Lecture 01 ① 7/2/19

Grading :-
Midtearm = 20 %.
final = 60 %.
Project = 20 %.

Language :- C++ OR Java.

Compiler :-
- Is a translator.
- Read all Set and then translate.
- Compiler is also used to create executable file.

Interpreter :-
- Is a translator
- Read line by line and then translate.

Lexical Analyser :- first phase of Compoter

Lecture 02 ② 11/2/19

* The Scanner :-
→ Group of character is called Lexemes.
→ Keyword jo hote hain un ka token bhi Wohi hota hai
→ ERROR ko talash karta hai.

* The Parser
→ Make tree of token.
→ Cheack phrases → Syntax.
→ $a^* \rightarrow \epsilon, a, aa$, $a^+ \rightarrow a, aa, aaa, \ldots$.
→ Syntax k ~~and~~ ander error talash karta hai.

* Semantic Analyze
* Intermediate Code
→ int take 2 byte 16 bits.
→ double take 4byte 32 bits.

* Optimize
→ Improve the code.
→ a = a+1 => a++

Lecture 03 ③ 13/2/

* Identifier is store in Symbol table

SYMBOL TABLE.

Assignment Token was Some.

front hand and backhand Compiler is use hardware.

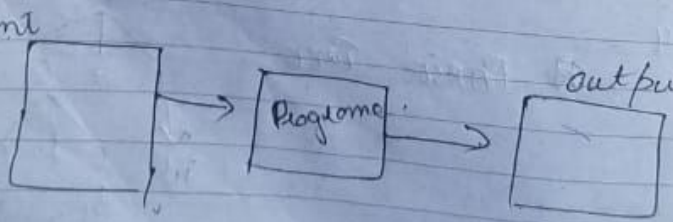
Rules. Step# 3 Rules.

* if Prec low then "Shift".
* if Prec high then "Reduce"

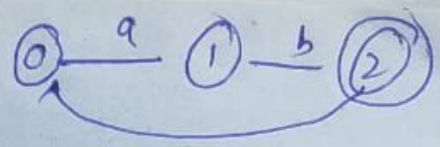
* id reduce with recepect to grammer.
* No terminal Not reduce.

CLR(1) and LAR(1) are Skip.

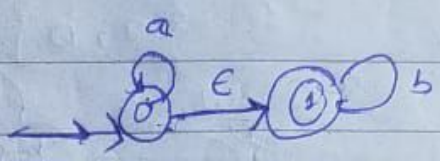
Implement ~~Lexer~~ Lexical Analysis. my language.



$(ab)^+$

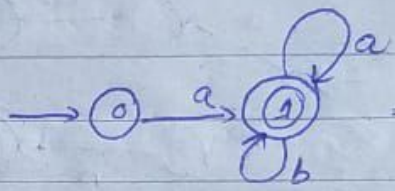


(7) $a^* b^*$



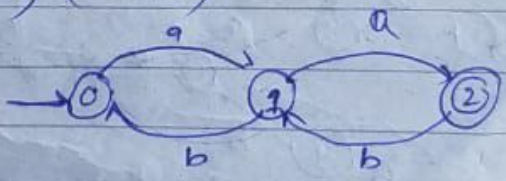
(11) $a^+ b^+$

(8) $a(a|b)^*$

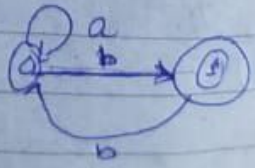


= {a, aa, ab, bbaa ...

$a\,(a|b)\,(a|b)$



(10) $(a^* b^*)^* b$



{b, ab, abbb, aabbb, aabbbb ...}

Ab

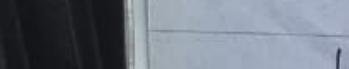
(4)

Lecture # 4 · 14/2/19

→ Scanner :- It depend on finite State Automat.. It Scans left to Right · It track the words and generate tokens. and these token's allocation are Called Lexems. Take Character as a input and from the Source code.

→ The Scanning process :-

Specify what needs to recognize : Some to are easy to identify e.g assignment operator $ Some are Complex e.g set of possible identifier. is very large and infinite Solution is : Recognize a pattern.



is main ek letter compulsary hai or us ki baad multiple letter ho Sakta hain or us k baad yeh hum. digits main Kate hain to digits bhi multiple ho sakta hai but digits per onay k baad dubara wapas. piche nahi ja sakta but letter (letter Id) ka iss Expression main. wapas onge jasakta hain

~~(a|b)* (b|a)*~~

iv) (a|b) (a|b) = {aa, ab, bb, ba}

x) $a^* bab^*$ = {ba, aba, bab, aaba, babb

xi) $a^+ b^+$ = {ab, aabb, abab — }.

xii) $(ab)^+$ = {ab, abab, ababab, — }

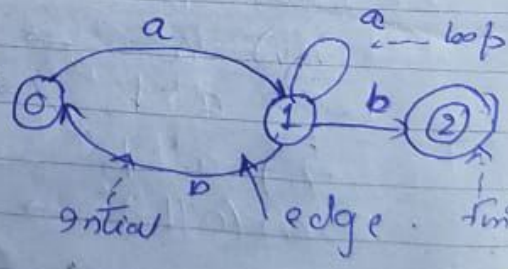
xiii) ~~$(a|b)^* (ab)^* b^*$~~ = {ϵ, ab, abbb} = $(ab^+)^*$

$(ab)^*$ = {ϵ, ab, abab — }.

lecture # 9 6/3/19

Elements of T.D.

