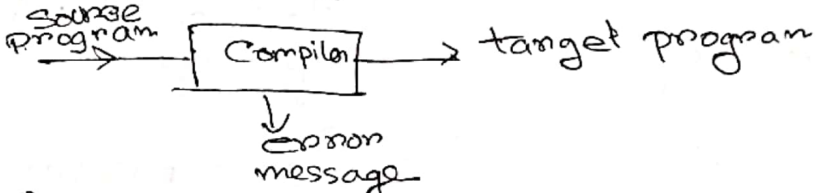
C.W. Compiler design

23-06-22

Compiler: A compiler is a program that reads a program written in one language - the source language - and translate it into an equivalent program in into another language - the target language



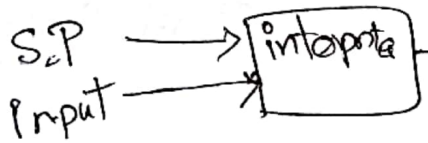
Importance:

1. Report any error in the source program that it detects during the translations process.

If the target program is an executable mechine language program. It can then be called by the user to process inputs and produce output.

Fig: Running the tp.

Interpreter: An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on input supplied by the user

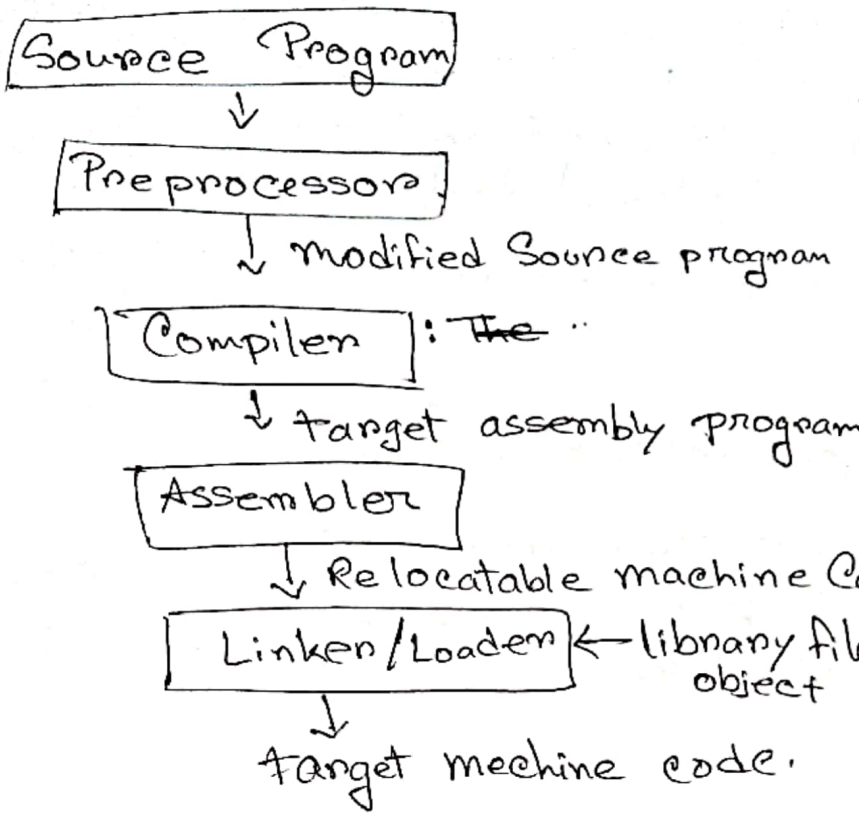


Difference btn compiler & interpreter:

	Compiler		Interpreter
1.	Compiler Code runs faster	1.	Interpreter code runs slower.
2.	It links between code files into runnable program.	2.	No linking of files or machine code generation
3.	Program code is already translated into machine code	3.	Interpreter are easier to use specially for beginners.
4.	The code execution time is less. It is based on language translation linking loading model.	4.	It is based on interpretation model.
5.	It takes entire program	5.	It takes a single line of code
6.	Display all error after compilation	6.	Display all error of each line one by one.
7.	Example: C, C#, Java, C++	7.	Example: PHP, Javascript Perl, Ruby, Python

Language processing system:-

L.P.S:



Preprocessor:- read through source code & prepare it for compilation.

After preprocessing the newly filtered file is passed to the compiler. takes the preprocessed file. and use it to generate corresponding assembly code

Assembler:. From the compiler the new assembly code is passed to the assembler. Assember assemble the code into object code

Linker: Takes all object codes. libraries passed to it & link. them togeter in a single execution file

C.W.

~~SAD~~ CD

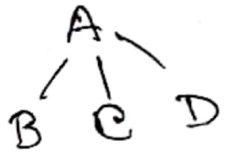
27.06.202

Phase: logically interrelated operation.

2 phase.

a) Anlysis: (Non mechine

b) Synthesis: (mechine



Lexical

LA → Scanner: Reads source program & turn the S.P. to a sequence of token.

source prog.

↓

L.A → Scanner

↓ token

Syntax → Parsing.

↓ expression

Semantic.

↓ statement

Intermediate Code Generation.

↓

Code Optimization

↓

Code generation

↓

target prog

S. A: makes tree.

semantic: shorten the syntax parsing.

I.C.G: - final machine language is produce

C~~O~~D - Improves.

C.G -.

Symbol table manager.

error handler.

Present

3 - Bithi

8 - Nadim

10 - Natasha

13 - Icncy

16 - Nithun

24 - Jannat

26 - Fahin

27 - Mou

33 - Nishi

Abs

34 - Shonno

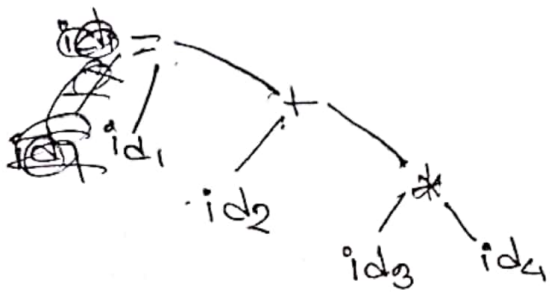
42 - Prem

19 - Nilay

position = initial + rate * 60

LA: $id_1 = id_2 + id_3 * id_4$

S.A:



C.O:

$temp_1 := id_3 * 60.0$

$id_1 := id_2 + temp_1$

C.G: Assembly Code

MOV F id_3, r_2

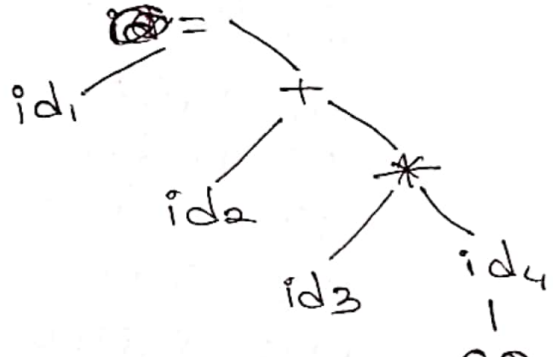
MUL F *60.0, r_2

MOV F id_2, r_1

ADD F r_2, r_1

MOV F r_1, id_1

Semantic A:



I.C.G:

$temp_1 := inttoreal(60)$

$temp_2 := id_3 * temp_1$

$temp_3 := id_2 + temp_2$

$id_1 := temp_3$

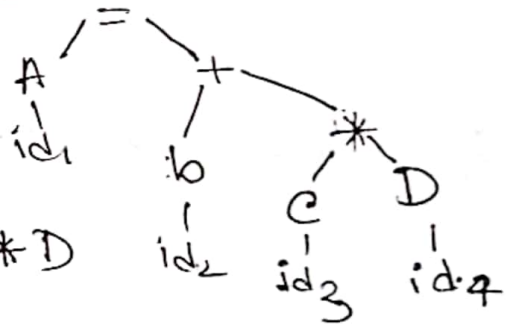
Assembly language func

MOV, MUL, MOVF-

ADD DIV . F→function.

SUB

R→ register.



A = b + C * D

I.C.G:

$temp_1 := id_4 * id_3$

$temp_2 := id_2 + temp_1$

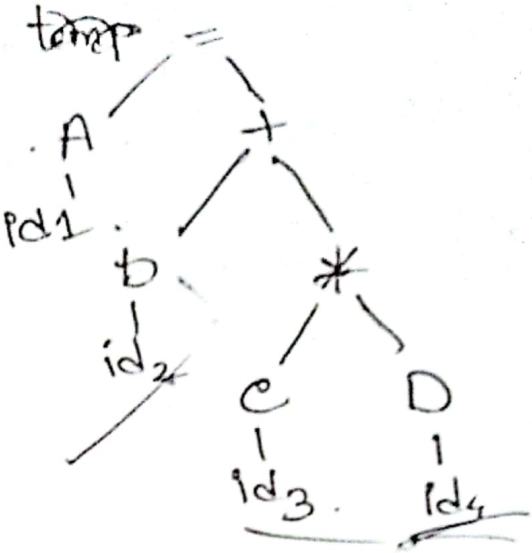
$id_1 := temp_2$

C.O:

$temp_1 = id_4 * id_3$

$id_1 = id_2 + temp_1$

Assembly:



MOV F id_4, r_1

MOV F id_3, r_2

~~ADD~~ ~~id_3 r_2, r_1~~

MUL F $r_2 * r_1$

MOV F id_2, r_3

ADD F r_1, r_3

MOV F r_3, id_1

~~Answer~~

H.W. $A = B + C * 75$

27-06-22

107 - Bithi
108 - Nadim
110 - Natasha
113 - Jency
116 - Mithun
124 - Jannat
126 - Fahim
127 - Mou
133 - Nishi

SAD

System types

1. Physical :- tangible, Abstruct
2. Open closed
3. Man-made Information system : manage data for particular organization
   * Formal
   * Informal : employee based
   * Computer based : System that depends on comput

MIS

Processing logic
Computer
Human Being
Database

"Automated"

upto Real time Business Subsystem

7- Bithi
8 - Nadim
10 - Natasha
13 - Jency
16 - Mithun
24 - Jannat
27 - Mou
33 - Nishi
42 - Prema

DSS classification

Passive
Active
Co-operative.

DSS -> user
-> assistance.

A -> bC

(I)

Grammar has 4 components.

① A set of terminal symbol. "Token"

② The terminal are three elementary symbol of language defined by the grammar.

② A set of non-terminal symbol. "Syntactic variable"

③ Set of productions, each production consist of non terminals, called head/left side of production,

an arrow and a sequence of terminal and/or non-terminal called -body / right side of production.

4. A designation of one of the non-terminal as the start symbol.

list = digit
list = list + digit
list = list - digit
digit = 0|1|2|3|4|5|6|7|8|9

9-5+2, 3-1 op 7

The bodies of three production with non-terminal list as head equalently can be grouped -

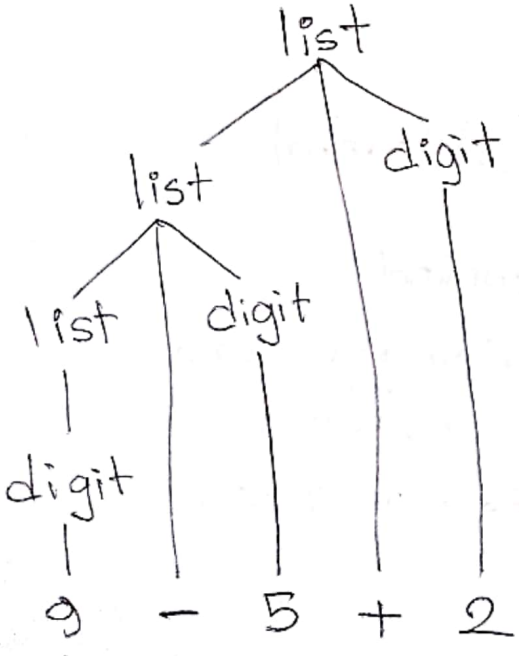
list = list + digit | list - digit | ~~dis~~ digit

9 - 5 + 2

(a) 9 is a list production, since 9 is a digit.

