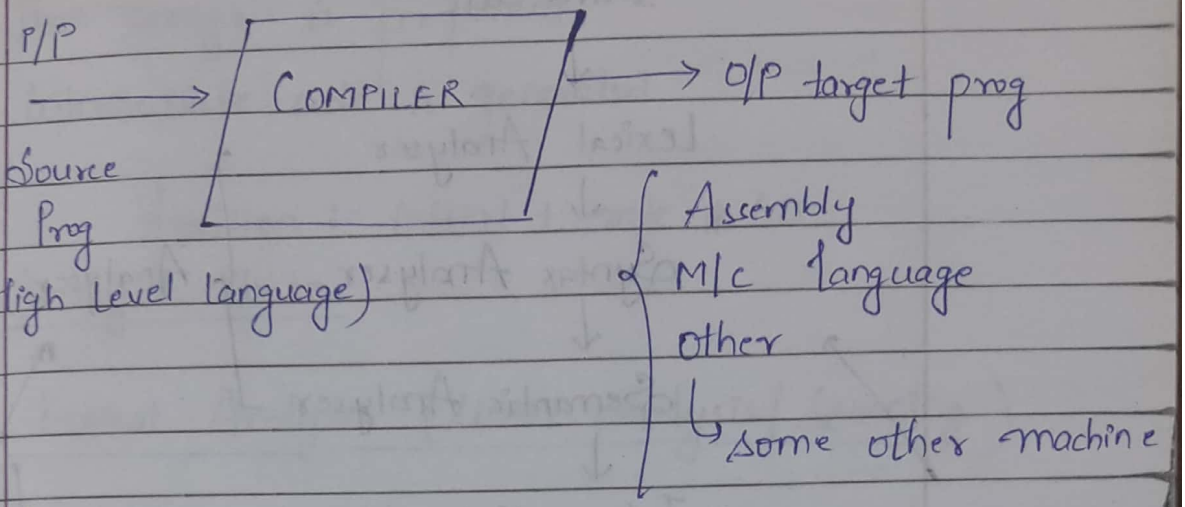
Page No.
Date : / /



Assembly :-

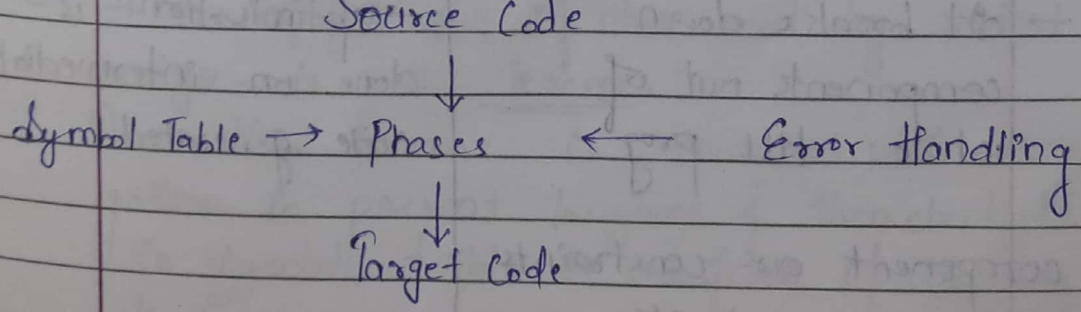
i/p → compiler → assembly language → assembler → target

Compiler v/s Interpreter

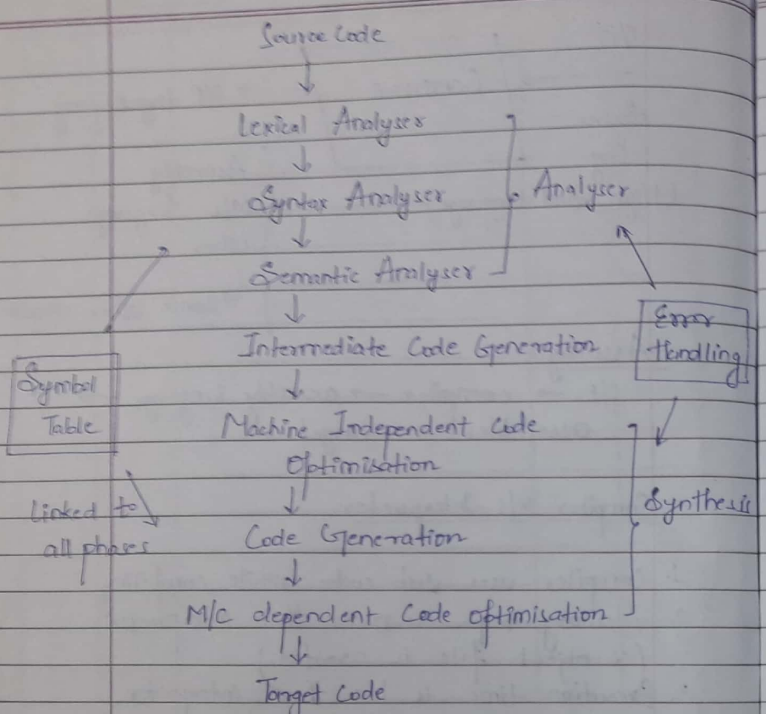
1. Compiler uses sub codes while compiling.
2. Memory req. is more than interpreter (∵ object file is created)
3. Execution time is less than interpreter

∵ Compiler has, execution as well as compilation time & interpreter has one execution time only

PHASES OF COMPILATION



Date : / /



Compiler consist of 2 main phases

Compiler

Analysis Phase	Synthesis Phase
→ It breaks down components out of whole HLL prog	→ optimisation is done on intermediate code generated.
✓ component are constraints such as int, etc. identifier, etc.	

Date : / /

✓ Then analysis is performed.

✓ intermediate code is generated

position := initial + rate * 60

Analysis Phase

① Lexical Analysis (Linear analysis/ Scanning)

```
void main ()
{
    float initial, rate, position;
    position = initial + rate * 60;
}
```

• Linear analysis :- bcz it reads each char from left to right i.e. linear scan from left to right.

→ It then converts into some meaningful token. Token are acc. to diff° languages.

→ Symbol table contains all keyword from where token is checked during token generation.

→ A meaningful set of char read by lexical analyser is a lexeme.
Then, this lexeme is matched with existing pattern in present language & then check it in symbol table to confirm if it was a keyword.

Page No.
Date : / /

Eg. void matches variable name pattern but also present in Symbol Table.

∴ It is a keyword.

→ It only validates lexeme with tokens of source language.

→ lexeme is current set checked.

→ token is stored constraints.

Eg.

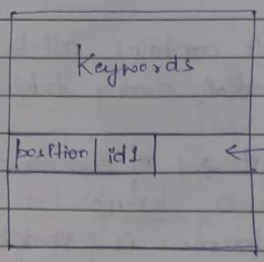
void → lexeme	; → lexeme
Keyword → token	Semicolon → token

(variable name)

→ As soon as a new identifier is found, it is entered in symbol table.

→ if encountered next time, ptr is linked to that ptr stored in ST.

Symbol Table :-



Page No.
Date : / /

O/P of Lexical Analysis

<position, id1, ptr> ↘ of ST

< t, plus sign >

Errors it will not report :-

main ( )

validate ↙ ↘ if not present then it is not a problem in this phase.

→ It only checks stored tokens.

Page No.
Date : / /

(2) SYNTAX ANALYZER

also, Hierarchial Analysis → ∵ tree is traversed.
Parsing

Src code → LA —token→ Syntax Analyser → Parse Tree

FA / RE

Grammar (CFG)
- all rules related to a particular language used.
- ✓ It helps in building meaningful sentences.

(if able to make a parse tree successfully then it is a valid sentence.)