(a|b) a^* b



① States: Position in T.D are known as circles. There are three types of state:

→ Ambiguity grammer give One or More Parse tree.
→ Operato → Operator he lekhta hai
→ Number → Convert into digit, String or list.
→ SDT → Syntax-directed translation.
→ Compiler Walk On Postfix expression.
→ 9-5+2 => 95-+2+ 95-2+
→ 9-(5+2) = 9-(52+) => 952+-
→ Solve brakets first.
→ SDD = Syntax Directed Definition.
→ SDT Syntax directed Translation Schemes (SDTs)
→

Lecture # 7 27/2/19

Repeat lecture No 1, 2

Off → 28/2/19

Lecture # 8 4/3/19

Chp# 3 - Sentic Analysis

Regular Expression

R.E is a short hand notation that specifies a set

if $L(r)$ & $L(s)$ are two language.

i. $L(r) \cup L(s) \rightarrow r/s$

ii. $L(r)\ L(s) \rightarrow rs$

iii. $(L(r))^* \longrightarrow r^*$

Eg:- Consider a language $\Sigma = \{a, b\}$.

i) $a|b \rightarrow \{a, b\}$

ii) $b(a|b) \rightarrow \{ba, bb\}$

iii) $a^* \longrightarrow \{\epsilon, a, aa, aaa\}$.

iv) $a^+ \rightarrow \{a, aa, aaa - \}$.

v) $a^* b^+ \rightarrow \{b, bb, bbb, ab, abb, aab, -\}$

vi) $a^* b^* \rightarrow \{\epsilon, a, b, ab, aa, bb, aaa, aaaa, ab, aabbb, aba, bbab, -\}$

vii) $(a|b)^* \rightarrow \{\epsilon, b, bbb, a, aaaa, ab$ (all result same as $(a|b)^+$ except ϵ)

viii) $(a|b)^+ \rightarrow$

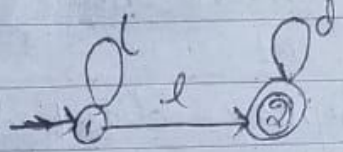
→ unique result hone zaroori hain

Lecture # 05 — 19/2/19

→ BNF - Backus-Naur Form.

→ CFG → Context free form

→ L $l^+ d^+$



a

num

num12.

→ Lecture # 06 — 21/2/19

→ Derivation for a language L and its grammer G.

→ 1+9 = (left most)

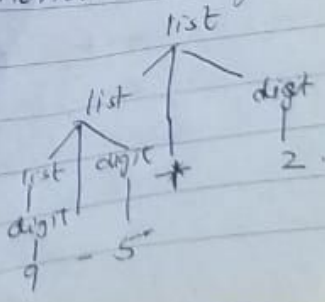
list = list + digit = digit + digit = l+dgt

= 1+9

→ ~~tag~~ leaf Node ~~but~~ are terminal hone gai

→ Parse tree.

9 - 5 + 2



1. Initial or Start State :-
   Start State → (0) OR → (1)
   OR → (1)

2. Intermediate State :
   State b/w initial & final.

3. Find State
   End State of automata (0) OR (n+1)

(2) Edge :-
   Representation by arrows, used to connect two states.

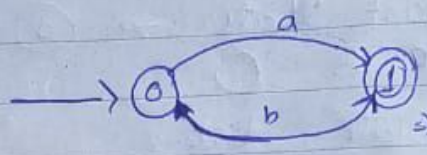
3. label :-
   Each State contains a Symbol called label.

4. Loops :-
   loops are denoted by ares and used to represent clean clauses.

fuck you!

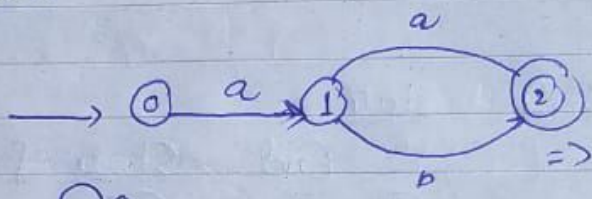
Consider $\Sigma = \{a, b\}$.

1. a|b



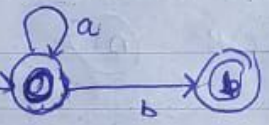
$\Rightarrow \{a, b\}$.

2. a(a|b)



$\Rightarrow \{aab, ab \dots\}$

3. a^*b

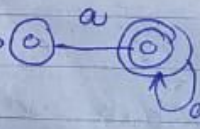


$\{\epsilon, ab, aab, aaab \dots\}$

(4) a^*

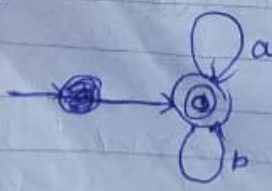
$\Rightarrow \{\epsilon, a, aa, aaa, \dots\}$