(b) 9-5 " " " " " 9 " " list & 5 is a digit

(c) 9-5+2 " " " " , " 9-5 is a list & 2 is a digit

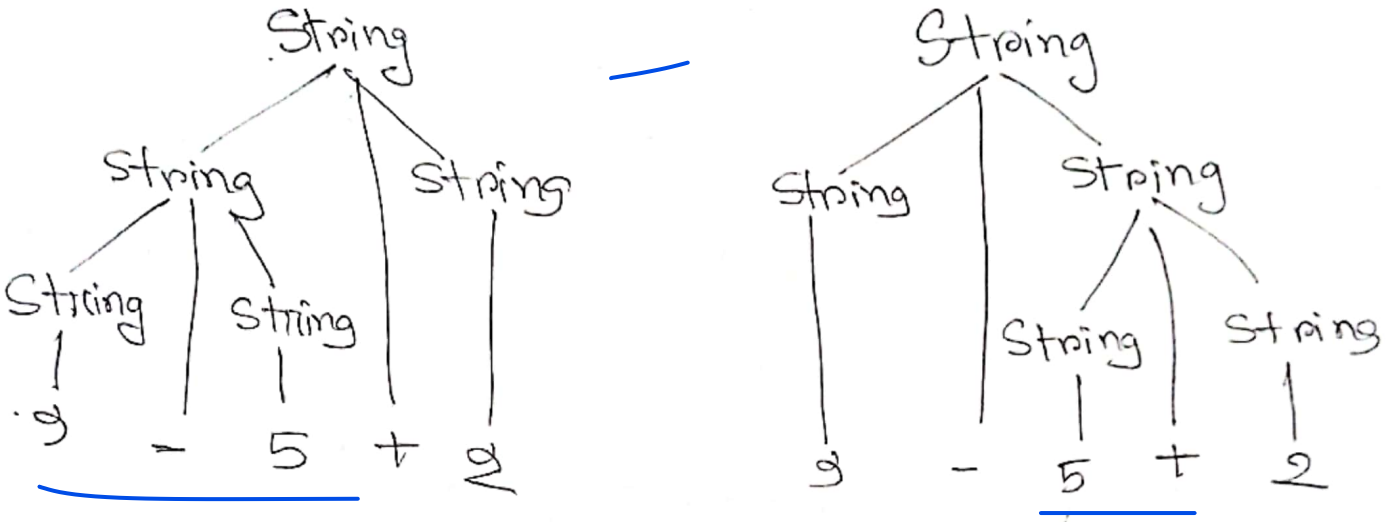


Ambiguity: A grammar can have more than one parse tree generating a given string of terminals, such a grammar is said to be ambiguous.

9-5+2

(9-5)+2 9-(5+2)

String = string + String | String - String | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Parse tree: A parse tree show how the start symbol of a grammar derives a string in the language If non-terminal A has a production then $A \rightarrow XYZ$ then the parse tree may have interior node labeled X, Y and Z from left to right.

Formally, CFG has following properties:

1) The root is labeled by a ~~terminal~~ start symbol.
2) each leaf is labeled by a terminal.
3) Each interior node is labeled by a non-terminal.
4) If A is nonterminal labelling som interior node and $X_1, X_2, \dots X_n$ are the labels of the children of that node from leaf to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$.

Flow Control : maintaining of speed of data transmission between two devices. Datalink layer.

Protocol — Noiseless -> Stop and wait, Simplest
Channels.
— Noisy -> stop & wait ARQ, Go-Back-N-ARQ, Selective Repeat ARQ.

Stop & Wait Protocol.

CPU gene bus generates acknowledgment signal.

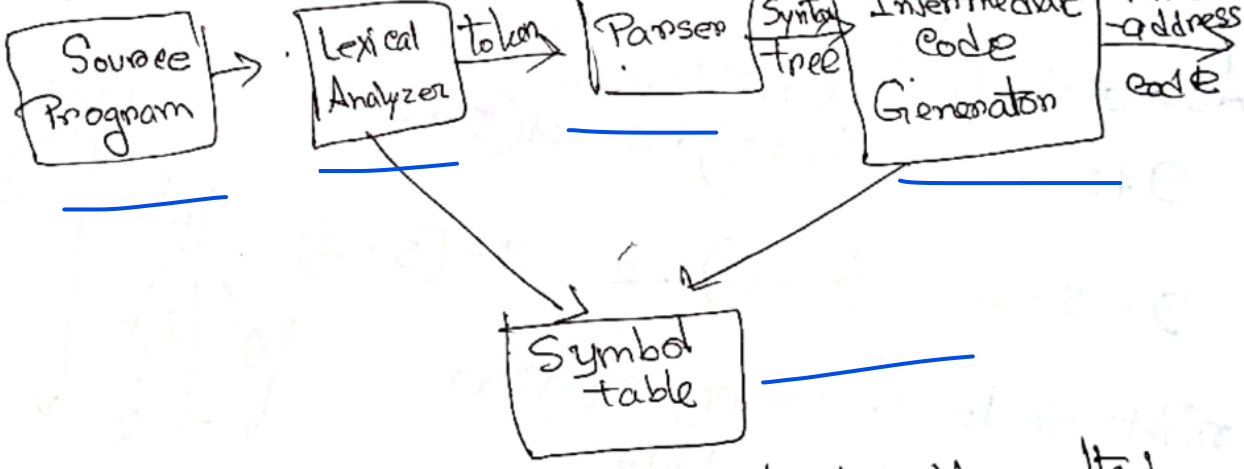
Noisy -> Stop & Wait ARQ :

Stop-And-Wait ARQ = Stop-and Wait + time out timer + sequence number.

Ramp sinusoidal,

C.W. C.D -> 03-067-22

A model of Compiler Front-end



10 - Nata
13 - Jen
16 - Mith
24 - Jaunt
26 - Thim
27 - Mou
33 - Nishi
34 - Srno
42 - Prema
Late: 8 - Nandini
19 - Nilay

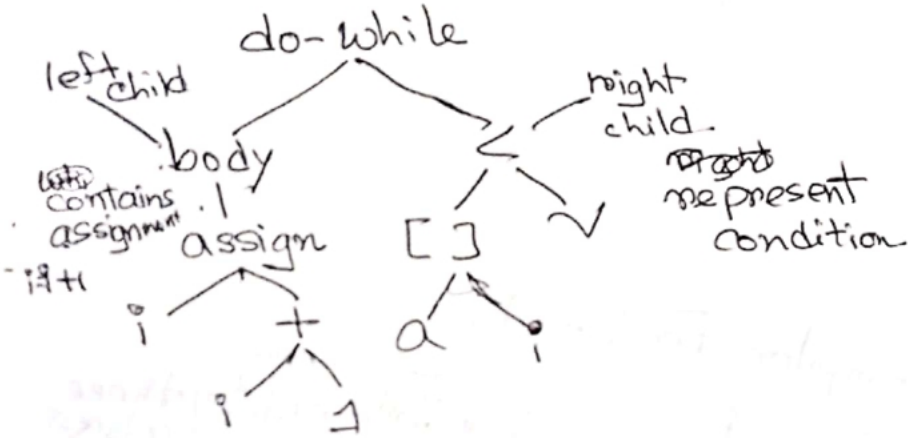
A. L.A allow a translator to handle multicharacter constructs like identifiers which are written as sequence of character but are treated as unit called token during syntax analysis.

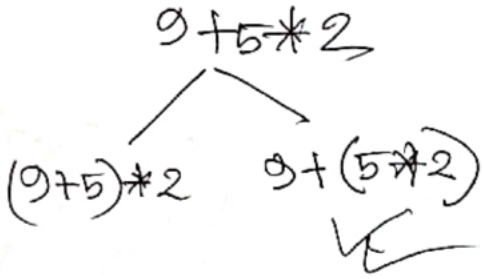
In fig. the parser produce a syntax tree. Then it's farther translated into three address code.

I.C.G:- Two forms: 1. Abstract Syntax tree
or
Simply Syntax tree.
Represents hierarchical syntactic structure of source program

Some Compiler combines parsing and i.c.g into one component.

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

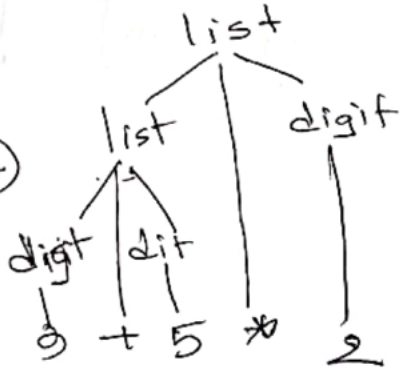




Associativity of Operator:-

9+5+2 (9+5)+2 9+(5+2)

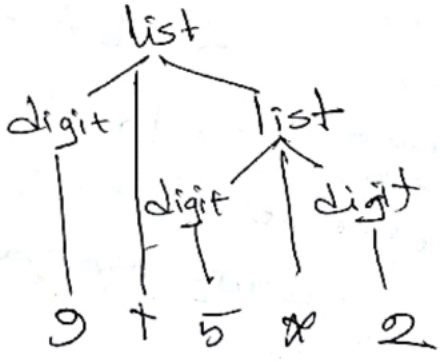
9-5-2 (9-5)-2 9-(5-2)



right → letter = right/letter
letter → a|b|c|... |y|z

Precedence

a=b=c



CD

04-07-22
Monday.

Four common arithmetic operator & precedence table

Left associative: + −	7
" " : * /	8
→Factor → digit | (expr).	10
	13
→term → term * factor.	16
| term / factor	24
| factor	26
	27
	34
	42
	15

the resulting grammar :

expr → expr + term | expr - term | term

term → term * factor | term / factor | factor.

factor → digit | (expr)

postfix Notation

(i) If E is a variable or constant, then the postfix notation for E is E itself.

(ii) If E is a expression of the form E_1 op E_2, where op is any binary operator then the postfix notation for E is $E_1' E_2'$ OP, where E_1' and E_2' are the postfix notation for E_1 and E_2 respectively.

(iii) If E is a parenthesize expression of the form (E_1) the postfix notation for E is the same as the postfix notation for E_1.

example: (9-5+2)

→ By rule 1, 9, 5, 2 are constants.

→ " " 2, translation 9-5 is 95-

→ 95-2+

9-(5+2)

9-52+

952+-

(9+5)-(4-6)

95+ - 46-

95+ 46--

By rule 1, 9,5,2 are constants

" " 2, translation 5+2 is 52+

→ 9 52+-

Syntax-directed translation / translation scheme.

S.D.T. is done by attaching rules or program fragments to production in a grammar. eg.

Consider expr generated by the product

expr → expr, + term sum of two subexpression

→ We can translate expr by exploiting its structure as in the following pseudo-code.

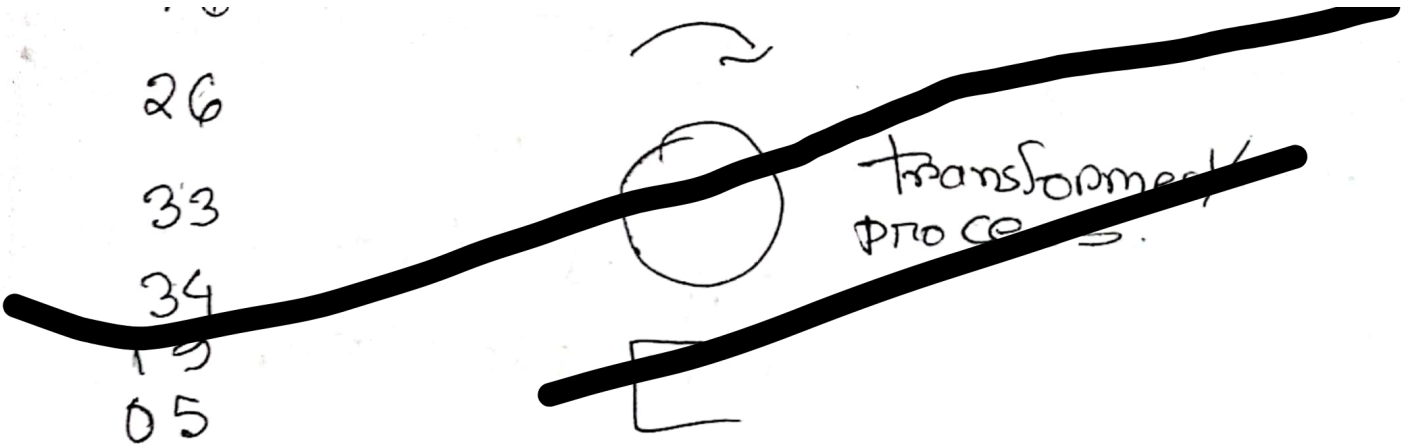
translate expr:
translate term)
handle +;

Translation scheme:

A SDT defination associates:

(i) With each grammar symbol, a set of attributes and
(ii) With each production, a set of semantic rules for computing values of the attributes associate with symbol appearing in production.

26
33
34
05



Decision tree, table,

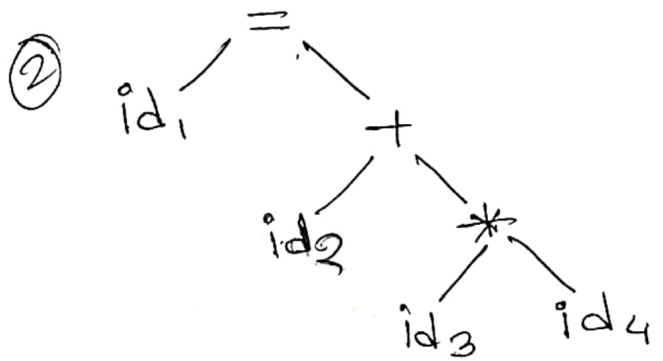
Compiler Design

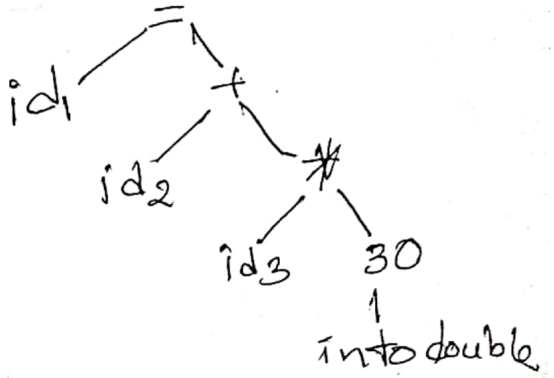
Gain Slm day-1

Source code → Compiler → Object code.
HLL MLL
S.C → P.P → C → A → L

value = value + rate * 30

① $id_1 = id_2 + id_3 * id_4$

② 

③ 

④
$temp_1 := int\ to\ double(30)$
$temp_2 := id_3 * temp_1$
$temp_3 := id_2 + temp_2$
$id_1 := temp_3$

⑤
$temp_1 := id_3 * 30.0$
$id_1 := id_2 + temp_1$

or

$temp_1 = id_3 * int\ to\ double(30)$
$id_1 = id_2 + temp_1$

⑥ Assembly

Ammulator

```
MOV X, 30.0
MUL X, X
MOV X, Y
ADD Z, X
MOV M Z
```

cannot be done on random variable.

