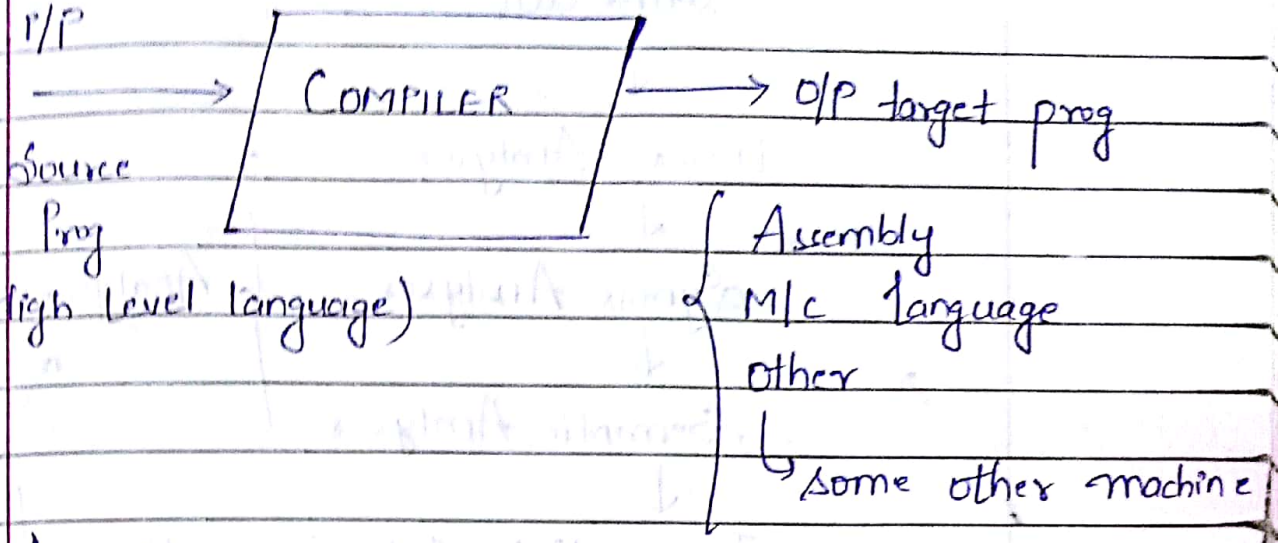


Assembly :-

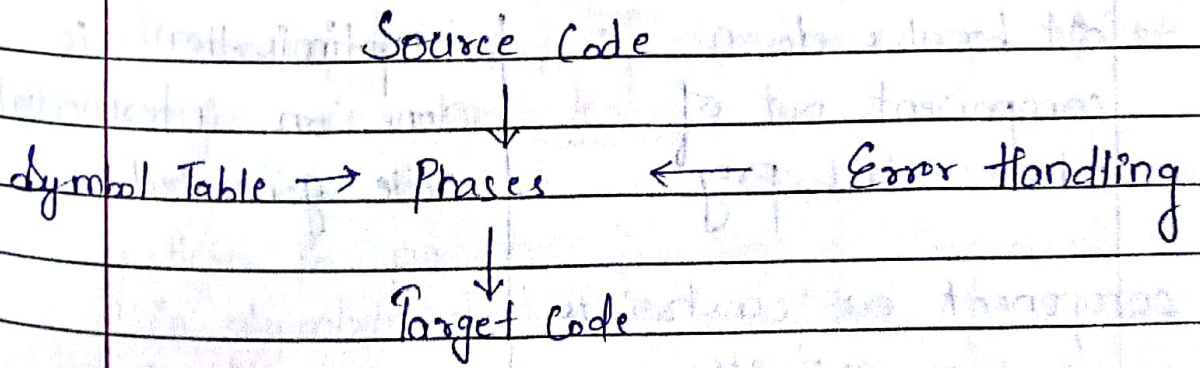
i/p → compiler → assembly language → assembler → target

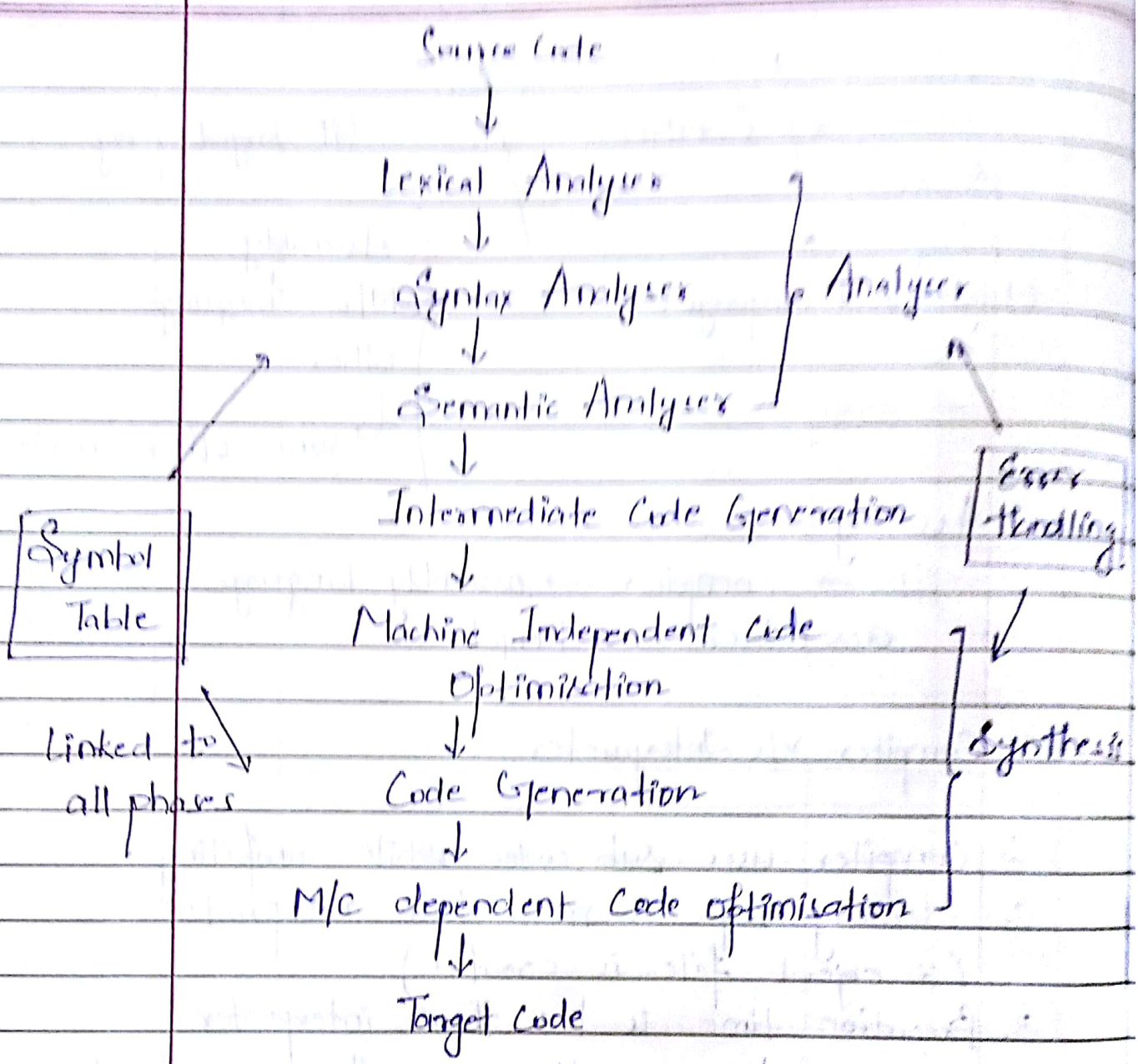
Compiler v/s Interpreter

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





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.	

✓ 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^n 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.

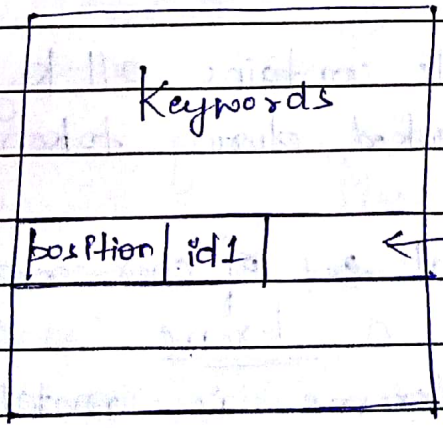
Eg. void → lexeme	; → lexeme
Keyword → token	Semicolon → token

→ (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

< +, plus sign>

Errors it will not report :-

main ()

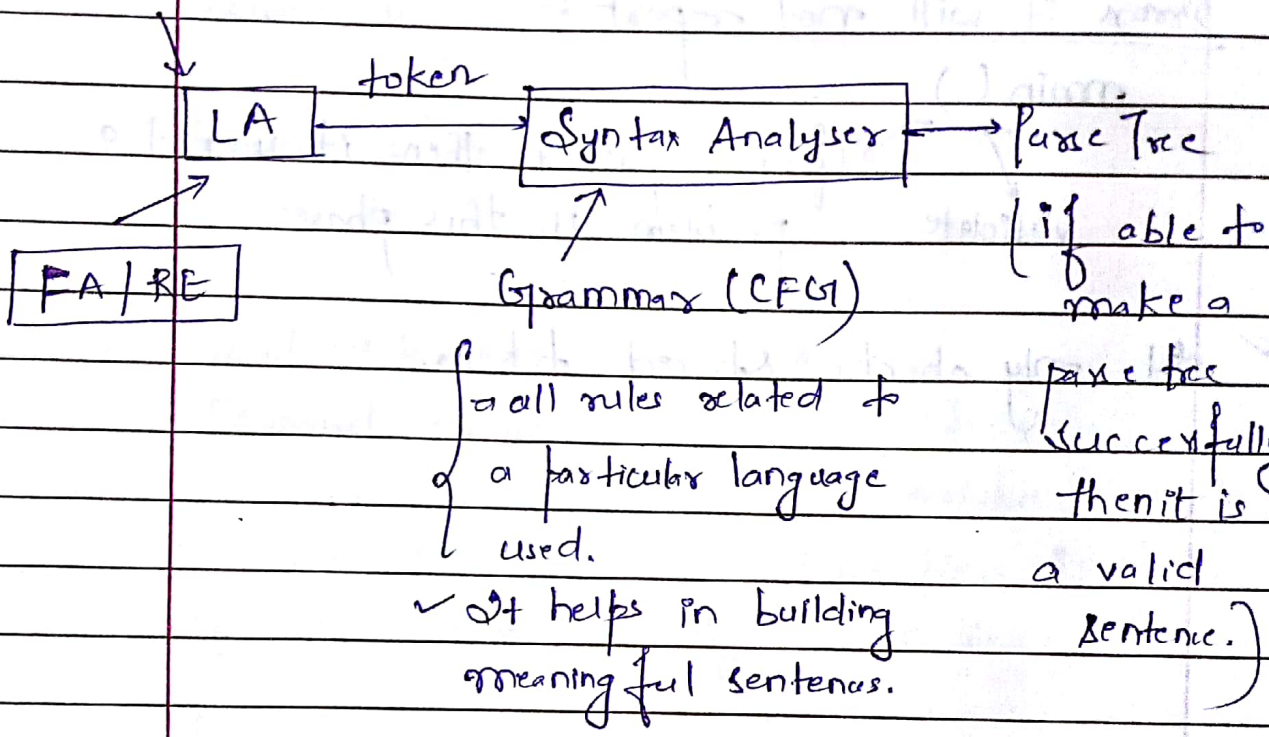
validate → if not present then it is not a problem in this phase.

