Dt. _____
Pg. _____

06/01/21

Compiler Design

→ Lexical Analysis
→ Parsing / Syntax Analysis
→ Semantic Analysis
→ ICG (Intermediate Code Generation)
→ Code Optimization
→ Code Generation

Reference Book:
→ Compiler Principles, techniques & tools - Ullman Aho Sethi

Compiler → Translates a program written in one language into an equivalent program in another language.
(Source Program → Target Program)

Errors :-
Lexical → Spelling
Syntax → Grammer
Semantics → Meaning

Lexical Errors (Spelling)
→ Invalid words
Eg: int a = @3o; (@ → invalid character; 3o → invalid identifier)
int 1x2; (identifier)
String S = "Hell (invalid string)

Syntax Errors (Grammatical)
→ Improper structure
Eg: ;int a, b
int a, b
main()
{
}

Semantics Errors (meaning)
Eg: int a[-20];
int a, b; c = a+b;
Sum(1,2,3); Sum(int)

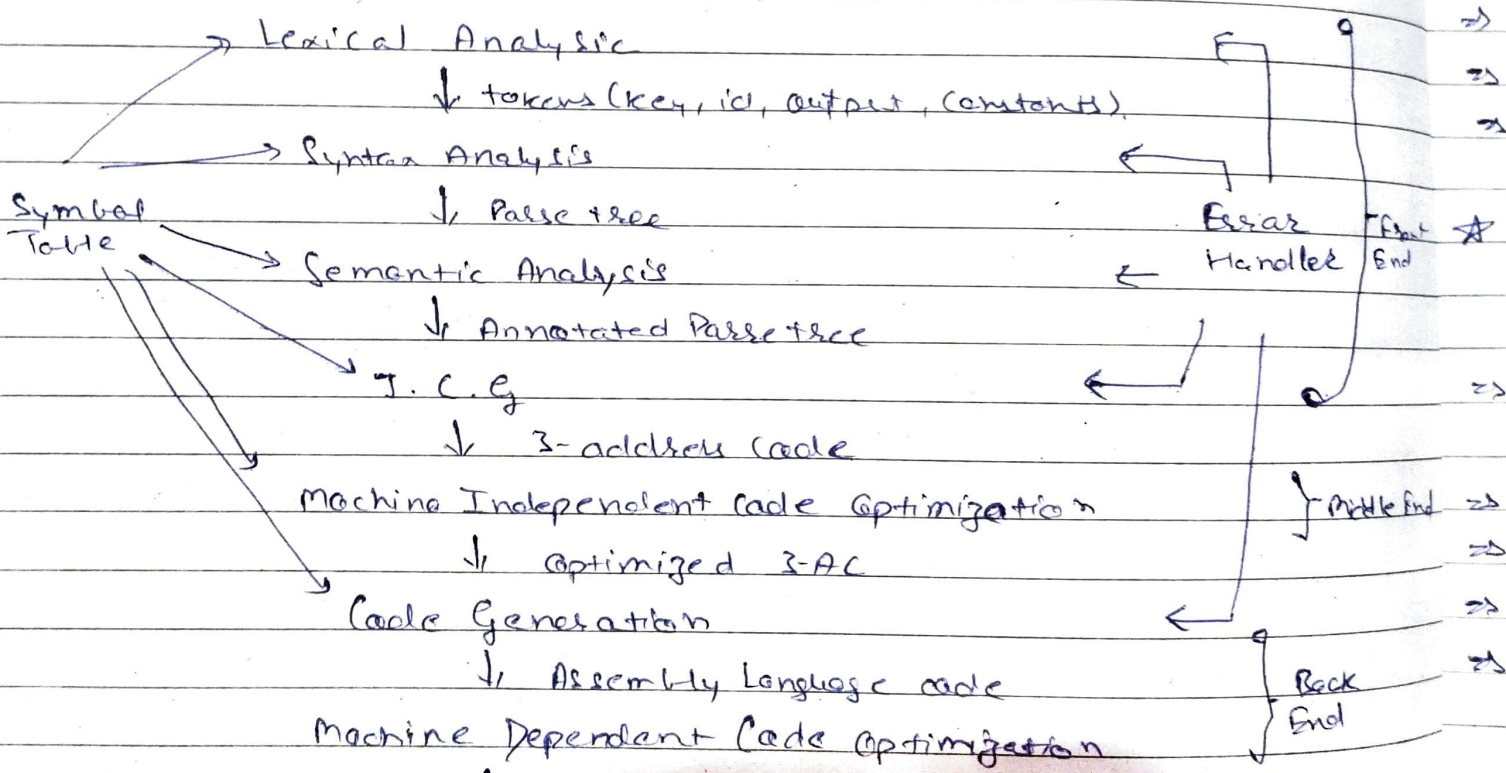
1st Compiler → FORTRAN by Backus Nov

Dt. _____
Pg. _____

Notes:

1. The translation process carried out by a compiler is called compilation.
2. Compilation can be done in different modules & each module is known as phase of the compiler.

Phase Diagram of Compiler.

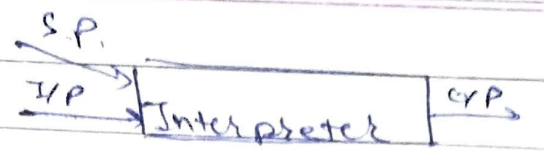


08/01/22 CD

⭐ Interpreter → Instead of producing a target program as a result of translation, an interpreter directly executes the operations specified in the source program on i/p supplied by the user.

Dt. _____

Pg. _____



* Difference b/w High Level Language & Low level language

HLL	LLL
=> Programmer friendly	=> M/c friendly
=> Requires Compiler / Interpreter	=> Executed directly
=> Easy to understand	=> Hard to understand
=> Portable & M/C Independent	=> Non-portable & M/c dependent
=> Simple	=> Complex

* Difference b/w Compiler & Interpreter

Compiler	Interpreter
=> Converts entire source code into executable m/c code.	=> Translate SC line by line.
=> Take time to analyze SC	=> Less time to analyze
=> Overall execution time is faster	=> Overall execution time is slow
=> Debugging is hard	=> easier.
=> Generate intermediate Object code	=> No intermediate code
Ex: C, C++, Java	Ex Python, Perl

* Assembler → Translate assembly code into machine code

~~Compiler~~

Dt. _____
Pg. _____

★ Difference b/w Compiler & Assembler

Compiler	Assembler
⇒ S/C to M/C	⇒ Assembly code → M/C
⇒ I/P is S/C	⇒ I/P is assembly
⇒ whole program at once	⇒ can't do it at once
⇒ O/P is binary	⇒ O/P are target compiler, i.e., mnemonic (ADD, MUL)
Ex GNU	Ex - C, C++, Java,
⇒ 2 phases	⇒ 6 phases

★ Language Processing System:

↓ Source Program (HLL)

Pre-processor : Does macro expansion

↓ modified S.P (Pure HLL)

Compiler

↓ Assembly language

Assembler

↓ M/C

Linker / Loader

↓ Target M/C code

Linker: Link & combine the objects generated by compiler into single executable program file. It is responsible to combine & link all the modules of the program if written seperately

Loader: Load the program from secondary memory to main memory.

