficult in terms of system like, a lot of bg work must be done before implementation.

* PP accepts CFG in the form of LL(1) → Look ahead character can be predicted.
  - L → left to right scanning
  - L → LMD

Bottom up approach → CFG is accepted in the form of LR
- L → left to right scanning
- R → RMD in reverse

string → reduced to (S)

* Common problems for predictive parsers:

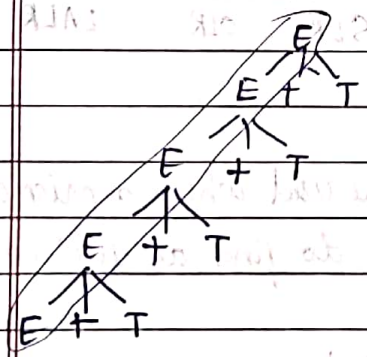
① Left recursion → we have to eliminate it.

let $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

a+a+a+a



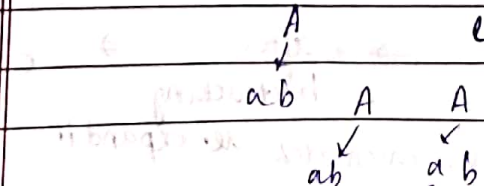
→ left recursion.

② Left factors → should it be eliminated, do it

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

left factors



Eliminate Left Recursion

$\alpha, \beta, \gamma \rightarrow$ can be set of any terminal, ε_0, non-terminals

* check if the given CFG is of the form

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

if it is, then it is a left recursive

so, by retaining the original meaning, we eliminate the "direct left recursion" as from the CFG.

$$\Rightarrow A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon_0 \quad \text{(changed to right recursive)}$$

Left Factoring:

If G is in the form $A \to \alpha\beta_1 | \alpha\beta_2$ replace with

$A \to \alpha A'$

where $A' \to \beta_1 | \beta_2$

example: $S \to 0S1 | 01$

$S \to 0S'$

$S' \to S'|1$

* $S \to iEtS | iEtSeS | e$

$E \to id$

$S \to iEtSS' | e$

$S' \to \varepsilon_0 | eS$

$E \to id$

G:- $S \to 0S'$

$S' \to S1 | 1$

$S \xrightarrow{\text{Lmd}} 0S' \xrightarrow{\text{Lmd}} 0S1$

First(α)

First(S) = {0}

First(S') = {0,1}

Follow(S)

Follow(S) = {1,$}

Follow(S') = {$,1}

First(α)

NT	First
E	{(, id}
E'	{+, ε_0}
T	{(, id}
T'	{*, ε_0}
F	{(, id}

$E \to TE'$

$E' \to +TE' | \varepsilon_0$

$T \to FT'$

$T' \to *FT' | \varepsilon_0$

$F \to (E) | id$

Algorithm:

Compute First(α)

1. First(terminal) = {terminal}
2. $A \to \alpha$; $\alpha \overset{*}{\Rightarrow} \varepsilon_0$ (one or more derivations)

then, First(A) = {ε_0}

3. $A \to BCD$ where $B, C, D \overset{*}{\Rightarrow} \varepsilon_0$ only then $A \to \varepsilon_0$.

$B \to a \mid \varepsilon_0$

$C \to c \mid \varepsilon_0$

$D \to d \mid \varepsilon_0$

First (A) = {a, c, d, ε}

First (B) = a, ε

First (C) = c, ε

First (D) = d, ε

Compute Follow (S):

Procedure

1. place $ in follow (S), if S is the start symbol
2. If G is in form $A \to \alpha B \beta$, where First(β) doesn't contain ε_0, then add everything in First(β) except ε_0 to Follow(B)
3. If G is in form $A \to \alpha B$ or $A \to \alpha B \beta$, where First(β) contains ε_0, then add everything in Follow(A) to Follow(B)

Q1. $E \to TE'$

$E' \to +TE' \mid \varepsilon_0$

$T \to FT'$

$T' \to *FT' \mid \varepsilon_0$

$F \to (E) \mid id.$

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

4 $E \to TE'$ $(A \to \alpha B \beta)$ or $E \to TE'$ $A \to \alpha B$

$\alpha \to \varepsilon_0$, $\beta \to E'$

First (E') = {+, ε}

So, add + to follow (T)

① F(E) ⇒ F(E')

② F(T) ⇒ First(E') − ε

③ F(E) ⇒ F(T)

④ F(T) ⇒ First(E') − ε

⑤ F(E') ⇒ F(T)

* $E' \to +\ TE'$ (or) $E' \to +TE'$, $A \to \alpha B$ → can ignore cuz $A = B = E'$ (Right recursive grammar)
$(A \to \alpha B\beta)$

2nd step from procedure: $First(E') = \{+, \varepsilon_0\}$
so, $Follow(T) = \{+\}$

now for 3rd step in the procedure
$E' \to +T E'$
if $E' \Rightarrow \varepsilon_0$
so, $E' \to +T$
$\therefore$ follow $(E') \Rightarrow$ follow(T)

* $T \to F T'$ $First(T') = \{*, \varepsilon_0\}$ (or) $T \to F T'$ $(A \to \alpha B)$
$(A \to \alpha B \beta)$
2nd step so, $Follow(F) = First(T') - \varepsilon_0 = \{*\}$
so, follow(T) → Follow(T')
3rd step so, $Follow(T) \Rightarrow Follow(F)$

* $T' \to *FT'$ $first(T') = \{*, \varepsilon_0\}$ (or) $T' \to *FT'$ $(A \to \alpha B)$
$(A \to \alpha B \beta)$
follow$(F) \Leftarrow$ first$(T') - \varepsilon_0$. x ignore

* $F \to (E)$
$(A \to \alpha B \beta)$

Q2.* $S \to (L) \mid a$ compute first & follow
$L \to ,S \mid b$

	first	follow
S	{(, a}	{$,)}
L	{, b}	{)}

$S \to (L)$
$A \to \alpha B \beta$

Q3: a) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon_0$

$B \rightarrow \varepsilon_0$

	first	follow
S	a, b	{$}
A	ε_0	{a, b}
B	ε_0	{b, a}

1. Is the grammar suitable to be passed using predictive parsing method? Modify the G if needed.

$S \rightarrow aAbA \mid aAbc \mid ScA$

$A \rightarrow aAbab \mid b$

Left factoring $\Rightarrow$ $S \rightarrow \underbrace{aAb}_{\alpha} A \mid \underbrace{aAb}_{\alpha} c \mid ScA$