- O/P only checks stored tokens.

(2). SYNTAX ANALYZER

also, Hierarchial Analysis → ∵ tree is traversed.

Parsing



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's.

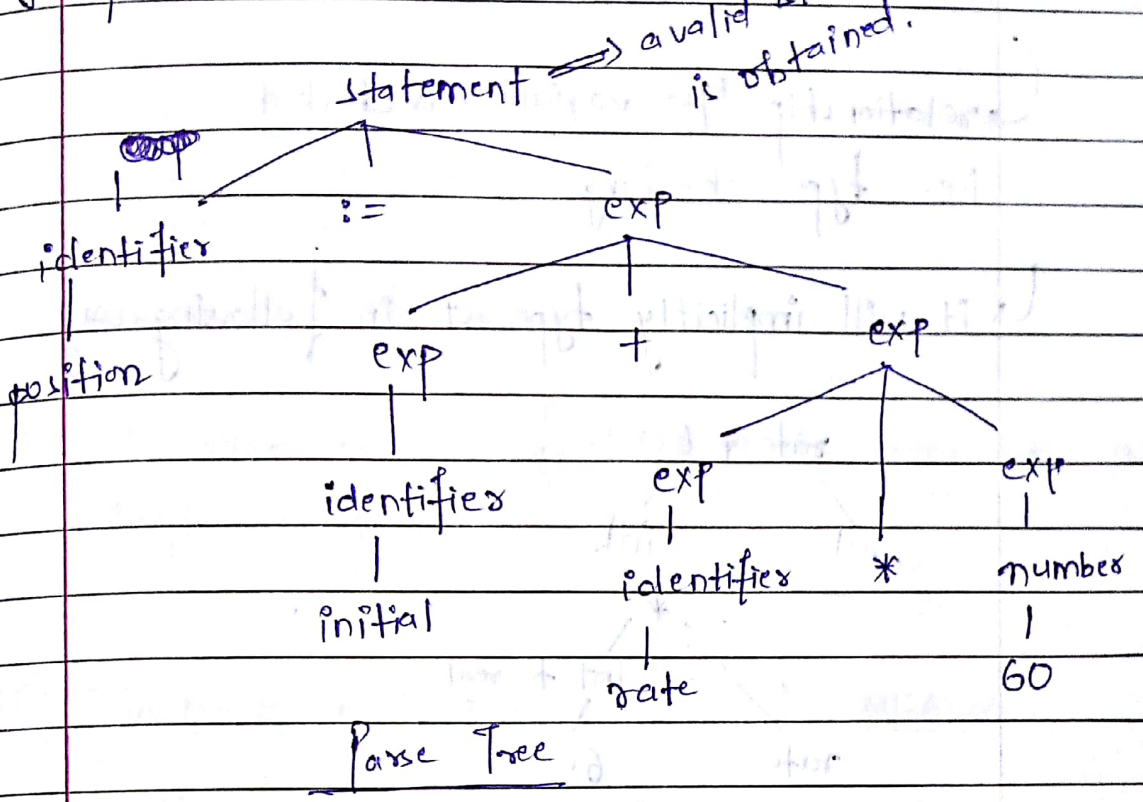
Statement

1. identifier$_1$ = exp_2
2. while (exp) do stm_2

i/p string

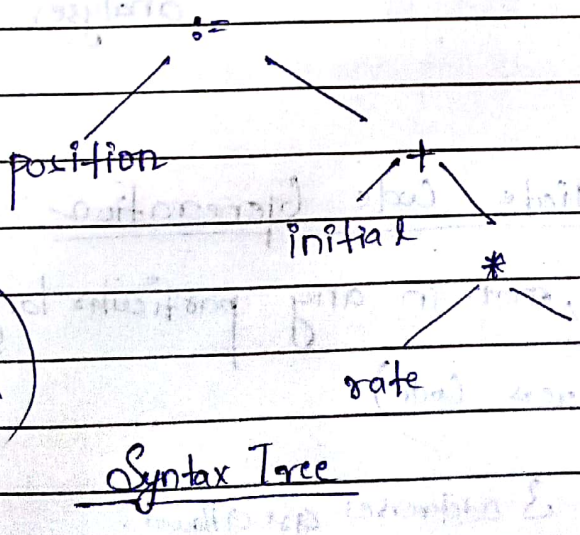
Page No.
Date : / /

Eg. position := initial + rate * 60



Parse Tree

Compressed representation of Parse Tree is Syntax Tree.



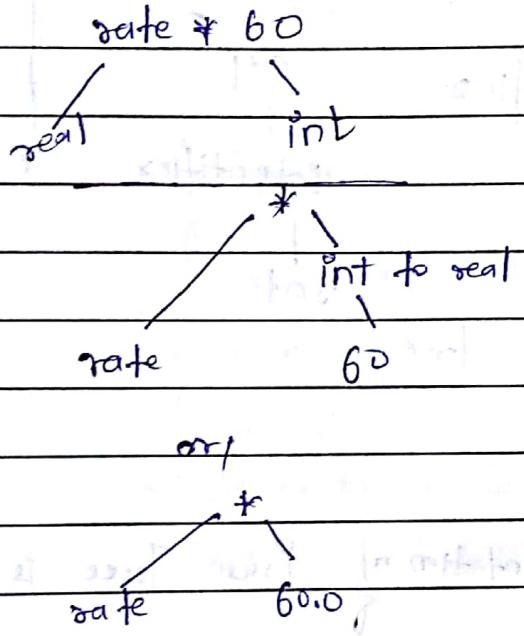
* level should be differentiated

Syntax Tree

③. Semantic Analysis

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

↳ it will implicitly typecast in following way



(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 : / /

- Use temporary variables to ensure 3-address code

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

- Use temp 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. `fi ( 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 in correct char by a correct char
5. ✓ transposing adjacent char

Page No.
Date : / /

Eg. 193.
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.
④th recovery mode.

Eg. E 9 −4
exponent ← locally handeled. & replaced by 9 E −4
⑤th recovery mode..

Terms related to Automata.
1. Symbol → any char 0, 1, a, b, # --etc.
2. (S) 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, ∈, ab, abc

Page No.
Date : / /

6. Suffix (n+1) c, bc, abc, ε

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

8. proper suffix ⇒ ≠ lg.

^ *

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)*

(id=)

converting to definition.

letter → A|B|C---|Z|a|b|---|z
digit → 0|1|2|---|9

Page No.
Date : / /

Now,

identifier → letter (letter | digit)*

Eg. (number)

exponent, fraction,

digit → 0|1|--|9

sign → +|-

digits → digit digit* or (digit+)

op_fraction → ∈ | . digits

op_exp → ∈ | E(+|-|∈) digit

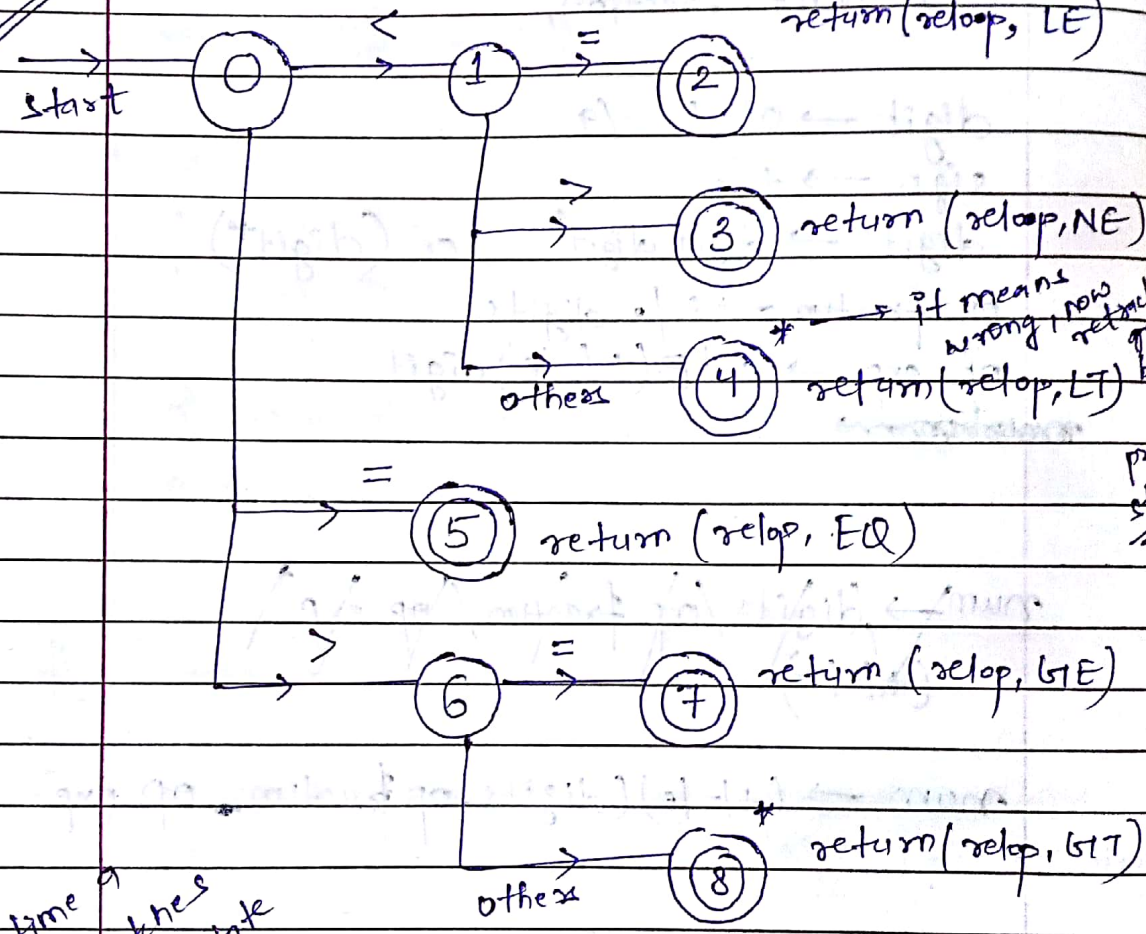
num → (digits /op_fraction / op_exp) (+|-|∈)