Dt. _____

Pg. _____

Ex: a = b + c * d

[| a |]

↓

id op id op id op id PM

↓

Syntax A.

↓

S

/ | \

id = E ;

/ | \

E + E

| / | \

id E * E

| |

id id

↓

Semantic A.

↓

S.att Rule

/ | \ \

id = E.att ;

/ | \

E.att + E.att

| / | \

id E.att * E.att

| |

id id

$$\left[\begin{array}{l} S \rightarrow id = E; \\ E \Rightarrow E+E \mid E*E \mid id \end{array}\right]$$

⇒ Parse Tree

⇒ Annotated / Decorated P.T.

Dt.
Pg.

↓

I.C.G

↓

$t_1 = c * d$

$t_2 = b + t_1$

$a = t_2$

↓

Code Optimization

↓

$t_1 = c * d$

$a = b + t_1$

↓

Code Generation

↓

```
MOV  C, R0      (Assembly Code)
MUL  d, R0
ADD  b, R0
MOV  R0, a
```

11/01/21 C.P (3)

★ Lexical Analysis:

→ Reads HLL program & breaks the program into tokens where tokens are the logical unit of programming lang.

→ It also identifies lexical errors present in the given program

Ex: int a, ✓ int @a ; ✗

★ Syntax Analysis:

→ It groups all the tokens together & verifies their grammatical structure with the help of context free grammer (CFG)

Ex: int a, ✗

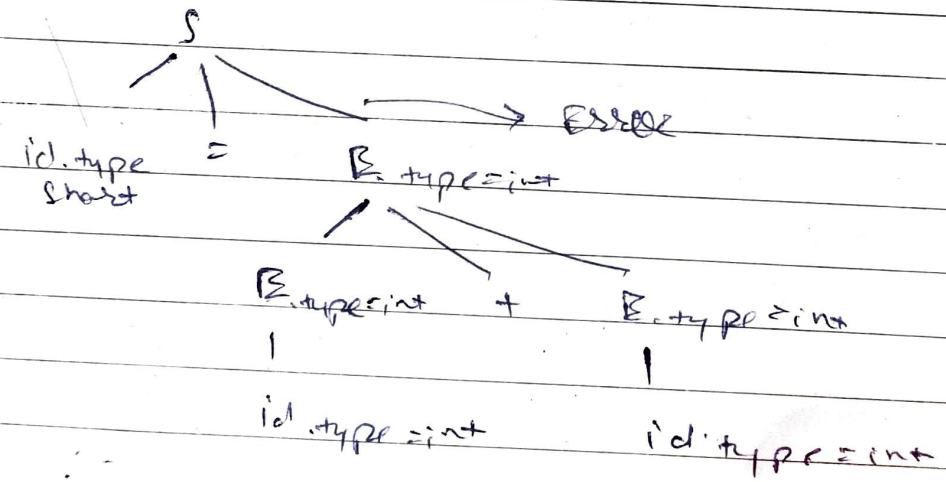
→ If the tokens are syntatically correct then it produces parse tree as O/P otherwise returns Syntax error

Ex. short a, b;
int c, d;
a = c + d;

Semantic Analysis:

→ Takes parse tree as input & produces annotated parse tree as o/p.
→ Type checking of the expression statements
→ Verifies any variable is used without declaration or not.
→ Re-declaration of some variable multiple time in the same scope.
→ Mismatch of formal & actual parameter.

Ex.



Intermediate Code Generation:

→ I.C. is a universal code from which generation of machine code for every machine becomes easy. This is known as retargeting
→ No need to change complete compiler, only change the back end of the compiler.
→ Optimization on the intermediate code is also possible.

Dt. _____
Pg. _____

→ The main form of I.C. are
a) 3-Address Code
b) Syntax Tree
c) Postfix Code

★ Code Optimization:
→ Improves the efficiency of target code by eliminating the unnessary code.

★ Code Generation:
→ Generate target program code

★ Single Pass & Multipass Compiles

Single Pass:

Multipass

Pass: One complete scan of the source program is known as the pass.

→ F.E. : HLL → Intermediate
M.E. : Optimize I.C
B.C. : I.C. → Target Program

Dt. _____

Pg. _____

☆ Symbol Table: Data structure that maintains information about identifiers & contents present in the program.

id	token	type	size	scope
:				

→ In the case of identifier, ST stores information type of the identifier, memory, scope (global/local).

→ In case of functional identifier: name of the function, return type, no of arguments, type of arguments, way of passing

13/01/21

CD

(4)

☆ Compiler Construction Tools →

a) Scanner Generator: Lexical analyzer (using Regular expression) Ex. LEX

b) Parser : Syntax analyzer, Ex YACC (Yet Another Compiler Compiler)

c) Syntax directed translation engine: to traverse parse tree & generate IC. Collection of routines

d) Code generator: Generate Target Code

e) Data flow analysis engine: Optimization

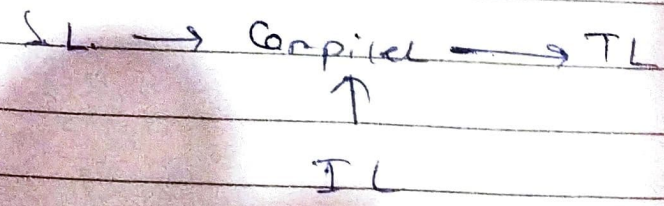
f) Compiler construction toolkit: Set of routines for constructing phases of compiler.

☆ Bootstrapping: Used to construct a compiler

a) Source lang

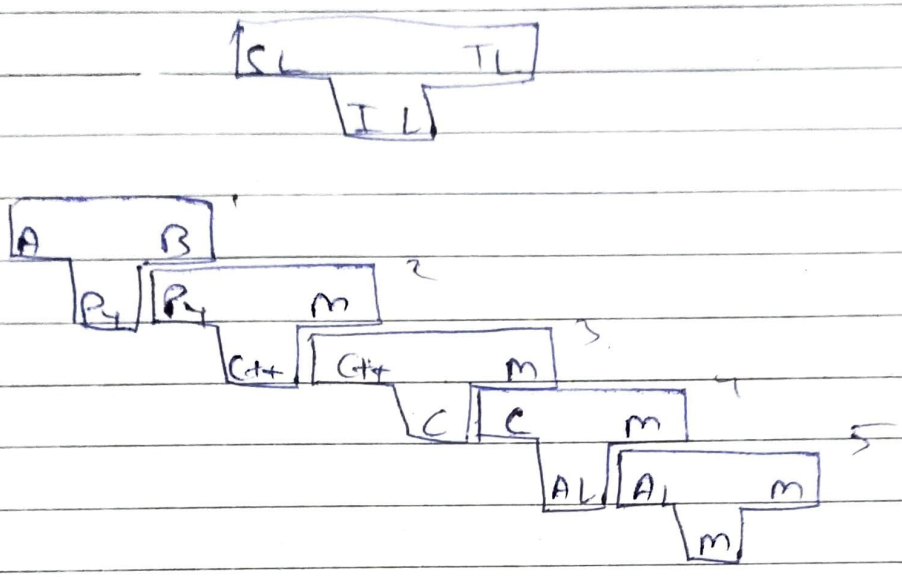
b) Target lang

c) Implementation lang



Dt. _____
Pg. _____

T-Diagram - The compilation process of compiler can be represented using T- Diagram.