(5) a^+



$\Rightarrow \{a, aa, aaa \dots\}$

$(a|b)^*$



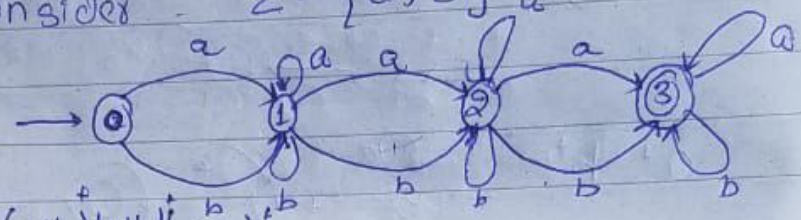
$= \{\epsilon, ab, aabb \dots\}$

Lecture # 10

7/3/19

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

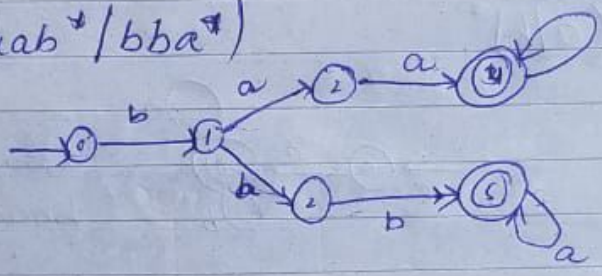
i) =



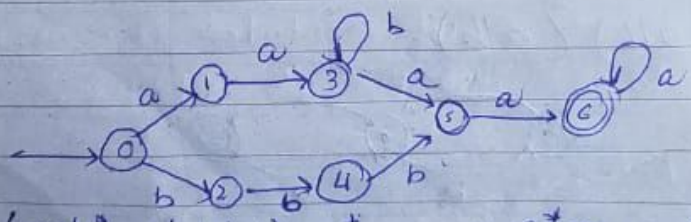
$(a|b)^+(a|b)^+(a|b)^+$

$b(aab^*/bba^*)$

ii) =

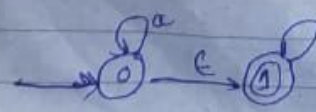


iii)



$(aab^*a \mid bbb)\, a^+ \quad \because a^*a = a^+$

iv)



a^*b^*

```
Begin
    a,b : integer
      c : String
  Input c
  Input a,b
  If (b <>0)
   a = a+b
   print c.
   print "Ans = ", a.
End.
```

Program → Begin declaration_list
Statement_list End.

declaration_list → declaration | declaration_list

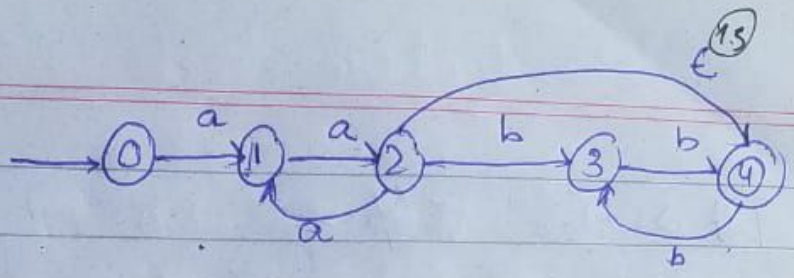
declaration → variable_list : type

variable_list → variable | variable_list, variable

variable → letter | variable alphanum

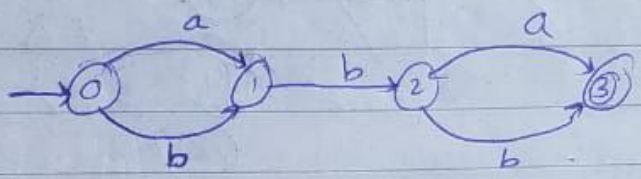
$aa^+ b^*$

v)



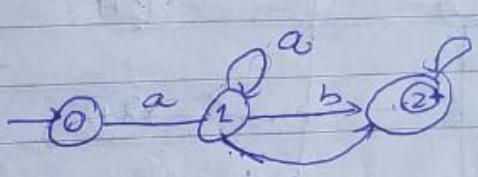
$(aa)^+ (bb)^*$

vi)



$(a|b)\ b\ (a|b)$

vii)