Data must be stored in resistor first.

$e^{j2\pi}$

$\cos 2\pi + j \sin 2\pi$

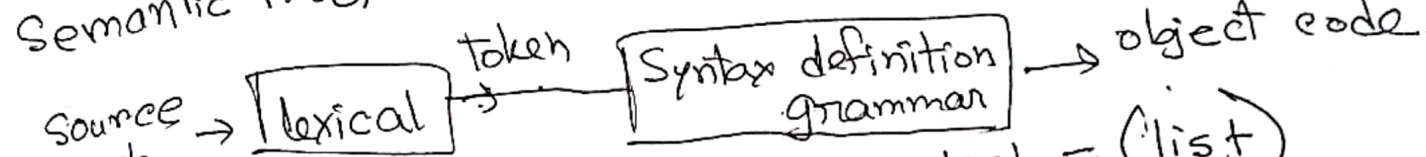
1 + 0.

CD - 03-08-22

Simple one pass compiler.

Grammar → Syntax → context free grammar. (CFG)

Semantic info.



endpoint start

$G = (T, N, S, R)$ — R: Rules/Production

T: terminal, N: non terminal

list = (list)

list = list + digit

list = list - digit

list = digit

digit = 0|1|2|...|9|

T = digit

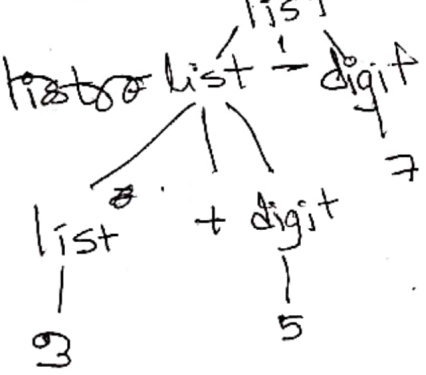
NT = list

S = list.

*, ÷

+, −

3 + 5 - 7



Alphabet

A0 (0, 1).

L0 = {0|1|100|10|---}

A0 (a, b, c)

L0 = {a, ab, bc, aaaab ---}

AL (source code of e) · L2 = {any expression of e}

list = list - digit

= list + digit - digit

= digit + digit - digit

= 3 + 5 - 7

Left Most

3 + (5 - 7) Associativity

↳ Priority.

list = list + digit

= list - digit + digit

= 3 5 - 7 + 3

Right Most

Ambiguous grammar

Find IDFT of $x(k) = \{2, 1+j, 0, 1-j\}$

We know, $X(n) = \frac{1}{N} \sum_{k=0}^{N-1} x(k) e^{j\frac{2\pi}{N} \cdot kn}$

$= \frac{1}{4} \sum_{k=0}^{3} x(k) e^{j\frac{2\pi}{4} k \cdot n}$

$= \frac{1}{4} \left[x(0) e^{j\frac{2\pi}{4} \cdot 0 \cdot n} + x(1) e^{j\frac{2\pi}{4} \cdot 1 \cdot n} + x(2) e^{j\frac{2\pi}{4} \cdot 2 \cdot n} + x(3) e^{j\frac{2\pi}{4} \cdot 3 \cdot n} \right]$

$= \frac{1}{4} \left[x(0) + x(1) e^{j\frac{\pi}{2} \cdot n} + x(2) e^{j\pi n} + x(3) e^{j\frac{3\pi}{2} \cdot n} \right]$

Syntax directed translation.

prefix
postfix
infix

postfix

3	3
+	+ 3
7	+ 37
−	− 37+
2	− 37+2

∴ 37+2−

3+7−2 Prefix

2 − 7 + 3

2	2
−	− 2
7	− 27
+	+ 27−
3	+ 27−3+

∴ +3−272

Semantic production rules

Syntax (grammar)

expr → expr$_1$ + digit.
digit → 1

Semantic (value).

expr.t = expr.t || digit.t || '+'
digit.t = '1'

String I/P:
8+5−2

7 * 3 + 5

S → aSb | c | ab

a a b c b

S → aSb
→ aaSbb
→ aacbb

acab

S → aSb
→ acb

list = list + digit

list = list - digit

list = digit

digit = 0|1|2|...|9

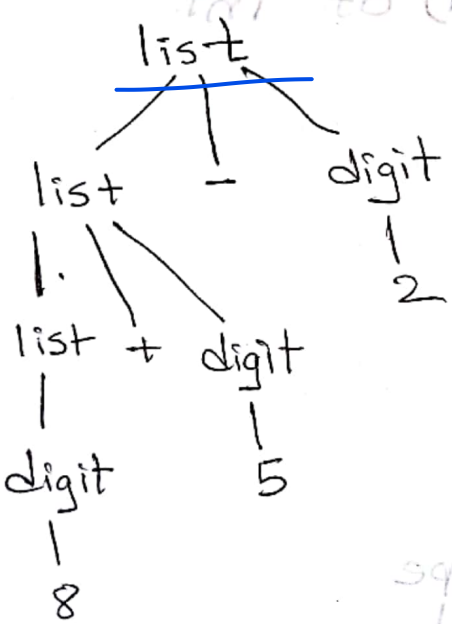
I/P string: 8+5-2

list = list - digit

= list + digit - digit

= digit + digit - digit

= 8 + 5 - 2



list.t = list.t || digit.t || '+'

list.t = list.t || digit.t || '-'

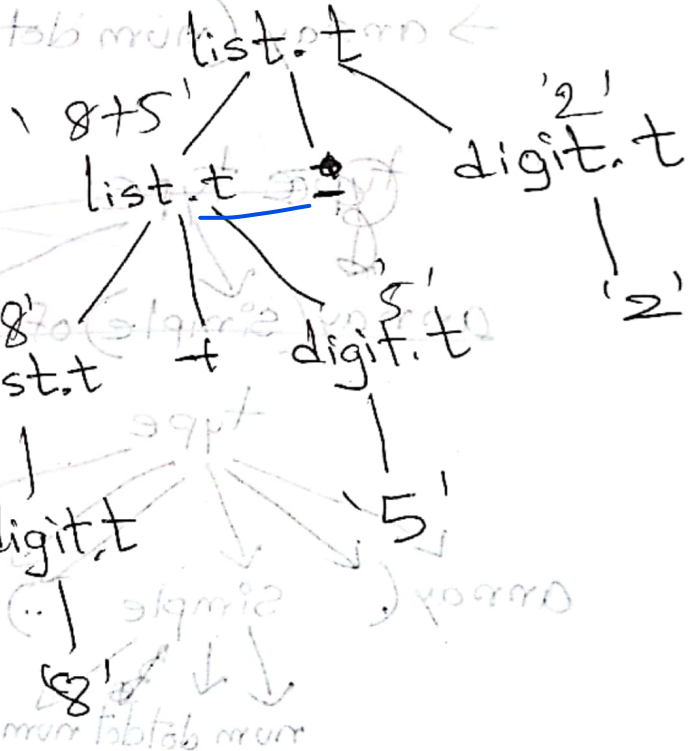
list.t = digit.t

digit.t = '0'

digit.t = '1'

digit.t = '9'

8+5-2



Parsing.

Gramm ar:

type → simple | id | array (simple) of Type

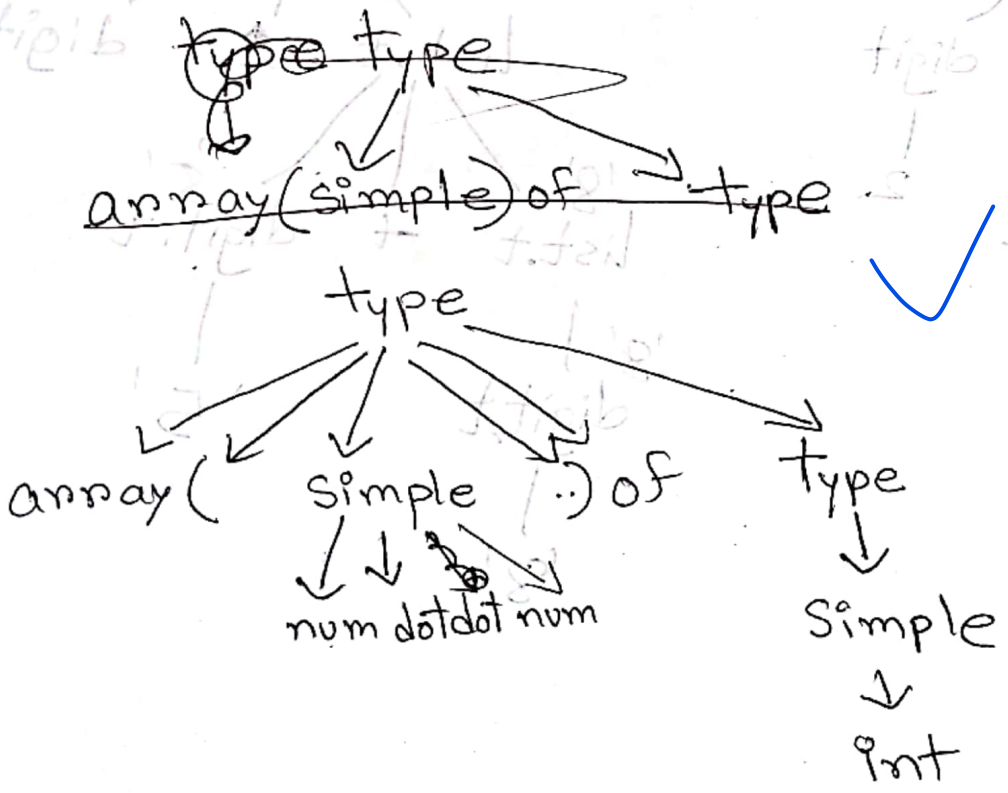
simple → int | char | num dotdot num

Input string : array (num dot dot num) of integer

type → array (simple) of type

→ array (num dotdot num) of simple

→ array (num dotdot num) of int.



Select name, marks+dp*5/15 as total_marks
from lab;

H.W. Sheet, → all data input

C.W 06-08-22
Saturday.

C.D.

Top down parser

Bottom up parser

① ② ③
G = {S → aSb | c | ab}

Input: acb, aabb

S → aSb
→ acb

S → aSb
→ aaSbb
→ aacbb

S → aSb
→ aabb

S → aSb | aabb
→ a aSbb | aabb
→ aaaSbbb | aabb
aacbb | aabb
aaabbb | aabb

this one is wrong. As here we see 2nd one is different.

S/acb ⇒ aSb | acb
~~⇒ acb | acb~~
⇒ a aSbb | acb
backtrack
⇒ acb | acb

production line
step by step
2Q.

8
10
13
16
24
27
33

18-05
-19
15

abs
→ 7.
26
34
bt. 42
~~15~~

④ bt
① b.t
② b.t
③ bt

$A \Rightarrow a\alpha$
$\rightarrow a\beta$

$A \Rightarrow abc\alpha$
$\rightarrow abc\beta$

$A|a\beta \Rightarrow a\alpha|a\beta$ (1)
$A|a\beta \Rightarrow a\beta|a\beta$

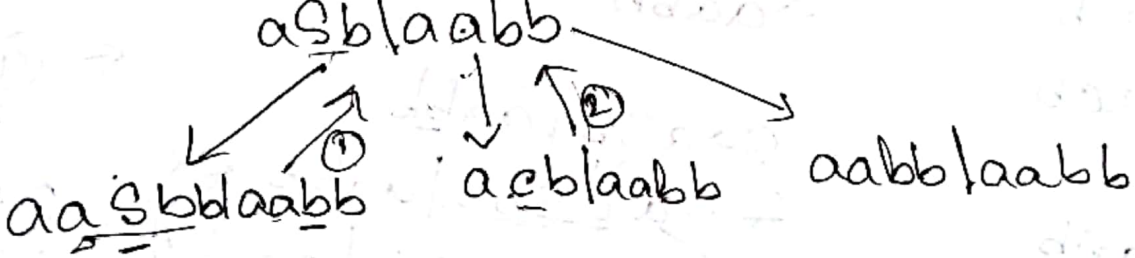
$S \rightarrow aSb|aabb$ (5) b+
$\rightarrow acb|aabb$

$\rightarrow aabb|aabb$

$S \rightarrow aSb|c|ab$

$S|aabb$

$aSb|aabb$ (1) (2)
$\Rightarrow aaSbb|aabb$
$\Rightarrow acb|aabb$
$\Rightarrow aabb|aabb$



To remove the backtracking problem we do left-factoring.