Bootstrapping is a process in which a simple language is used to translated more complicated program which in turn may handle more complicated program. This complicated program can further handle even more complicated program & so on.

Cross compiler is a compiler which runs on one machine but it produces target code of another machine

★ Lexical Analysis Phase (Scanner):
→ Reads S.P one character at a time from left to right.
→ It breaks the program into tokens.
→ It identifies lexical errors
→ It eliminates blank & comment lines in the program
→ It ~~gives~~ stores the symbol table.
→ It maintains line no.

Dt. _____

Pg. _____

Tokens: Tokens describe category of the I/P string. (keyword, identifier, operator, string, constants, punctuation mark).

keywords: if, else, for, while

identifiers: x, y, x200, y13, _x, z7

operators: +, -, *, /, =, <, >, >=, etc

Constants: 100, 200, 3.5, 7.6, 9.7

Punctuation mark: ; , : { }, ( ), [ ]

Patterns: Rules of the tokens

Lexeme: Sequence of characters in the source program that are matched with patterns of the tokens.

Ex(i) [int] [a] [=] [4] [;]

Keyword id pm id p.m → lexemes

(ii) [int] [a] [=] [3 0] [;]

→ invalid lexeme, lexical error

keyword id op pm

15/02/21 C.D. (5)

Lexical Analyzer / Scanner

Manual Code — Tool Based - LEX / FLEX

Dt. _____
Pg. _____

✯ Rules of the Tokens:

a) Regular Expression:

Ex. $L = \{a, b\}$ R.E. $= (a+b)^*$

* → Kleen Closure

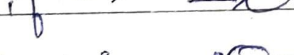
+ → Positive Closure

b) Finite Automata

Deterministic finite Automata: For each state & input, there will be exactly single transition

Keywords:

if =>


else =>


for =>


while =>


int =>


Identifiers:

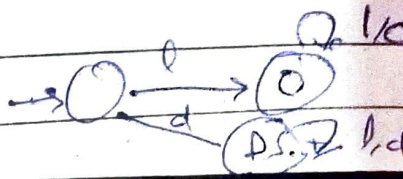
Symbols (i) letters $l = \{a, \ldots z\}$

(ii) digits $d = \{0 \ldots 9\}$

Rules of Identifiers : $l(l+d)^*$

→ Should start with a letter

→ No limit on length.



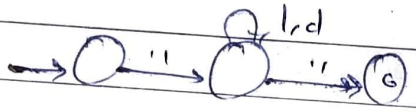
Dt. _____

Pg. _____

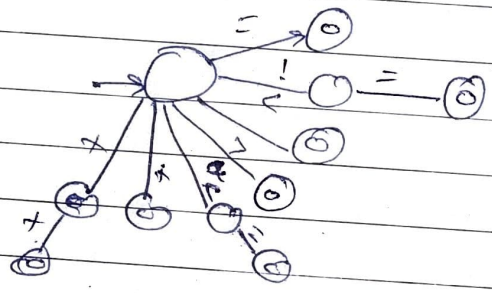
→ Length of the identifier should be ≥ 2

$l\,(l+d+\epsilon)^3\,l$ → RE.

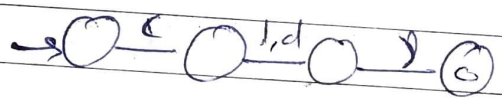
String :- "$(l+d)^*$"



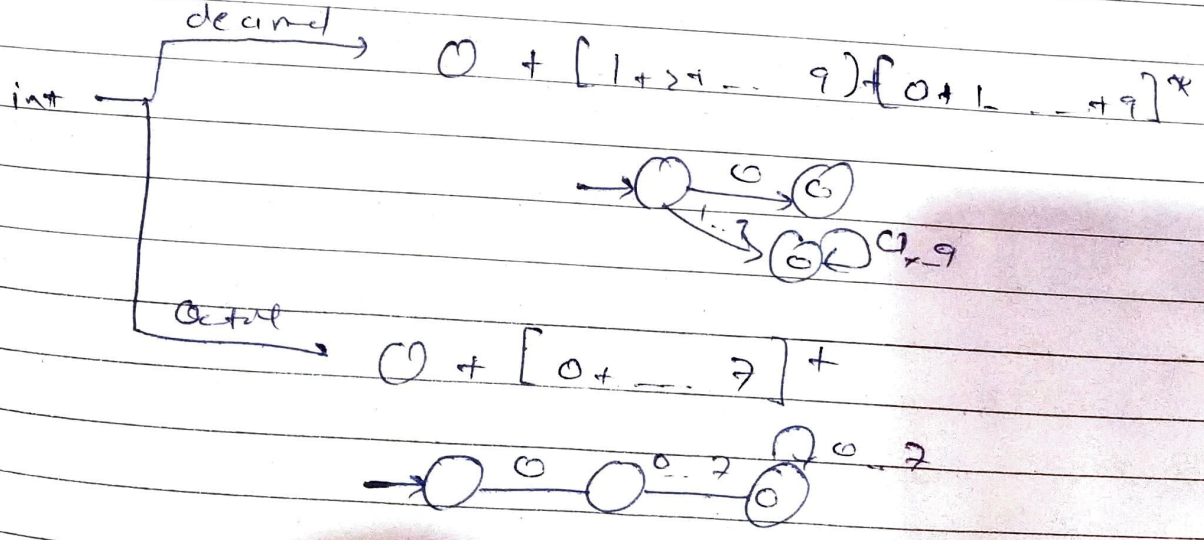
Operators:



char : $(l+d)$

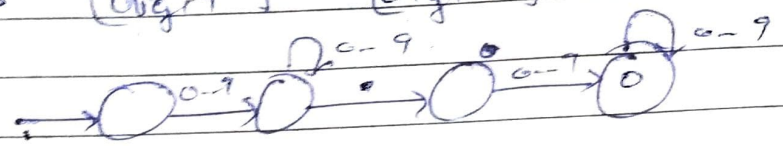


Constants:

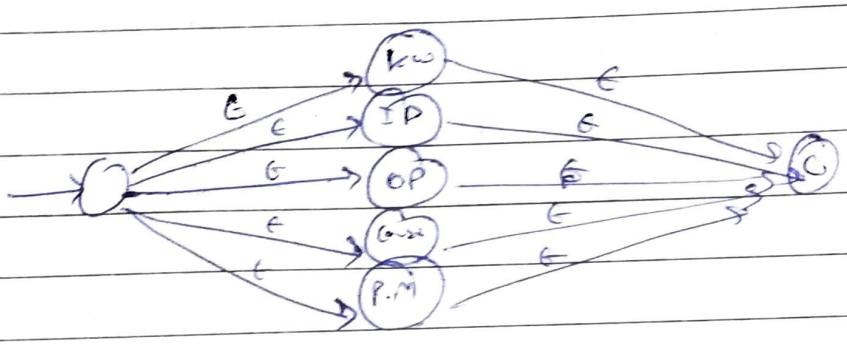


Dt. _____
Pg. _____

Real → [digit]$^+$. [digit]$^+$



Entire Rule Combined



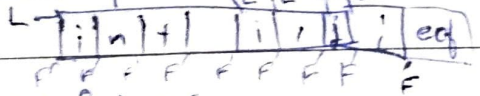
Input Buffering: Code stays in secondary memory & at execution only the main memory

a) One Buffer Scheme

b) Two " "

a) One Buffer Scheme

We use two pointers Lexeme Beginning & forward pointer



b) Two Buffer Scheme

19/01/21 CD (6)

☆ Problems Related to Lexical Analysis

a) Identify whether the given program format gives the L.F. or not

Ex. (1) if (a > b)
{
a = c + c;
}

No error in L.A.

" int a = 0 2 9; Here L.A. is present

Dt. _____

Pg. _____

Q) No. of tokens in given program format

(i) for | (| i=0 | ; | i<=10 | ; | i++ |) | # of tokens = 13

(ii) printf | (| "welcome" |) | ; | # of tokens = 5

(iii) for | (| a | > | b |) |
{ | a=b+1; | * addition cannot
}

Non tokens: Comments, preprocessor directive, blanks, tab, new line, macros, etc

(iv) int a, b, c;

Label	token id	Type	mem	Scope
a	id			
b	id			
c	id			

Not known to LA

Q.1 Find L.E.

(a) int, a b ;

(b) int a = $ 10;

(c) String s = " Hello

(d) float a = 2.3;

(e) float b = 3...4;

Q2. # of tokens

(a) int a = 1, b = 2;

(b) a = b c d ; +

(c) printf ("%d %d", x, y);

(d) while (i <= 100) {
a = a++;
} b = c + d; /* addition of two no.

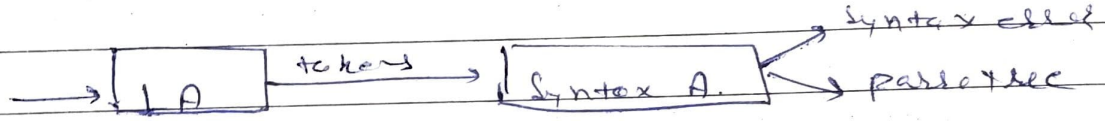
Dt. _____
Pg. _____

Q3. int main()
{
int i, n;
for (i=0; i<n; i++)
}

→ No problem in L.A.
Syntax error

20/01/21 C.D. (3)

* Syntax Error / Parser:



Syntax Error
1. Lack of tokens to form the statement
Ex int a,b
2. More no of tokens
Ex int a,, b;
3. Tokens not arranged properly
Ex (i;) ; int , ab

(i.) main()).
{
int a, b
; pf ("Hello")
}}

Dt. _____

Pg. _____

☆ Rules of the Statement: Written using Context free Grammer. Regular grammer have less power than CF G & they cannot handle comparision syntax.

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

V = Set of variables (No. of Terminals) (A, B, C, D, E) capital

T = " Terminals - (a, b, c, d ...)