Expression

1. Any identifier is an expression
2. Any number "
3. If exp_1 & exp_2 are expressions then,

   $exp_1 + exp_2$, $exp_1 * exp_2$, (exp_1) } are exp'.

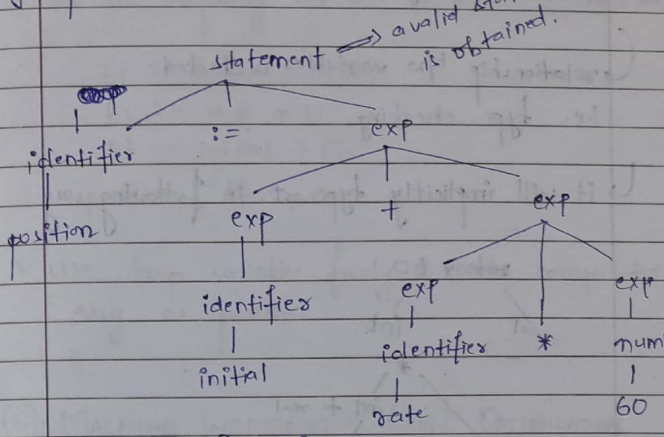
Statement

1. $identifier_1 = exp_2$
2. while (exp) do stm_2

Page No.
Date : / /

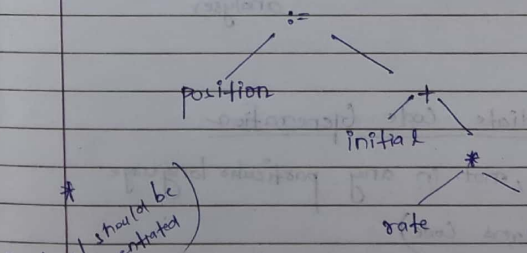
i/p string

Eg. position := initial + rate * 60



Parse Tree

Compressed representation of Parse Tree is Syntax Tree.



(Level should be differentiated *)

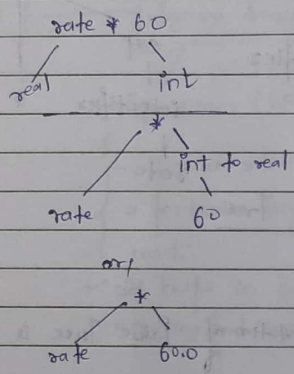
Syntax Tree

Date : / /

③. Semantic Analysis

↳ relationship b/w variables is checked
i.e. type checking

↳ it will implicitly typecast in following way



or

(annotated Parse Tree) → o/p of Semantic analyser.

④. Intermediate Code Generation

↳ a code, not in any particular language

↳ (3-Address Code)

↓

maximum 3 addresses are allowed.

Eg. LHS := = + =

① ② ③

Page No.
Date : / /

↳ We use temporary variables to ensure 3-address code

t1 := int to real (60)
t2 := rate * t1
t3 := initial + t2
position := t3

↳ use tmp variables freely to make computation as easy as possible.

⑤. MACHINE INDEPENDENT CODE OPTIMISATION

↳ This optimisation works on all machines.

t1 := rate * 60.0
position := initial + t1

Book – Ch-3

Page No.
Date : / /

⟹ LEXICAL ANALYSER

It also keeps new line character which helps in returning the line no. containing error.

Eg. `if ( a = 6)` — closing bkt, no, equal

identifier | identifier | open bkt

error handling — not matching to any pattern

Eg. abc@@# cdf

not allowed in identifier

valid longest match (abc) lexeme → identifier.

@@# → thrown away

cdf → again identifier

* panic mode ⇓ it will throw them.

Recovery mode

1. panic mode recovery
2. deleting an extra char
3. Inserting a missing char
4. replacing incorrect char by a correct char
5. transposing adjacent char

Page No.
Date : / /

Eg. 123.

matching no. but not a complete no.

∴ To continue it takes inserting a missing char recovery mode.

Eg. 123,48 → locally it will replace it with . to validate it.

(4)th recovery mode.

Eg. E 9 −4 → locally handeled. E replaced by 9 E −4 (exponent)

(5)th recovery mode..

Terms related to Automata

1. Symbol → any char 0, 1, a, b, # -- etc.
2. (Σ) Alphabet → set of symbols
3. String → finite seq of symbol
4. Language → set of strings generated by a regular expression.
5. (abc) prefix (n+1) if n symbols including a, ε, ab, abc

6. Suffix (n+1) c, bc, abc, ε

7. proper prefix ⇒ abc i.e. w ≠ string not included. (Here, w = abc)

8. proper suffix ⇒ Ng.

precedence: * > · > +

· — concatenation

+ — union

ϕ → null set containing ● { }

ε → empty string.

Regular definition

regular expressions are assigned a name which is used, while writing R.E. containing those exp^n.

Eg. / ⇒ same as +

letter + digit only with first letter.

R.E. = (a|b|c|---|z|A|B|---|Z)(a|b|c|---|z|A|B|---|Z|0|1|---|9)*

(i/=)

converting to definition.

letter → A|B|C---|Z|a|b|---|z

digit → 0|1|2|---|9

Now,

identifier → letter (letter | digit)*

Eg. (number)

exponent, fraction,

digit → 0|1|---|9

sign → +|-

digits → digit digit* or (digit+)

op_fraction → ε|. digits

op_exp → ε|E(+|-|ε) digit

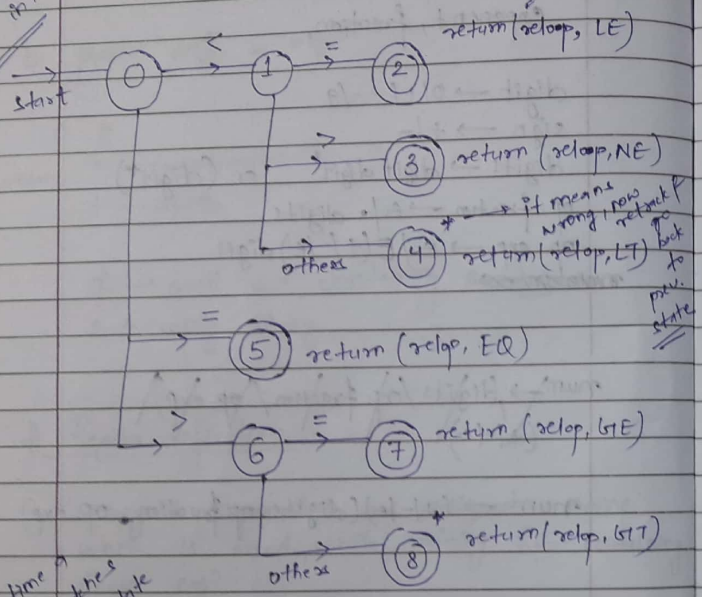
num → digits / op_fraction / op_exp / (+|-|ε)

num → (+|-|ε)(digits . op_fraction . op_exp)

Page No.
Date : / /

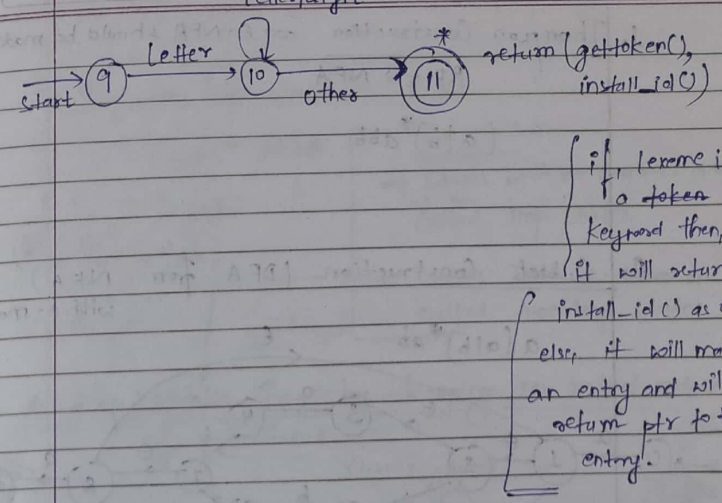
Transitional diagram

(1). relational operators allowed in mysql <, <=, >, >=, =, <> (not equal to)