$A \Rightarrow a\Lambda$

$A \to aaB \mid aB \mid cdg \mid cdeB \mid cdfB$

$A \to aA' \mid cdB$

$A' \to bB \mid B$

$B' \to g \mid eB \mid fB$

$A \to \cdot a A'$

$A' \to \alpha \mid \beta$

$S \to aSb \mid c \mid ab$

$S \to aS' \mid c$

$S' \to \cdot Sb \mid b$

Parser

① Top down

② Bottom up

$S \to AS$ ①

$S \to AB$ ②

$A \to a$ ③

$B \to bC$ ④

$B \to dB$ ⑤

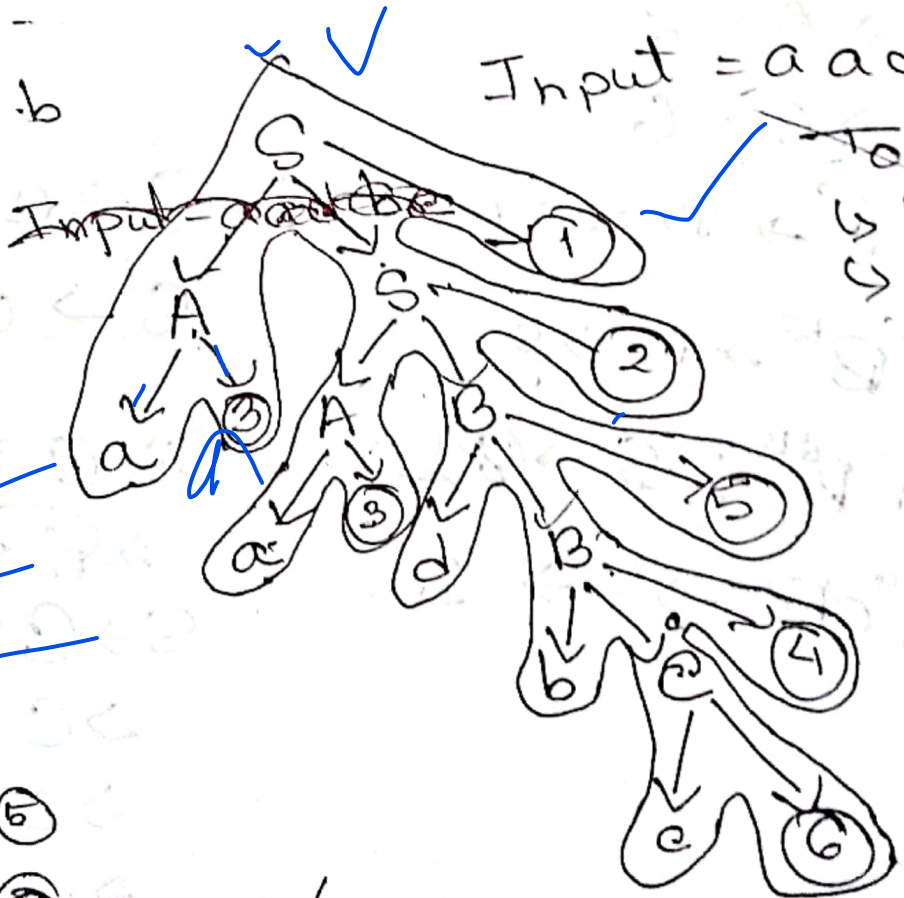
$C \to c$ ⑥

Input = a a d b c

Top down

↳ up to down

↳ left to Rig



O/P: 3 3 6 4 5 2 1

3 3 6 4 5 2 1

aadbc

Bottom-up → - down to up
Right to left

3 3 6 4 5 2 1

$S \to S\alpha \mid c \to (1)$

$S \to A\alpha$

$A \to Sb \mid c$ } (2)

left Recursion

↳ Compiler Cannot handle

But can handle Right Recursion

$a \to A = 3$

$a \to A = 3$

$c \to C = 6$

$bC^{c} \to B \Rightarrow 4$

$dB^{bc} \to B \Rightarrow 5$

$A^{a}B^{dB} \to S \Rightarrow$ ~~1~~

$A^{a}SAB \to S \to$ ~~1~~

$S \to cS'$ $S \to S\alpha$

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

$\to c\alpha\alpha$

~~$S \to cS'$~~

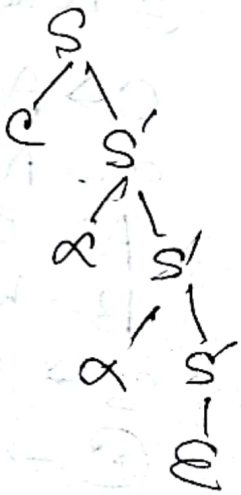
~~$\to \alpha\alpha S'$~~

$S \to cS'$

$\to c\alpha S'$

$\to c\alpha\alpha S'$

$\to c\alpha\alpha$



terminal

$S \to S\alpha \mid c$

terminal

$S \to cS'$

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

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$

$A \to SdA' \mid fA'$

$A' \to cA' \mid \varepsilon$

$S \to (SdA' \mid fA')a \mid b$

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$S \to SdA'a \mid fA'a \mid b$

$S \to fA'aS' \mid bS'$

~~$A \to Ac \mid Aad \mid bd \mid f$~~

~~$A \to AA'$~~

$S' \to dA'aS' \mid \varepsilon$

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$

~~$A \to cA$~~

$S \to Aa \mid b$

$A \to A\alpha \mid \beta$

$A' \to \alpha A' \mid \varepsilon$

$S \to Ac\alpha \mid Sd\alpha \mid f\alpha \mid b$

$S \to Sd\alpha \mid Ac\alpha \mid f\alpha \mid b$

~~$S \to dbS' \mid Ac\alpha \mid f\alpha$~~

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

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$

$A \to SdA' \mid fA'$

$A' \to cA' \mid \varepsilon$

$S \to SdA'a \mid fA'a \mid b$

$S \to fA'aS' \mid bS'$

$S' \to dA'aS' \mid \varepsilon$

DBMS

207: 1 (b)

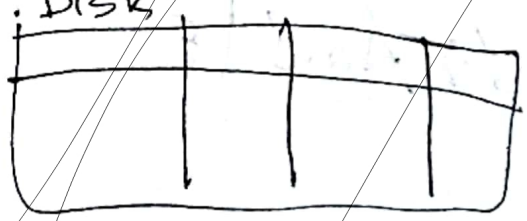
Cassette

Music	Price	Sl	Add	Duratio

author

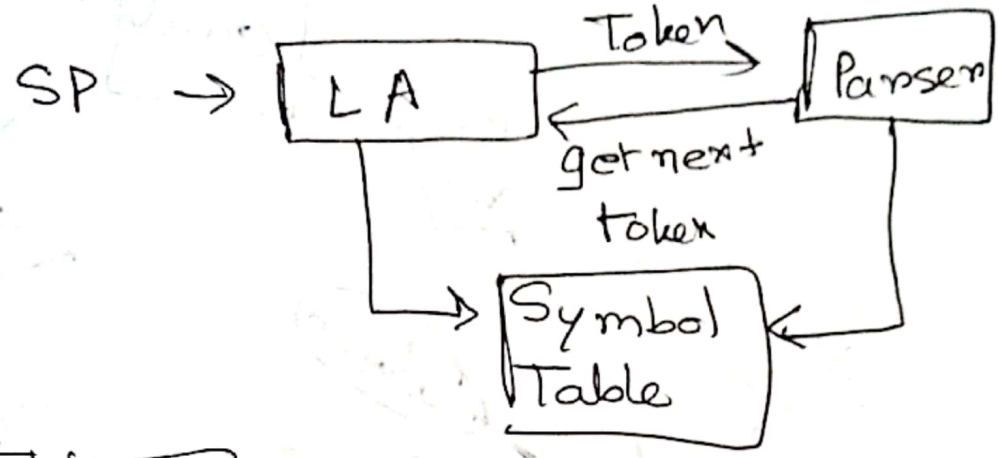
name	address	URL

Disk



10-08-22 - CD

Lexical Analyzer.



int | var1 | = | 10 | ; ← each input of statement is a lexeme.

lexeme → একটা pattern এ পাঠালে → Token
match করলে

Token types → keyword, identifier, operator, constant, special-operation (end1).

~~একটা lexem e; = pattern(token)~~

token = pattern(lexeme)

pattern decide করে source program এর lexeme কে কি mean করে।

ab = 10
id id
id const

a =
ab =
ab1=
identifier

Set of possible symbol is regular expression : possible structure / form symbols

a^* = ∈, a, aaa, a a a a — 0 or more

Clear closure

$x?$ = A x or nothing. 1 or nothing

a^+ = a, aa, aaa, — 1 or more

$x_1 | x_2$ = OR.

→ $(x_1 | x_2)^*$ = ∈, x_1, $x_1 x_2$,

x_i ∈, $(x_1 | x_2)$, $(x_1 | x_2)(x_1 | x_2)$, $(x_1 | x_2)(x_1 | x_2)(x_1 | x_2)$

$x_1 . x_2 \rightarrow x_1 . x_2$

digit = alphabet

identifier = digit (digit | number)* ← formula

a1

x_2,

val1 ;

sum ;

Regular definition:

digit → [A, Z), [a, z]

number → [0 .. 9]

stmt → if expr then start

| if expr then else stmt

| ∈

expr → term (cost) nothing term

| term operator

term → id | number.

id → [A, z], [a, z]

number → [0, 9]

~~operator~~ → {+, -, /, *, <, >, =}

! , !=

~~id → nothing~~ no

number → [0-9]

id → [A-Z, a-z]

operator = {<, >, ==, >=, <=, !=}

if → if

else → else

if{

}

else{

}

for(int a=1; a<5; a++){

}

while

Loop

Regular expression.

stmt → stmt | type | order | inde

type → stmt → for type ;

order

output → values

values → [A-Z, a-z]

order → [<, >, <=, >=]

inde → [++, --]

type → [int]

statement → while expr then output

expr → type value operator constant

output → constant.

---

type → [int]

value → [a]

operation → [<, >, <=, >=]

constant → [·10]*

while → while

stmts → for(expr op expr op expr) then
stmt

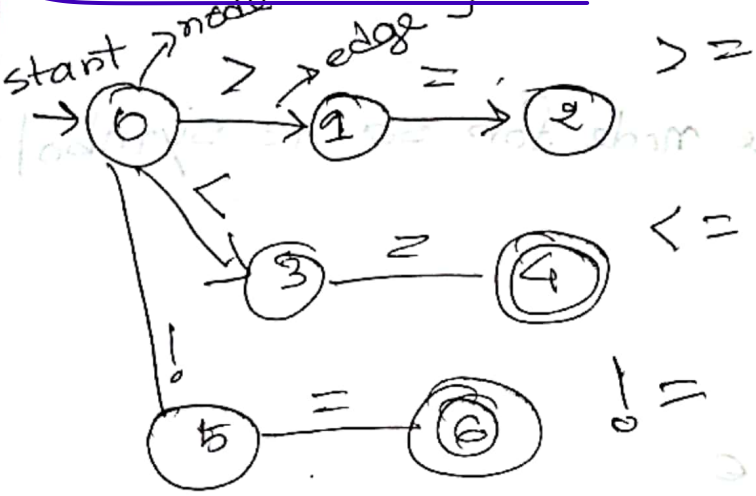
expr → term op term | term op term
op term | ε

term → digit | (digit | numb)* | numb.

>=
<=
!=

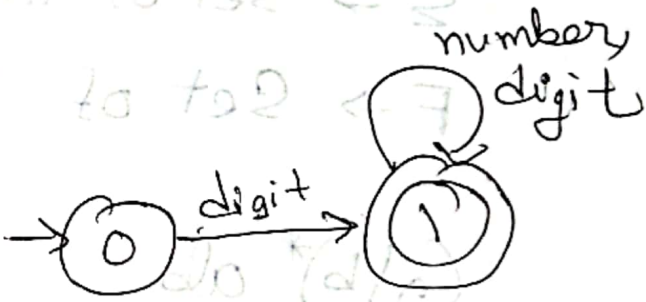
<
>

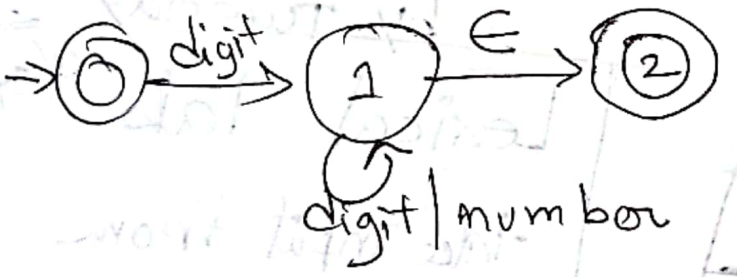
Transition diagram.



Relational Operator.

id = digit (digit | num)*





Finite Automata :- recognizer of language
~~that~~ using regular expr. to find token

NFA | DFA

can accept ε

can create multiple node for same symbol.