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

Page No.
Date : / /

Transitional diagram

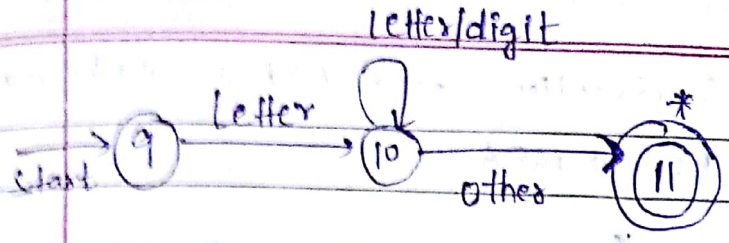
(1). relational operators allowed in pascal <, <=, >, >=, =, <> → not equal to



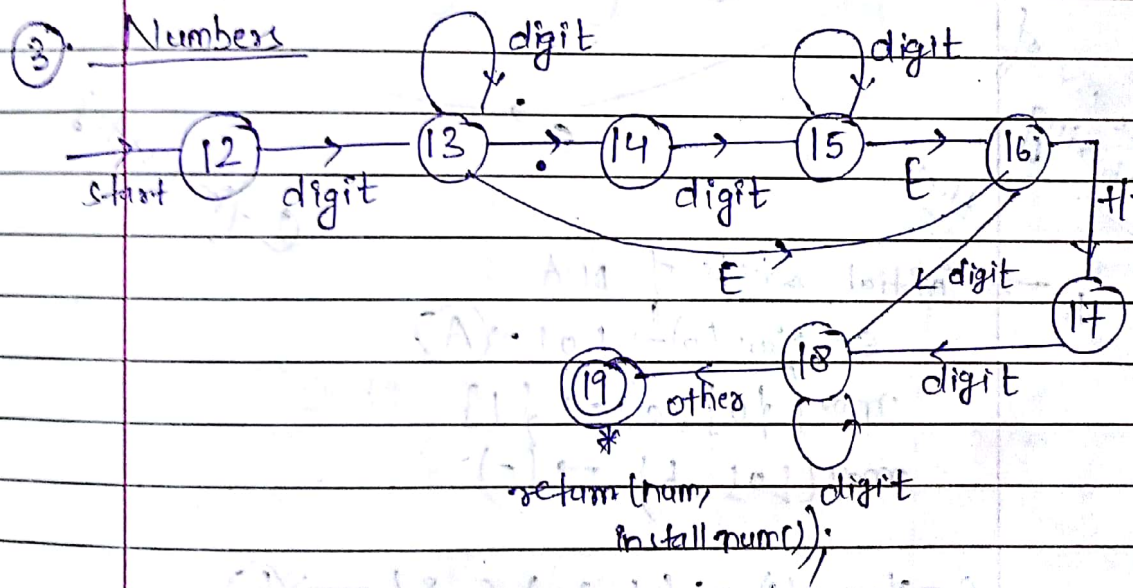
Every time a lexeme matches starting at state by (6).

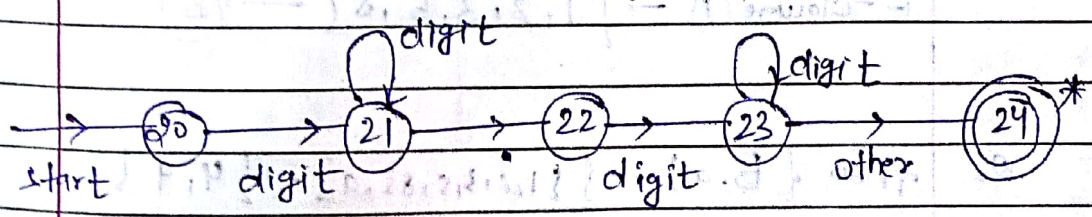
(2). Identifier → starting with alphabets only, followed by a numeral or alphabets.

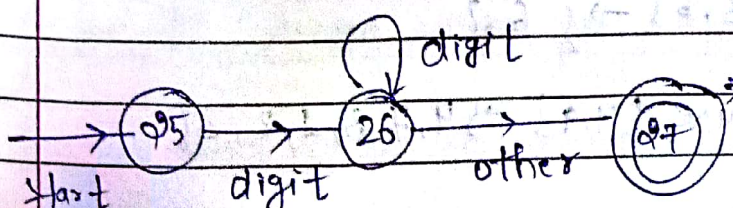
Page No.
Date : / /



if, lexeme is a ~~token~~ keyword then, it will return install_id() as 0 else, it will make an entry and will return ptr to that entry.







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

Page No.
Date : / /

3. Regular Expression to DFA

(a|b)* abb#

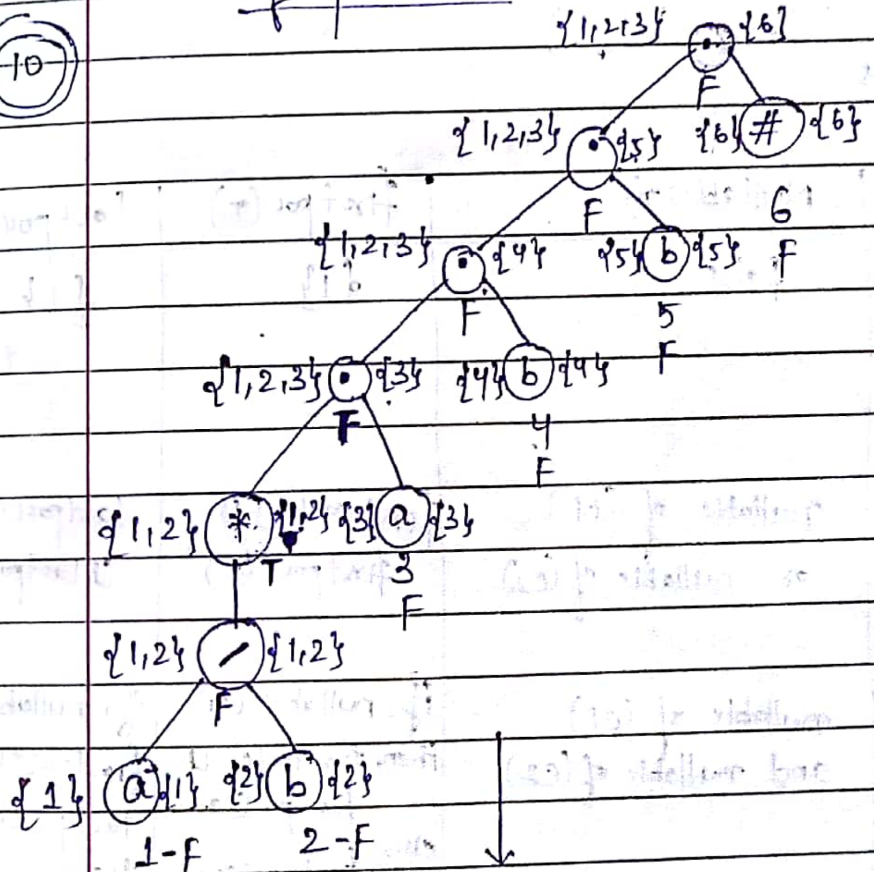
In NFA :-

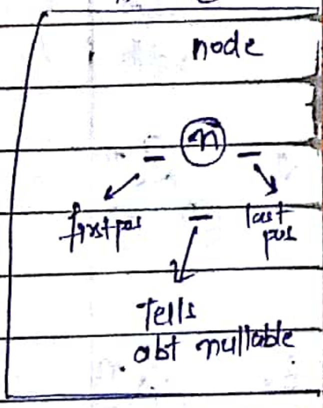
⇒ States with no ε-moves are called Imp. States.

⇒ States with ε-moves are called non-Imp states.

(*) follow this dir^n. if parentheses corresp.. make it 1 part.

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





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

Every node (interior or exterior) is a expression.

Page No.
Date : / /

Step 3: Calculate following fn for the tree

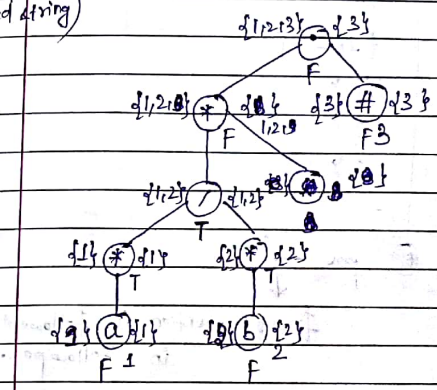
1. Nullable → calculated for each and every node
2. firstpos → calculated for each node (first symbol in generated string)
3. lastpos → " (last symbol in a generated string)
4. follow → for some nodes only.
   it belongs to ⊙ → concatenation node
   ⊛ → clear node

→ if a language generates 'ε' along with any of it is called nullable.

RULES

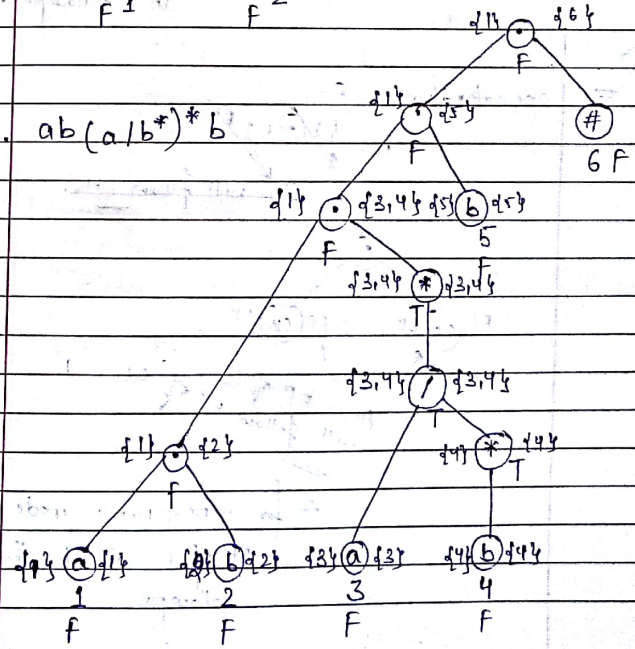
		Nullable(n)	firstpos(n)	lastpos(n)
1.	n is a leaf (labeled with ε)	false	{i}	{i}
2.		nullable of (c1) or, nullable of (c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
3.		nullable of (c1) and nullable of (c2)	if, nullable (c1) then, firstpos(c1) ∪ firstpos(c2) else, firstpos(c1)	if, nullable(c2)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

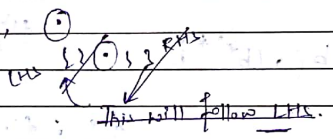
• (a|b)*abb#

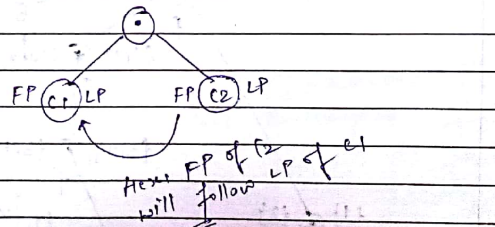
pos	follow
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4}
{4}	{5}
{5}	{6}
{6}	–

⇒ follow of (*) pos. →

first position is followed up in followpos.

