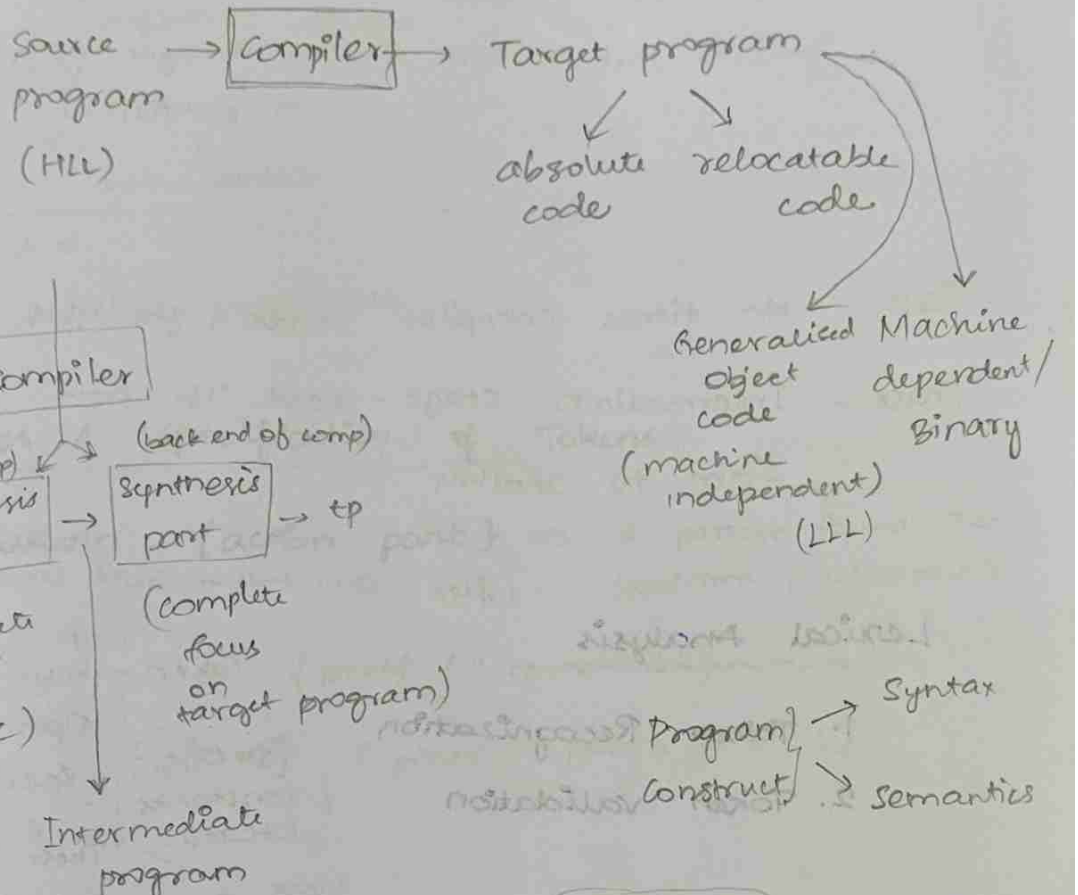


Program construct → Syntax
→ Semantics

I/p → target program → O/p

Analysis of the src

1. Linear analysis (lexical analysis)
2. Hierarchical analysis (Syntax analysis) (CFG)
3. Semantic analysis

(natural language) recursive enumerable - Type 0 most generous language

CSG - Type 1

CFG - Type 2

Regex - Type 3 least generous

In Java, Intermediate code - byte code

Analysis part - Javac

Synthesis part - JVM (which changes according to platforms)

Pass - No. times compiler reads the i/p

Phase - Intermediate stage - read i/p from one form to another

(flex - Linux tool to build lexical analyser)

Lexical Analysis

1. Token Recognization
2. Token validation

Token - keywords, Operators, Special chars. (These 3 are specific tokens (bcoz predefined))

Identifiers, constants, (literals) (user defined)

Token specification

- Lexeme - seq. of symbols → m ma mai main
- Pattern - structure of token
- Token

lex tool

lex specification (*.l) → lex → lex.yy.c (produces 'C' equivalent version)

lex.yy.c → C compiler → o/p

Ex,

```
$ lex f1.l
$ cc lex.yy.c
$ ./a.out
```

Lex program structure

```
%{
    definition section
%}
%%
    rule section
%%
Auxilary procedures
```

Rules & specification of Tokens

pattern {action part} => If pattern found then perform {action part}

Ex,

---> a {printf ("character");}

only a

---> [a-z] { printf ("from a to z"); }

from 'a' to 'z'