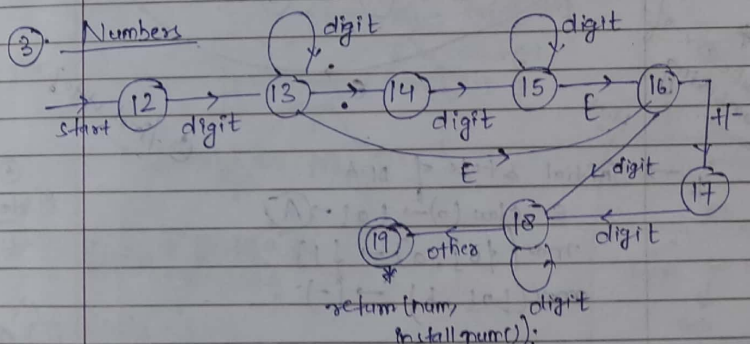


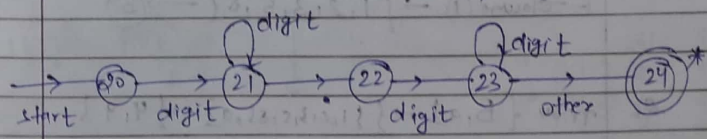
Every time a lexeme matches a starting at state by (6).

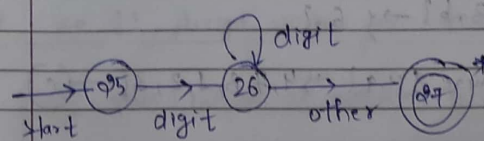
(2). Identifier → Starting with alphabets only, followed by a numeral or alphabets.



(3). Numbers







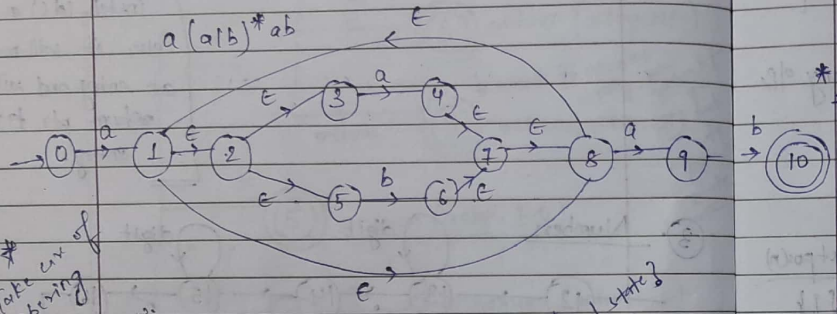
Regular Expression to NFA

1. Thompson Construction → ε-NFA should be made
   RE → NFA

   $(a|b)^*abb$

2. Subset Construction (DFA from NFA) with ε-moves

$a(a|b)^*ab$



* Take care of numbering

Hint:

① final state

→ initial state of DFA

ε-clos(0) → {0} → A

mov({0}, a) → {1}

mov({0}, b) → {-}

ε-closure(1) → {1, 2, 3, 5, 8} → B

mov{B, a} → {{1,2,3,5,8}, a} → {4, 9}

mov{B, b} → {6,}

ε-closure {4,9} → {4, 7, 8, 1, 2, 3} → C

* if directly mentioned then draw NFA w/o ε.

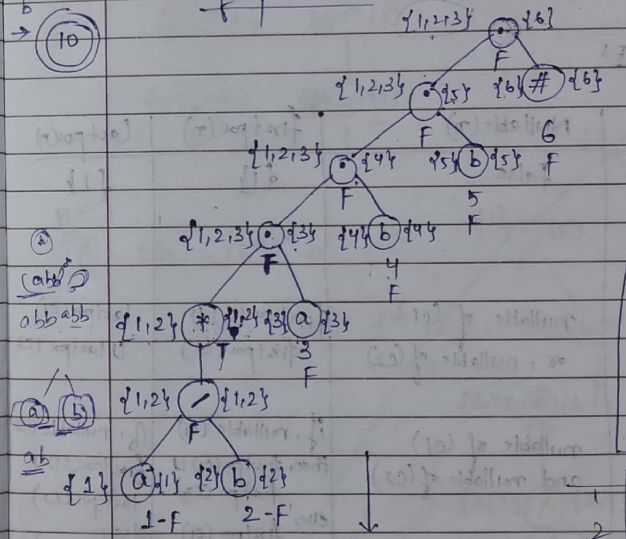
3. Regular Expression to DFA

$(a|b)^*abb\#$

In NFA :-
- ⇒ States with no ε-moves are called Imp. states.
- ⇒ States with ε-moves are called non-Imp states.

① follow this dir^n. if parenthesis comes it stops. create

* Step I : Make Tree by keeping in mind order of precedence



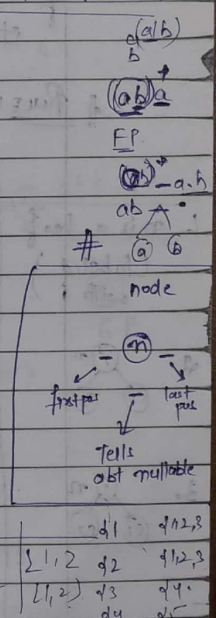
(a|b)*abb#

abb abb

ab

* Step II : Assign unique pst. to leaf nodes.

Every node (interior or exterior) is a expression.



	d1	d1,2,3
2	{1,2} d2	d1,2,3
	{1,2} d3	d4
		d5
		d6

Page No.
Date : / /

Step 3: Calculate following fn for the tree

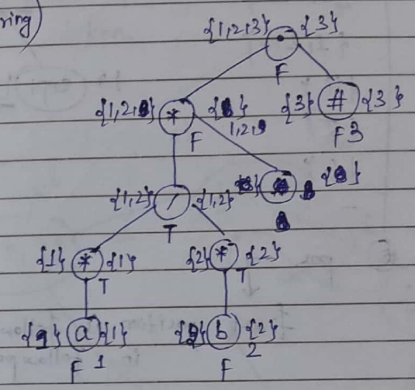
1. Nullable → calculated for each and every node
2. firstpos → calculated for each node (first symbol in a generated string)
3. lastpos → " (last symbol in a generated string)
4. follow → for some nodes only.
   it belongs to ⊙ → concatenation node &
   ⊛ → clears node

→ if a language generates 'ε' along with any o/p, it is called nullable.

RULES

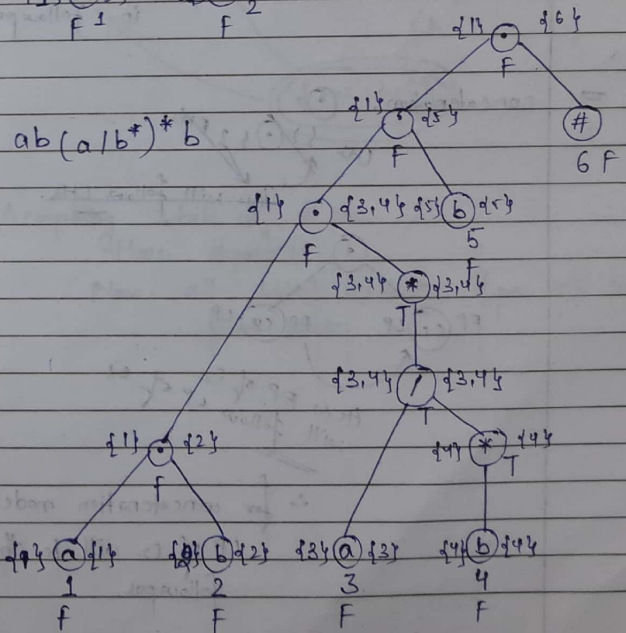
		Nullable(n)	firstpos(n)	lastpos(n)
1.	n is a leaf (labeled with i)	false	{i}	{i}
2.		nullable of (c1) or, nullable of (c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
3.		nullable of (c1) and nullable of (c2)	if, nullable(c1) then, firstpos(c1) ∪ firstpos(c2) else, firstpos(c1)	if, nullable(c2) then, lastpos(c1) ∪ lastpos(c2) else, lastpos(c2)
4.		True	firstpos(c)	lastpos(c)

Q. Eg. $(a^*/b^*)^*\#$



H.W.