DFA
NFA
DFA to NFA
NFA to DFA
e-NFA to NFA
RE to NFA

NFA

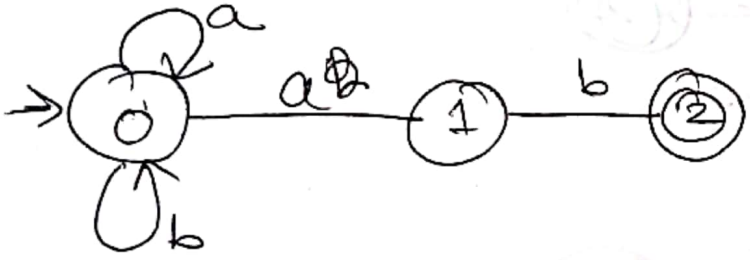
S → states

S_0 → initial state

Σ → Set of input symbol

F → Set of Final state.

$(a|b)^* ab$



$S = \{0, 1, 2\}$

$S_0 = 0$

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

$F = \{2\}$

	a	b
0	{0, 1}	{0}
1	{ }	{2}
2	{ }	{ }

by Tuesday

Lexical lab C/C++

take input from file → tokenize the input

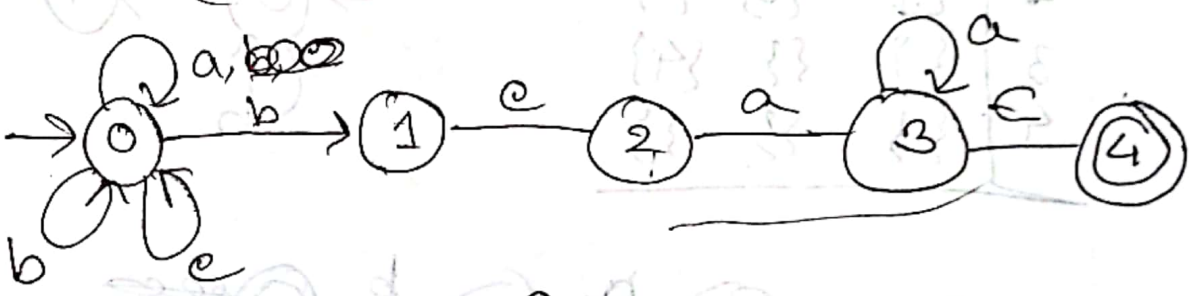
C.W

DBMS lab

select count(*) as total_students,
group by → যে আলাদা set অনুযায়ী গ্রুপ
উপর base করে।

C.W. Compiler Design 13.08.22

$(a|b|\epsilon)^* bca^+$

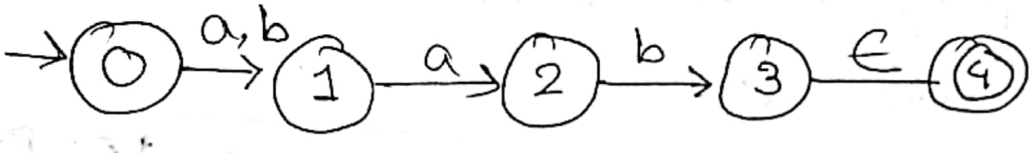


	a	b	c	ε
0	{0}	{0,1}	{}	{2}
1	{}	{}	{2}	{}
2	{3}	{}	{}	{}
3	{3}	{}	{}	{4}
4	{}	{}	{}	{}

ε এর জন্য
table এ
ir

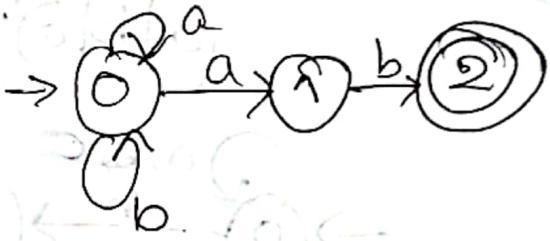
(a|b)ab

NFA

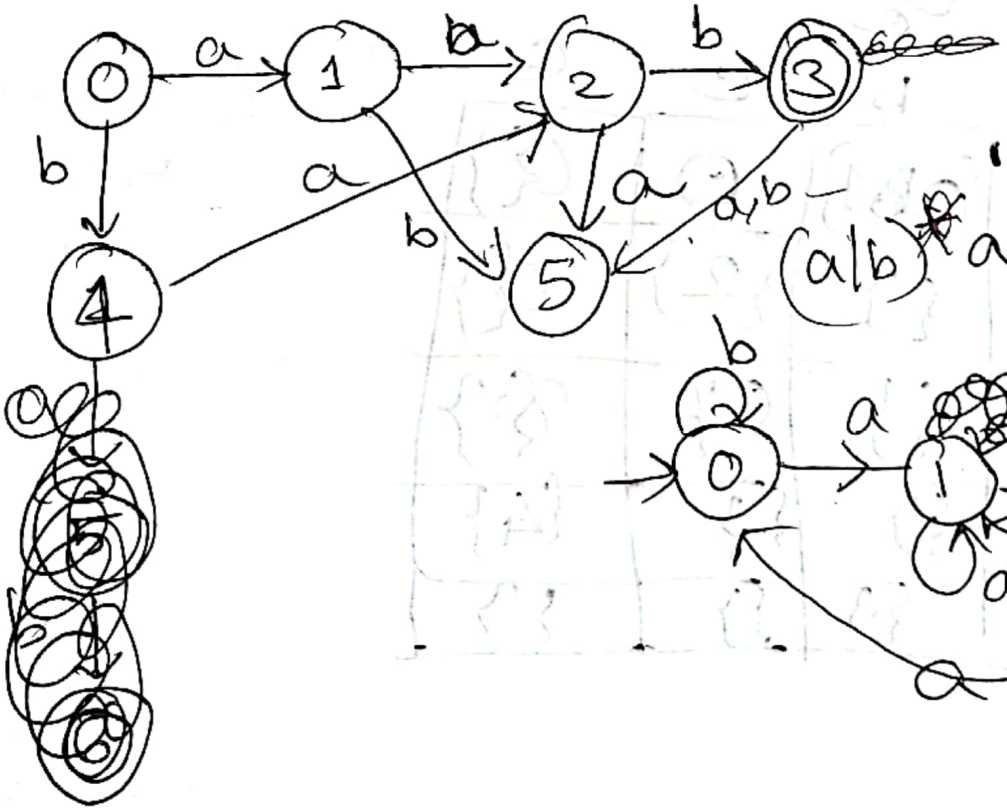


	a	b	∈
0	{1}	{1}	{ }
1	{2}	{ }	{ }
2	{ }	{3}	{ }
3	{ }	{ }	{4}
4	{ }	{ }	{ }

$(a|b)^* ab$



DFA

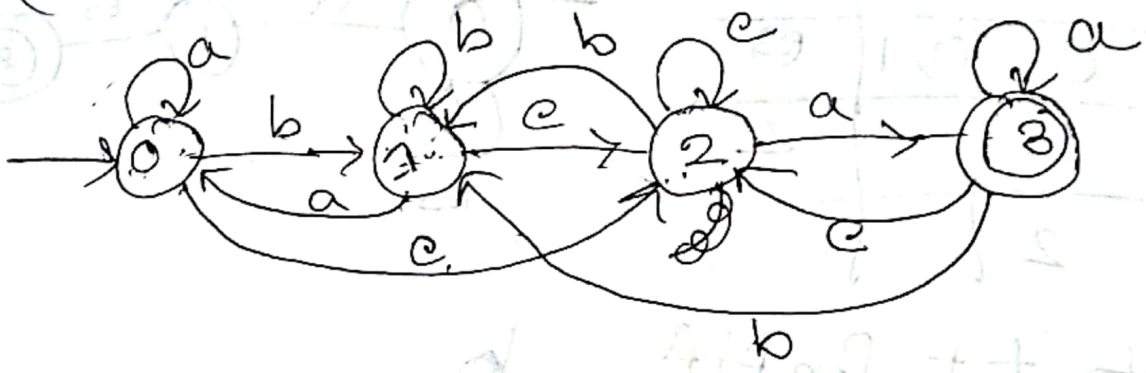


$(a|b)^* ab$

abca
abcbca
b

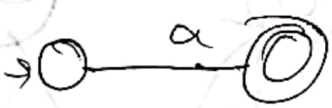
abbca

(a|b|c)* bca+



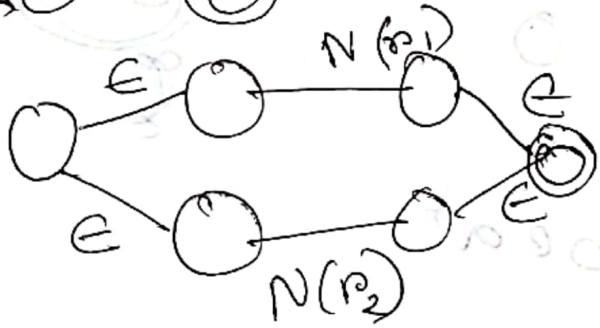
RE → NFA

r(a)