⇒ concatenation, ⊙



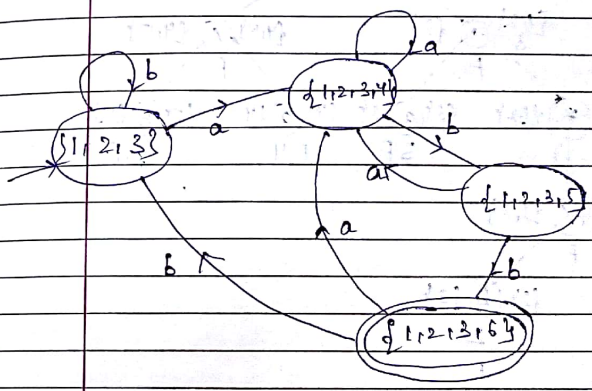


∴ for concatenation node, LP of c2 will be the followpos.

Page No.
Date : / /

Rules :-

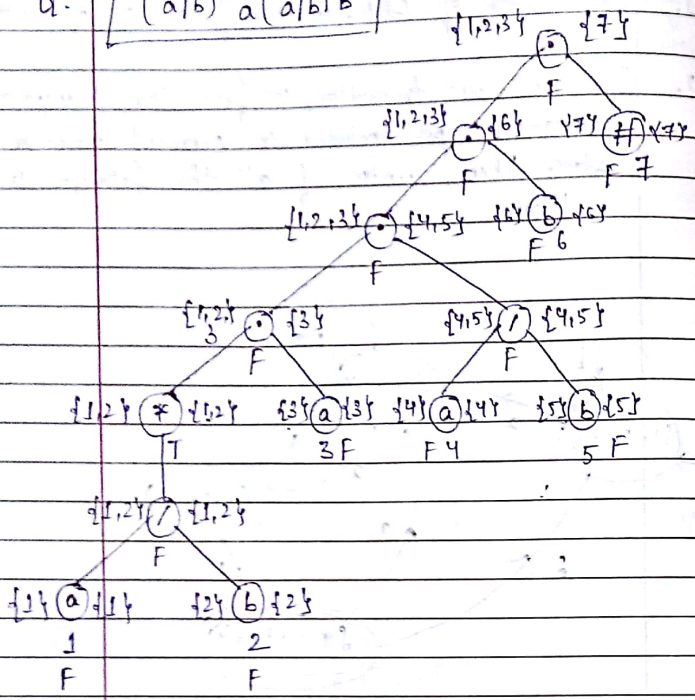
1. → start state of DFA, consist of all (FP) of root node.
2. To find out transition, we find that transition variable or pos included in current node or state. Then, union of all the followpos is taken.



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

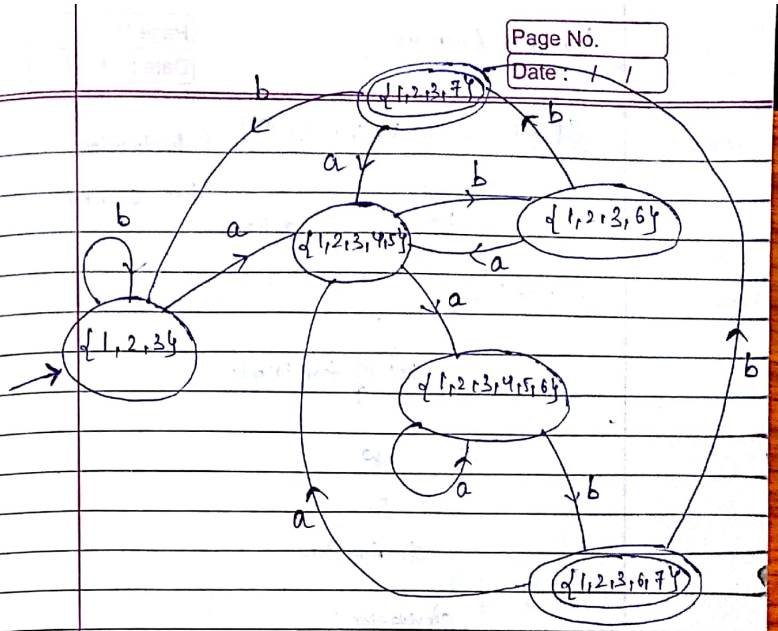
Page No.
Date : / /

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}	–



Page No.
Date : / /

Syntax Analysis

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

Non-terminals, Terminals, Start Symbol

$L(G) = \{ \quad \}$

Set of terminals

$S \xrightarrow{*} \omega$

derivation

derivation

Leftmost, Rightmost

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

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

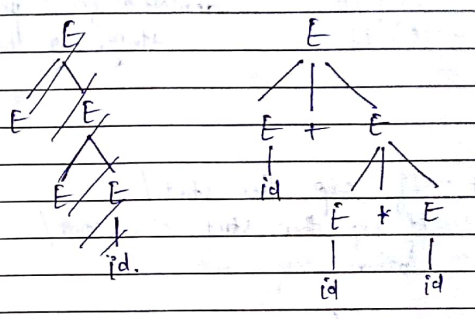
string → $(id + id * id)$

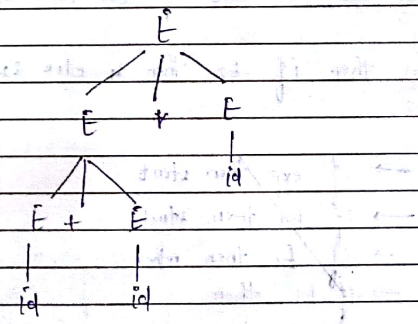
Page No.
Date : / /

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:-

$E \rightarrow E+E \mid T$ — only one left most derivation for a particular string, eg Right most

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

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

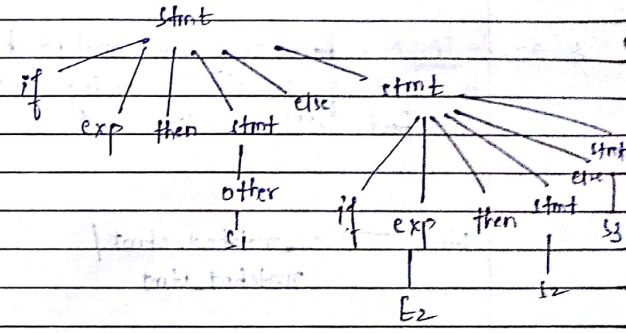
Page No.
Date : / /

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

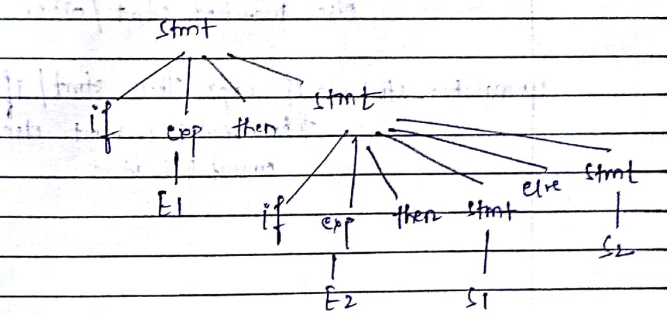
→ if E1 then S1 else if E2 then other else stmt

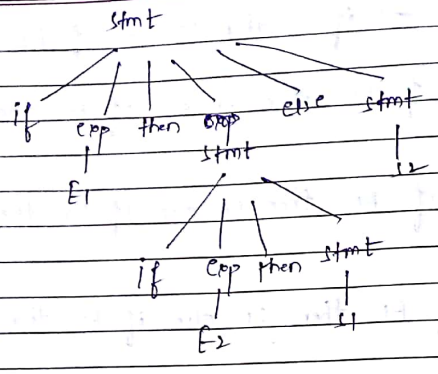
→ if E1 then S1 else if E2 then S2 else other

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



(2)





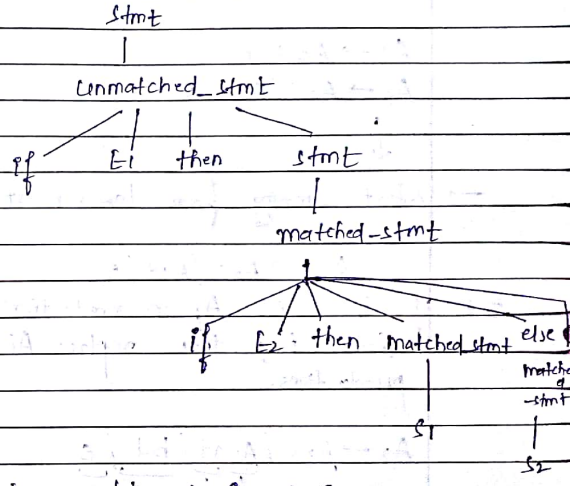
* Acc. to PASCAL, ~~last~~ every else matches to its near most if, which is coming incorrect in ~~last~~ 2nd derivation.

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' → αA' / ε

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

Page No.
Date : / /

Eg. $S \rightarrow A\alpha \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

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

↓

→ Start solving from lower most production.

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

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

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

$A_2 \rightarrow bdA_2' \mid A_2'$

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

Now,

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

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

Page No.
Date : / /

(ii) $A' \rightarrow \alpha'$

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 $\rightarrow \alpha\beta_1 / \alpha\beta_2$ { we left factor this grammar }

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 \longrightarrow A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 / \beta_2$

Decision of which production to choose is deferred for future.