$S \rightarrow aAbS' \mid ScA$

$S' \rightarrow A \mid c$

Left Recursive $S \rightarrow \underbrace{aAbS'}_{\beta} \mid \underbrace{ScA}_{\alpha}$

$S' \rightarrow A \mid c$

$S \rightarrow aAbS'S'' \mid aAbS'S''$

$S'' \rightarrow cAS'' \mid \varepsilon_0$

Final Ans $\Rightarrow$ $S \rightarrow aAbS'S''$

$S' \rightarrow A \mid c$

$S'' \rightarrow cAS'' \mid \varepsilon_0$

$A \rightarrow aAb \mid ab \mid b$

2. Compute First and Follow of all non-terminals

a) $S \rightarrow ABe$

$A \rightarrow dB \mid aS \mid c$

$B \rightarrow AS \mid b$

First (S) → {d, a, c}
First (A) → {d, a, c}
First (B) → {d, a, c, b}

Follow (S) = {$, a, b, c, d, e}
Follow (A) = {a, b, c, d}
Follow (B) = {e, a, b, c, d}

$S \rightarrow \underset{\alpha}{A} \underset{B}{B} \underset{\beta}{e}$ $\quad G \rightarrow \alpha B \beta$

first(B) to Follow(A)

d, a, c, b →

$A \rightarrow \underset{\alpha}{d} \underset{\beta}{B}$ $\quad A \rightarrow \underset{\alpha}{a} \underset{B}{S}$

follow (A) → follow (B)

B → AS

b) $S \rightarrow asbs \mid bsas \mid \varepsilon_0$

$S \rightarrow \underset{\alpha}{a} \underset{B}{s} \underset{\beta}{bs}$ $\quad \alpha B \beta$

First (S) = {a, b, ε_0}
Follow (S) = {$, b, a}

first (β) = b ↓ follow (S)

c) $S \rightarrow +SS \mid *SS \mid a$

First (S) = {+, *, a}
Follow (S) = {$, +, *, a}

$S \rightarrow \underset{\alpha}{+} \underset{B}{S} \underset{\beta}{S}$

first (S) → follow (B)

3. Check whether the G is suitable for predictive parser. Modify the G if required. Compute FIRST and FOLLOW of non-terminals.

a) $S \rightarrow \underset{\alpha}{S(S)S} \mid \underset{\beta}{\varepsilon_0}$

First (S) = {ε_0, (}
Follow (S) = {$,), (}

$S \rightarrow S'$

First (S') = {(, ε_0}

$S' \rightarrow (S)S S' \mid \varepsilon_0$

Follow (S') = {(, $,)}

b) $S \to S+S \mid S+ \mid (S) \mid S* \mid a$

$S \to S \$ S' \mid (S) \mid *a$

$S' \to +S \mid + \mid *$

$S \to S S' \mid (S) \mid a$

$S \to (S) S'' \mid a S''$

$S'' \to S' S'' \mid \varepsilon$

c) $S \to CC$

$C \to \alpha C \mid d$

d) $S \to A$

$A \to BC \mid DBC$

$B \to Bb \mid \varepsilon$

$C \to C \mid \varepsilon$

$D \to a \mid d$

* $S \to CC$

$C \to cC \mid d$

cdd$

predictive parsing table

$A \to \alpha$

compute First(α)

M[X, a]

NT Termin.

	Stack	Input	Action
$S \to CC$	S $	cdd$	
M[S, c]	cC$	cdd$	
M[S, d]	cC$	cdd$	→ pop
$C \to cC \mid d$	Cc$	dd$	
M[C, c] M[C, d]	d $	dc$	dd$ → pop
S: $S \to CC$	$S \to CC$	C$	d$
C: $C \to cC$	$C \to d$	d$	d$ → pop
		$	$

end of parsing.

* $E \to TE'$

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

$T \to FT'$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

NT/ip	+	*
E		
E'	$E' \to +TE'$	
T		
T'	$T' \to \varepsilon$	$T' \to *$
F		

stack	I
E $	id
TE' $	id
FT'E' $	id
id T'E' $	id
T'E' $	
E' $	
+TE' $	
TE' $	
FT'E' $	
id T'E' $	
T'E' $	
*FT'E' $	
FT'E' $	

ε means pop out

$E \to TE'$
$E' \to +TE' \mid \varepsilon_0$
$T \to FT'$
$T \to FT'$
$T' \to *FT' \mid \varepsilon_0$
$F \to (E) \mid id$

if $A \to \alpha$
compute first(α)
if first(α) contain ε_0,
calculate follow(α)
& include it.

NT/ip	+	*	(	id	)	$
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to \varepsilon_0$	$E' \to \varepsilon_0$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to \varepsilon_0$	$T' \to *FT'$			$T' \to \varepsilon_0$	$T' \to \varepsilon_0$
F			$F \to (E)$	$F \to id$		

Stack	Input	
$E\ \$$	$id+id*id\$$	
$TE'\ \$$	$id+id*id\$$	
$FT'E'\ \$$	$id+id*id\$$	
$id\ T'E'\ \$$	$id+id*id\$$	→ pop
$T'E'\ \$$	$+id*id\$$	
$+TE'\ \$$	$+id*id\$$	→ pop
$TE'\ \$$	$id*id\$$	
$FT'E'\ \$$	$id*id\$$	
$id\ T'E'\ \$$	$id*id\$$	→ pop
$T'E'\ \$$	$*id\$$	
$*FT'E'\ \$$	$*id\$$	→ pop
$FT'E'\ \$$	$id\$$	→ pop
$id\ T'E'\ \$$		

ε_0 means pop out

pop T: $E'\ \$$ | $\$$
pop E: $\$$ | $\$$
pop: $\$$ | $\$$

Construction of predictive parsing table:

* For each production $A \to \alpha$ in grammar do the following:

  i) For each terminal a in First(α) add $A \to \alpha$ in M[A, α]

  ii) If ε_0 is in First(α) then for each terminal b in Follow(A) add $A \to \alpha$ to M[A, b]. If ε is in First(α) & \$ is in Follow(A), add $A \to \alpha$ to M[A, \$] as well.