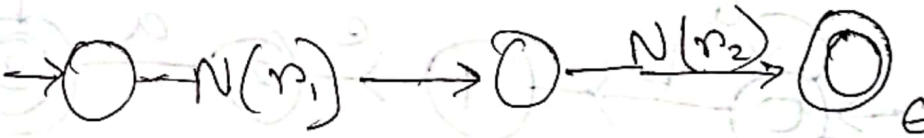


∈

$r_1 | r_2$

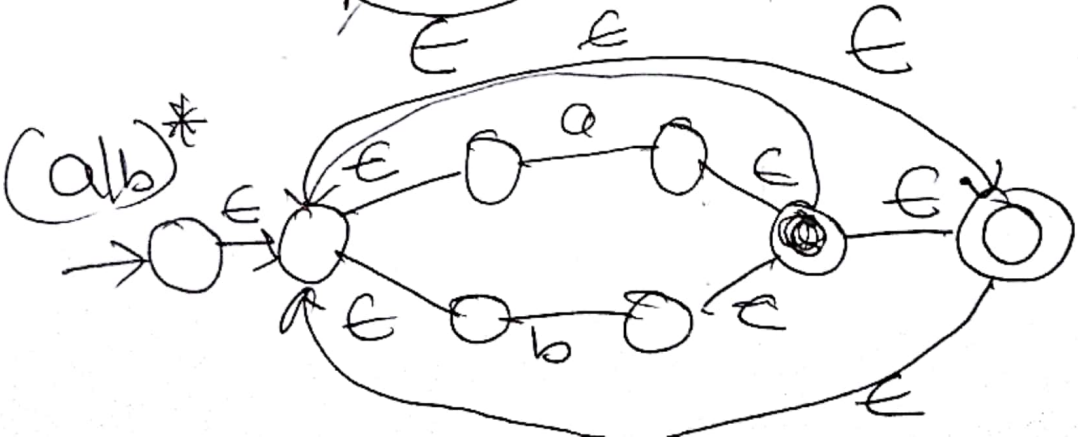


$r_1 r_2$

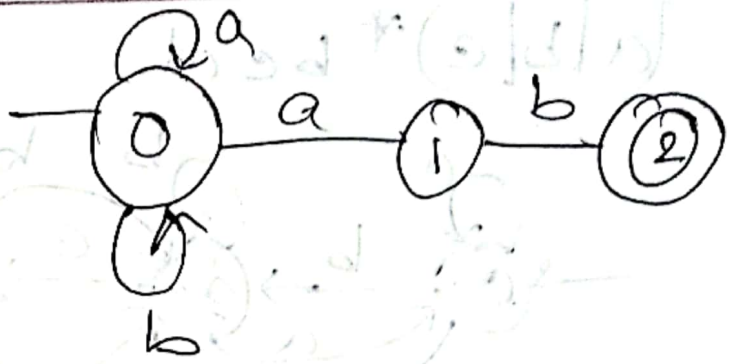


a*

(a|b)*

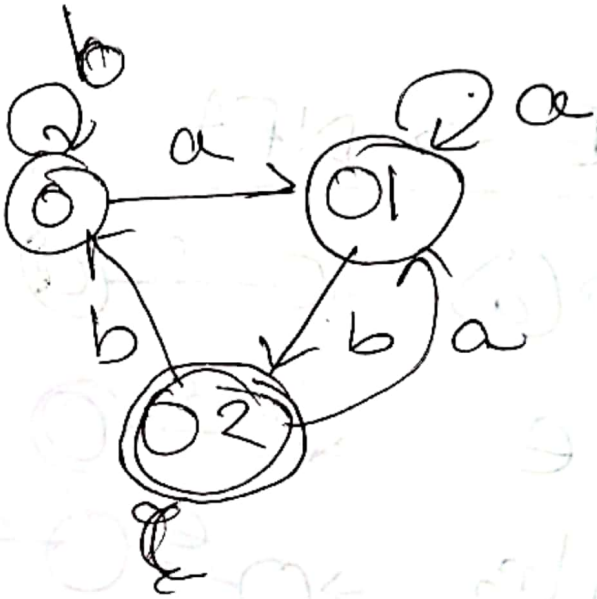


	a	b
0	0, 1	
1		2
2		

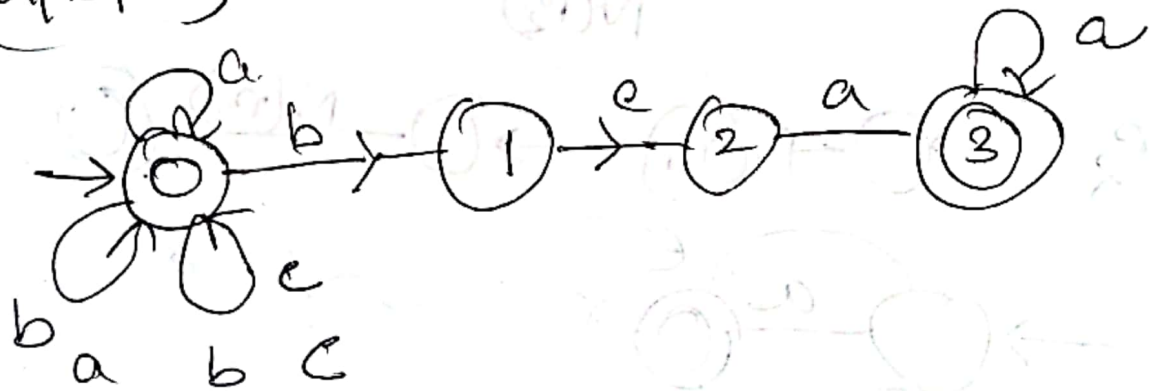


to t.t for DFA

	a	b
0	01	0
01	01	02
02	01	0

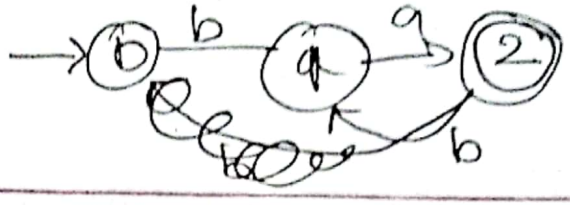


$(a|b|c)^* b c a^+$



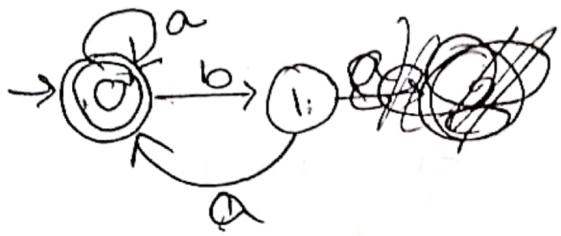
0
1
2
3

ϵ
a
aba
aababa



self

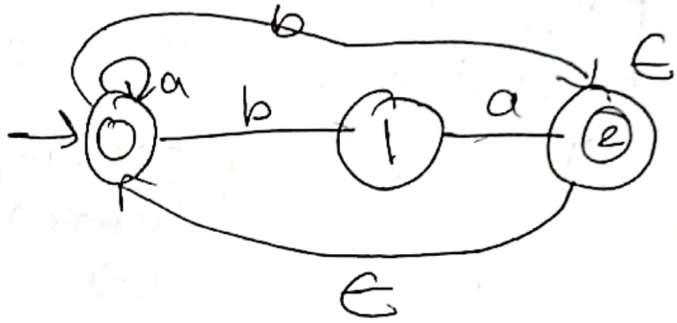
$a^*\cdot(ba)^*$



	a	b
0	0	1
1	0	{}

	a	b
a	0	1
1	a	{}

	a	b
0	0	1
1	2	
2		



C.W → DBMS -> 16-08-22 Tuesday

slide-12

Keys

Determining keys from E-R set.

$K \subseteq R$ - In relational algebra -> · U হবে।
যাতে only 1 time values থাকে।

Strong entity set.

Weak entity set

Relationship set.

C.W

CDL (1)

17-08-22

```
int main() {
    int val1, val2;
    float div;
    scanf("%d", &val1);
    scanf("%d", &val2);
    div = val1 / val2;
    printf("Result = %f", div);
}
```

int -> keyword
main -> function
val1 -> id
val2 -> id
float -> keyword
scanf -> func

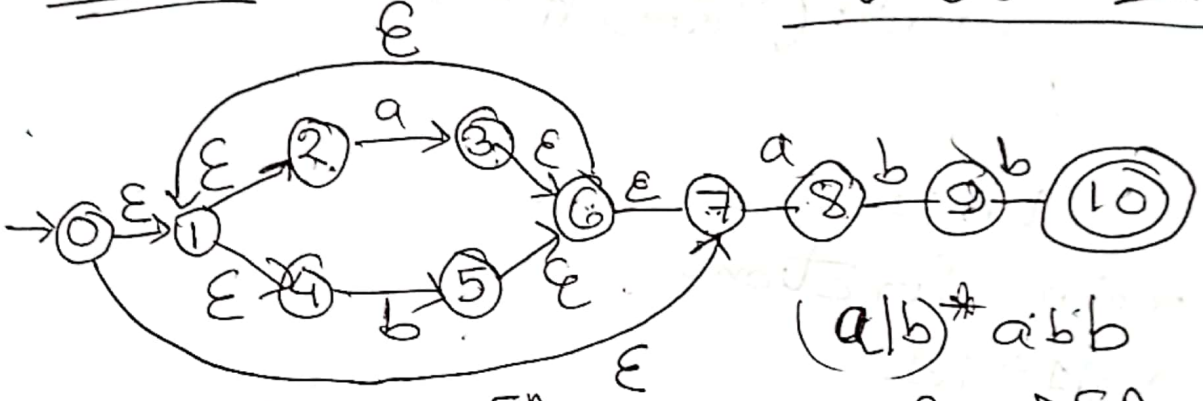
Input : A string.

a) count number of words, letter, digits & other characters
b) separate letter, digit and other characters.
c) Count how many vowels & consonent
d) Find out which vowels & consonents present.

e) ~~Dfa~~ Divide the given string into ~~to~~ two substring where one string contains the word started with vowel & other with consonant.

C.D CD

20-08-21



$(a|b)^* abb$

NFA to DFA

++ for NFA

	a	b	ε
→ 0			1
1			2, 4
2	3		
3			6
4		5	
5			6
6			7
7	8		
8		9	
9		10*	
* 10	{}	{}	{}

NOT Possible

++ for DFA

	a	b
0		

ε-closure

ε-closure({0}) = {0, 1, 2, 4, 7}

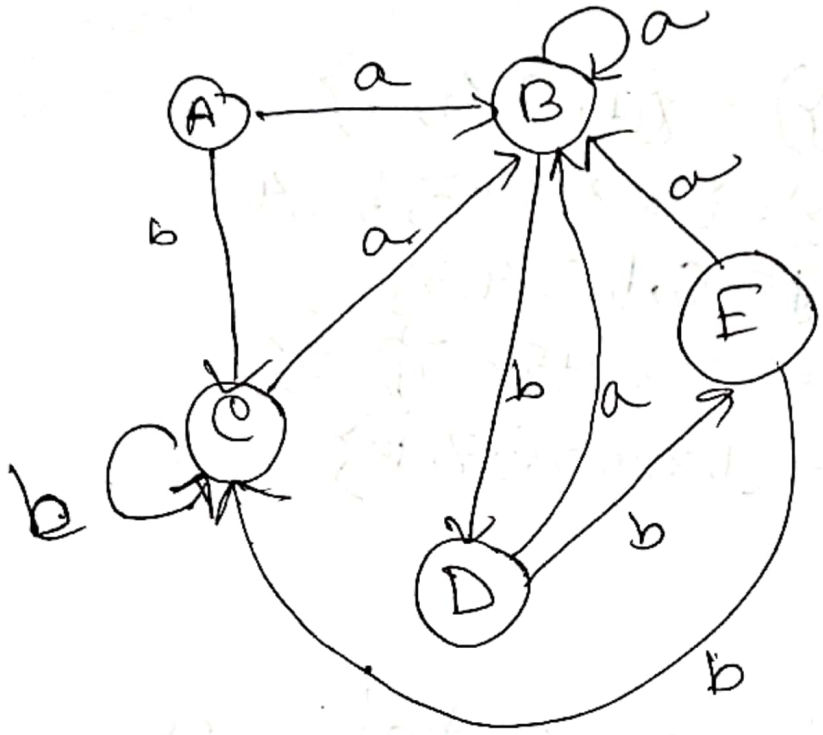
ε-colosure({3,8}) = {3, 8, 6, 7, 1, 2, 4}

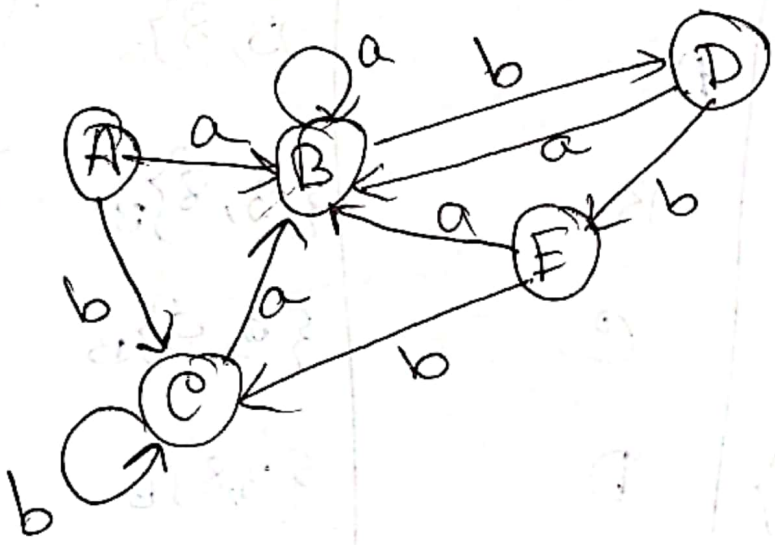
ε-closure({5}) = {5, 6, 1, 2, 4, 7}

ε-closure({5,9}) = {1, 2, 4, 5, 6, 7, 9}

ε-closure({5, 10}) = {1, 2, 4, 5, 6, 7, 10}

starting node	NFA State	DFA state	a	b
→ 0	{0, 1, 2, 4, 7}	A	{3, 8}B	{5}C
{3,8}	{1, 2, 3, 4, 6, 7, 8}	B	{3, 8}B	{5, 9}D
	{1, 2, 4, 5, 6, 7}	C	{3, 8}B	{5}C
	{1, 2, 4, 5, 6, 7, 9}	D	{3, 8}B	{5, 10}E
	{1, 2, 4, 5, 6, 7, 10}	E	{3, 8}B	{5}C





① DFA
→ 1 ~~sim~~ symbol 1 state policy.

Parsing

Recursive Decent
Predictive
Top down LL
Bottom up LR · SLR.

· syntax analyzer এ token গুলা দিলে কাজ করে।

Left Recursion

$A \to A\alpha \mid \beta$

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

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

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

left Recursion কে recursive decent parsing করতে চাই left factoring করে করতে হবে।

Left Factoring.

$A \to \alpha\alpha\beta \mid \alpha\alpha\gamma$

i/p → ααγ. In 1st option ααβ, 1st 2nd matched ও Aα ও does not. ∴ Backtrack to 1st position & then select the 2nd option ααγ.

$A \to \alpha\alpha X'$
$X' \to \beta \mid \gamma$

Absar

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

left Recursive

$\therefore \quad E \to T\,E' \quad -(1)$

$E' \to +TE' \mid \varepsilon \quad -(2)$

$T \to F\,T' \quad -(3)$

$T' \to *FT' \mid \varepsilon \quad -(4)$

$F \to (E) \mid id \quad -(5)$

Left factoring

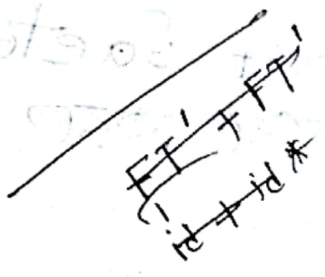
$S \to iEtS \mid iEtSeS \mid a$

$E \to b$

$S \to iEtSS' \mid a$

$S' \to \varepsilon \mid eS$

$E \to b$



I/P: id + id * id

$E \xrightarrow{1} T\,E'$

$\xrightarrow{2} T + T\,E'$

$\rightarrow T + F\,T'\,E'$ (3)

$\rightarrow T + F * F\,T'\,E'$ (4)

$\rightarrow T + F * F\,\varepsilon\,E'$ (4)

$\rightarrow T + F * F\,\varepsilon\,\varepsilon$ (2)

$\rightarrow F\,T' + F * F$ (3)

$\rightarrow F\,\varepsilon + F * F$

$\rightarrow F + F * F$

$\rightarrow id + id * id$

E'
T'
*
F → *
F
T
E → +
T' → ε
F → id
T
E

Stack implementation → Recursive descent parsing.

Procedure	Input
E	id + id * id
T	id + id * id
F	id + id * id
Push/Move	id + id * id
T'	id + id * id
E'	id + id * id
Push/Move	id + id * id
T	id + id * id
F	id + id * id
Push/Move	id + id * id

T'	id + id * id
push/move	id + id * id
F	id + id * id
push/move	id + id * id -
T'	id + id * id
E'	id + id * id

Recursive decent parsing-

① Remove left recursion

② Do stack implementation

$A \to A\alpha \mid \beta$

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

CD

Predictive Parsing.