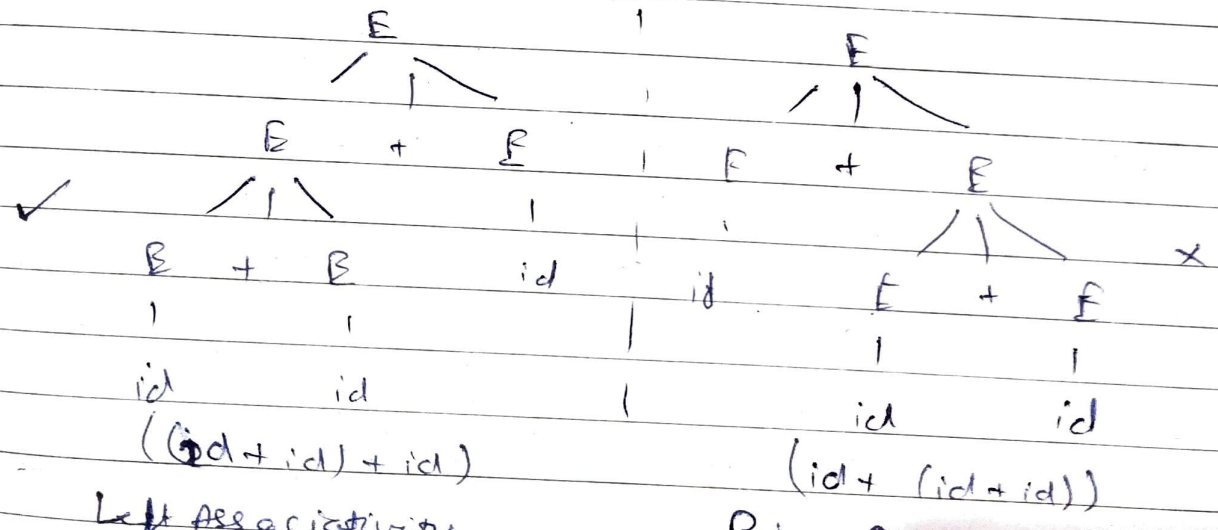
P = No of productions

S = Starting Symbol

Ex. $S \rightarrow id = E$

$E \rightarrow E+E \;/\; E*E \;/\; id$

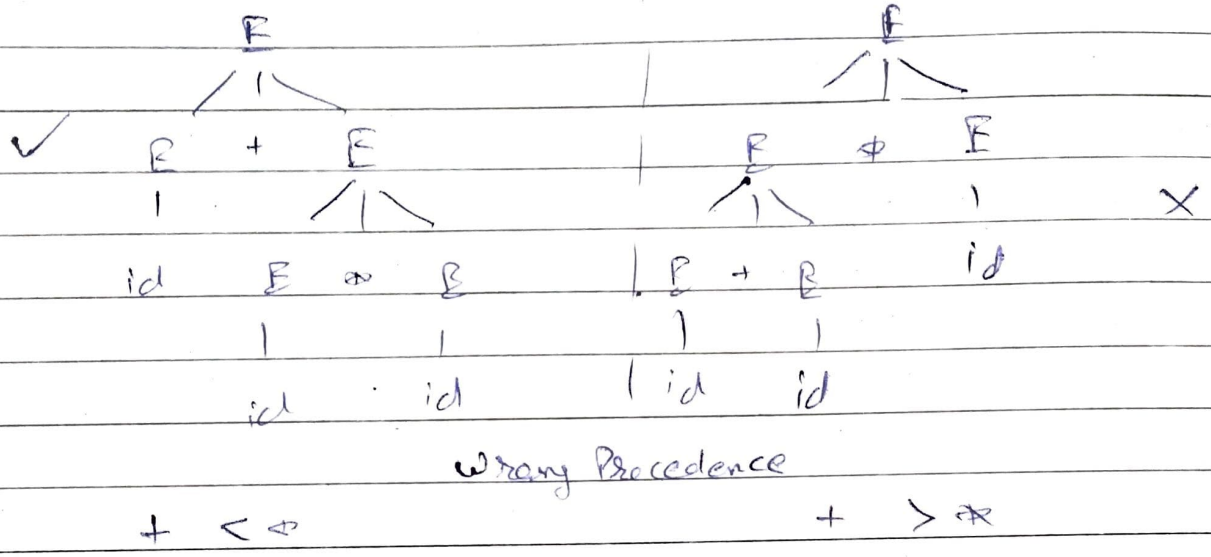
Ex. :) id + id + id



Wrong Ambigious Grammer

Dt. _____
Pg. _____

Ex: (i) id + id * id



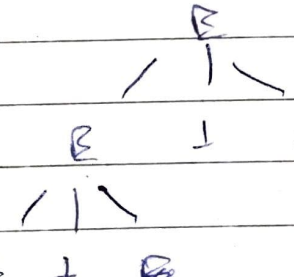
Wrong Precedence

+ < *

+ > *

* Remove Ambiguity:

E → E + id / id (Left Recursion)



* Solve Precedence

E → E + T / T

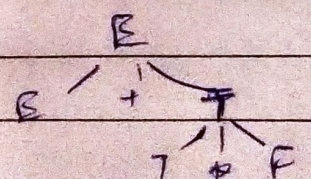
T → T * F / F

F → id

+ -> +

* -> *

+ < *



Dt. _____

Pg. _____

Ex. Power ↑ => (2 ↑ (3 ↑ 4)) R → L

E => E + T / T

T => T * F / F

F => G ↑ F / G

G => id

Associativity

+ => +

* => *

↑ <. ↑

Precedence

+ < * < ↑

Ex. Boolean Expression:

Exp → Exp or Exp / Exp & Exp / NOT Exp / True / False

E → E or F / F

F → F & G / G

G → Not G / True / False

Precedence

OR < And < Not

Associativity

OR => OR

And => AND

Q.1. A → A $ B / B

B → B ## C / C

C → C @ D / D

D → d

a) Ambiguity

b) Associativity of O/P

c) Precedence "

Ex. E → E * F / F + E / F

F → F - F / id

Associativity

* => *

+ <. +

- - (Ambiguous Grammer)

22/01/21 C.D.

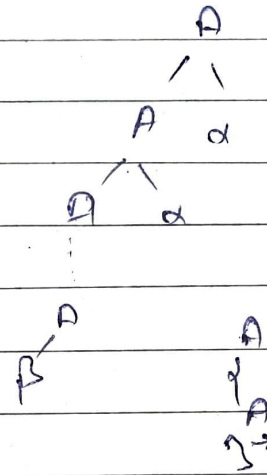
Grammar

Ambiguous → N.A.

Recursion

L.R

$A \to A\alpha / \beta$



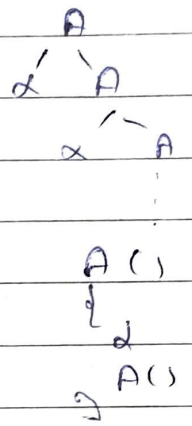
Infinite Loop

Not desired in parser

Language: $\beta\alpha^*$

R.L

$A \to \alpha A / \beta$



Language: $\alpha^*\beta$

Change Grammar:

$A \to \beta A'$

$A' \to \epsilon / \alpha A'$

Right Recursion

Ex. $E \to E+T / T$

$T \to T*F / F$

$F \to id$

$A \to \beta A'$

$A' \to \epsilon / \alpha A'$

Conversion:

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to id$

Dt. _____

Pg. _____

Ex. $S \to SOS \mid S \mid \epsilon \mid 01$

Generalize form :

$A \to A\alpha_1 / A\alpha_2 / A\alpha_3 \ldots / \beta_1 / \beta_2 / \beta_3 \ldots$

$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' / \ldots$

$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A'$

Left Recursion

Direct LR

$A \to A\alpha / \beta$

Indirect LR

$A \to B\alpha$

$B \to A\gamma / \nu$

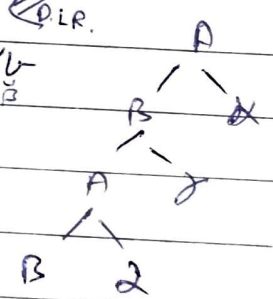
D.LR.

$B \to B\alpha\gamma / \nu$

$\beta = \nu\beta'$

$\beta' = \alpha\gamma\beta' / \epsilon$

$A \to \beta\alpha$



Ex. $S \to Aa / \nu$ (I LR)

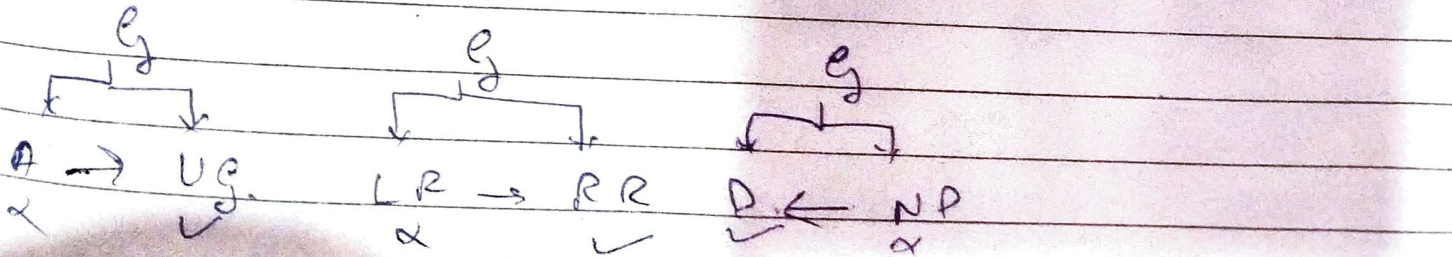
$A \to Sd / a$

$A \to Aad / \nu d / a$ (LR)

$A \to \nu dA' / aA'$ (Non-LR)