→ $aa^* b^*$

→ $a^+ b^*$

$a^+ b$

$aa^* b^*$ ✓

$a^+ a^* b^+$

~~$a a^*$~~

$a^+ a^* b^*$

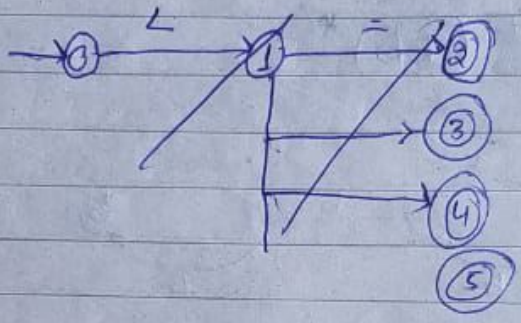
$aa^* | aa^* bb^*$

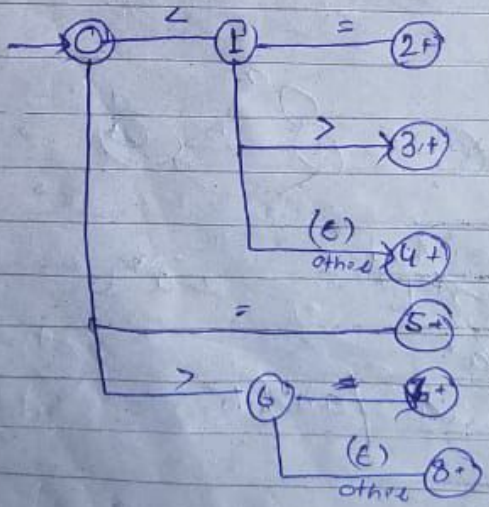
$a^+ | b^+$

$a^+ b^+ | a^+$

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

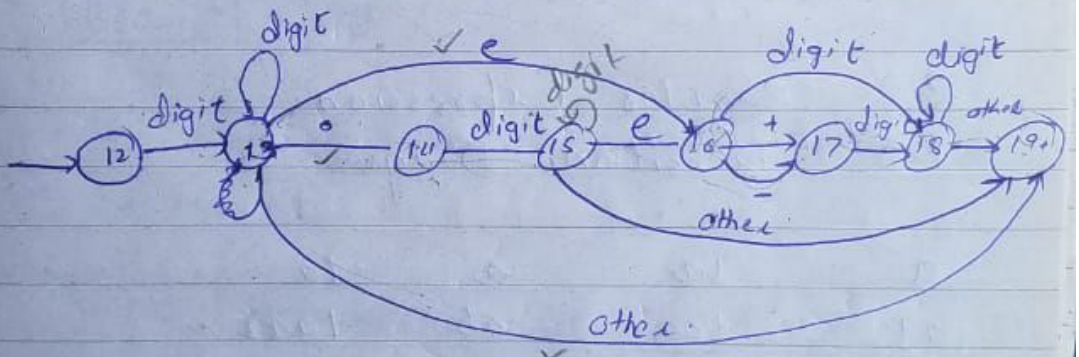
TD for Relop..





T.D for I.D / Key words

Compiled T.D for Num :-



* = Other

12.12

$12 \times 10^{+25}$

21-March-2019
Mid term

(16)

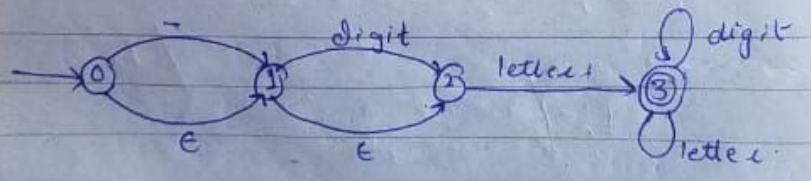
lecture # 11 13/3/19

T.D

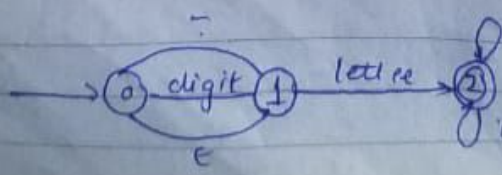
* Write R.E & T.D for 1-D which can start with (-) or digit or letter and followed by letter and followed by letter or digit in any order/sequence, but for being a valid T.D. it must contain at least one letter.

R.E Regular Expression.
T.D Transition Diagram.

-a	1a	a	-1a
-a1	1aa	ab	-1aa2
-ab1	1a1aa	ab1	
-ab1a			



R.E = (-/ϵ)(digit/ϵ)letter(digit/letter)*



→ Write CFG for the language whose specification are given as follow:-

- Consider the name of language ASL (A Simple language)
- An ASL prog start with the word "BEGIN" and finished with word "END"
- The program consists of declaration and statement
- The declaration start with variable-list followed by a "coln" and a "type"
- Type should be "integer" or "char" or "real" or "string"
- Statement should be "assignment" Statement "print statement" "input Statement" and "IF Statement"
- A Sample program of language is given as follow:-

alphanum ⟶ letter | digit.

letter ⟶ a | b | --- | z | A | B --- | Z

digit ⟶ 0 | 1 | 2 | --- | 9.

type ⟶ integer | char | real | string

statement list ⟶ assign-statement | Print-statement | input-statement | if-statement

assign statement ⟶ variable = expression.

expression ⟶ expression | bin-op expr | unary op expr | (expr) | num | variable

bin-op ⟶ + | - | * | /

unary-op ⟶ - | +

num ⟶ digit | num

Print-statement ⟶ Print Print list

Print_list → print_item | Print - list

Print_item → expression | "text"

text ⟶ text_ch ; text.

text_ch ⟶ Any Printable ASCII Char

Input_Statement ⟶ Input Variable_list.

if - Statement ⟶ if (cond) stmt | if (cond) stmt else stmt

Cond ⟶ term ~~relop~~ relop term

term ⟶ variable | num

relop → < | > | <= | >= | <> | ==

relop = relational operator.

Terminal → Not further expand.

Lecture 12

18/3/

Syntax Analyer = Parser

- Terminal leaf node pe hun gai

→ Passing main problem hogi jab ek se zayad tree hun gai

→ Ek se zyda ko loop chalta hai wo Ambiguity hota hai_

Mid Q/A

1. $(a^* bc)^*$
2. $(ab/b)^+$
3. $(a/ab)^* /ab/a)^*$

$(a/ab)^*(ab/a)^*$
$(a^*/ab)^*$
$[ab^*/a^*)^+$

1. $(101^*)^*$
2. $(.1/a)^+$
3. $(11/0)^* (0/00)^*$

lecture 13 .