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

* $S \to iEtSS'$ → can't do bcoz its not in form of LL(1)
  $S' \to eS \mid \varepsilon_0$
  $E \to id$

NT \ i/p	i	t	e	id	$
S	$S \to iEtSS'$				
S'			$S' \to eS$		
E					

Recursive descent parsing (RDP):

- consists of a set of procedures

* $S \to +SS \mid *SS \mid a$

input $+*aaa\$$ construct the parse table.

NT | i/p	+	*	a	$
S	$S \to +SS$	$S \to *SS$	$S \to a$	

$A \to \alpha$

$S \to +SS$

α

First $(+SS) = +$

$M[S, +]$

stack	input	
$S\$$	$+*aaa\$$	
$+SS\$$	$+*aaa\$$	→ pop
$SS\$$	$*aaa\$$	
$*SSS\$$	$*aaa\$$	→ pop
$SSS\$$	$aaa\$$	
$aSS\$$	$aaa\$$	→ pop
$SS\$$	$aa\$$	
$aS\$$	$aa\$$	→ pop
$S\$$	$a\$$	
$a\$$	$a\$$	→ pop
$\$$	$\$$	→ passed

Predictive parsing algorithm:

* set ip point to the first symbol of w;
set X to top stack symbol;
While (X <> $) { /* stack is not empty */
if (X is a) pop the stack & 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 → Y1 Y2 ... Yk) {
output the production X → Y1 Y2 ... Yk;
pop the stack;
push Yk, ... Y2, Y1 onto the stack with Y1 on top;
}
set X to the top stack symbol;
}.

If M[A,α] = blank, skip input symbol;
If M[A,α] = synch (;), pop NT on stack top
If token on stack top mismatches input symbol, pop stack.

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) iff and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

- For no terminal a do both α and β derive strings beginning with a.
- At most one of α or β both derive empty string.
- If $\alpha \Rightarrow \varepsilon$, then β does not derive any string beginning with a terminal in Follow(A).

i) $S \to S(S)S \mid \varepsilon$

$S \to S'$

$S' \to (S)SS' \mid \varepsilon$

↓ $A \to \alpha \mid \beta$

Follow(A) doesn't contain terminal with the first(α).

$S \to S'$:- first(S') = (, ε

follow(S) = (,), $

S → S' / A → x

First(S') = (, ε

$S' \to (S)SS'$ first((S)SS') ⇒ (

follow(S') ⇒ $,), (

Repeats terminals so, not LL(1) grammar.

NT \ i/p	)	(	$
S	$S \to S'$	$S \to S'$	$S \to S'$
S'	$S' \to \varepsilon$	$S' \to (S)SS'$ / $S' \to \varepsilon$	$S' \to \varepsilon$

M[S, (]

M[S, ε]

$S' \to (S)SS'$ M[S', (]

$S' \to \varepsilon$

ii) $S \to iEtSS'$

$S' \to eS \mid \varepsilon$

$E \to id$

$S \to iEtSS'$

first(iEtSS') = i

Follow(S) = e, $

$S' \to eS \mid \varepsilon$

First(eS) = e

Follow(S') = $, e

classmate
Date
Page

Bottom-up parsing:

(LR)

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

Input: ab
: a B
A B
S

Handle pruning: →
terminals involved in reduction in RMD is called a handle.

* $S \to 0S1 | 01 | 0$
i/p :- 00 11 $

Identify the handles

00 11 $
0 0 : X
0 0 1 : $S \to 01$ so shift
0 S 1 : 0S1
S

Conflicts during shift reduce parsing.

① Shift reduce → $S \to aAa | a$
② Reduce Reduce
I/p: aba
it can shift as well as reduce

$S \to aAa | aB$
$A \to b$
$B \to b$
i/p : aba
b → can be reduced to either A/B

$S \rightarrow +SS \mid *SS \mid a$

i/p :- $+ * aaa\$$

+	→	S	S
*	→	S	S
a	→	S	
a	→	S	

LR parsers

- L-R scanning of input
- RMD in reverse

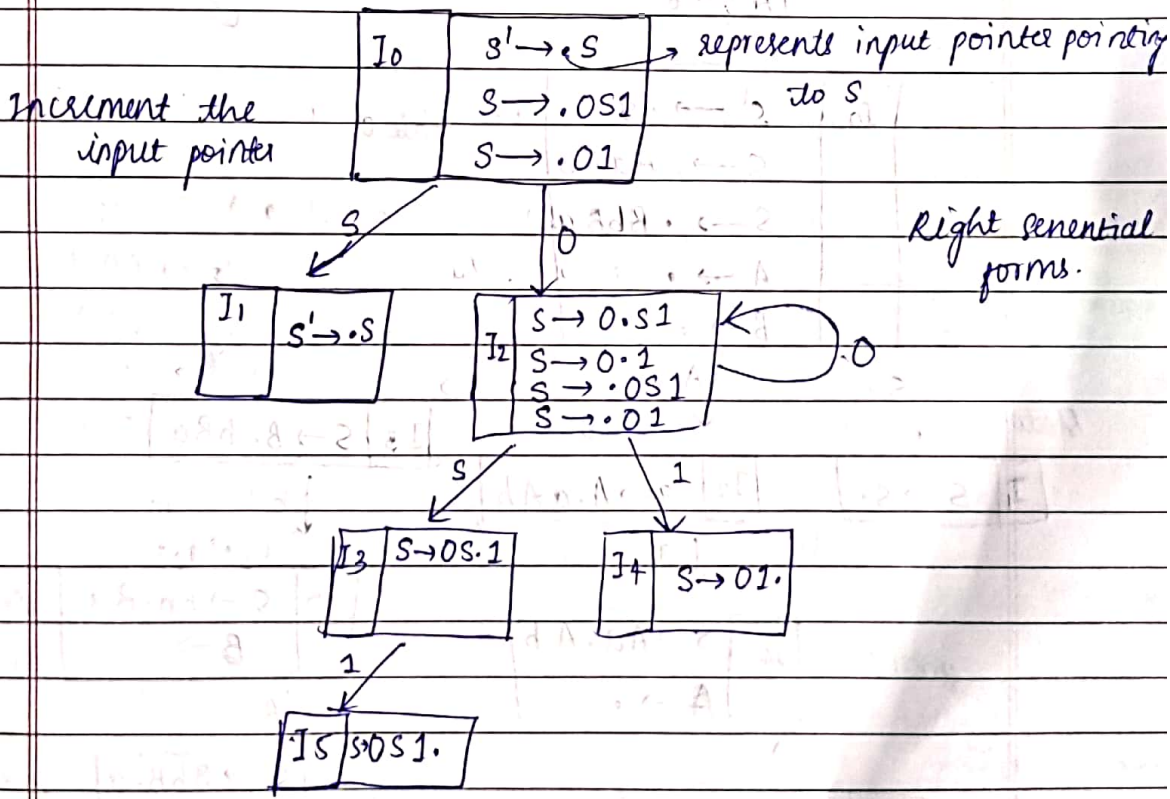
Simple LR parsers

- SLR
  - LR(0)
  - SLR(1)