Q. $ab(a/b^*)^*b$



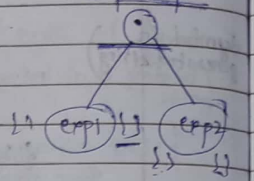
Page No.
Date : / /

find out positions

→ follow is ~~calculated~~ for all nodes but they are only calculated for ⊙ and * nodes.

• (a|b)*abb#

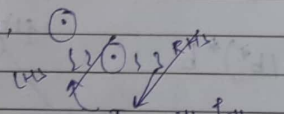
pos	follow
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4}
{4}	{5}
{5}	{6}
{6}	–

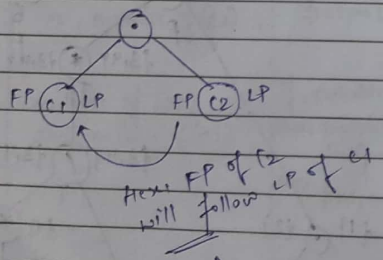


⇒ follow of * pos. →

first position is followed up in followpos.

⇒ concatenation, ⊙



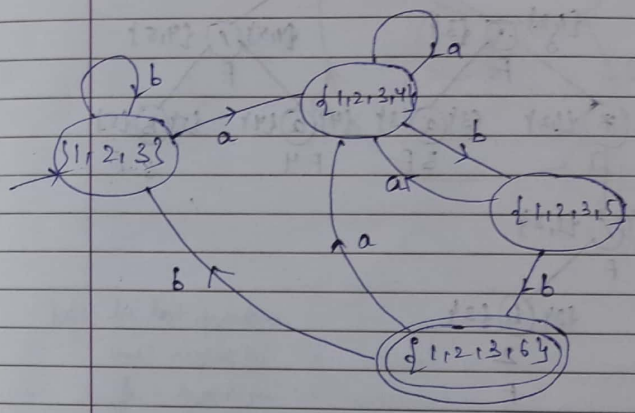


∴ for concatenation node, LP of c2 will be the followpos.

Page No.
Date : / /

Rules :-

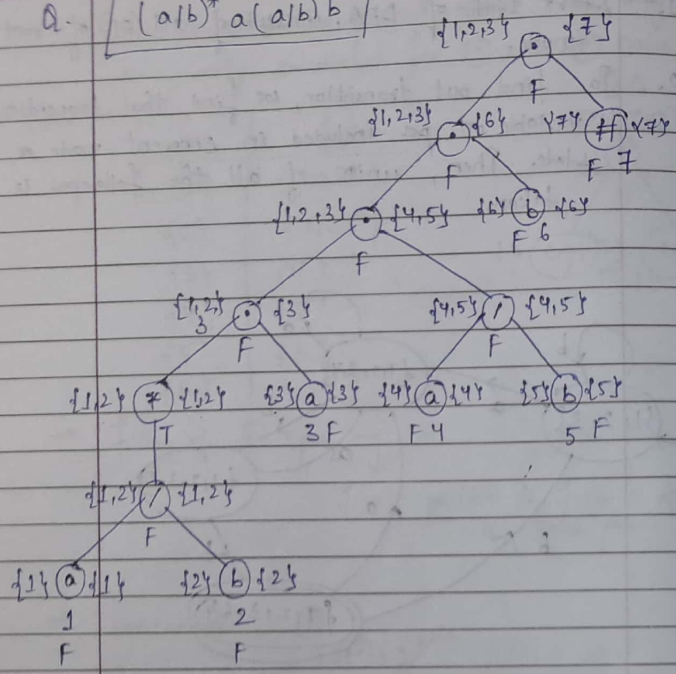
1. → Start state of DFA, consist of all (FP) of root node.
2. To find out transition, we find that transition variable or pos included in current node or state. Then, union of all the followpos is taken.



3. Accepting state ⇒ # pos.
Here, # pos = (6).
Now, all states including # pos will have final state.

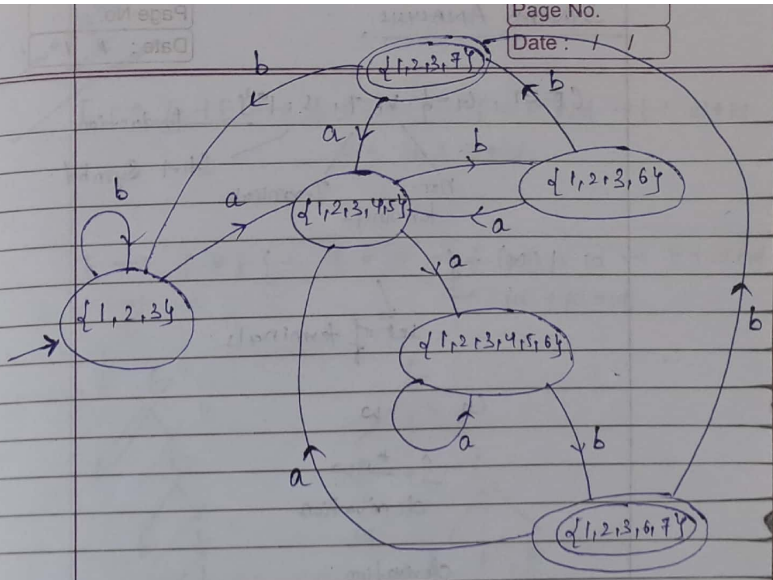
Page No.
Date : / /

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



follow up

pos	follow
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4,5}
{4}	{6}
{5}	{6}
{6}	{7}
{7}	-



left ki last position
mei right ki
first position
daal do

Syntax Analysis

Page No.
Date : / /

CFG, $G = \{ V, T, S, P \}$ — P: Production

V: non-terminals, T: Terminals, S: Start Symbol

$L(G) = \{ \ldots \}$ — set of terminals

$S \xrightarrow{*} w$ — derivation

derivation

Leftmost, Rightmost

- Graphically derivation is Parse tree.
- Ambiguity ⇒ Leftmost or Rightmost 2 or more parse tree for same string.

$E \rightarrow E*E / E+E / (E) / -E / id$

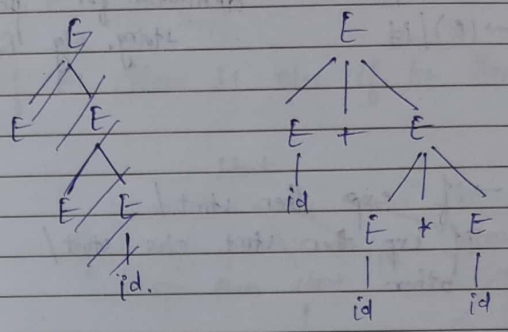
string → (id + id * id)

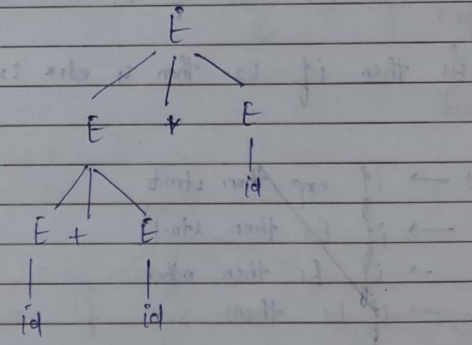
Page No.
Date : / /

two rightmost

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

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





⇒ Some parsers can take ambiguous trees but, they need to have some disambiguous rules to handle it. → Hectic task.