---> a^* (kleen closure)

zero or more occurences of 'a'

---> a^+ (positive closure)

1 or more occurences of 'a'

$[a-z]^*$ $[a-z]^3$

$[0-9]^+$ ---> for integer

Ex of program,

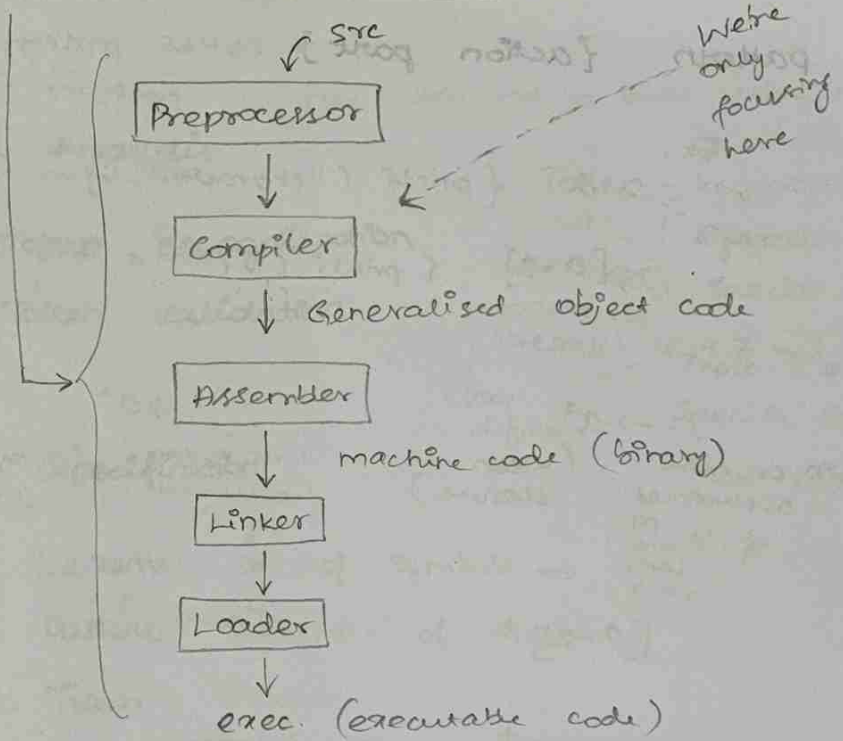
"f1.l"

```
%{
  # include <stdio.h>
%}
%%
[0-9]+  { printf ("%s is an integer", yytext); }
.       { }
%%
```

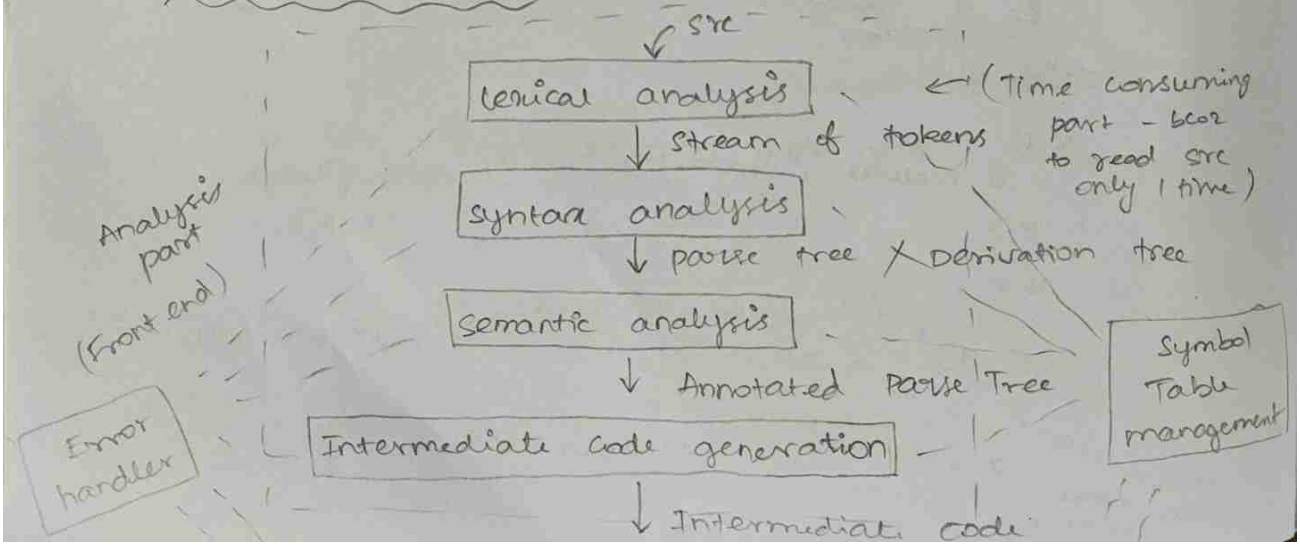
Other than above patterns.