LR(0) parsers

Ex: $S \rightarrow 0S1 \mid 01$

initial item :- Consider a new non-terminal that derives the start symbol.



closure of item sets

1. Initially add every item in I to closure(I).

2. If $A \to \alpha . B\beta$ is in closure(I) and $B \to \gamma$ is a production, then add $B \to .\gamma$ to closure(I), if it is not already there

Kernel items:- initial item $S' \to .S$, and all items whose dots are not at left end

Non-kernel items:- all items with their dots at left end except $S' \to .S$

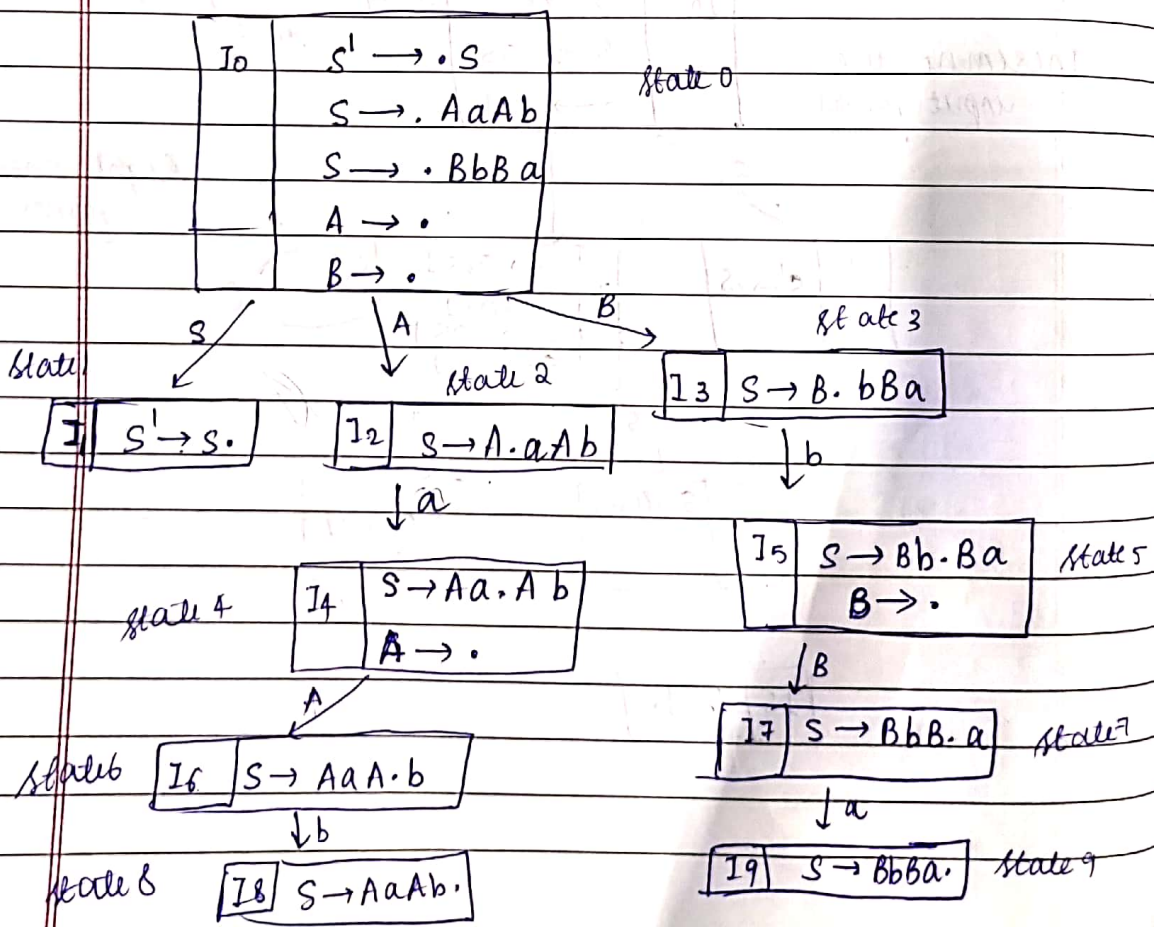
Example:

$S \to AaAb \mid BbBa$

$B \to \varepsilon$

$A \to \varepsilon$

Construction table



4 actions

① Shift ③ Accept
② Reduce ④ Error

classmate Date Page

SLR parse table / LR(0) parse table

States	ACTION ↓ (Terminals)			GOTO ↓ (Non terminals)			in original CFG
	a	b	$	S	A	B	
0	r3/r4	r3/r4		1	2	3	GOTO(I0,S) = I1
1			accept				
2	s4						GOTO(I0,A) = I2
3		s5					GOTO(I0,B) = I3
4	r3	r3			6		GOTO(I4,A) = I6
5	r4	r4				7	GOTO(I5,B) = I7
6		s8					
7	s9						
8			r1				
9			r2.				