Page No.
Date : / /

To handle this, we try to convert ambiguous grammar into unambiguous grammar.

Unambiguous grammar for last eg:-

E → E+E/T
T → T*F/F
F → (E)/id

→ only one left most derivation for a particular string. eg Right most

stmt → if exp then stmt /
if exp then stmt else stmt /
other.

non-ambiguous

(1). if E1 then (S1 else if E2 then S2) else S3.

ambiguous

(2). if E1 then if E2 then S1 else S2.

stmt → if exp then stmt
→ if E1 then stmt
→ if E1 then other
→ if E1 then

stmt → if exp then stmt else stmt
→ if E1 then stmt else (stmt)
→ if E1 then other else stmt
→ if E1 then S1 else stmt
→ if E1 then S1 else if exp then stmt else stmt

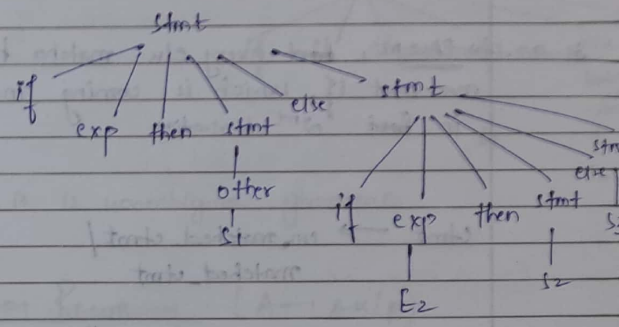
Page No.
Date : / /

→ if E1 then S1 else if E2 then stmt else stmt

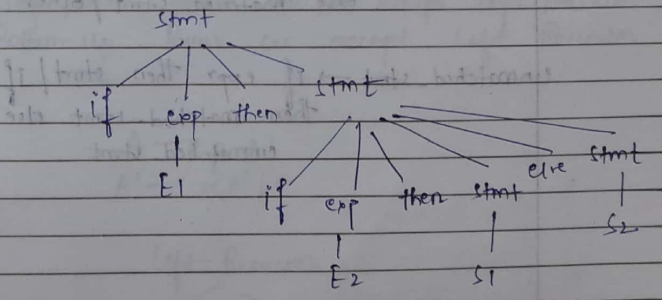
→ if E1 then S1 else if E2 then other else stmt

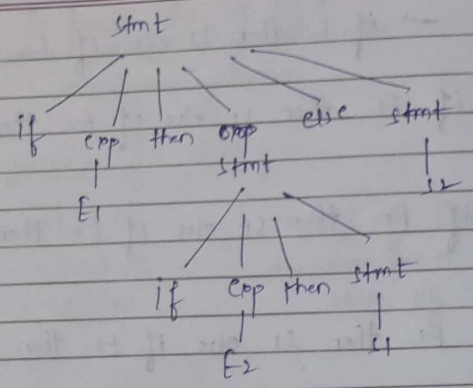
→ if E1 then S1 else if E2 then S2 else other

→ (if E1 then S1 else if E2 then S2 else S3)



(2)





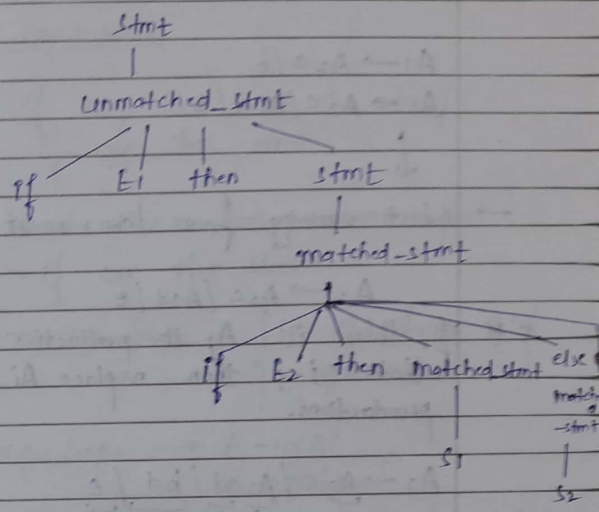
* acc. to PASCAL, ~~last~~ every else matches to its nearmost if, which is coming incorrect in ~~last~~ 2nd derivation.

stmt → un_matched_stmt / matched_stmt

matched_stmt → if expr then matched_stmt else matched_stmt / other

unmatched_stmt → if expr then stmt / if expr then matched_stmt else unmatched_stmt

Eg. if E1 then if E2 then S1 else S2.



✓ Now, it is unambiguous grammar.

LEFT RECURSION (A → Aα/β)

* Top-Down parser do not accept Left Recursion.
* Bottom-up Parser can accept Left Recursion.

A → β A'
A' → α A' / ε

Left-Recursion
- Immediate (Clearly visible)
- Non-Immediate (after substitution, it becomes visible.)

Page No.
Date : / /

Eg: $S \to A a \mid b$
$A \to Ac \mid Sd \mid \epsilon$

$A_1 \to A_2 a \mid b$
$A_2 \to A_2 c \mid A_1 d \mid \epsilon$

↓

→ Start removing from lower most production.

$A_2 \to A_2 c \mid A_1 d \mid \epsilon$

** If there is A_j in production of A_j such that $i<j$ then replace A_i with its productions.

$A_2 \to \underbrace{A_2 c}_{A\alpha_1} \mid \underbrace{A_2 ad}_{A\alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{\epsilon}_{\beta_2}$

$A_2 \to bdA_2' \mid A_2'$

$A_2' \to cA_2' \mid adA_2' \mid \epsilon$

Now,

$A_1 \to A_2 a \mid b$
$A_1 \to bdA_2'a \mid A_2'a \mid b$

$A_1 \to bdA_2'a \mid A_2'a \mid b$
$A_2 \to bdA_2' \mid A_2'$
$A_2' \to cA_2' \mid adA_2' \mid \epsilon$

Page No.
Date : / /

(i) $A' \to \alpha'$

LEFT FACTORING

stmt → if exp then stmt else stmt /
if exp then stmt

If 2 productions have common part α = if exp then stmt

$stmt \to \alpha\beta_1 \mid \alpha\beta_2$ { we left factor this grammar }

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

Decision of which production to choose is deferred for future.

Top-Down Parser

(i) left recursion is removed
(ii) left factoring is done

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

A. $T \to \underbrace{T * F}_{\alpha_1} \mid \underbrace{F}_{\beta_1}$ (left recursion)

$T' \to \alpha F T' \mid \epsilon$
$T \to FT'$

Date : / /

$E \to E+T \mid T$

$E \to TE'$

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

$F \to (E) \mid Id$

(a) $A \to Ba \mid c$

$B \to AA$

$C \to B \mid b$

Ans. $A_1 \to A_2a \mid A_3$

$A_2 \to A_1A_1$

$A_3 \to A_2 \mid b$

$A_2 \to A_1 A_1$

$A_2 \to A_2aA_1 \mid A_3A_1$

$A_2 \to A_3A_1A_2'$

$A_2' \to aA_1A_2' \mid \varepsilon$

$A_3 \to A_2 \mid b \to A_3 \to A_3 \to A_3\underbrace{A_1A_2'}_{\alpha} \mid \underbrace{b}_{\beta}$

$A_3 \to bA_3'$

$A_3' \to A_1A_2'A_3' \mid \varepsilon$

$A_1 \to A_2a \mid A_3$

$A_2 \to A_3A_1 \mid A_2'$

$A_2' \to aA_1A_2' \mid \varepsilon$

$A_3 \to bA_3'$

$A_3' \to A_1A_2'A_3' \mid \varepsilon$

Date : / /

IMP # Top-Down Parser

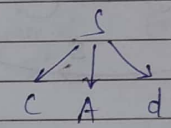
- Recursive - Decent Parser
- Predefine Parser
- Non-Recursive Predictive Parser

→ Recursive - Decent Parser → Grammar does not have left recursion, may have left factoring

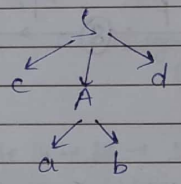
$S \to cAd$

$A \to ab \mid a$ $(w = cad)$

$w = cad$



$w = cad$



$w = cad$

Next symbol = b so backtrack
Reg symbol = d

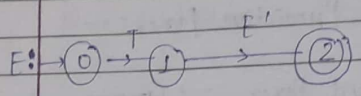
* This backtrack consumes lot of time.