Top-Down Parser

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

(1) $E \rightarrow E+T / T$
$T \rightarrow T * F / F$
$F \rightarrow (E) / id$

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

$T' \rightarrow \alpha F T' / \varepsilon$
$T \rightarrow F T'$

$E \rightarrow E + T / T$

$E \rightarrow TE'$

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

$F \rightarrow (E) / id$

(a) $A \rightarrow Ba / c$

$B \rightarrow AA$

$C \rightarrow B / b$

Ans. $A_1 \rightarrow A_2 a / A_3$

$A_2 \rightarrow A_1 A_1$

$A_3 \rightarrow A_2 / b$

$A_2 \rightarrow A_1 A_1$

$\downarrow$

$A_2 \rightarrow A_2 a A_1 / A_3 A_1$

$\downarrow$

$A_2 \rightarrow A_3 A_1 A_2'$

$A_2' \rightarrow a A_1 A_2' / \varepsilon$

$A_3 \rightarrow A_2 / b \rightarrow A_3 \rightarrow A_3 \rightarrow A_3 \underbrace{A_1 A_2'}_{\alpha} / \underbrace{b}_{\beta}$

$A_3 \rightarrow b A_3'$

$A_3' \rightarrow A_1 A_2' A_3' / \varepsilon$

$A_1 \rightarrow A_2 a / A_3$

$A_2 \rightarrow A_3 A_1 / A_2'$

$A_2' \rightarrow a A_1 A_2' / \varepsilon$

$A_3 \rightarrow b A_3'$

$A_3' \rightarrow A_1 A_2' A_3' / \varepsilon$

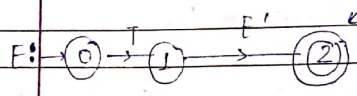
Page No.
Date : / /

Example of Predictive Parser

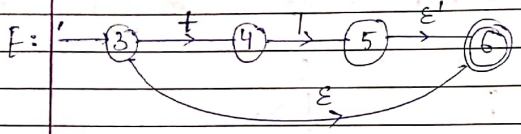
* Imp

$E \rightarrow TE'$

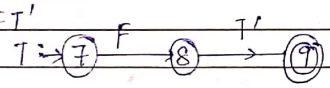
declare accept on reaching this state.



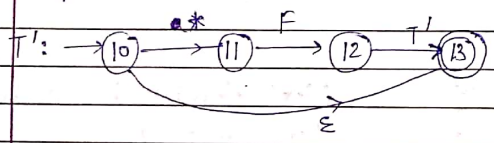
$E' \rightarrow +TE' / \varepsilon$



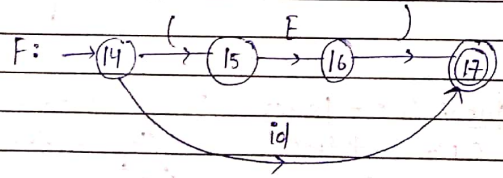
$T \rightarrow FT'$



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



$F \rightarrow (E) / id$



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

END

AND SO ON.

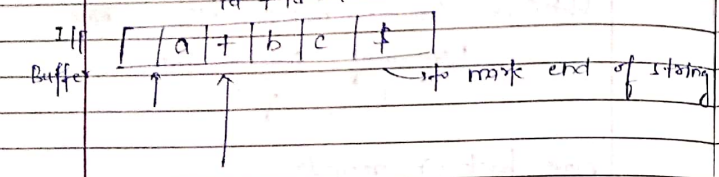
* Imp

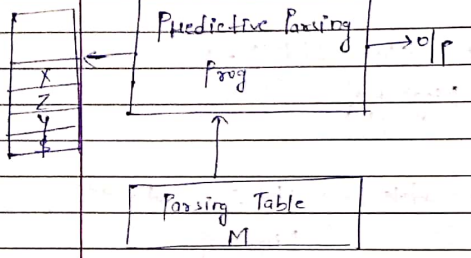
Remove left recursion by factoring it left factoring is present in predictive parser.

Page No.
Date : / /

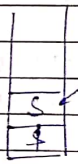
NON-RECURSIVE PREDICTIVE PARSER

id + id * id





initially,



E → TE'
E' → +TE' / ∈
T → FT'
T' → *FT' / ∈
F → (E) / id

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

Friday - Test

Page No.
Date : / /

FOLLOW is calculated only for non-terminals.

HP 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.
" " ∈ is ∈

Eg. A → ab / ∈ / BC

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

** RULES:

1. If X is a terminal first(X) = {X}
2. If X → ∈, first(X) → {∈}
3. X → Y1 Y2 — Yn

FIRST(X) → FIRST(Y1) ∪ FIRST(Y2) — ∪ FIRST(Yn) until, ∈ is there in Yi.

FIRST(E) = {(, id}	FOLLOW(E) = {$,)}
FIRST(E') = {+, ∈}	FOLLOW(E') = {$,)}
FIRST(T) = {(, id}	FOLLOW(T) = {+, *, $,)}
FIRST(T') = {*, ∈}	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 ∈
   i.e. A → αB
   then, follow (B) → Follow (A)

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

* PARSING TABLE :-

Terminals →

Non terminals ↓	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T' → ∈	T' → *FT'		T' → ∈	T' → ∈
F	F → id			F → (E)		

** Rules

Page No.
Date : / /

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 ∈ or production is A → ∈ then,
  also, all terminals of follow (A)
- A → terminal α.
  ↓
  production in terminal col.
- ~~write~~ accept in $ of start symbol.

* [Part 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 $

R → (E)
C → (

A → BC

$ 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; no. of look ahead symbol

left Scanning is done

left most derivate tree

Q. ① S → iEtSS' | a
S' → eS | ∈
E → b

② S → [SX] | a
X → +SY | 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 →

Non-terminals ↓	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]$
$ [SX	ab]$
$ [S	Xb]$
$ [a	Xb]$
$ X]	b]$
$ Yb]	b]$
$ b]	b]$
$]	]$
$	$

→ accepted.

Page No.
Date : / /

Questions

(1). $S \to A$

$A \to aB \mid Ad$

$B \to bBC \mid f$

$C \to g$

String (abfa)

FIRST(S) = {a}	FOLLOW(S) = {$}
FIRST(A) = {a}	FOLLOW(A) = {d, $} (S, A)
FIRST(B) = {b, f}	FOLLOW(B) = {g, d, $} (A, B) (X)
FIRST(C) = {g}	FOLLOW(C) = {g, d, $} (B)

	a	d	b	f	g	$
S	S→A					
A	A→aB A→Ad					
B						
C						

Imp.

LL-parser → left Recursion & left factoring

remove

Page No.
Date : / /

Q1. $S \to A$

$A \to aB \mid Ad$

$B \to bBC \mid f$

$C \to g$

w String (abfg)

$A \to \underset{\alpha}{Ad} \mid \underset{\beta}{aB}$

$A' \to aBA'$

$A' \to dA' \mid \epsilon$

$S \to A$

$A \to aBA'$

$A' \to dA' \mid \epsilon$

$B \to bBC \mid f$

$C \to g$

FIRST(S) = {a}

FIRST(A) = {a}

FIRST(A') = {d, ε}

FIRST(B) = {b, f}

FIRST(C) = {g}

FOLLOW(S) = {$}

FOLLOW(A) = {$}

FOLLOW(A') = {$}

FOLLOW(B) = {d, g, $}

FOLLOW(C) = {d, g, $}

	a	b	d	f	g	$
S	S→A					
A	A→aBA'					
A'			A'→dA'			A'→ε
B		B→bBC		B→f		
C					C→g	

Date : / /

String :

abfg

Stack	I/P
$S	abfg$
$A	abfg$
$A'Bf	abfg$
$A'B	bfg$
$A'CBb	bfg$
$A'CB	fg$
$A'Cf	fg$
$A'C	g$
$A'g	g$
$A'	$
$	$

→ accepted

Q2. E → UV | EBE | V | [E]

V → a | b

U → < | >

B → ? | ! | @

string : a@a!a

Ambiguous or not.

a@a!a

Page No.
Date : / /

Q2 left most

$E \to EBE \to VBE \to aBE \to a@E$

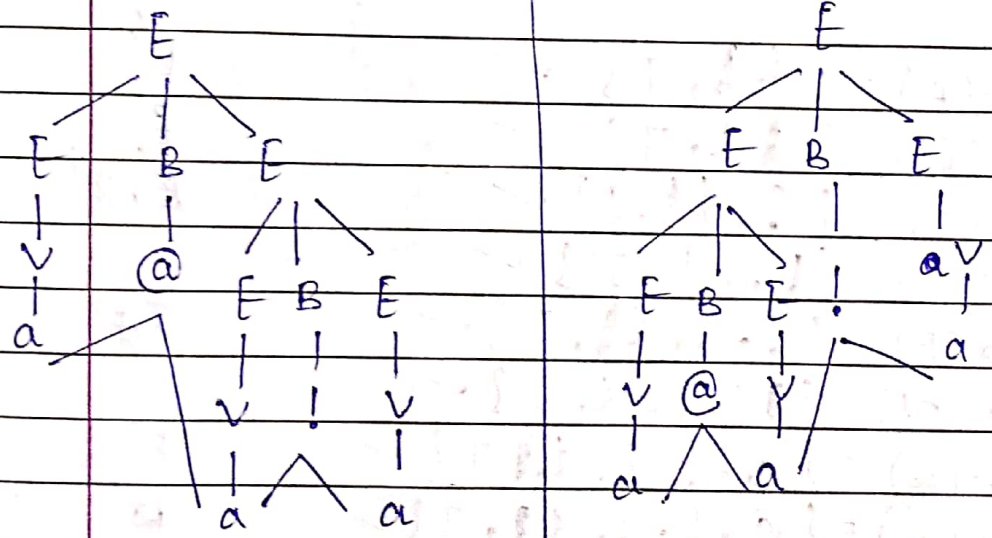
$\to a@EBE \to a@VBE \to a@aBE$

$\to a@a!E \to a@a!V \to \boxed{a@a!a}$

$E \to EBE \to EBEBE \to VBEBE \to aBEBE$

$\to a@EBE \to a@VBE \to a@aBE$

$\to a@a!E \to a@a!V \to \boxed{a@a!a}$



Q3. $E \to TE'$

$E' \to ?E \mid \epsilon$

$T \to FT'$

$T' \to !T \mid \epsilon$

$F \to WF'$

$F' \to @F \mid \epsilon$

$W \to UV \mid V \mid [F]$

$V \to a \mid b$ or

$U \to <\!/\!>$

Page No.
Date : / /

FIRST(E) = { [, < , a , b , ? } ✓
FIRST(E') = { ? , ϵ } ✓
FIRST(T) = { [, < , a , b , ? } ✓
FIRST(T') = { ! , ϵ } ✓
FIRST(F) = { [, < , a , b , ? } ✓
FIRST(F') = { @ , ϵ } ✓
FIRST(W) = { [, < , a , b , ? } ✓
FIRST(V) = { a , b } ✓
FIRST(U) = { < } ✓