Shift:
I2, a = I4
shift a & move to I4
⇓
s4

Reduce:
Reduce A → ε
Reduce B → ε

↓

I0: ① S → AaAb
② S → BbBa
③ A → ε
④ B → ε → this has . at right end

so, compute follow of A & B

	Follow
A	a, b
B	b, a

⇓

I0 { * A → . reduce ③ grammar
0, a → r3
0, b → r3 → grammar number
(follows computed for A)
B → . reduce ④ grammar
0, a } ⇒ r4

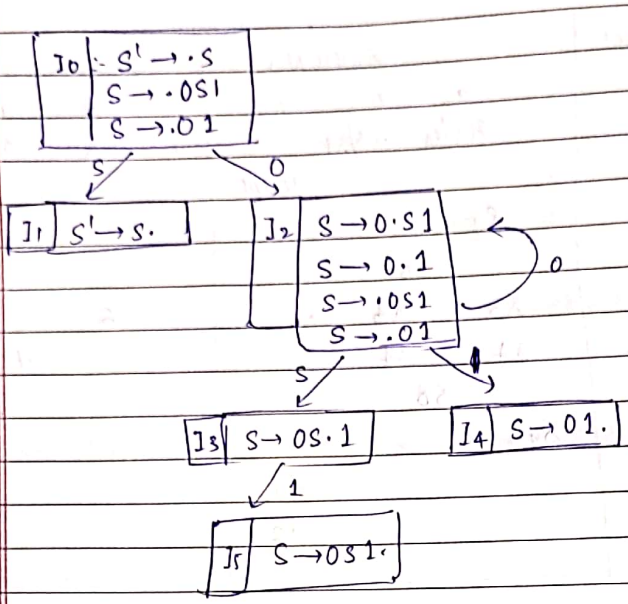
next I4:
A → .
follow of A = {a, b}
4, a } ⇒ r3
4, b }
reduce 3rd grammar

next I5
B → .
follow of B = {a, b}
5, a } ⇒ r4
5, b }

next I8
S → AaAb. reduce to ① grammar
follow(S) = $
8, $ ⇒ r1

next I9 reduce to ② grammar
S → BbBa. 9, $ = r2

4. $S \to 0S1 \mid 01$



$I_0, 0 = I_2$

Parse table.

States	Actions			GOTO
	0	1	$	S
0	S_2			1
1			accept	
2	S_2	S_4		3
3		S_5		
4	R_2	R_2		
5	R_1	R_1		

① $S \to 0S1$
② $S \to 01$

reduce

I_5:- $S \to 0S1.$

follow of $S = \{\$, 1\}$

5, $ ⇒ R_1

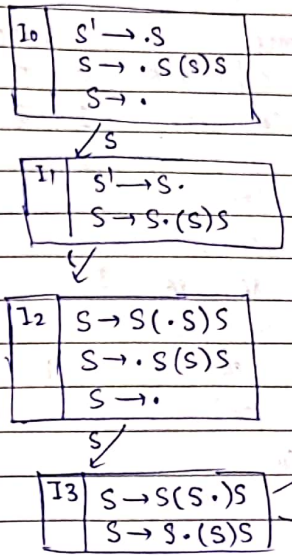
5, 1 ⇒ R_1

I_4: $S \to 01.$

4, 1 = R_2

4, $, R_2

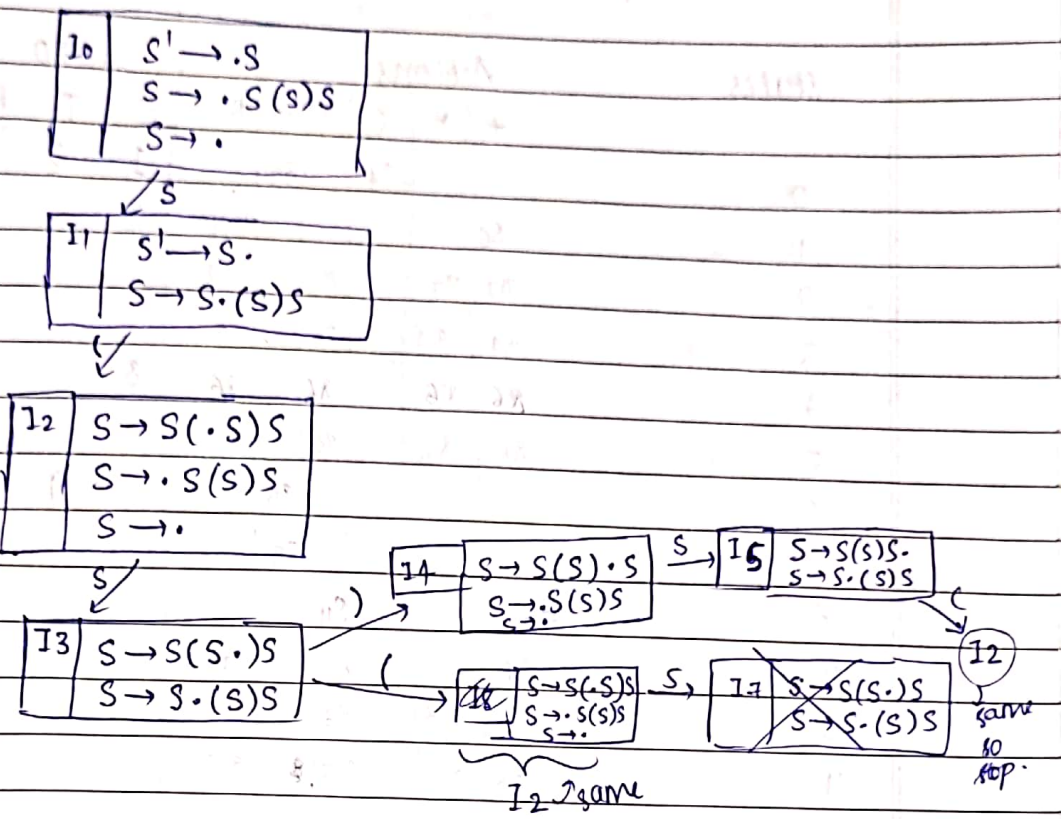
4. $S \to S(S)S \mid \varepsilon$



Parse table

States	Act
	(
0	
1	S_2
2	R_3
3	S_2
4	R_1
5	S_3

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



Parse table

① $S \rightarrow S(S)S$
② $S \rightarrow \epsilon$

State	Actions			GOTO
	(	)	$	S
0				1
1	S_2			
2	r_2	r_2	r_2	3
3	S_2	S_4		
4	r_2	r_2	r_2	5
5	S_2/r_1	r_1	r_1	

I_1, (⇒ I_2

I_2: $S \rightarrow \epsilon$.
follow of $S \Rightarrow \{\$, (,)\}$
2, $ } ⇒ r2
2,)
2, (

I_5: $S \rightarrow S(S)S.$
5, $ } ⇒ r1
5,)
5, (

I_4 : $S \rightarrow \epsilon$.
4, $ } ⇒ r2
4,)
4, (

* construct the parse table:

States	Actions						GOTO		
	+	*	(	)	id	$	E	T	F
0			S4		S5		1	2	3
1	S6					accept			
2	r2	S7		r2		r2			
3	r4	r4		r4		r4			
4							8		
5	r6	r6		r6		r6			
6					S5			9	
7					S5				10
8	S6			S11					
9	r1			r1		r1			
10	r3	r3		r3		r3			
11	r5	r5		r5		r5			

grammar :-
① E → E+T
② E → T
③ T → T*F
④ T → F
⑤ F → (E)
⑥ F → id

I2 : E → T.
Follow of E = {), $, + }
I2,) ⇒
I2, $ ⇒
I2, + ⇒
} r2

I5 : F → id.
follow of F = { *, +,), $ }
I5, *
I5, +
I5,)
I5, $
} ⇒ r6

I3 : T → F.
Follow of T = { *, +,), $ }
I3, *
I3, +
I3,)
I3, $
} ⇒ r4

I10 : T → T * F.
I10, *, +,), $ } ⇒ r3

I11 : F → (E).
I11, *
I11, +
I11,)
I11, $
} ⇒ r5

I9 : E → E+T.
I9,)
I9, $
I9, +
} ⇒ r1

for this parse table:

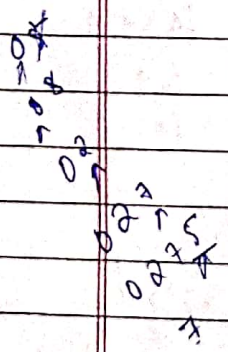
I/p: id * id + id.

0 & id has 5 so add 5 to stack

as id is reduced to F then pop the '5' from stack

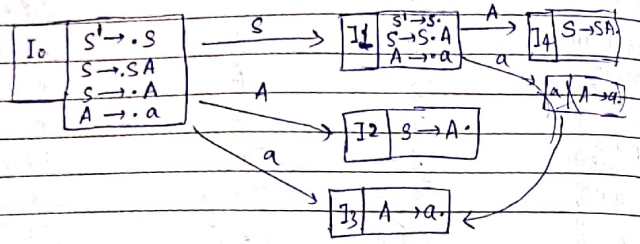
next

Stack	Symbol	Input
0		id * id + id $
05	id	* id + id $
03	F	* id + id $
02	T	* id + id $
027	T *	id + id $
0275	T * id	+ id $
0 2 7 10	T * F	+ id $
0 2	T	+ id $
0 1	E	+ id $
016	E +	id $
0165	E + id	$
	F	
0163	E + F	$
0169	E + T	$
0 1	E	$
	accept	



* construct LR(0) / SLR(0) parse table for the following G

① $S \to SA \mid A$
$A \to a$



States	Actions			GOTO	
	a	A	$	S	A
0	s3			1	2
1	s3		accept		4
2	r2	r2			
3	r3	r3			
4	r1	r1			

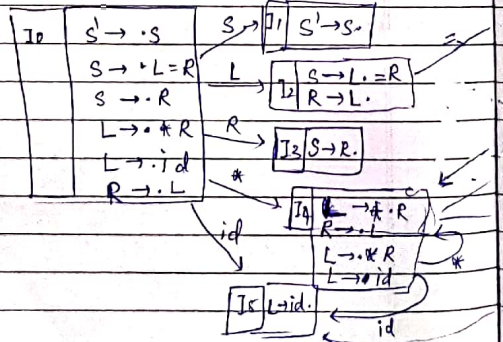
① S → SA
② S → A
③ A → a

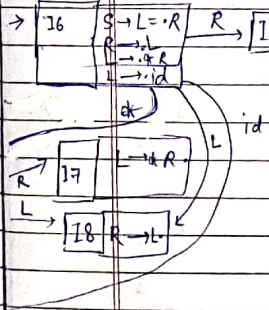
I2: S → A.
follow(S) = {$, a}
I2, $ } ⇒ r2
I2, a }

I3: A → a.
follow(A) = {$, a}
I3, $ } ⇒ r3
I3, a }

I4: S → SA.
I4, $ } ⇒ r1
I4, a }

② $S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$



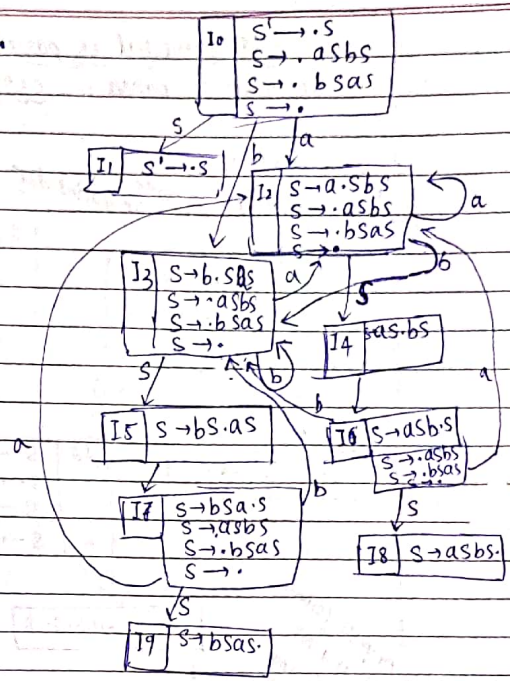


① S → L=R
② S → R
③ L → *R
④ L → id
⑤ R → L

I2: R → L.
follow(S) = $
follow(R) {$, =}
follow(L) = {$, =}
I2, $ } ⇒ r5
I2, = }

State	Actions				GOTO		
	=	*	id	$	S	L	R
0					1	2	3
1		s4	s5	accept			
2	s6/r5			r5			
3				r2			
4		s4	s5				7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