[redacted]

lecture 14 3/9/19

Predictive Parsing

* Before predictive parsing use have to do pre-processing.

first $ follow

oo first $ follow for preprocessing.

eg:-

$A \rightarrow \alpha | \beta$

$A \longrightarrow \alpha\beta$

$A \longrightarrow +\epsilon T$

follow of + is E.
follow of E is T.
follow of T is empty.
First of + is empty

first of E is +
First of T is +

Rules of FIRST (it contain Terminal)

* if X is a terminal then first(X)=X
* if X is a production or non-terminal then see its production to find out follow.

Rules:-

* for production if $A \longrightarrow \alpha B \beta$ if follow of B is β but β is a non-terminal so that all follows of β is also the follow of B

Condition:-

1) if follow (B) = A, $A \longrightarrow \epsilon$ then
   ↳ follow of (B) = first (A) + follow (A)

2) $A \longrightarrow \alpha B \beta$
   ↳ follow (B) = first (B)

3) $A \to \alpha B$
follow (B) = follow (A)

Example Of First & follow

first of ϵ → follow is ϵ

$E \to TE'$
$E' \to +TE' / \epsilon$
$T \to FT'$
$T' \to *FT' / \epsilon$
$F \to (E)$
$F \to id$ → follow ϵ

1) first of (E) = { (, id }
2) first (E') = { +, ϵ }
3) first (T) = { (, id }
4) first (T') = { *, ϵ }
5) first (F) = { (, id }

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

(E)
Terminal Non terminal Terminal

Note: { As 'E' is starting symbol so it's follow is $ }

Note { ϵ then choose follow of E and first of ϵ.

Note } E' follow
Note ϵ than choose of
E }.

follow of F → } T follow, T' first.

T → F T' (follow)

T' → FT / ϵ.

(follow)

Lecture 15 8/April/19

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

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

follow E = {$,)}.
(E') = {$,)}
(T) = {+, $,)}.

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

$(F) = \{*, +, \$,)\}$.

Terminal

Non Terminal	id	+	*	(	)	$
E	$E \to TE'$	–	–	$E \to TE'$	–	–
E'	–	$E' \to +TE'$	–	–	$E' \to e$	$E' \to e$
T	$T \to FT'$	–	–	$T \to FT'$	–	–
T'	–	$T' \to e$	$T' \to *FT'$	–	$T' \to e$	$T' \to e$
F	$F \to id$	–	–	$F \to (E)$	–	–

* Fill No terminal by 1st of A is ε. then. follow (A)

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

$F \to (E)$

(27)

$E' \to TE'$

$0	ccdd$	Shift c → s_3
$0c_3	cdd$	Shift c → s_3
$0c_3c_3	dd$	Shift d → s_4
$0c_3c_3d_4	d$	reduce r_3 → d → B
$0c_3c_3B_6	d$	reduc r_2 → B → cB

$E \to TE'$

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

$F(T') \to$

$F(E) = \{), \$ \}$

$F(E') = \{), \$ \}$

$F(T) = \{ +,), \$ \}$

$F(T') = \{ +, \$$

$F(F) = \{ *,$

(Test) 28

Draw canical collection & passing tree of

(a) $E \rightarrow iEtS$
$S \rightarrow eS \mid \epsilon$
$S \rightarrow a$

(B) $E \rightarrow iEtSeS$
$E \rightarrow a$

Lecture # 16 (29) 11/4/19.

SLR (simple

E → BB. ①
B → CB | d. ② ③

Step 1

E' → E — Augment
E → BB .
B → cB | d .

Step 2 #

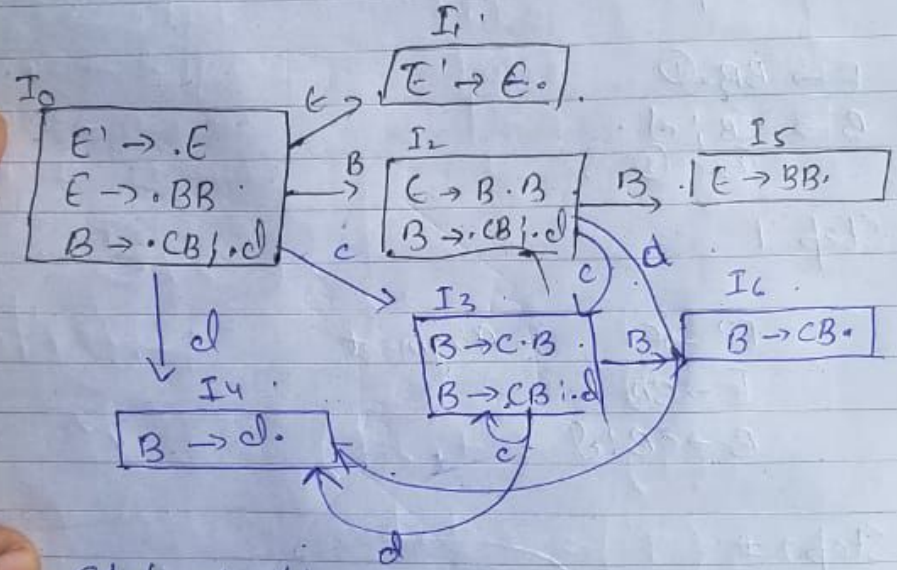
E' → .E .
E → . BB ①
B → . CB | . d
② ③

LR Rule Item represent .
L-R = left to right

Step 3 #

E' → . B .
E → . BB ①
B → . CB | . d
② ③

Conical.