→ Predictive - Parser (No-backtrack required) : left Recursion removed + left factoring + Recursive - Decent Parser.

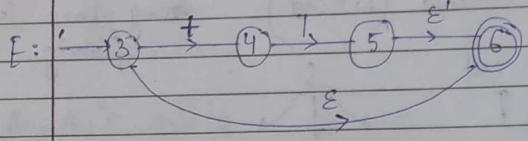
Page No.
Date : / /

Example of Predictive Parser

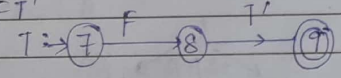
$E \rightarrow TE'$



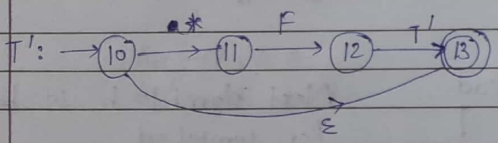
$E' \rightarrow +TE' / \varepsilon$



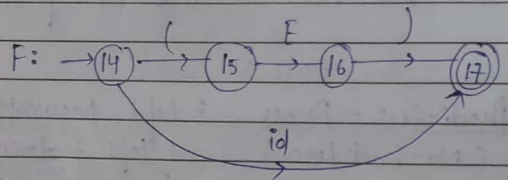
$T \rightarrow FT'$



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



$F \rightarrow (E) / id$



if transition for non-terminals, we jump to procedure of that terminal.

Page No.
Date : / /

id + id * id

State 0 → State 7 → State 14 → 17 → END

Diagram of T — Diagram of F — END

Move back in recursion

→ State 8 → State 10 → State 13 →

Diagram of T' — END

→ State 9 → State 1 → State 3 → State 4

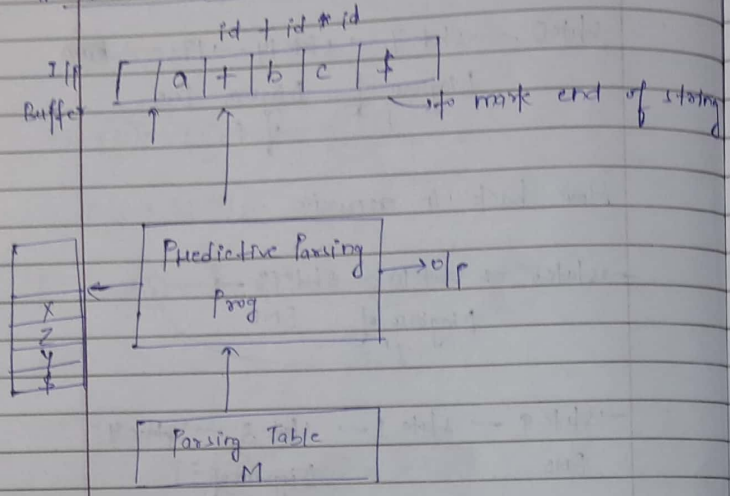
END — Diagram of E'

→ State 7 → State 14 → 17 → State 8 → State 10 →
→ State 11 → State 14 → State 17 → State 12 → State 10
→ State 13
END

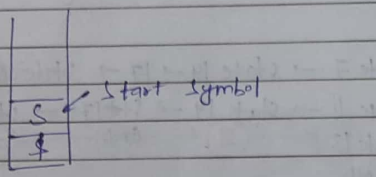
AND SO ON.

Page No.
Date : / /

NON-RECURSIVE PREDICTIVE PARSER



initially,



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

- Calculate FIRST for each terminals as well as non-terminals.

Friday - Test

Page No.
Date : / /

- FOLLOW is calculated only for non-terminals.

IMP - FIRST of a non-terminal is the first terminal of string of terminals i.e. sentence derived from it.

- FIRST of terminal is a terminal.
- " " ϵ is ϵ

Eg. $A \rightarrow ab / \epsilon / BC$

FIRST of A will include, a and first of B, if B is ϵ then first of C.

** Rules :-

1. If, X is a terminal first(X) = {X}
2. If, $X \rightarrow \epsilon$, first(X) $\rightarrow$ {ϵ}
3. $X \rightarrow Y_1 Y_2 \dots Y_n$

FIRST(X) $\rightarrow$ FIRST(Y_1) $\cup$ FIRST(Y_2) $\dots \cup$ FIRST(Y_n) until, ϵ is there in Y_i.

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

Page No.
Date : / /

** Rules for Follow

1. S → start symbol ; FOLLOW (S) = { $ }
2. A → αBβ
   follow (B) → first (β) U FOLLOW (A)
   ↓
   if β contains ε
   or, A → αB
   then, follow (B) → FOLLOW (A)

↳ (if A → αBβ
   if follow of B i.e. first (β) contains ε
   then follow (A) also.

* PARSING TABLE :-

Non terminals ↓ / Terminals →	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)		

Page No.
Date : / /

** Rules

↳ How to fill Parsing table :

→ Take one production at a time.

→ A → α
in row of 'A', write production A → α in all terminals of first (α).
if, α contains ε or, production is A → ε then,
also, all terminals of follow (A)

→ A → terminal α.
↓
production in terminal col.

→ ~~Write accept in $ of start symbol.~~

* (Put reverse string in stack)

Stack	I/P string
$E	id + id * id $
$E'T	id + id * id $
$E'T'F	id + id * id $
$E'T'id	id + id * id $
$E'T'	+ id * id $
$E'	+ id * id $
$E'T+	+ id * id $

Page No.
Date : / /

$ E'T	id * id $
$ E'T'F	id * id $
$ E'T'id	id * id $
$ E'T'	* id $
$ E'T'F*	* id $
$ E'T'F	id $
$ E'T'id	id $
$ E'T'	$
$ E'	$
$	$

** accepted string

* ✓✓ Grammar Generated is LL(1) → Parser
- L → Left Scanning is done
- L → Leftmost derivate tree
- 1 → no. of look ahead symbol

Page No.
Date : / /

Q. ① S → iEtSS' | a
S' → eS | ∈
E → b

② S → [SX] | a
X → tSY | Yb | ∈
Y → SXc | ∈

① S → iEtSS' | a
S' → eS | ∈
E → b

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

PARSING Table :-

Terminals →

Nm ↓	i	t	a	e	b	$
S	S → iEtSS'		S → a			
S'				S' → eS S' → ∈		S' → ∈
E					E → b	

Stack	I/P string
$ S	S →

** LMP

∴ two production are there.
⇒ ambiguity
⇒ not a LL(1) grammar.
⇒ fails.