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

$E \to T\ E'$

$E' \to +\ T\ E' \mid \varepsilon$

$T \to F\ T'$

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

$F \to (E) \mid id$

~~Predictive~~

First & Follow

-explore 1st Nonterminal.

1. 1st terminal / ε

$F = \{ (, id \}$

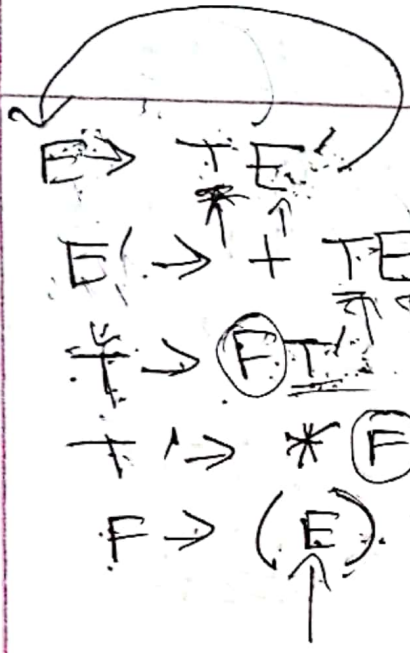
$T' \to \{ *, \varepsilon \}$

$T \to \{ (, id \}$ Ans of F as F is nonterminal

$E' \to \{ +, \varepsilon \}$

$E \to \{ (, id \}$

Follow: যার follow বের করব তার পরে কে আছে। বের করব ডান পাশ দেখে।



$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

পরে কেউ না থাকলে আগে follow এটার follow হবে

3য় follow এটার পরে থাকলে first হবে 3য় follow.

$E' \rightarrow$

Follow

$E \rightarrow \{), \$ \}$

$E' \rightarrow \{), \$ \}$

$T \rightarrow \{ +,), \$ \}$

$T' \rightarrow \{ +,), \$ \}$

$F \rightarrow \{ *, +,), \$ \}$

C.N

2 types protol UDP & TCP

TCP Transmission Control Protocol

↳ Highly reliable

↳ Flow control

↳ Connection Oriented

↳ wired so can expend by increasing

30.08.22 DBMS

30.08.22

13 netbeans → tools → plugin → php

16 New project → netbeans

24

27

33 c/xampp/htdocs

34 welcome.php

42

16

C.D

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

	First	Follow
E	(, id	), $
E'	+, ε	), $
T	(, id	+,), $
T'	*, ε	+,), $
F	(, id	*, +,), $

Left recursion remove

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

LL(1) ঃ

Parsing Table

Grammar	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)				

ε থাকলে follow set এ চলে যাব। ↑ ↑
E' এর first এ ε থাকায় তাই এর follow তে যাব।

stack	input	output
$E	id+id*id$	
$E'T	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
$E'T'id	id+id*id$	F→id
$E'T'	+id*id$	
$E'	+id*id$	T'→ε
$E'T+	+id*id$	E'→+TE'
$E'T	id*id$	
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F→id

Applica
Presented
Sessio
Transpor
Networ
Datalink
Physi

Stack	Input	Output
$E'T'	*id$	
$E'T'F*	*id$	T'→*FT'
$E'T'F	id$	
$E'T'id	id$	F→id
[illegible]	$	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	$	
$E'T'	$	T→ε
$E'	$	E→ε
$		

class

1→ 1 (12:10 -1)

3→ 1 (12:10 -1)

6→ 1 (11:20 -12:10)

13→ 2 (10:20 -12:20)

20→ 2 (11:30 - 1:30)

22→ 1 (12:30 - 1:00)

30→ 1 (

lab

17 - 2(12:10 - 2:40)

About ICE
in CSE

CD 31-08-22

C.U

LL(1) : Left থেকে 1টা করে derivation

left to right scan Left most derivative

Left factoring → Not LL(1)

S → iEtS | iEtSeS | a

E → b

~~left~~

S → iEtSS' | a

S' → ε | eS

E → b

First
i, a
e, ε
b

Follow
{e, $}
{e, $}
{t, }

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

first এর গুলা নিব, আর ε থাকলে ε এর জন্য follow গুলা নিব।

do stack implementation

LL(2)

$S \to aABc$
$A \to Abc \mid b$
$B \to d$

abbcdc

$S \to aABc$
$\to a\underline{Abc}dc$
$\to abbcdc$

$S \to aABc$
$\to aAdc$
$\to aAbcdc$
$\to abbcdc$

2 types of parse tree

$S \to aABc$
$\to aAbcBc$
$\to abbcBc$
$\to abbcdc.$

While expanding do expansion of 1 ~~terminal~~ non-terminal at a time

Shift Reduce parsing: id +id *id

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

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

shift reduce এর কাজটা একটা একটাই করে stack-এ নিবে। এরপর grammar দেখে reduce করবে

Stack	Input	Output
$	id + id * id $	shift id
$ id	+ id * id $	Reduce E→id
$ E	+ id * id $	shift +
$ E +	id * id $	shift id
$ E + id	* id $	Reduce E→id
$ E + E	* id $	Reduce E→E+E
$ E	* id $	shift *
$ E *	id $	shift id
$ E * id	$	reduce E→id
$ E * E	$	reduce E→E*E
$ E	$	Accept

non terminal →

2 types of Conflicts

(1) Shift reduce Conflict

(2) Reduce Reduce Conflict

বেশিরভাগ কম্পাইলার এই conflict ও

Operator Precedence Parsing

দুটো reduce করার ক্ষেত্রে কোনটার precedence আগে আসে।

Non-terminal. Same row left to right.

Power (↑/↑)

( )

* /

+ −

① No-terminal

(E X E) id

Operator → operand এর উপর dependent.

E → EAE | (E) | -E | id

A → + | - | * | / | ↑

g / f	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	≥	<	<	>	>
id	>	>	>	>	>	>	<	>	>
(	<	<	<	<	<	<	<	>	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

এই পাশের খুলা left এ আছে E left E আগে

$E \rightarrow E A E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid \div \mid \uparrow$

(2)

	Stack	input	comment
1	$	id+id*id$	shift id
2	$id	+id*id$	Reduce E→id
3.	$E	+id*id$	shift +
4	$E+	id*id$	shift id
5	$E+id	*id$	Reduce E→id
6	$E+E	*id$	shift *
7	$E+E*	id$	shift id
8	$E+E*id	$	reduce E→id
9	$E+E*E	$	reduce E→E*E
10	$E+E	$	reduce E→E+E
11	$E	$	Accepted.

Rules & Procedure:

① DO NOT compare with Non-terminal
If last element (right most data) is nonterminal then ignore & go to its left.

② In line 3 we skip E & compare $ with + ∵ $ < + so we shift
the condition here is - L.H.S < RHS then shift else reduce

③ In line 6, we compared + & * as + < * so, shift

④ In line 9, * is compared with $ ∴ * > $ so we reduce. Same case for line 10. (+ > $).

	id	+	*	$
id	—	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	A

Operator precedence relations.

of attribute.

CD

Top down parser.

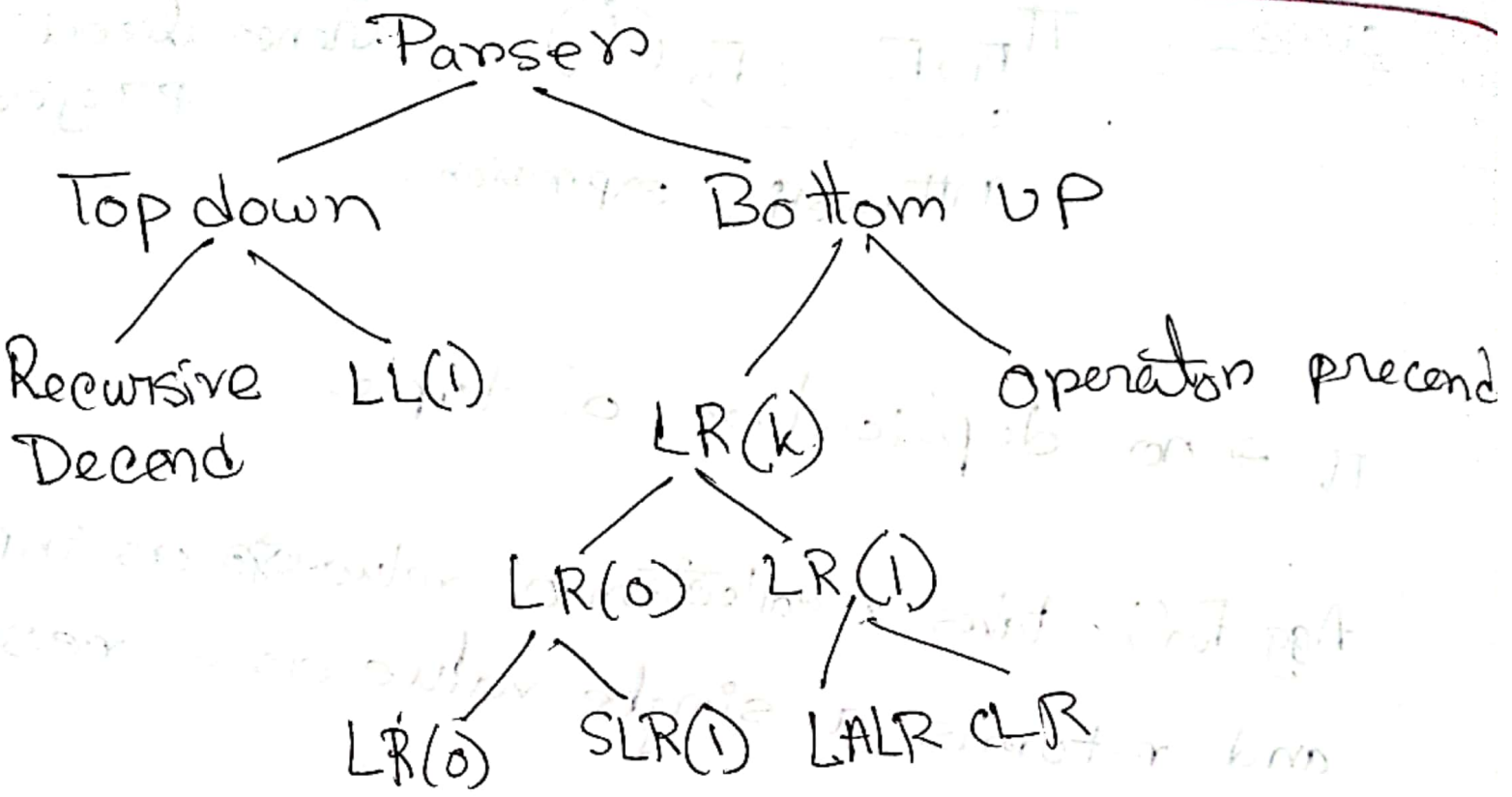
→ LL(0)

→ LL(1) → 2ta equation

LL parser → operator

→ shift

→ predicate



LR(0)

$E \rightarrow T + E$ (1) | T (2)

$T \rightarrow id$ (3)