Step # 4

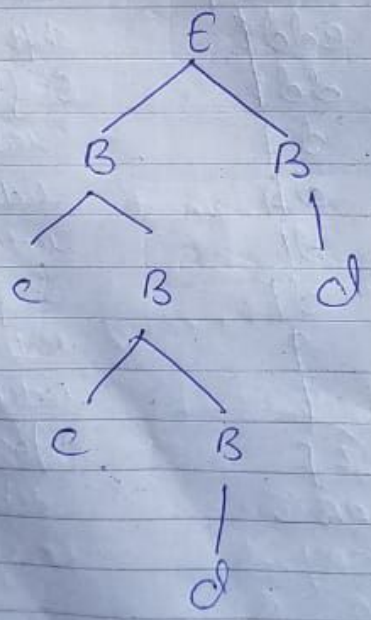
State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	γ_3	γ_3	γ_3		
I_5	γ_1	γ_1	γ_1		
I_6	γ_2	γ_2	γ_2		

Step # 5

Stack	Input	Action
$0	ccdd$	shift "c" & goto State "3"
$0c3	cdd$	shift "c" & goto S_3
$0c3c3	dd$	shift "d" → S_4
$0c3c3d4	d$	reduce r_3 B-d
$0c3c3B6	d$	reduce r_2 B→cB
$0c3B6	d$	reduce r_2 B-cB
$0B2	d$	shift "d" → S_4
$0B2d4	$	reduce r_3 B-d
$0B2B5	$	reduce r_1 E-BB
$0E1	$	Accept

Step# 6

Parse Tree

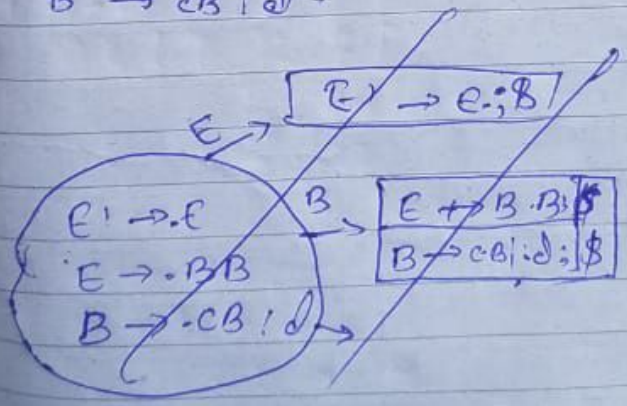


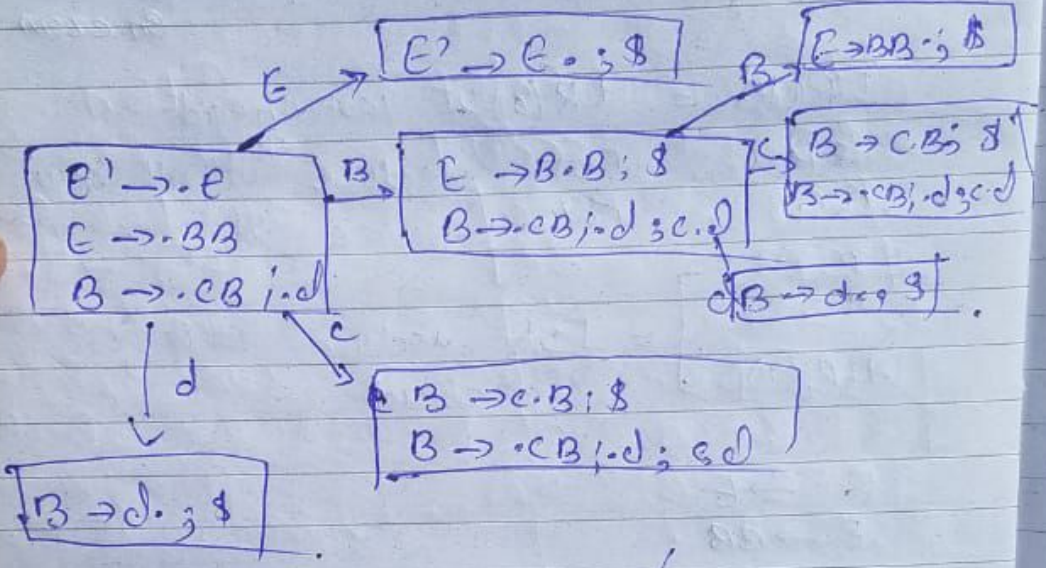
Lecture # 17. 24/4/19

CLR(1) $ LALR(1). → Not included in exam

LR(1) = LR(0) + lookahead
LR(0) = Put reduce in full row
SLR(1) = Put reduce in follow of LHS
CLR(1)
$
LALR(1) } = Put reduce only in lookahead