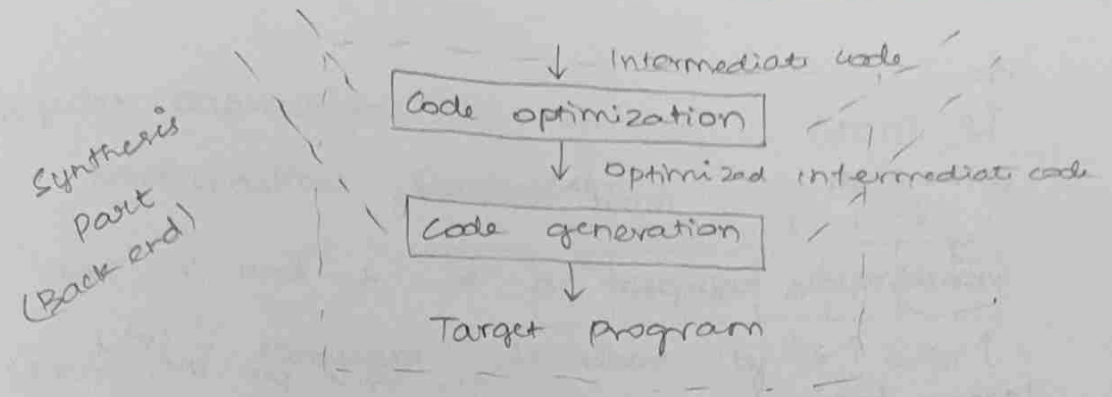
```
int main(){
    yylex();
    return 0;
}

int yywrap(){
    return 1;
}
```

Language Processing System



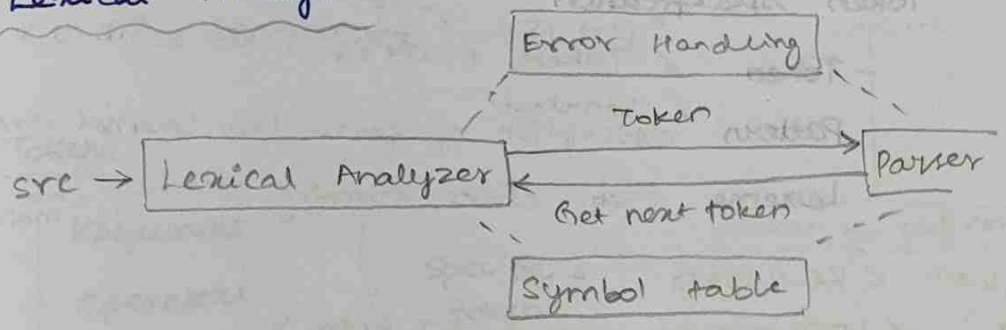
Phases of compiler (Structure of compiler)





Error handler & Symbol Table management phases supports all the 6 phases in the compiler

1. **Lexical Analysis** :



① Scanning - char by char, left to right

② Lexical Analysis (actual) - Token recognization
- Token Validation
- Keep track of line no.
- Identifier - make entry to Symbol table (after recognition) (keeps pointer to the symbol table)
- Error handling
- Stripping out delimiters (white spaces, tab, newline, comments)

Token recognization: keyword, operator, identifier, literal, spl. char

Done on behalf of other parts of compiler

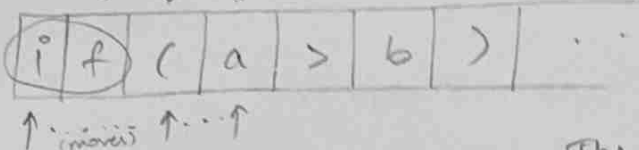
For efficiency, in scanning process instead of requesting char by char, request them in bulk and store it in buffer like in next pg

for,

if (a>b) . . .

forward ptr

Input Buffering



lexeme_begin ptr

The fwd ptr increments till a pattern is found with lexeme_begin ptr & forward ptr

(2.1) Token recognization

Token specification — type, optional

- **Token** < Token name , Token attribute >
- **Pattern** (structure/description of form how Lexemes are framed)
- **Lexeme** (seq. chars forming tokens) {m, ma, mai, main - token}