3. $S \to aSbs \mid bsas \mid a$

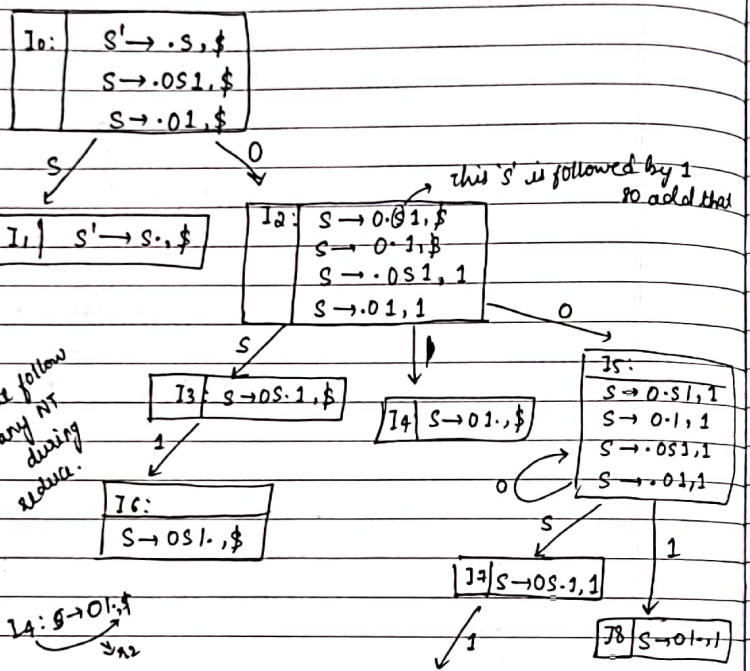


More powerful LR parser

canonical LR parser :- CLR(1)

$S \to OS1 \mid 01$

this $ is followed by $ so, write $ in initial dⁿ



No need to compute follow of any NT during reduce.

① $S \to OS1$

② $S \to 01$

I4: $S \to 01.,\$$ → r2

parse table

States	Action			GOTO
	0	1	$	S
0	S2			1
1			accept	
2	S5	S4		3
3		S6		
4	S5	S8	r2	
5				7
6			r1	
7		S9		
8			r2	
9				

use this rule

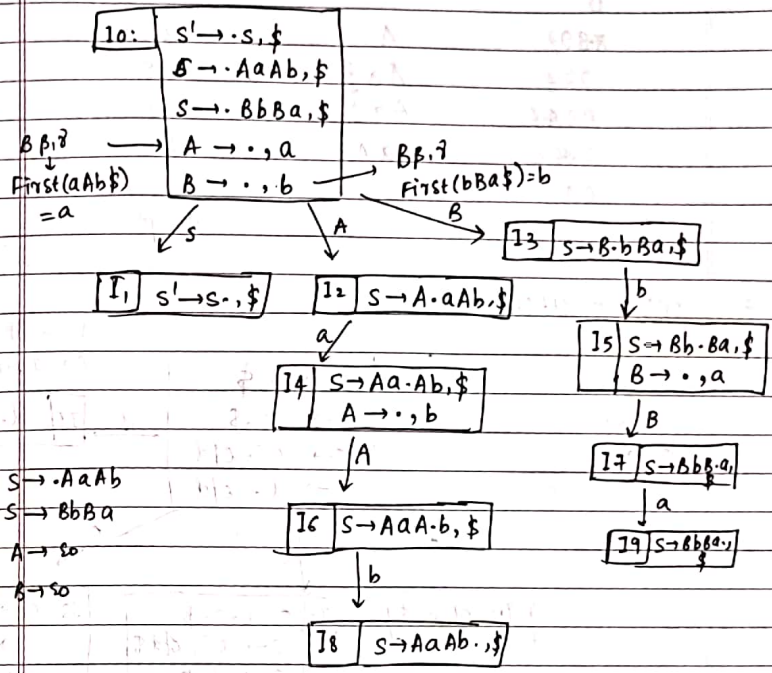
$A \to \alpha \cdot B\beta, \gamma\}$

$B \to \cdot a$, First($\beta\gamma$)

* $S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$



① $S \to AaAb$

② $S \to BbBa$

③ $A \to \varepsilon$

④ $B \to \varepsilon$

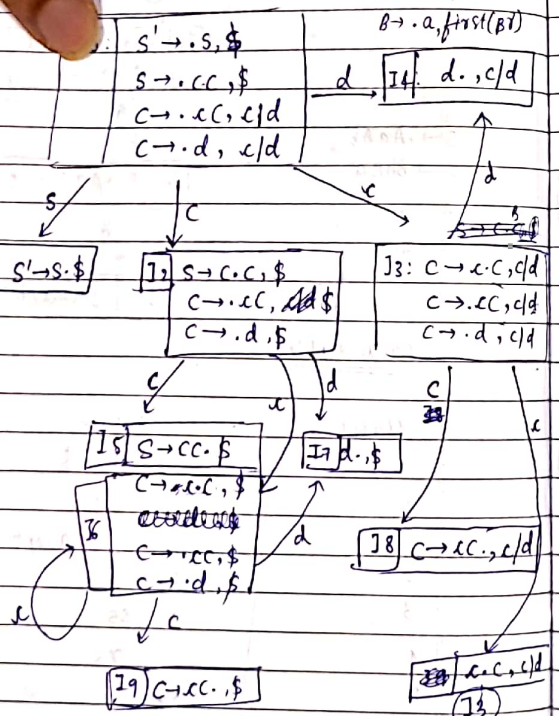
parse table:

States	Action			GOTO		
	a	b	$	S	A	B
0	r3	r4		1	2	3
1			accept			
2	S4					
3		S5				
4		r3				6
5	r4					7
6		S8				
7	S9					
8			r1			
9			r2			

	Symbol	input
		a b $
02	A	a b $
024	Aa	b $
0246	AaA	b $
8	AaAb	$
	S	
		accept

* construct CLR( ) table for

i) S → CC
C → cC
C → d



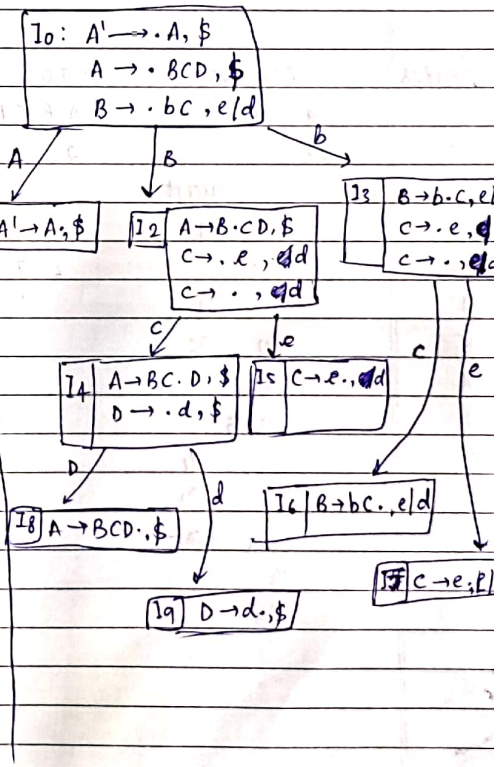
States	ACTIONS			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