$A' \to adA' / \epsilon$

$S \to Aa / \nu$



Left factoring → Converting NDTM to DTM

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2 / \beta_3$

Eg $S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

$\Downarrow$

$S \rightarrow iEtSS' / a$

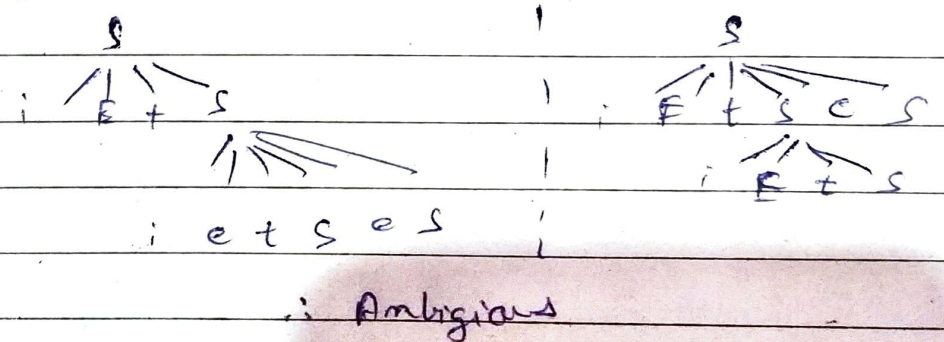
$S' \rightarrow \epsilon / eS$

$E \rightarrow b$

Note: Removing the Non Deterministic nature of grammar doesn't remove ambiguity.

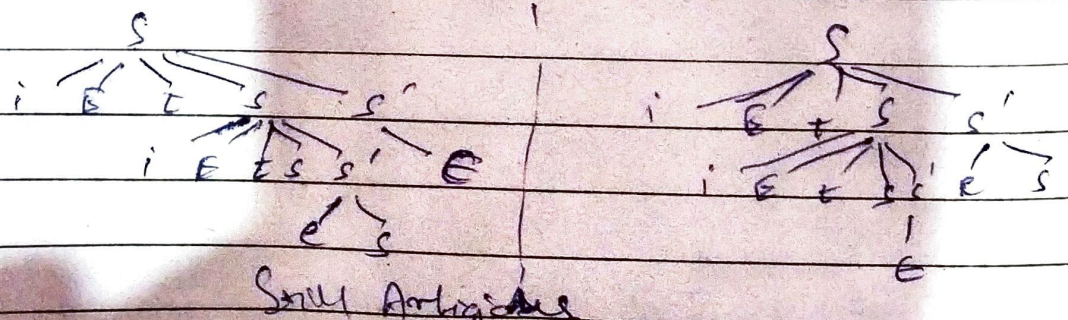
Eg → iEtS iEtSeS

(1) $S \rightarrow iEtS / iEtSeS / a$



∴ Ambigious

(2) $S \rightarrow iEtSS' / a$ $\quad S' \rightarrow \epsilon / eS$



Still Ambigious

Dt. _____

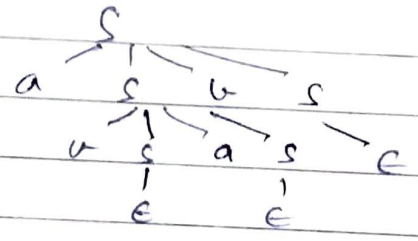
Pg. _____

Eg: $S \rightarrow aSbS$
$/ aSaSb$
$/ abb$
$/ b$

25/01/21 C.D. (9)

Eg: $S \rightarrow aSbS / bSaS / \epsilon$ → Multiple Productions

abab



To determine which production to select - First set is calculated.

★ First Set ( ) : Rules to calculate first set

a) First (terminal) = terminal

b) $A \rightarrow \alpha$

First $(A) = \{a\}$

c) $A \rightarrow B\alpha$, $B \not\rightarrow \epsilon$ (directly or indirectly)

First (A) = First (B)

d) $A \rightarrow B\alpha$, $B \rightarrow \epsilon$

First $(A) = \{$ First $(B) - \epsilon \} \cup \{$ First $(\alpha) \}$

Ex: (i) $S \to aA$ First(S) = {a}

$A \to b/\epsilon$ First(A) = {b, c}

$B \to d$ First(B) = {d}

Ex: (ii) $S \to AB$ First(S) = First(A) = {a}

$A \to a$ First(A) = {a}

$B \to b/\epsilon$ First(B) = {b, ϵ}

Ex: (iii) $S \to AB$ First(S) = {First(A) − ϵ} ∪ First(B) = {a, b}

$A \to a/\epsilon$ First(A) = {a, ϵ}

$B \to b$ First(B) = {b}

Ex (iv): $S \to AaAb/BbBa$ First(S) = {First(A) − ϵ} ∪ First(a) ∪ {First(B) − ϵ} ∪ First(b)

$A \to \epsilon$

$B \to \epsilon$

= {{ϵ − ϵ} ∪ a} ∪ {(ϵ − ϵ) ∪ b} = a, b

First(A) = {ϵ} First(B) = {ϵ}

Q1. $A \to Bb/Cd$

$B \to aB/\epsilon$

$C \to dC/\epsilon$

★ Follow Set(): Rules to calculate Follow Set

a) Follow(S) = {$}

b) $A \to \alpha B \beta$, $\beta \to \epsilon$

Follow(B) = First(β)

c) $A \to \alpha B$

Follow(B) = Follow(A)

d) $A \to \alpha B \beta$, $\beta \to \epsilon$

Follow(B) = {First(β) − ϵ} ∪ {Follow(A)}

Dt. _____

Pg. _____

Ex: (i) $E \to TE'$

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

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to id$

• Follow $(E) = \{\$\}$

Follow $(E') =$ Follow $(E) = \{\$\}$

Follow $(T) = \{$First $(E') - \epsilon\} \cup \{$Follow $(E)\}$

$= \{(+, \epsilon) - \epsilon\} \cup \{\$\} = \{+, \$\}$

Follow $(T') =$ Follow $(T) = \{+, \$\}$

Follow $(F) =$ ~~Follow (T)~~ $\{$First $(T') - \epsilon\} \cup$ Follow (T)

$= \{(*, \epsilon) - \epsilon\} \cup \{+, \$\}$

$= \{*, +, \$\}$

Ex: (ii) $S \to aABb$

$A \to c/\epsilon$

$B \to d/\epsilon$

Follow $(S) = \{\$\}$

Follow $(A) = \{$First $(B) - \epsilon\} \cup$ First (b)

$= \{(d, \epsilon) - \epsilon\} \cup \{b\} = \{d, b\}$

Follow $(B) =$ First $(b) = \{b\}$

Q.1 $S \to A$

$A \to aB/Ad$

$B \to bBC/f$

$C \to g$

Q.2. $S \to iE+SS'/a$

$S' \to eS/\epsilon$

$E \to b$

★ Passing Techniques

a) Universal Passing
- CYK Passing $O(n^3)$
- Earley's Passing

b) Top-Down Passing
- With Backtracking
  - Brute force
  - Recursive Descent Predictive Passing
- W/O Backtracking
  - Non-Recursive Descent Predictive Passing

c) Bottom-up Passing
- LR Passing
  - LR(0)
  - SLR(1)
  - CLR(1)
  - LALR(1)
- Operator Precedence Passing