right side reduce করে left এ যাবে

first এ explore করতে হবে, Augmentation

$E' \rightarrow \cdot E$

$E \rightarrow \cdot T + E$

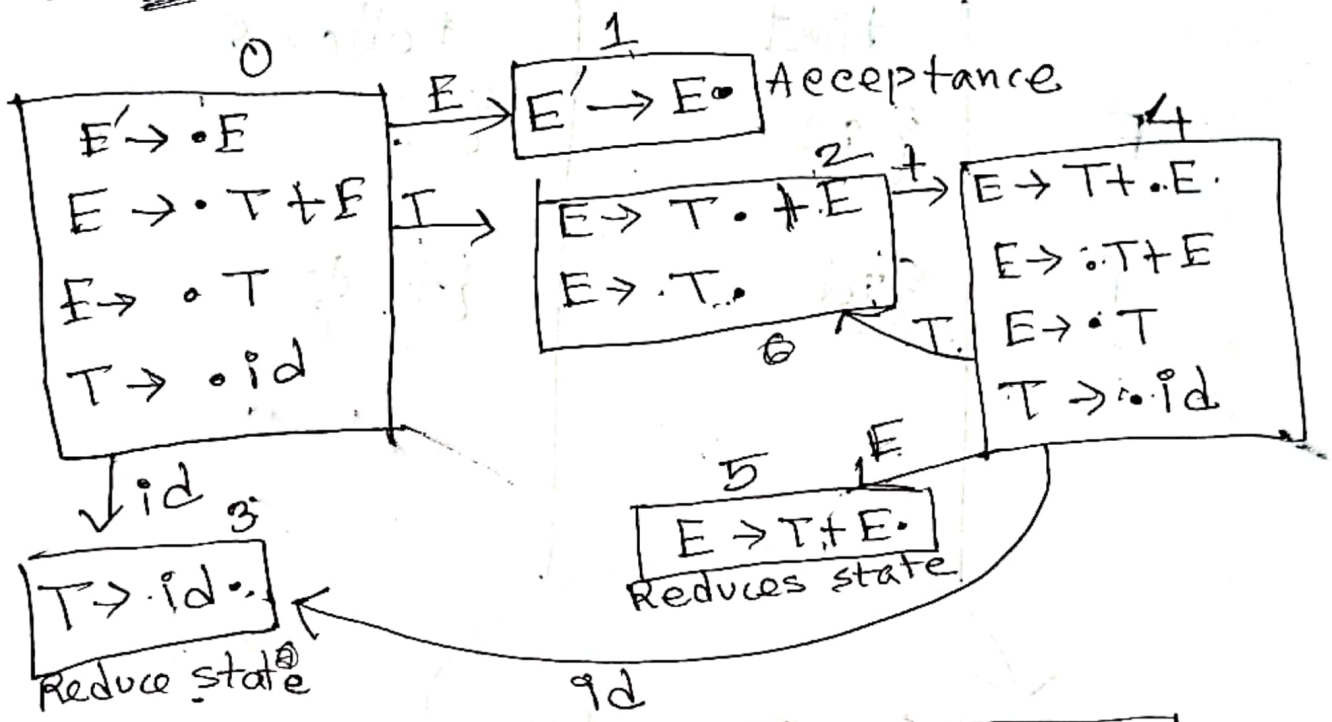
$E \rightarrow \cdot T$

$T \rightarrow \cdot id$

Non-terminal থেকে যত গুলো explore করতে হবে

$E \to E+T \;(1) \mid T \;(2)$

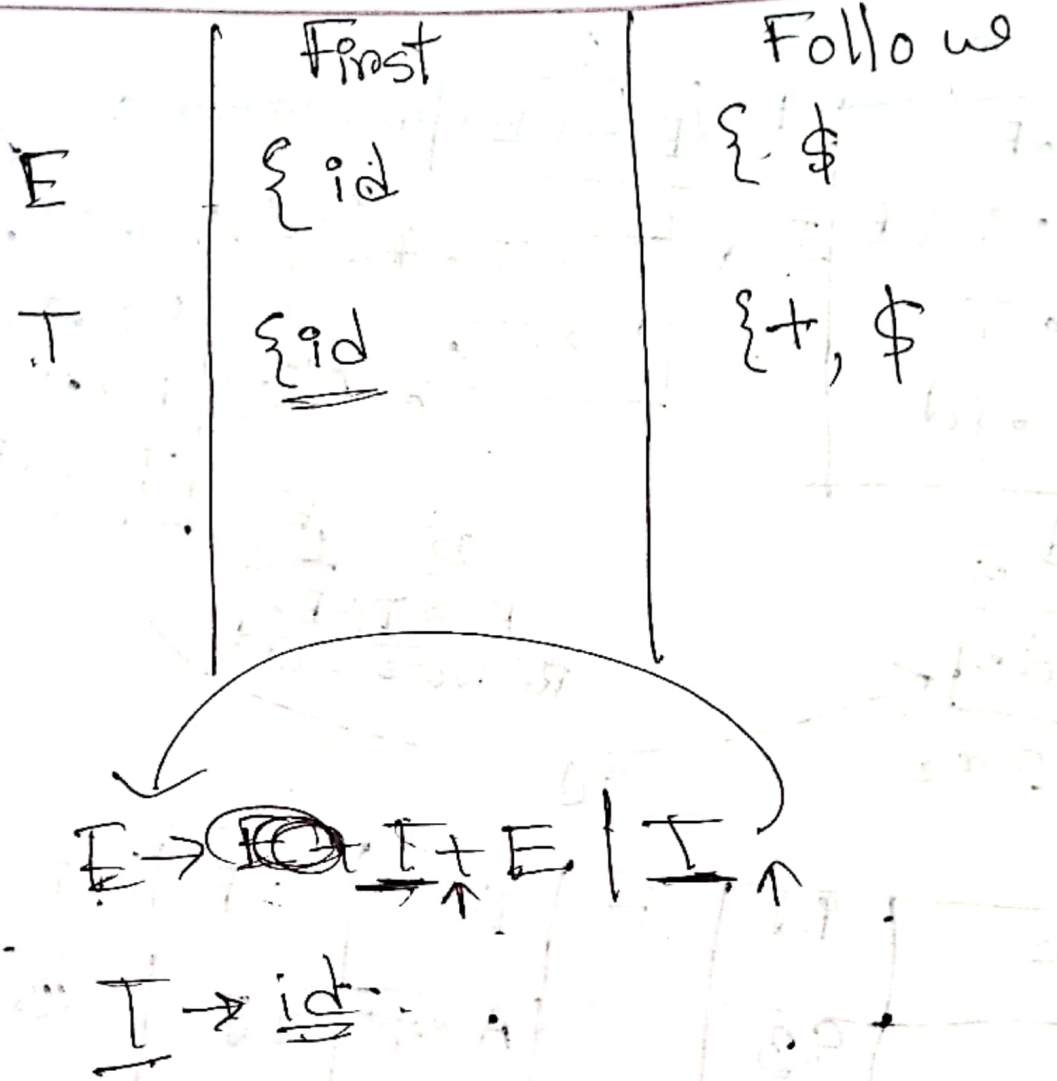
$T \to id \;(3)$



State	id	+	$	E	T
0	S_3		Accept	1	2
1					
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

LR(0) → reduce থুলা সব মাইগুলাই যাবে

SLR(1) → only E এর follow set এ যাবে, reduce only e



SLR(1) → reduction ধুলাকে follow এর মধ্যে দিও।

LR(0), SLR(1) ✱

08.09.22

Systax Directed Translation

parser.

parse tree vs syntax tree.

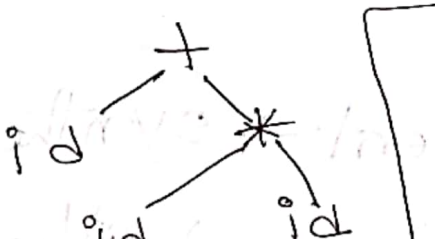
only Vadim absent

input : id + id * id

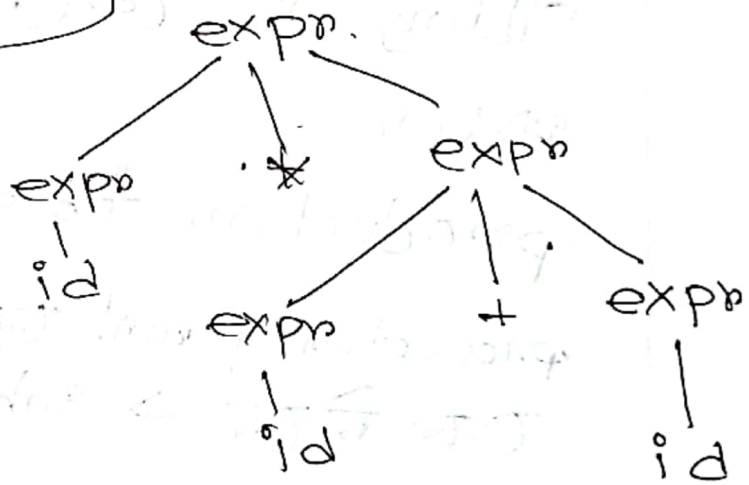
expr = expr + expr | expr * expr

expr = id

leaf → operand
root → operator.



syntax tree



Parse tree.
It is made with grammar.

S Attribute Vs L Attribute

Synthesized $S \to a\ S'\ bC$ bottom up approach
inherited

- right থেকে value নিতে পারবে না
- ~~যেকে each~~ parent থেকে directly নিতে পারবে।
- Sibling এর থেকে only left থেকে নিতে পারবে।
- production to parent → synthesized
- production parent থেকে নিলে and sibling থেকে নিলে → inherited

Synthesized → S attribute

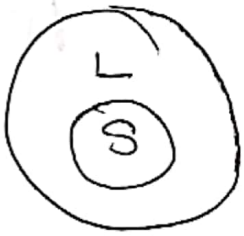
synthesized & inherited → L attribute

L

A → X Y Z {Y.val = A.val , Y.val = X.val , Z.val = Y.val}

S → MN {S.val = M.val + N.val} ↔ both S & L

M → PQ {M.val = P.val * Q.val , P.val = Q.val}



S ⊂ L

Not L

3 address code

Quadruple

Triple

Indirect triple

A = −B * (C+D)

Intermediate code

T1 = −B

T2 = C+D

T3 = T1 * T2

A = T3

Quadruple

	Operator	Operand 1	Operand 2	Result
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	T3		A

~~(4) Triple~~

Triple No need for Result.

	Operator	operand 1	operand 2
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Statement

i. ①

ii. ②

iii. ③

iv. ④

2020 1 (b) (i) compiler
(ii) Interpreter
(iii) compiler + compiler
(iv) compiler + interpreter
↳ most used

Bootstraping language to design.

**What is symbol table: data structure (↳ linear, non-linear,) that stores information generated during lexical analysis phase

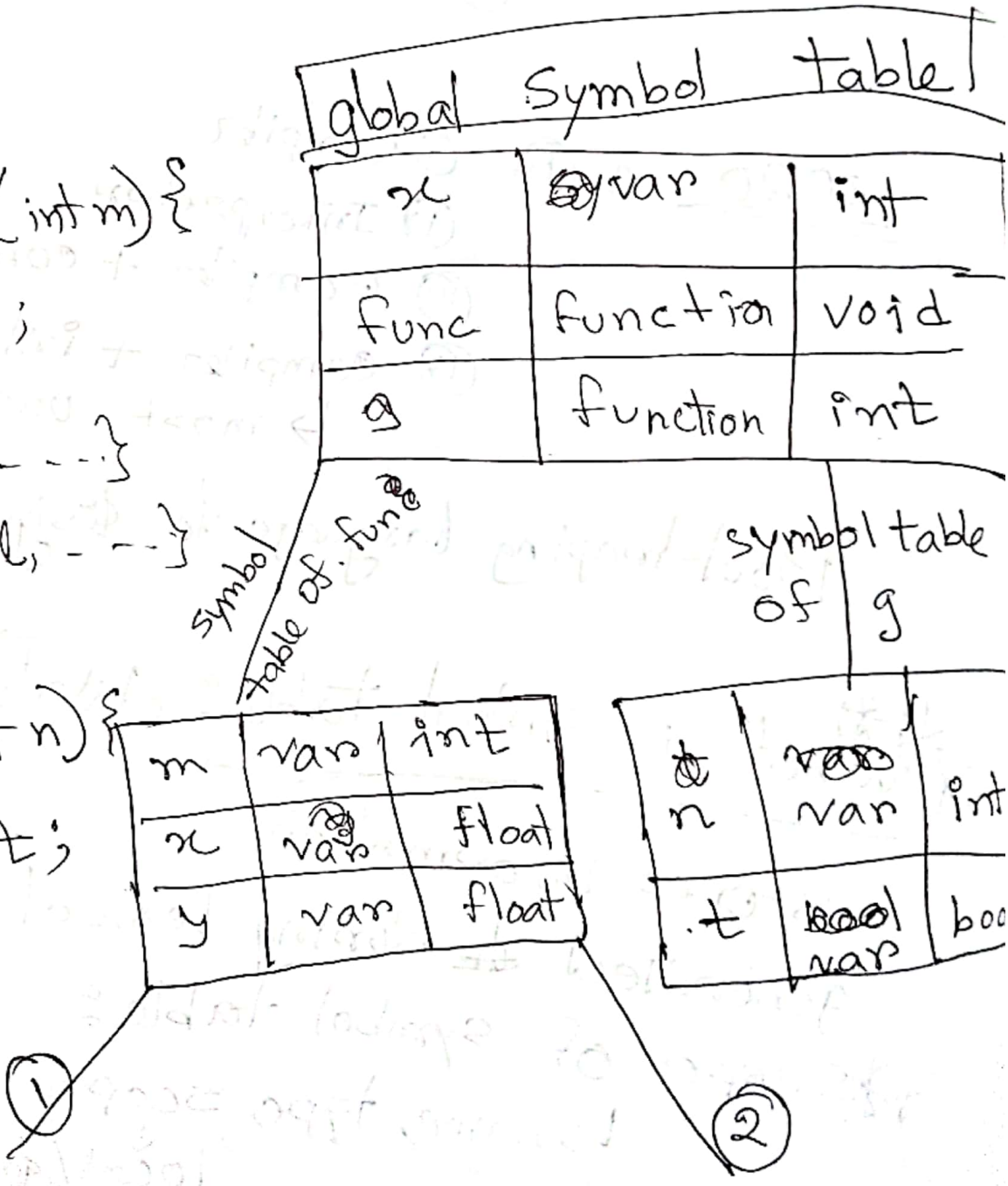
**Scope of symbol table:
↳ name, type, scope
↓
local/global

Used in full compilation process - in

lexical analysis, semantic, syntax, code-optimization, generatation in all the steps.

For generating error it has the most use.

```
int x;
void func(int m){
float x,y;
-----
{int i,j,---}
 {int x,k,---}
}
int g(int n){
   bool t;
}
```



Inner scope Symbol table

① I.S 1 s.t

i	var	int
j	var	int

② I.S.Q s,t

x	var	int
l	var	int