CD$
C → c
D → d

parse table

States	ACTIONS				GOTO			
	b	e	d	$	A	B	C	D
0	s3				1		2	
1				accept				
2	s5/r4		r4				4	
3	s5/r4		r4				6	
4		s9					8	
5	r3		r3					
6	r2		r2					
7				r1				
8				r5				
9								

ii) A → BCD
B → bC
C → e | ε
D → d

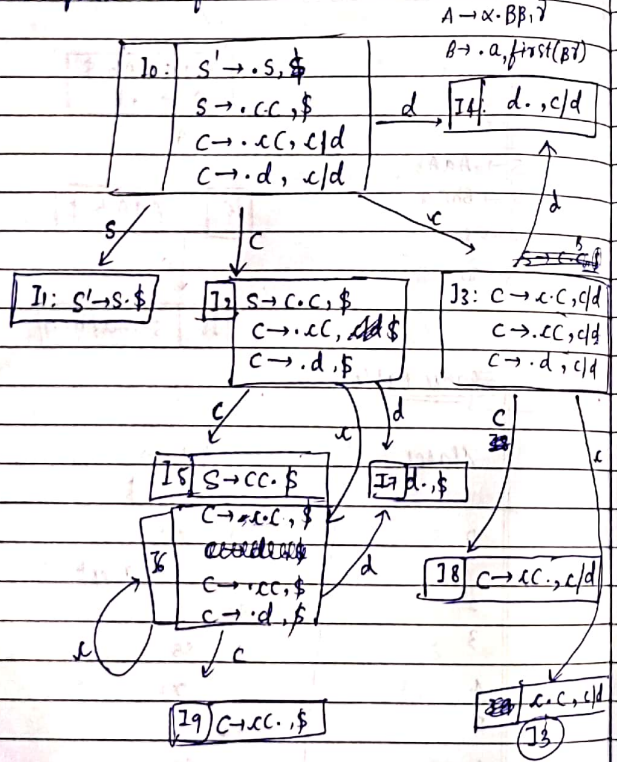


Input: ab$

stack	symbol	input
0		ab$
02	A	a b$
024	Aa	b $
0246	AaA	b $
02468	AaAb	$
01	S	
		accept

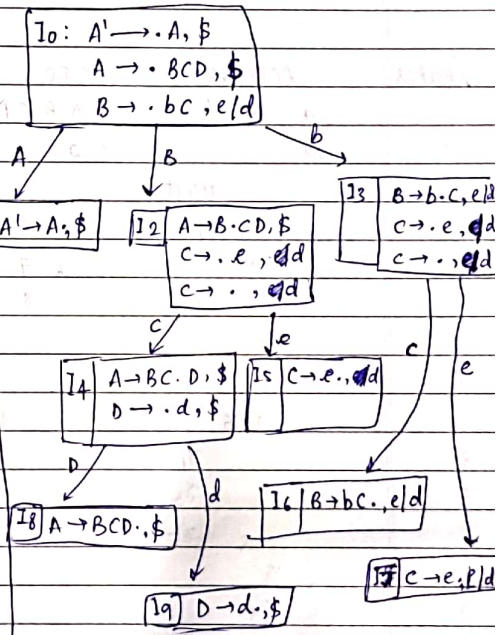
* construct CLR(1) parse table for

i) S → CC
C → cC
C → d



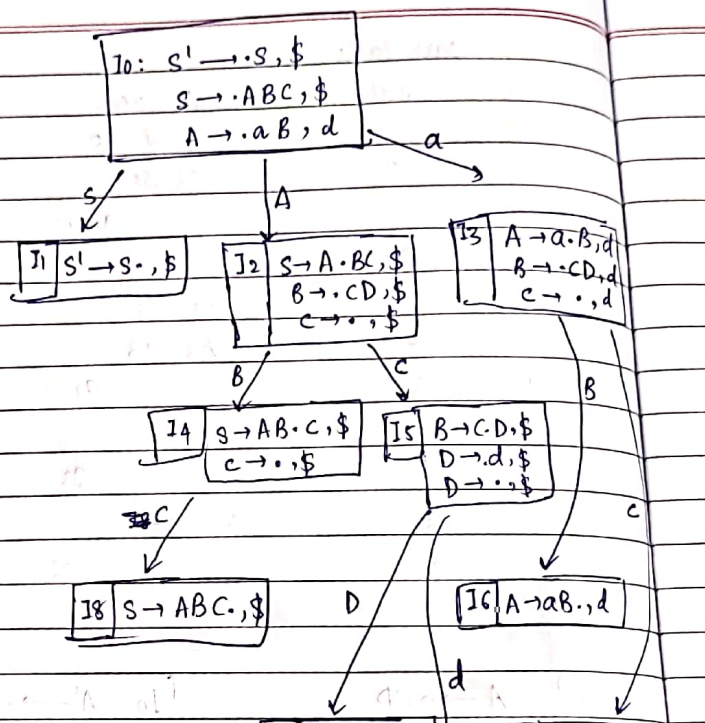
States	ACTIONS			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

ii) A → BCD
B → bC
C → e | ε
D → d



States	ACTIONS				GOTO			
	b	e	d	$	A	B	C	D
0	s3				1	2		
1				accept				
2	s5/r4		r4				4	
3	s5/r4		r4				6	
4			s9					8
5	r3		r3					
6			r2	r2				
7				r1				
8				r5				

iii) $S \to ABc$
$A \to aB$
$B \to CD$
$C \to \varepsilon_0$
$D \to d \mid \varepsilon_0$



States	ACTIONS			GOTO				
	a	d	$	S	A	B	C	D
0	s_3			1	2			
1			accept					
2		r_4				4	5	
3	r_4					6	7	
4		r_4				8		
5	s_{10}	r_6					9	
6	r_2							
7	s_{12}/r_6					11		
8		r_1						
9		r_3						
10		r_5						
11	r_3							
12	r_5							