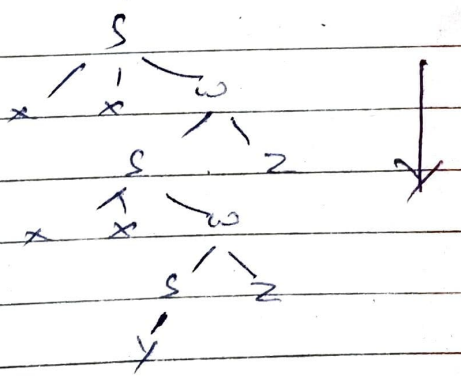
Ex.(1) $S \to xxw$

$S \to y$

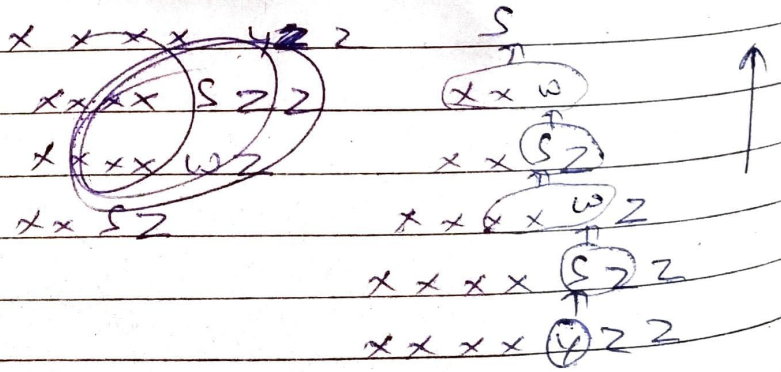
$w \to Sz$

$xxxxyzz$

Top-Down



Bottom-up



Dt. _____

Pg. _____

☆ Top-Down Parsing

→ Construct the parse table which start from the starting symbol & proceeds until the required string is produced by replacing every non-terminal by its corresponding R.H.S.

→ It uses left most derivation in the construction of parse tree.

☆ Bottom-Up Parsing

→ Construct the parse tree starting from the given string & proceeds until the starting symbol is reached by replacing the substring by its corresponding L.H.S. non-terminal.

→ It uses right most derivation in reverse order.

Top-Down Parsing

with Backtracking

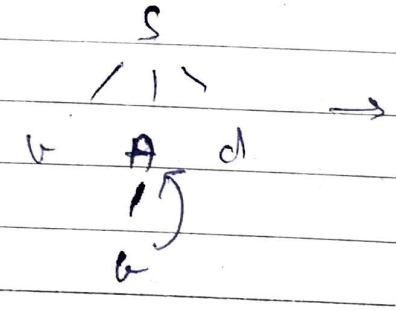
1. Brute force Parsing $O(2^n)$:

Ex. : S → bAd / aB

A → b / c

B → ccd / dd c

bcd:



→ Exponential Time Complexity so not used in practical compilers

→ Higher Power of Parser: No. of grammars a parsing technique can handle is known as power of a parser

→ Powerful compared to all the remaining techniques as it can handle A → ~~bAg~~

without Backtracking

1. Recursive Descent Predictive Parsing
   → Very simple to design
   → First compiler – FORTRAN

Ex: E → num T
T → * num T / ∈
E(), T()

```
Procedure E()
{
  if (lookahead == num)
  {
    advance();
    T();
  }
  else {error();}
  if (lookahead == $)
  { Pf("Sucess"); }
  else {error();}
}
```

```
Procedure T()
{
  if (lookahead == *)
  {
    advance();
    if (lookahead == num)
    {
      advance();
      T();
    }
    else {error();}
  }
  else {Null;}
}
```

```
advance()
{
  lookhead = Next token;
}
```

```
error()
{ Pf("Syntax error"); }
```

error essar

34 ✗ ✗ 34

Dt. _____

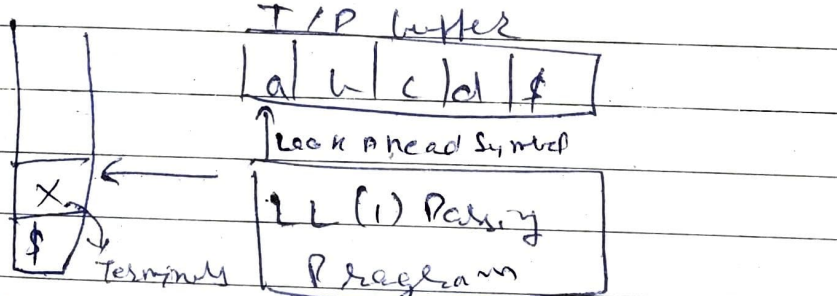
Pg. _____

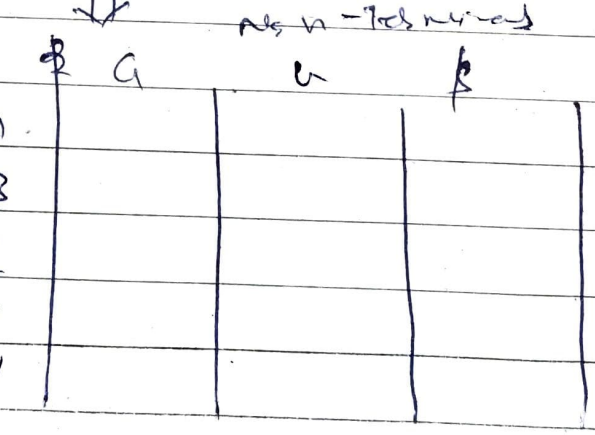
→ Recursive Program uses runtime memory (Activation record)
→ Not advised, may occupy lot of runtime memory in the compile time itself
→ Very difficult to run recursive program.
→ To avoid this, we use non recursive descent parsing.

29/01/21 C.D. (11)

2. Non-Recursive Descent Predictive Parsing or LL(1) Parsing

LL(1)
- Left to right
- Length of lookahead symbol.
- Left most derivation





LL(1) Parsing Table

Length of table = # of Non Terminal * # of Terminals

Dt.
Pg.

Let x be the symbol on the top of the stack & a is look ahead symbol then.

a) if $x = a = \$$
I/P string is valid & parsing is successful

b) if $x = a \neq \$$
Pop x from stack & increment I.A.S

c) If x is non-terminal on the stacks, then parser takes the decision from the parsing table
$[x, a] \Rightarrow x \to U\ V\ W$
x is replaced by U V W such that U appears on the top of stack

d) if $[x, a] \Rightarrow$ → blank
Syntax error, & return to the error handler.

→ To apply this algo initially starting symbol will be pushed on the stack.