Follow(E) = { $,] } ✓ — E'
Follow(E') = { $,] } ✓ — E,
Follow(T) = { ? , $,] } ✓ — E, T'
Follow(T') = { ? , $,] } ✓
Follow(F) = { ! , ? , $,] } ✓ — T, F'
Follow(F') = { ! , ? , $,] } ✓
Follow(W) = { @ , ! , ? , $,] } ✓ — F
Follow(V) = { @ , ! , ? , $,] } ✓ — W
Follow(U) = { a , b } ✓

	?	!	@	[	]	a	b
E				E→TE'		E→TE'	E→T
E'	E'→?E				E'→ϵ		
T				T→F¬T'		F¬F¬T'	T→F
T'	T'→ϵ	T'→!T			T'→ϵ		
F				F→WF'		F→WF'	F→W
F'	F'→ϵ	F'→ϵ	F'→@F		F'→ϵ		
W				W→[E]		W→V	W→
V						V→a	V→
U							

Page No.
Date : / /

FIRST(E) = { [, <, a, b, } ✓
FIRST(E') = { ?, ϵ } ✓
FIRST(T) = { [, <, a, b, } ✓
FIRST(T') = { !, ϵ } ✓
FIRST(F) = { [, <, a, b, } ✓
FIRST(F') = { @, ϵ } ✓
FIRST(W) = { [, <, a, b, } ✓
FIRST(V) = { a, b } ✓
FIRST(U) = { <, } ✓

FOLLOW(E) = { $,] } ✓ E'
FOLLOW(E') = { $,] } ✓ E
FOLLOW(T) = { ?, $,] } ✓ E, T'
FOLLOW(T') = { ?, $,] } ✓
FOLLOW(F) = { !, ?, $,] } ✓ T, F'
FOLLOW(F') = { !, ?, $,] } ✓
FOLLOW(W) = { @, !, ?, $,] } ✓ F
FOLLOW(V) = { @, !, ?, $,] } ✓ W
FOLLOW(U) = { a, b } ✓

	?	!	@	[	]	a	b
E				E→TE'		E→TE'	E→TE'
E'	E'→?E				E'→ϵ		
T				T→FT'		T→FT'	T→FT'
T'	T'→ϵ	T'→!T			T'→ϵ		
F				F→WF'		F→WF'	F→WF'
F'	F'→ϵ	F'→ϵ	F'→@F		F'→ϵ		
W				W→[E]		W→V	W→V
V						V→a	V→b
U							

a@a!a

Page No.
Date : / /

Stack	I/P
$E	$a@a!a$
$E'T	a@a!a$
$E'T'F	a@a!a$
$E'T'F'ω	a@a!a$
$E'T'F'V	a@a!a$
$E'T'F'A	A@a!a$
$E'T'F'	@a!a$
$E'T'F@	@a!a$
$E'T'F	a!a$
$E'T'F'ω	a!a$
$E'T'F'V	a!a$
$E'T'F'A	A!a$
$E'T'F'	!a$
$E'T'	!a$
$E'T'!	!a$
$E'T	a$
$E'T'F	a$
$E'T'F'ω	a$
$E'T'F'V	a$
$E'T'F'A	A$
$E'T'F'	$
$E'T'	$
$E'	$
$	$

→ string accepted

<	>	$
E→TE'	E→TE'	
		E'→∈
T→FT'	T→FT'	
		T'→∈
F→ωF'	F→ωF'	
		F'→∈
ω→UV	ω→UV	
U→<	U→>	

Page No.
Date : / /

Q.4 $S \to X|b$
$X \to S|a$
] left recursion.

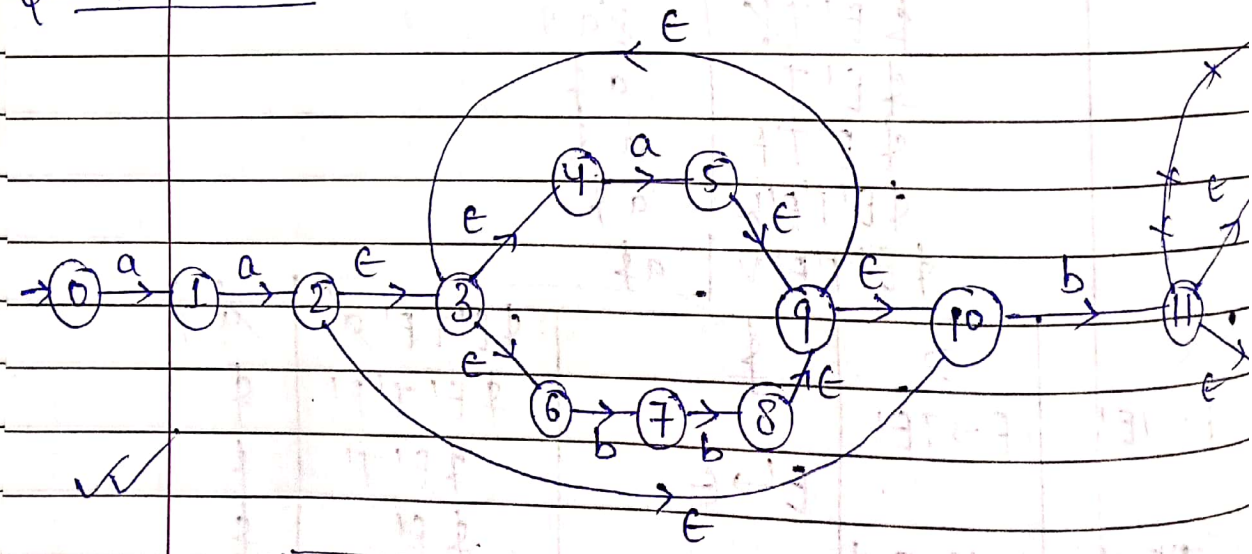
Q. $aa(a|bb)^*b(a|b)$ — Thompson

LL-Parser :-

Q. SLR, 1. $F \to id(P);$
$P \to Pf\ id | id$

2. $T \to B | \{L\}$
$L \to TL | B$
$B \to a|b.$

Q. THOMPSON :



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

Que DFA — from NFA
— direct algo (Parse
— direct from mind

Page No.
Date : / /

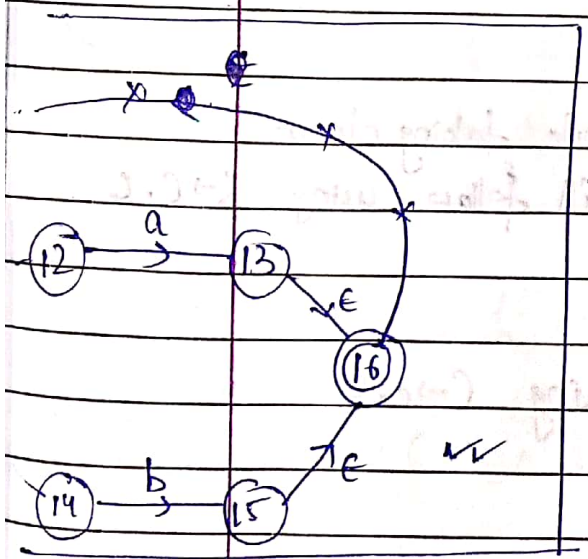
Q4. Left-Recursion

$S \to X \mid b$
$X \to S \mid a$

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

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

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



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. $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

i/p string → abbcde

abbcde
aAbcde (handle)
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

(string that can be obtained by replacing RHS 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. $E \rightarrow E+E \mid E*E \mid (E) \mid id$

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

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"

Page No.
Date: / /

Shifting [illegible]

$ E + E *	Reduce
$ E + E * id	E
$ E + E * E	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.

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

id

Steps: First we make operator-precedence Table & then derive any string using it.

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

Invalid cases

1. id (
2.) id
3. id id
4.) (
5. $ $
6. ($

Page No.
Date: / /

$ <· a and a ·> $

id ·> a [In given grammar, id has higher precedence than any terminal i.e. $+, *, \uparrow$, etc.]

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

Precedence: high → low; Left to right associative

	id	+	*	$
id	—	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	—

→ help to find handle

→ (end of string)

Operator precedence table.

i/p → $ id + id * id $

⇒ $ <· id ·> + <· id ·> * <· id ·> $

⇒ $ <· id ·> $

⇒ $ <· + <· id ·> * <· id ·> $

⇒ $ <· + <· * <· id ·> $

⇒ $ <· + <· * ·> $

⇒ $ <· + ·> $ ⇒ $ $ final

We'll check precedence b/w only all two symbols & will replace them with precedence sign.

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

↑ : right associative.

Page No.
Date : / /

id * (id ↑ id) − id / id

	+	−	*	/	↑	id	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
−	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗	–	–	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	–
)	⋗	⋗	⋗	⋗	⋗	–	–	⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖	–	–

next string starts → $

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

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

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

$ ⋖ * ⋖ (⋖ ↑ ⋗) ⋗ − ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋖ (≐) ⋗ − ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ * ⋗ − ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ − ⋖ id ⋗ / ⋖ id ⋗ $

$ ⋖ − ⋖ / ⋖ id ⋗ $

$ ⋖ − ⋖ / ⋗ $

$ ⋖ − ⋗ $

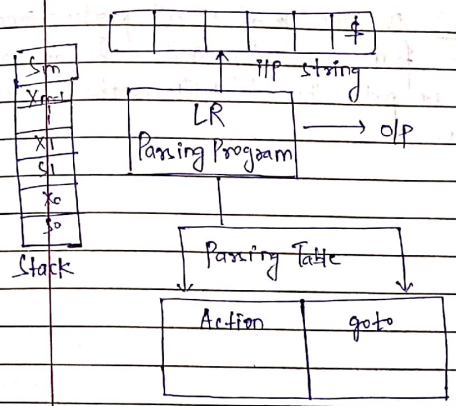
$ $ ✓

LR-Parsers

Page No.
Date : / /

LR-Parsers
- SLR (Simple LR)
- 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

derives start symbol of a given grammar with another symbol with unit production.

i.e. $E' \to E$, E is start symbol

(• is shifted)

$E' \rightarrow .E$

↳ 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 '.'. & we put this '.' with added productions.

Q. $E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

A. 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): $F \rightarrow .(E)$, $F \rightarrow .id$

→ all included in I_0.