E' → E
E → BB
B → cB | d



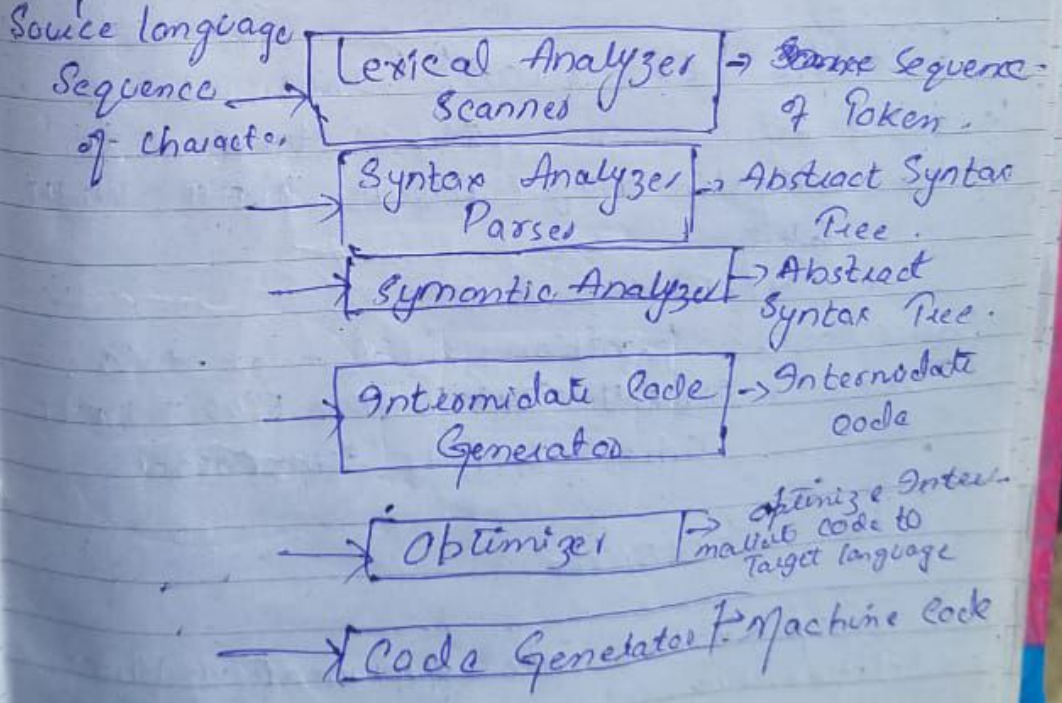

E' → .E
E → .BB
B → .CB ; .d
E' → E. ; $
E → B.B ; $
B → .CB ; .d ; c.d
E → BB. ; $
B → C.B ; $
B → .CB ; .d ; c.d
B → d. ; $
B → c.B ; $
B → .CB ; .d ; c.d
E
B
C
d
c
B
c
d

Lecture # 1

Compiler :

A Compiler a Software. That transfer computer code in programming language (the Source Code) into another computer language (the target language), C, C++, java etc.

- A compiler translate the Source language into target language.



→ Change the abstract Syntax tree

INTERMEDIATE CODE GENERATOR :→

- Translates from abstract-Syntax tree to intermediate code -
→ One possibility in 3-address code. each instruction involves at most 3. Operands.

```
temp1 = intfloat(60)
temp2 = rate * temp1
temp3 = intial + temp3
position = temp3.
```

Optimizer

→ Tries to improve code to
- → Run faster
- → Be Smaller
- → Consume less energy

The Scanner :-

→ Read character from Source program
→ Group it into lexemes.
(Sequence of character that go together)
→ lexemes corresponds to a token, Scanner return next token (Plus may be some additional information) to the Parser.
→ Scanner also discover lexical error.
(e.g erroneous character such as # in java)

Example lexems and Token.

lexems : ; = index temp 21
Token : Semi colon Assign INDENT INDENT INT-LIT.
64.32
INT-FLOAT
Source code :- position = initial + rate * 60 ;
Token = INDENT ASSIGN INDENT PLUS INDENT
Times INT-LIT SEMI COLON

The PARSER :-

→ Group token into "grammatical Phases" discovering the underlying structure

of the source program.

→ Find Syntax error.

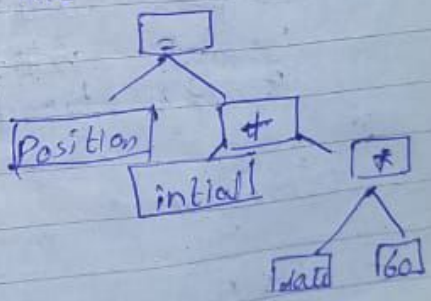
→ Might find some "Static Semantic" errors (e.g User can undeclared variable, or variable that are multiple declare)

→ Might generate code, or build some intermidiate representation of the program, such as an abstract-syntax tree.

Source Code :-

Position = intial + rate * 60 ;

Abstract - Syntax Tree



SEMANTIC ANALYZER

→ cheack more "Static Semantic" error.

Try optimize This (for Speed).

```
int Sumcale (int a, int b, int N)
{
  int i;
  int x, y;
  x = 0
  Y = 0
  for( i=0 ; i2=N, i++)
  {
    x = x+(4*a/b)*i+(i+1)*(i+1);
    n = n+b*y;
  }
  return x;
}
```

```
int i, x, y;
n = (x+(4*a/b)*i+(i+1)*(i+1)+b*y
```

Some type of optimizer

1. Constant propogation
2. Algebric ~~expression~~ simplification
3. copy propogation
4. Common Sub expression elemination
5. Dead code elemination
6. loop invariant removal.