Ex. $S \to A\ B$ → length of parsing table
$A \to a$
$B \to b$

Apply parsing for the string ab by considering following grammer in LL(1) parsing table & also identify total no. of actions

	a	b	$
S	S→AB		
A	A→a		
B		B→b	

Dt. _____

Pg. _____

Stack	I/P buffer	Action
$ S	a b $	1
$ B A	a b $	2
$ B a	a b $	3
$ B	b $	4
$ b	b $	5
$	$	6. Accept

E. $E \to TE'$

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

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to id / (E)$

I/P string $\to$ id + id * id

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

Stack	I/P	Action
$ E	id + id * id $	1
$ E' T	id + id * id $	2
$ E' T' F	id + id * id $	3
$ E' T' id	id + id * id $	4 Pop
$ E' T'	+ id * id $	5
$ E'	+ id * id $	6
$ E' T +	+ id * id $	7 Pop
$ E' T	id * id $	8
$ E' T' F	id * id $	9
$ E' T' id	id * id $	10 Pop
$ E' T'	* id $	11
$ E T' F *	* id $	12
$ E T' F	id $	13

Dt. _____
Pg. _____

$ E T' id	id $	14 Pop
$ E T'	$	15
$ E'	$	16
$	$	17. Accept

01/02/21 CD (12)

* Construct LL(1) Passing Table for the Grammar

E1. $E \to TE'$

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

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to id / (E)$

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

First(E) = {id, (}
First(E') = {+, ε}
First(T) = {id, (}
First(T') = {*, ε}
First(F) = {id, (}

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

→ If first has ε, we fill using follow
→ No multiple entry → LL(1)
→ Once grammar is LL(1) it is unambiguous, but not true

Dt. _____

Pg. _____

→ Left Recursive Grammar is not LL(1)

→ NP / Non left factored → Not LL(1)

→ Single production for each N.T. → LL(1)

→ $A \to \alpha_1 / \alpha_2 / \alpha_3 \ldots \alpha_n$

First(α_1) ∩ First(α_2) ∩ ... ∩ (First(α_n) ≠ ∅ → LL(1)

→ $A \to \alpha / \epsilon$

First(A) ∩ Follow(A) ≠ ∅ → LL(1)

Ex.: (i) $S \to aSbS / bSaS / \epsilon$

LL(1) grammer is not

First($aSbS$) = {a}

First($bSaS$) = {b}

First(ϵ) = Follow(S) = {a, b, $}

[S, a] => $S \to aSbS$, ~~$S \to \epsilon$~~

[S, b] => $S \to bSaS$, $S \to \epsilon$

[S, $] => $S \to \epsilon$

Ex: (ii) $S \to aABb$

$A \to c / \epsilon$

$B \to d / \epsilon$

Single grammer production is always LL(1)

Follow(A) = First(B) = {b, d}

Ex: (iii) $S \to aB / \epsilon$ follow(S) = follow(C) = follow(B) = {$}

$B \to bC / \epsilon$

$C \to cS / \epsilon$

Dt. _____
Pg. _____

G.1 $S \to AB$
$A \to d / \epsilon$
$B \to b / \epsilon$

G 2. $S \to aSA / \epsilon$
$A \to c / \epsilon$

G 3. $S \to A$
$A \to Bb / Cd$
$B \to aB / \epsilon$
$C \to cC / \epsilon$

03/02/21 CD (13)

G.4. $S \to AB$
$A \to ab / ac$
$B \to bb / ba$
LL(2) as not

~~First(S) = {a b} {a c}~~ Always LL(2)
~~First(A) = {a b} {a c}~~
~~First(B) = {b b} {b a}~~
First(ab) = {a b}
First(ac) = {a c}
(bb) = {b b}
{b a} = {b a}
∴ LL(2) ✓
LL(1) ✗

Power of LL(2) parser is more than LL(1) parser

Dt. _____
Pg. _____

Bottom-Up Parsing Tree (Shift Reduce Parser)

→ Based on handle detection & handle pruning mechanism.

Handle: Substring in the I/P string that is matched with RHS part of the grammer.

→ When handle is detected, replace it by corresponding LHS non-terminal. This is known as handle pruning.

Ex (i) $S \to xxw$

$S \to y$

$w \to Sz$

$xxxxyzz$ - (5)

$xxSzz$ - (4)

$xxwzz$ - (3)

$xSzz$ - (2)

Szz - (1)

# of handles = 5

Actions in Bottom-up

- Shift: Shift the symbol from I/P buffer into the stack
- Reduce: When a handle is detected on the stack, replace it by its ~~left~~ corresponding left non-terminal.
- Accept: If stack contain the starting symbol & look ahead symbol, I/P string is accepted.
- Error: Parser halts w/o accepting the string

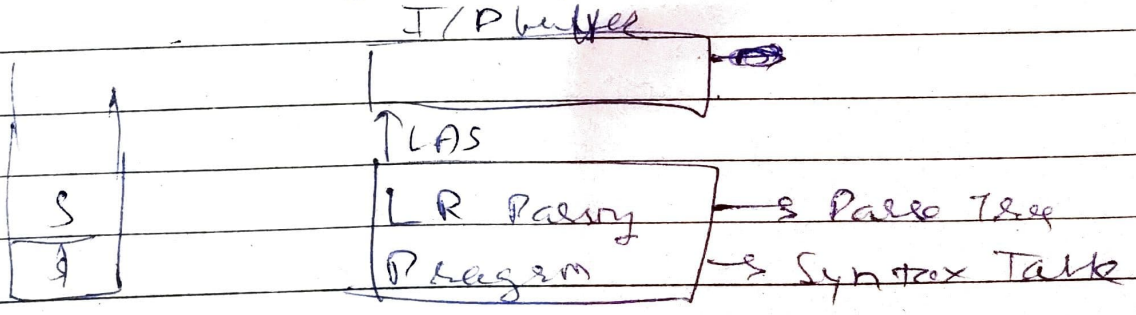
Q. How many shift reduce actions taken by shift reduce parser to reduce the I/P string

int id, id;
S → TL;
T → int / float
L → L, id / id

Shift = 5
Reduce = 4
of operation = 10

Stack	I/P Buffer	Action
$	int id, id; $	Shift
$ | int	id, id; $	Reduce
$ | T	id, id; $	Shift
$ | T) id	, id; $	Reduce
$ | T | L	, id; $	Shift
$ | T | L |,	id; $	Shift
$ | T | L |, | id	; $	Reduce
$ | T | L	; $	Shift
$ | T | L |;	$	Reduce
$ | S	$	Accept

LR Parsing Technique :



Dt. _____

Pg. _____

Terminals → / State ↓	Action			Goto		→ Non-Terminals
	a	b	$	S	B	
I_0						
I_1						
.						
I_n						

a) Initially push initial / start state of the parsing table onto the stack.

b) Let S be the state on the top of the stack & a is the L.A.S. in I/P buffer, then LR parser will take decision from the LR parsing table as follows.

i) if Action [S, a] = Shift J

Shift a onto the stack & then shift J onto the top of the stack & increment the L.A.S.

ii) if [S, a] = Reduce (A → β)

then pop 2*|β| (2 times of length of β Symbol from the stack) & push A onto the stack & then push goto of (J, A) onto the top of the stack.

iii) If [S, a] = Accept

I/P string is valid → stop parsing

iv) If [S, a] = Blank

Syntax error → return to error handler