Closure (item) ≡ together makes closure (E)

I_0 → represent initial state

• transitions from I_0:-

transition (E): I_1: $E' \rightarrow E.$
$E \rightarrow E.+T$
(closure of + was null + NP)

transition (T): I_2: $E \rightarrow T.$
$T \rightarrow T.*F$
(closure of * was null.)

I_3: $T \rightarrow F.$

I_4: $F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$
(from closure of E)

I_5: $F \rightarrow id.$

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.

Now states, 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: $T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

Transitions for I_4

I_8 : $F \to (E \cdot)$
$E \to E \cdot + T$ = (6) IMP.
$E \to T \cdot$
$T \to T \cdot * F$

∵ part of I_2 & already included.
∴ not to be included in state (I_8)
only transitions made for I_2 in dfg.

$T \to t \cdot$ = I_3

$F \to (\cdot E)$
Also note closure.
{ all productions. } together ≡ I_4

$F \to id \cdot$ ≡ I_5

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

Transitions for I_6

I_9 : $E \to E + T \cdot$
$T \to T \cdot * F$
$T \to F \cdot$ ≡ I_3
~~$F \to (\cdot E)$~~
$F \to (\cdot E)$
{ E-closure } ≡ I_4
$F \to id \cdot$ ≡ I_5

∴ I_9 : $E \to E + T \cdot$
$T \to T \cdot * F$

Transition for I_7

I_{10} : $T \to T * F \cdot$
$F \to (\cdot E)$
{ E-closure } ≡ I_4
$F \to id \cdot$ ≡ I_5

∴ I_{10} : $T \to T * F \cdot$

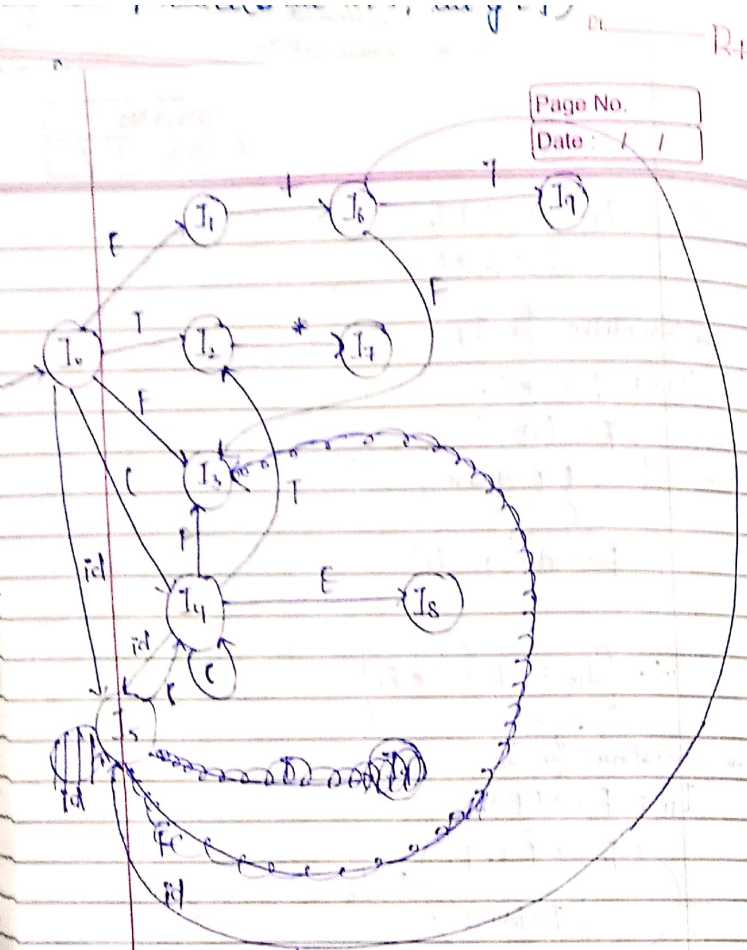
Transition for I_8

I_{11} : $F \to (E) \cdot$
$E \to E + \cdot T$
{ T-closure } ≡ I_6

∴ $I_{11} = F \to (E) \cdot$

Transition for I_9

I_{12} : $T \to T * \cdot F$
{ F-closure } ≡ I_7



** and make further states

Page No.
Date : / /

Q. 0: S' → S
1: S → L=R
2: S → R
3: L → *R
4: L → id
5: R → L

I_0 : S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1 : S' → S.

I_2 : S → L.=R
R → L.

I_3 : S → R.

I_4 : L → *.R
R → .L

I_5 : L → id.

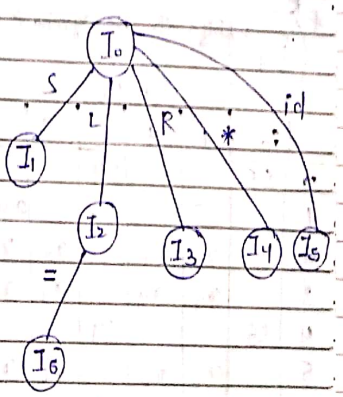
I_6 : S → L=.R
R → .L

I_7 : L → *R.

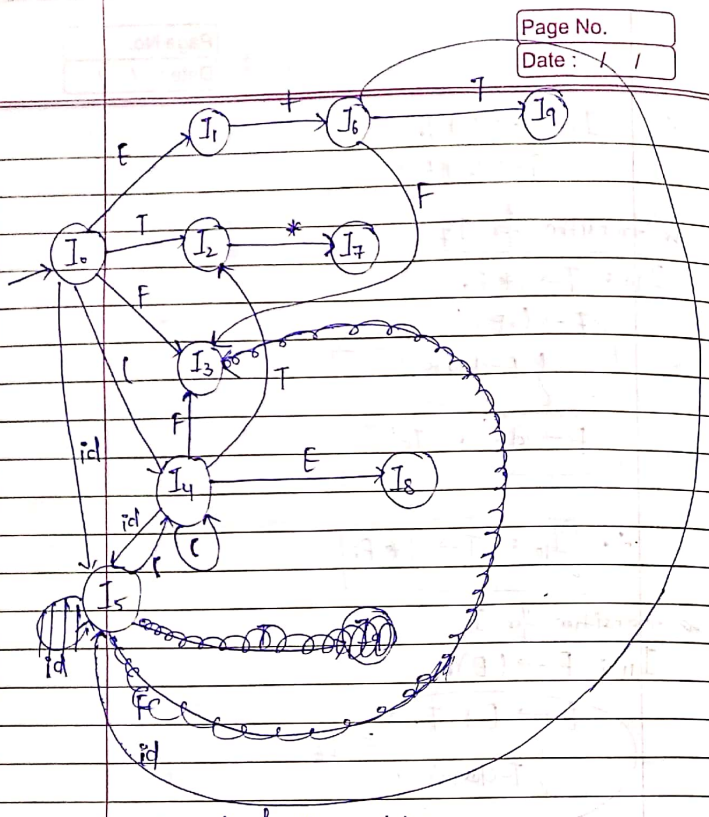
I_8 : R → L.

I_9 : S → L=R.

for L, H will go to I_8



int dot-product(int x[], int y[])



** and make further states

Page No.
Date : / /

S' → 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 : $L \to id.$ ✓ ✓

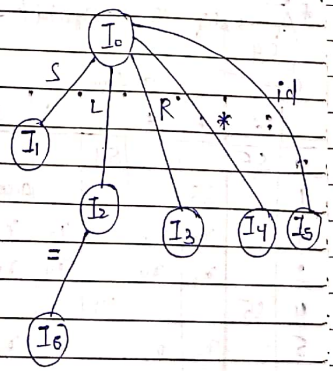
I_6 : $S \to L=.R$ ✓
$R \to .L$ ✓

I_7 : $L \to *R.$ ✓

I_8 : $R \to L.$ ✓

I_9 : $S \to L=R.$ ✓

for L, H will go to I_8



int dot-product(int x[], int y[])

FIRST | FOLLOW

IMP = { We'll not consider E' while constructing parsing table. }

Follow(E') =

FIRST(E) = { (, id }

FIRST(T) = { (, id }

FIRST(F) = { (, id }

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

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

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

IMP { when, will get E' → . will make $ col = accept of that production.

1 2 ... 11 → Represents states.

* → S4 → shift the i/p symbol to stack & move to state (4).

	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		r_2	S7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S5			S4			8	2	3
5		r_6	r_6		r_6	r_6			
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			

r_1: E → E+T
r_2: E → T
r_3: T → T*F
r_4: T → F
r_5: F → (E)
r_6: F → id

∴ No possibility of ambiguity

→ Name all productions ← F → id, F → id

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

IMP!

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

Q.

FIRST(S) = { *, id }

FIRST(L) = { *, id }

FIRST(R) = { *, id }

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			9
6						9
7	r_3		r_3			
8	r_5		r_5			
9			r_1			

set of items.

F → T.

F

A → B

S → (A B)

L = R (A B)

r1 : E → E+T
r2 : E → T
r3 : T → T*F
r4 : T → F
r5 : F → (E)
r6 : F → id

Stack	I/P	Action
0	id + id * id $	Shift
0 id 5	+ id * id $	Reduce (r6)
0 F 3	+ id * id $	r4
0 T 2	+ id * id $	r2
0 E 1	+ id * id $	shift
0 E 1 + 6	id * id $	shift
0 E 1 + 6 id 5	* id $	r6
0 E 1 + 6 F 3	* id $	r4
0 E 1 + 6 T 9	* id $	shift
0 E 1 + 6 T 9 * 7	id $	shift
0 T 9 * 7 id 5	$	r6
0 E 1 + 6 T 9 * 7 F 10	$	r3
0 E 1 + 6 T 9	$	r1
0 E 1	$	accept

Reverse rightmost derivative

rules:

1. if r : A → α.
will pop 2*len(α) from stack & will replace with LHS i.e. (A)

when, S4, (S6) comes it represents, shift i/p to stack and go to state 6.

Que. S → AS | b
A → SA | a

S' → S
S → AS | b
A → SA | a

FIRST(S) = {a, b}
FIRST(A) = {a, b}

FOLLOW(S) = {a, b, $}
FOLLOW(A) = {a, b}

I_0 : S' → .S
S → .AS
S → .b
A → .SA
A → .a

I_1 : S' → S.
A → S.A

I_2 : S → A.S
S → .AS
S → .b
A → .SA
A → .a

I_3 : A → SA.

I_4 : A → S.A
A → .SA
A → .a

I_5 : S → AS.

I_6 : S → b.