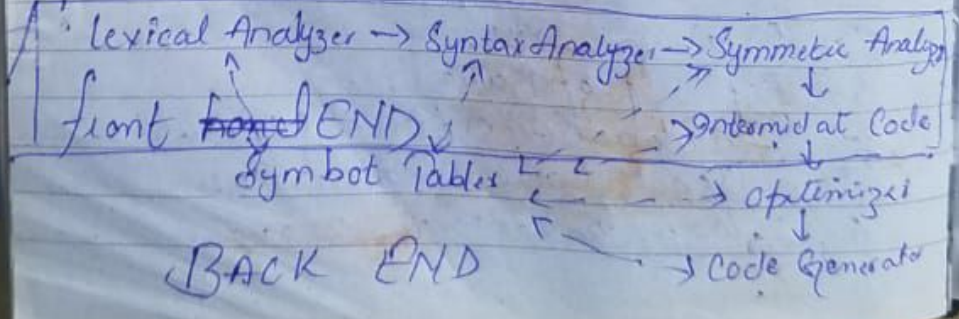
7. Strength Reduction

Code Generator

Generate object code from (optimized) intermediat code.

```
.data
c1:
  .float 60.0
.text
  l.s   $f0, data
  mul.s $f0, c1
  l.s   $f2, intial
  add.s $f0, $f0, $f2
  s.s   $f0, position
```

SYMBOL TABLES



1. Keep tract of name declared in the program.
2. Separate level for Scope
3. Used to analyze Static Symantics
   1. Variables should not be declared more than once in a Scope
   2. Variables should not be used before being declared
   3. Parameter type for method should match method declaration

Translation of a Statement

result = intial + rate * 100

↓

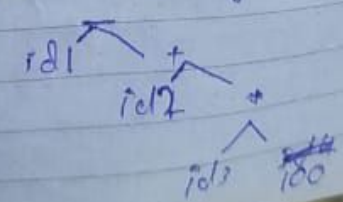
Lexical Analyzer

↓

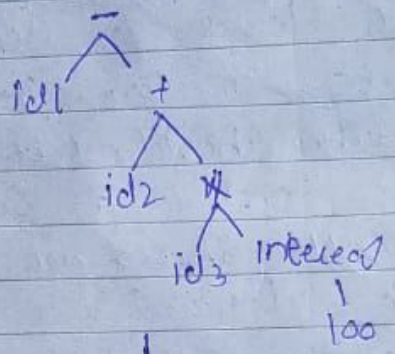
id1 = id2 + id3 * 100

↓

Syntax Analyzer



Semetic Analyzer



Intermediate Code Generator

temp 1 = inttoreal (100)
temp2 = id3 * temp 1
temp3 = id2 + temp 2
id1 = temp3

Lecture # 2

→ The Scanner :

Input = character from the Source prog —
Output = token

→ The Scanner has read some input and is on the way to identifying what kind of token has been read.

→ Scanning process

- assignment operator is easy to identify
- Others are more complex
  The set of possible identifiers is very large or even infinite

Q How would the Scanner recognize an identifier
A By Regular expression come to the rescue!

- Definition :

→ A machine that determines whether a given string belongs to a language is called a finite automation

Backus - Naur Form (BNF)
Context free Form (CFG)

→ Terminal Symbol : bold face string if, num, id.

→ Nonterminal Symbol, grammer Symbol. italicized names, list, digit, A, B

→ Grammar $G = (N, T, P, S)$

N = a set of nonterminal Symbol.
T = " " terminal Symbol, tokens
P = " " production rules
S = a start Symbol, $S \in N$

Lecture # 18 (45) 6/5/19

OPERATER PRECEDENCE PARSER.

Step.-

1. Check for valid grammer.
2. Op. proces relation table.
3. Parse given string (stack)
   E.g. - id + id + id.
4. Generate parse tree

$E \rightarrow E+E$

$E \rightarrow E*E.$

$E \rightarrow id.$

$E \rightarrow E+E \mid E*E \mid id.$

+, - Some presenside · but low presensencli for *, ÷

*, ÷ Some presencit.

Step # 2

OP PARSE RELATION TABLE.

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	error	>
$	<	<	<	Accept

RREC Rule.

1. id (a,b, etc), num → high.
2. $ → low
3. (left) + > + (Right).
4. + < *
5. (left) * > * (Right)
6. + > $
7. id > *
8. id [blank/empty] id
9. $ Accept $
10. if same operator then check left operator that is greater.

Step #

STAC
$
$id
$E
$E+
$E+id
$E+E
$E+E*
$E+E*i
$E+[E*E
$[E+E]
$E

* E+E

Step # 4

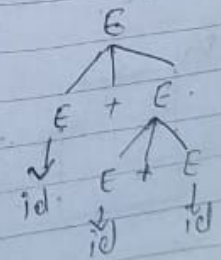
Step # 3.

STACK	Relation	Input	Comment
$	<	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 $	Shift *
$ E + E *	<	id $	Shift id.
$ E + E * id	>	$	Reduce E → id.
$ E + E * E	>	$	Reduce E → E * E.
$ E + E	>	$	Reduce E → E + E.
$ E	Accept	$	

* E + E → E

Step # 4

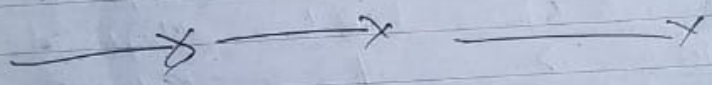
PARSE TREE



Rules. Step# 3 Rules.

* If Prec low then "Shift".
* if Prec high then "Reduce."

* id reduce with respect to grammer.
* No terminal Not reduce.



CLR(1) and LAR(1) are Skip.

Implement ~~task~~ Lexical Analysis. any language.