$X \to Yb$

Page No.
Date : / /

(2.) $S \to [SX] \mid a$

$X \to +SY \mid Yb \mid \epsilon$

$Y \to SXc \mid \epsilon$

FIRST(S) = {[, a}	FOLLOW(S) = {$, +, [, a, b,], c}
FIRST(X) = {+, ε, [, a, b}	FOLLOW(X) = {], c}
FIRST(Y) = {ε, [, a}	FOLLOW(Y) = {], b, c}

Parsing Table:

	[	]	a	+	b	c	$
S	S→[SX]		S→a				
X	X→Yb	~~X→ε~~ X→ε	X→Yb ✓	X→+SY ✓	X→Yb	~~X→ε~~ X→ε	
Y	Y→SXc	Y→ε	Y→SXc		Y→ε	Y→ε	

IMP Rule :-

$X \to \alpha\beta$..

if, α contains ε then, we'll put this production in first of further non-terminals or terminals.

[ab]

Page No.
Date : / /

$ S	[ab]$
$ [SX]	[ab] $
$ [aX]	~~a~~b] $
$ X]	b] $
$ Yb]	b] $
$ b]	~~b~~] $
$]	~~]~~ $
$	$

↳ accepted.

Bottom-Up Parsing

Page No.
Date : / /

→ Here, from a string, we try to get a start symbol.

→ Here, we try to match our string with RHS of a production & replaces it with LHS of that production.

Two types of grammar

- Operator Precedence grammar
- LR grammar

Eg.
$S \rightarrow aABe$
$A \rightarrow Abc / b$
$B \rightarrow d$

i/p string → abbcde

abbcde
↓
aAbcde
↓
aAde
↓
aABe
↓
S

→ this string belongs to the language generated by the grammar.

→ To find a suitable handle is a tough task & this technique is called handle pruning

Strings that can be obtained by corresponding replacement is called handle

Page No.
Date : / /

→ Here, we get rightmost derivative in reverse order.

→ All parsers of Bottom-Up-Parsing works on shift-reduce principle & are called shift-reduce parsers.

Q. $E \rightarrow E+E / E*E / (E) / id$

$id_1 + id_2 * id_3$	$id_1 + id_2 * id_3$
$\Rightarrow E + id_2 * id_3$	$\Rightarrow E + id_2 * id_3$
$\Rightarrow E + E * id_3$	$\Rightarrow E + E * id_3$
$\Rightarrow E * id_3$	$\Rightarrow E + E * E$
$\Rightarrow E * E$	$\Rightarrow E + E$
$\Rightarrow E$	$\Rightarrow E$

Stack	i/p	Action
$	$id_1 + id_2 * id_3 \$$	shift id_1
$ id_1	$+ id_2 * id_3 \$$	Reduce $E \rightarrow id_1$
$ E	$+ id_2 * id_3 \$$	shift +
$ E +	$id_2 * id_3 \$$	shift id_2
$ E + id_2	$* id_3 \$$	Reduce $E \rightarrow id_2$
$ E + E	$* id_3 \$$	

** When top of stack is replaced by LHS, it is known as reduce action.

Here, we have 2 problems.
either we can shift * or reduce $E \rightarrow E+E$.
This situation is called "shift Reduce Conflict"

Solving further

$ E + E *	id, $
$ E + E * id3	$
$ E + E * E	$
$ E + E	$
[$ E	$]

start symbol → string belongs to grammar.

Operator Precedence Grammar

1). There should not be any ε production in grammar.

2). No two non-terminals can occur together

Eg.

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid (-E) \mid id$

Steps: first we make operator-precedence Table & then parse any string using it.

- $a <\cdot b$ → b has higher precedence than a
- $a \doteq b$ → a has equal precedence as b.
- $a \cdot> b$ → a has higher precedence than b

→ $ <· a or a ·> $

→ id ·> a [for given grammar, id has higher precedence than any terminal i.e. {(, *, +, ↑, etc.}]

Eg. $E \to E+E \mid E*E \mid id$

[→ high to same symbol precedence
left to right association]

	id	+	*	$
id	–	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	–

→ help to find handle

→ (end of string)

operator precedence table.

(we'll check precedence b/w all two symbols & $ will replace them with precedence sign)

I/P → $ id + id * id $

→ $ <· id ·> + <· id ·> * <· id ·> $

⇒ $ <· id ·>

⇒ $ <· + <· id ·> * <· id ·> $

⇒ $ <· + <· * <· id ·> $

⇒ $ <· + <· * ·> $

⇒ $ <· + ·> $ ⇒ $$ final

Consider <· as opening bkt $ ·> as closing. Now, whenever we get ·>, find closest <· & we'll get an appropriate handle

↑ is right associative.

Page No.
Date : / /

id * (id ↑ id) - id / id

	+	-	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗	-	-	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	-
)	⋗	⋗	⋗	⋗	⋗	-	-	⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖	-	-

$ ⋖ id ⋗ * ⋖ (⋖ id ⋗ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ id ⋗ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ ↑ ⋖ id ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (⋖ ↑ ⋗) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (≐) ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋗ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ - ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ - ⋖ / ⋖ id ⋗ $

$ ⋖ - ⋗ $

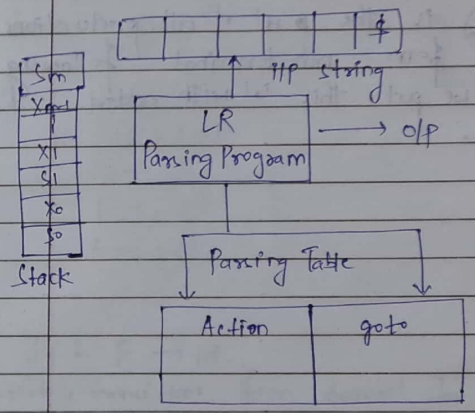
$ $

LR-Parsers

Page No.
Date : / /

LR(simple LR) — CLR (Canonical) — LALR

* grammar generated is LR(1) only. (1 lookahead symbol)
  - (left to right scanning)
  - (Right most derivative)



~ Left factoring & Left Recursion is not req. to be removed.

~ LR grammar is a superset of LL grammar.

Augmented grammar

derive start symbol of a given grammar with another symbol with unit production
i.e. $E' \to E$, E is start symbol

Page No.
Date : / /

$E' \rightarrow .E$

↳ this '.' added in a production makes it an item.

Eg. $E' \rightarrow .EXa$

↓

'.' = it mean our parser is expecting given string to be derived from production 'EXa'. In this case.

Rule

#Closure (item) is the set of all productions derived from non-terminal following '.'. If we put this '.' with added productions.

Q. $E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Now, add, $E' \rightarrow E$ to make it augmented.

Now. $I_0 : E' \rightarrow .E$

Closure (E) { $E \rightarrow .E+T$, $E \rightarrow .T$

Closure (T) { $T \rightarrow .T*F$, $T \rightarrow .F$

Closure (F) { $F \rightarrow .(E)$, $F \rightarrow .id$

* Closure (item) ≡ together marked closure (E)

I_0 → represent Initial state

all included in I_0.

• transitions from I_0 :-

(. is shifted)

Page No.
Date : / /

transition (E)

$I_1 : E' \rightarrow E.$

$E \rightarrow E.+T$

transition (T)

$I_2 : E \rightarrow T.$

$T \rightarrow T.*F$

$I_3 : T \rightarrow F.$

$I_4 : F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

** Now closure of E

$I_5 : F \rightarrow id.$

Now state, transitions from symbol I_1 :-

$I_6 : E \rightarrow E+.T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Transition from I_2 :-

$I_7 : T \rightarrow T*.F$

$F \rightarrow .(E)$

$F \rightarrow .id$

→ closure of null + key

→ symbols which I_0 can get for transition include the symbols following '.'. These symbols can be terminal as well as non-terminal.

→ make a new state for each transition symbol & add closure of each item in each state, which have been included.

→ closure of * key null.

Page No.
Date : / /

transitions for I_1

I_6: $F \to (E.)$
$E \to E.+T$ = (3) IHP.
$E \to I.$ } ∵ part of I_2 & already included
$T \to T.*F$ } ∴ not to be included in state (I_6)
only transitions made to I_2 in dfg.