I_7 : A → a.

func learn dat

Pg. B+

Page No.
Date : / /

$S' \to S$

$r_1 : S \to AS$

$r_2 : S \to b$

$r_3 : A \to SA$

$r_4 : A \to a$

I_0 : $S' \to .S$, $S \to .AS$, $S \to .b$, $A \to .SA$, $A \to .a$

I_1 : $S' \to S.$, $A \to S.A$, $A \to .SA$, $A \to .a$, $S \to .AS$, $S \to .b$

I_2 : $S \to A.S$, $S \to .AS$, $S \to .b$, $A \to .SA$, $A \to .a$

I_3 : $S \to b.$

I_4 : $A \to a.$

I_5 : $A \to SA.$, $S \to A.S$, $S \to .AS$, $S \to .b$

I_6 : $A \to S.A$, $A \to .SA$, $A \to .a$, $S \to .AS$, $S \to .b$

I_7 : $S \to AS.$, $A \to S.A$

I_8 : $A \to SA.$

$A \to .a$	4
$A \to .a$	5
$S \to .AS$	6
$S \to .b$	7

Page No.
Date : / /

$S \to a, b, \$$

$A \to a, b, \$$

	a	b	$	A	S
0	s3, s4	s3, s4	s3	2	1
1	s4, s3	s4, s3	accept, s3	5	6
2	s3, s4	s3, s4	s3	2	7
3	r2	r2	r2		
4	r4	r4			
5	s3, s4	s3, s4	s3	2	7
6	s3, s4	s3, s4	s3	5	6
7				5	

	a	b	$	A	S
0	s4	s3		2	1
1	s4	s3	accept	5	6
2	s4	s3		2	7
3	r2	r2	r2		
4	r4	r4	r4		
5	s4, r3	s3, r3	r3	2	7
6	s4	s3		5	6
7	r1, s4	r1, s3	r1,	5	6

func beggin dat - one-1 }

Questions

Page No.
Date : / /

(1). S → A
A → aB | Ad
B → bBC | f
C → g

String (abfg)

FIRST(S) = {a}	FOLLOW(S) = {$}
FIRST(A) = {a}	FOLLOW(A) = {d, $}
FIRST(B) = {b, f}	FOLLOW(B) = {g, d, $}
FIRST(C) = {g}	FOLLOW(C) = {g, d, $}

	a	d	b	f	g	$
S	S → A					
A	A → aB A → Ad					
B						
C						

Ans. LL parser → Left Recursion & left factoring

Page No.
Date : / /

Q.1. S → A
A → aB | Ad
B → bBC | f
C → g

String (abfg)

A → Ad | aB
α β

A' → aBA'
A' → dA' | ∈

S → A
A → aBA'
A' → dA' | ∈
B → bBC | f
C → g

FIRST(S) = {a}	FOLLOW(S) = {$}
FIRST(A) = {a}	FOLLOW(A) = {$}
FIRST(A') = {d, ∈}	FOLLOW(A') = {$}
FIRST(B) = {b, f}	FOLLOW(B) = {d, g, $}
FIRST(C) = {g}	FOLLOW(C) = {a, g, $}

	a	b	d	f	g	$
S	S → A					
A	A → aBA'					
A'			A' → dA'			A' → ∈
B		B → bBC		B → f		
C					C → g	

func begin dat - num.1 }

Questions

Page No.
Date : / /

(1). S → A
A → aB | Ad
B → bBC | f
C → g

String (abfg)

FIRST(S) = {a}	FOLLOW(S) = {$}
FIRST(A) = {a}	FOLLOW(A) = {d, $}
FIRST(B) = {b, f}	FOLLOW(B) = {g, d, $}
FIRST(C) = {g}	FOLLOW(C) = {g, d, $}

	a	d	b	f	g	$
S	S→A					
A	A→aB A→Ad					
B						
C						

WIP. Q.1. parser → left recursion & left factoring

Page No.
Date : / /

S → A
A → aB | Ad
B → bBC | f
C → g

String (abfg)

A → Ad | aB
α β

A' → aBA'
A' → dA' | ∈

S → A
A → aBA'
A' → dA' | ∈
B → bBC | f
C → g

FIRST(S) = {a}	FOLLOW(S) = {$}
FIRST(A) = {a}	FOLLOW(A) = {$}
FIRST(A') = {d, ∈}	FOLLOW(A') = {$}
FIRST(B) = {b, f}	FOLLOW(B) = {d, g, $}
FIRST(C) = {g}	FOLLOW(C) = {a, g, $}

	a	b	d	f	g	$
S	S→A					
A	A→aBA'					
A'			A'→dA'			A'→∈
B		B→bBC		B→f		
C					C→g	

int dot_product(int x[], int y[])

func ...

CANONICAL LR PARSER

Page No.
Date : / /

S' → S

S → CC : r1
C → cC : r2
C → d : r3

→ $ will follow (S).

I_0 : S' → .S, $ ✓
S → .CC, $ ✓
C → .cC, c/d ✓
C → .d, c/d ✓

I_1 : S' → S., $ ✓ (when making transition i.e. shifting dot, copy follow of parent production.)

I_2 : S → C.C, $ ✓
C → .cC, $ ✓
C → .d, $ ✓
] while taking closure, fill follow using S → C.C

I_3 : C → c.C, c/d ✓
C → .cC, c/d ✓
C → .d, c/d ✓
] using C → c.C

I_4 : C → d., c/d ✓

I_5 : S → CC., $ ✓

I_6 : C → c.C, $ ✓
C → .cC, $ ✓
C → .d, $ ✓

* SLR Parser is a weak Parser.
There are chances of both shift & reduce for same symbol.

⇒ Canonical Parser is comparatively strong than SLR.
(But no. of states in canonical parser are high.
∴ More computation.)

In canonical, to avoid shift & reduce, we use lookahead symbol.
∴ It has, LR(1) set of items

In SLR, we do not use lookahead.
∴ High chances of shift and reduce.
It has, LR(0) set of items.

Diff.

CANONICAL RULES:

A → α.Bβ, a
B → γ, first(β) & if, β is ε: follow(A)

I_7 : C → d., $ ✓

I_8 : C → cC., c/d ✓

I_9 : C → cC., $ ✓

func begin dat...

	c	d	$	S	C
0	S3	S4		1	2
1			accept		
2	S6	S7		5	5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

* LALR Parser

LALR Parser
(here, we will merge states whose productions are same and lookahead are diff$.

disadvantage
no. of states get reduced.

	c	d	$	S	C
0	S 36	S 47		1	2
1			accept		
2	S 36	S 47			5
36	S 36	S 47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

* LALR is less efficient than canonical

Semantic Analysis

⇒ When we put values of variables then it forms a annotated parse tree.

- inherited attribute : (inherit parent value)
- synthesized attribute : (combine values of two nodes)

annotated parse tree is i/p to intermediate code generation.

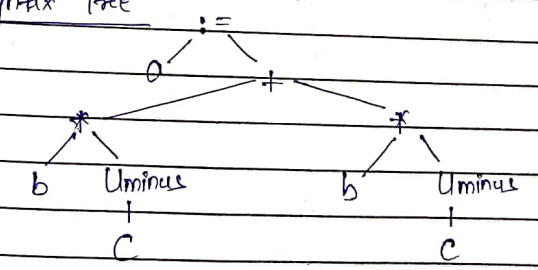
Intermediate Code Generation :-

i) Post-fix notation
ii) Syntax tree / DAG → Directed acyclic graph
iii) 3-address code

Eg. a := b * -c + b * -c (unary operator)

① abc -* bc -*+ := (post-fix)

② Syntax Tree



Func length dot

Page No.
Date : / /

DAG)

8).

(5) i. Address code.

1. $t1 = -c$
2. $t2 = b * t1$
3. $t3 = -c$
4. $t4 = b * t3$
5. $t5 = t2 + t4$
6. $a = t5$

{ all the temporary needs to be entered in symbol Table. } variables

address of rows only mentioned in case of * (a=b)

* Ways of storing & Address code in system.

i) Quadruples
ii) triples
iii) Indirect triples

(i) Quadruples (4 col)

	op	arg1	arg2	result
(0)	−	c		t1
(1)	*	b	t1	t2
(2)	−	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

these are ptr to symbol table for corresponding values.

(ii) Triples (3 col)

(instead of writing temporary variables, we mention address of rows containing those operations.)

	op	arg1	arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

IIT Kharagpur
TV Channel

(iii) Indirect Triples :

(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	-	c	
(15)	*	b	(14)
(16)	-	c	
(17)	*	b	(16)
(18)	+	(1)	(17)
(19)	=	a	(18)

Eg x[i] := y

x[i] (for LHS)

	op	arg1	arg2
(0)	[]:=	x	i
(1)	=	(0)	y

x = y[i]

y[i] (for RHS)

	op	arg1	arg2
(0)	=[]	y	i
(1)	=	x	(0)

Quadruple

x[i] = y

op	arg1	arg2	result
[]=	x	i	t1
=	y		t1

x = y[i]

op	arg1	arg2	result
=[]	y	i	t1
=	t1		x

param x1
param x2
param x3

Call p,3
return
return y.

y[i] = a

```
begin
  prod := 0;
  i := 1;
  do begin
       prod := prod + a[i] * b[i];
       i = i+1
     end
  while i<=20
end.
```

→ no need to write 3address code for keywords such as begin, end, int, float, etc.
→ this info is already present in symbol table.

→ Allot line no. to each 3-address code.

#3-Address code :-

(1) prod = 0
(2) i = 1
~~(3) if (i>20) goto (13)~~ IMP.
(4) $t_1 = i * 4$ // here we are assuming, int takes 4 bytes.
(5) $t_2 = a[t_1]$
(6) $t_3 = i * 4$ ⇒ real address is. (i*4) (index)
(7) $t_4 = b[t_3]$
(8) $t_5 = t_2 * t_4$
(9) $t_6 = prod + t_5$
(10) prod = t_6
(11) $i = i + 1$
(12) ~~goto(3)~~ if (i<=20) goto (4)
~~(13) exit~~

IMP

eg. for (i=0; i<5; i++)
{
}

(1) i = 0
(2) if (i >= 5) goto (L) → this label is end for loop or to be encountered in next steps.
(3) 1- address code for body of loop.
(n) i = i+1 // inc. statement at last
(L1) goto (2)
(L) Exit

→ Similarly for ~~do while~~ and while loop.
→ In case of do while include 'if' condition at last of body.

* By default assume ~~assume~~ array to be of int type i.e. 4 bytes
else, according to the given que.

```
for (i=0 ; i<10 ; i++)
    d++ = x[i] * y[i];

return d;
}
```

3 Address code :-

fn name.

(1) func begin dot product
(2) d=0
(3) i=0
(4) if (i>=10) goto (14)
(5) t1 = 4*i