$T \to F.$ = I_3

$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$
} all productions. together = I_4

Note: whole closure.

only 1
I_8: $F \to (E.)$
$E \to E.+T$

$F \to id.$ = I_5

transitions for I_6

I_9: $E \to E+T.$
$T \to T.*F$

$T \to F.$ = I_3

$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$
} E-closure = I_4

$F \to id.$ = I_5

Page No.
Date : / /

∴ I_9: $E \to E+T.$
$T \to T.*F$

transition for I_7

I_{10}: $T \to T*F.$

$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$
} E-closure = I_4

$F \to id.$ = I_5

∴ I_{10} : $T \to T*F.$

transition for I_8

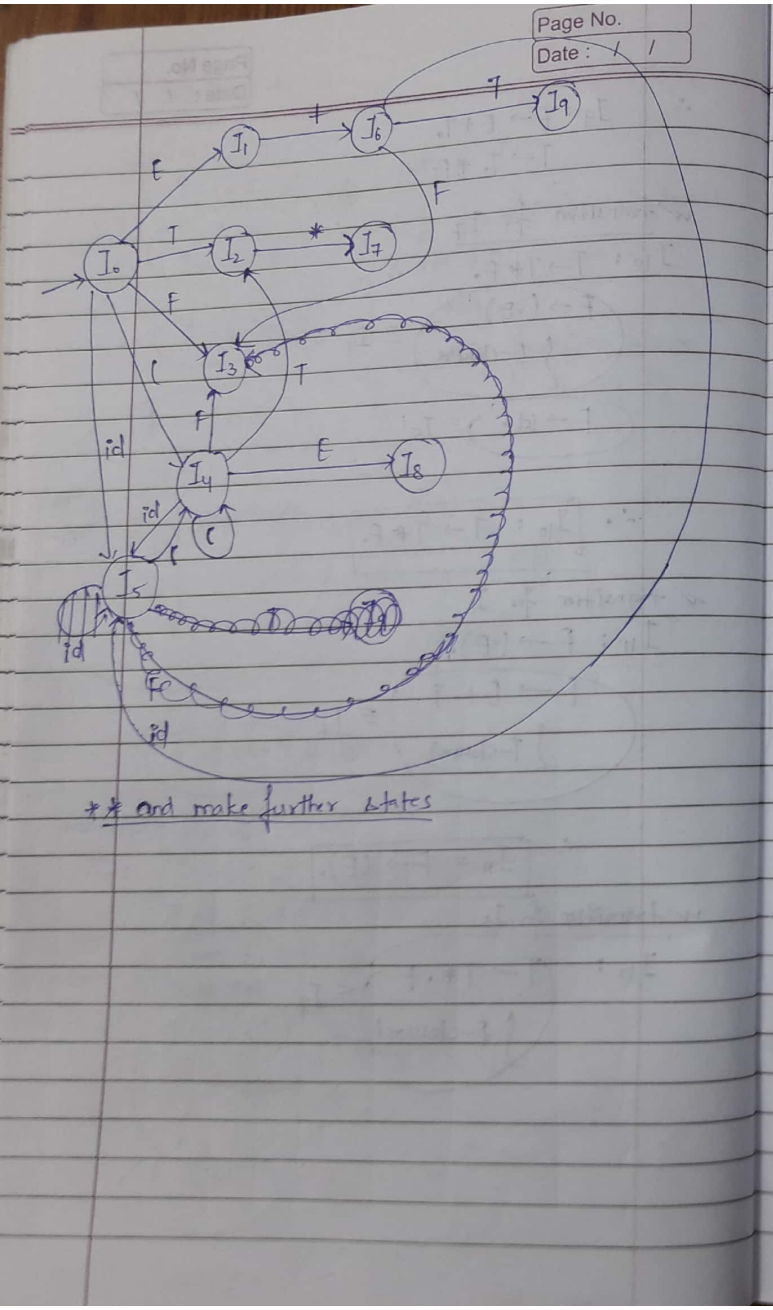
I_{11} : $F \to (E).$

$E \to E.+T$ = I_6
{T-closure}

∴ $I_{11} = F \to (E).$

transition for I_9

I_{12} : $T \to T*.F$ = I_7.
{F-closure}



** and make further states

$S' \to S$

Q. $\gamma_1: S \to L=R$
$\gamma_2: S \to R$
$\gamma_3: L \to *R$
$\gamma_4: L \to id$
$\gamma_5: R \to L$

I_0 : $S' \to .S$ ✓ ✓
$S \to .L=R$ ✓ ✓
$S \to .R$ ✓ ✓
$L \to .*R$ ✓ ✓
$L \to .id$ ✓ ✓
$R \to .L$ ✓ ✓

I_1 : $S' \to S.$ ✓

I_2 : $S \to L.=R$
$R \to L.$ ✓

I_3 : $S \to R.$ ✓

I_4 : $L \to *.R$ ✓
$R \to .L$ ✓

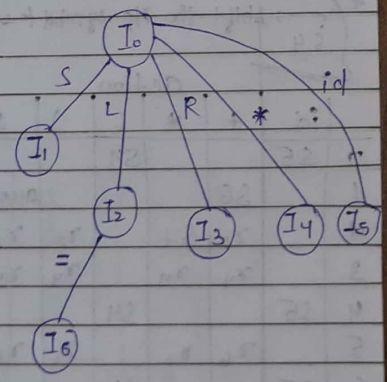
I_5 : $L \to id.$ ✓ ✓

I_6 : $S \to L=.R$ ✓
$R \to .L$ ✓

I_7 : $L \to *R.$ ✓

I_8 : $R \to L.$ ✓

I_9 : $S \to L=R.$ ✓
for, L, H will go to I_8



FIRST | FOLLOW

IMP // We'll not consider E' while parsing table.

IMP // Later, we'll get $E' \to E \cdot$ will make $ col = accept of that production.

~~Follow (E')~~ =

FIRST (E) = { (, id }

FIRST (T) = { (, id }

FIRST (F) = { (, id }

FOLLOW (E) = { $, +,) }

FOLLOW (T) = { $, +,), * }

FOLLOW (F) = { $, +,), * }

* 1, 9, 2, 11 → represents states.

* → shift the i/p symbol to stack & move to state (4).

S4

	action						goto			
	id	+	*	(	)	$	E	T	F	
0	S5			S4			1	2	3	$r_1: E \to E+T$
1		S6				accept				$r_2: E \to T$
2		r_2	S7		r_2	r_2				$r_3: T \to T*F$
3		r_4	r_4		r_4	r_4				$r_4: T \to F$
4	S5			S4			8	2	3	$r_5: F \to (E)$
5		r_6	r_6		r_6	r_6				$r_6: F \to id$
6	S5			S4				9	3	
7	S5			S4					10	
8		S6			S11					
9		r_1	S7		r_1	r_1				
10		r_3	r_3		r_3	r_3				
11		r_5	r_5		r_5	r_5				

→ Name all productions

$F \to id \cdot$ ← $F \to id$

→ if, we'll get '.' at the end (not augmented) then reduce it.

IMP.

$E \to T \cdot$ → reduce T with E i.e. production r_2 ($E \to T$) & write it in follow of (E)

Q. FIRST (S) = { *, id }

FIRST (L) = { *, id }

FIRST (R) = { *, id }

∴ No possible ambiguity

FOLLOW (S) = { $ }

FOLLOW (L) = { =, $ }

FOLLOW (R) = { $, = }

	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6, r_5			r_5			
3				r_2			
4						8	7
5	r_4			r_4			
6							9
7	r_3			r_3			
8	r_5			r_5			
9				r_1			

$F \to I$ T

I A B

S (A B) L = R (A B)