Token < KEYWORD , if >

ex: < IDENTIFIER , ptr to ST Entry >

Token recognition → Regular Expression

Token validation → Finite Automata

String - finite sequence of symbols - alphabets (Σ)

len of string - $|s|$ - no. symbols

Language - collection of strings over Σ

ϵ - empty string

Union, Concatenation, Kleene, Positive

Parts of a string

Prefix

Suffix

Substring (continuous)

Subsequence (not continuous)

Regular Expressions

Mathematical Symbolism

Let r and s be a regular expression

$L(r)$ - language described by r

ϵ denoting $L(\epsilon) = \{\epsilon\}$

$a \in \Sigma$, a is regex $L(a) = \{a\}$

$(r|s)$ be regex $L(r) \cup L(s)$ — or, union

$(r.s)$ be regex $L(r) . L(s)$ — dot, concat

$r*$ be regex $(L(r))^*$ — closure

Tokens

token → pattern

- Keywords
- Operators
- Special chars

} Specific token

- Identifiers
- Literals

} User defined

a	character a
"s"	string s
.	any char
^	beginning of line
#	end of line
[s]	one char in string s
[a-z]	any one char in range
[^s]	any char not in string s

(r - a symbol)

r*	zero or more occurences of r
r+	one or more occurences of r
r?	zero or one occurence of r
r{m:n}	between m & n occurences of r

Keywords

if → if

main → main

⋮

Operators

arith_op → + | - | * | / | %.

rel_op → < | <= | > | >= | == | !=

⋮

special char

punct → { | } | (|) | [|]

Identifiers

id → letter (letter | digit)* for specifying len can give ()^5 or (){5}.

letter → a | b | . . . | z (or) [a-z]

digit → [0-9]

Literals

- Numeric
  - Integer
  - real no / fractional
  - scientific

Pattern for Integers,

Integer → digit +

digit → 0 | 1 ... | 9

Pattern for Fractional numbers,

Fraction → digit +. digit +

digit → 0 | 1 ... | 9

Pattern for Scientific numbers.

Exponent → digit +. digit + (E(+|-)? digit +)

digit → 0 | 1 ... 19

num → Integer (Fraction)? (Exponent)?

num → [0-9]+ (.[0-9]+)? (E(+|-)? [0-9]+)?

Pattern for delimiters

delim → ws | tab | cr, newline

if (a[i] > N):

<KEYWORD, "if"> <SPLCHAR, '('>

<IDENTIFIER, ptr. to ST>

DFA :

All possible paths shd be defined

Transitions are unique

'ε' move is not used

NFA :

No need to define all possible paths

Transitions can be multiple

'ε' move is used

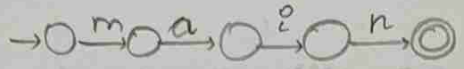
Token Recognization

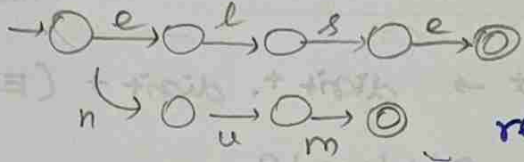
$N = \{Q, \Sigma, \delta, q_0, F\}$

Automata
- DFA
- NFA

Recognizer for keywords

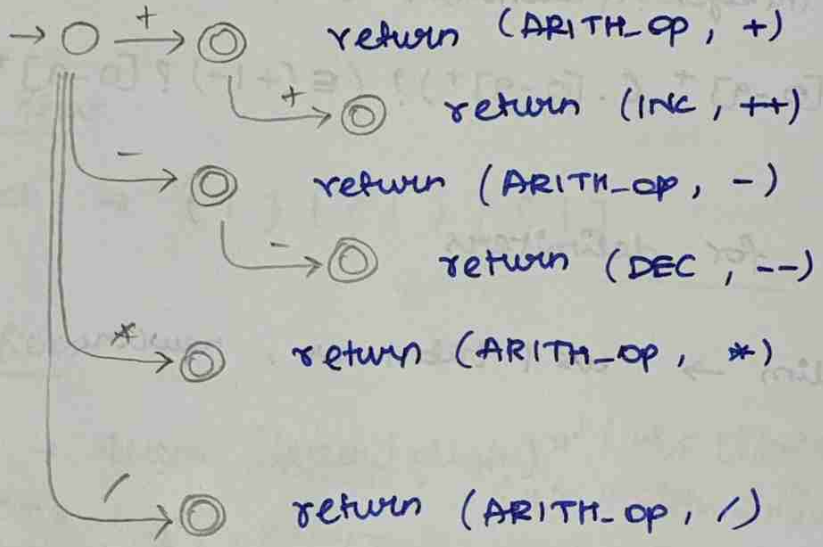
If → if  return (kw, "if")

main → main  return (kw, "main")

else → else  return (kw, "else")

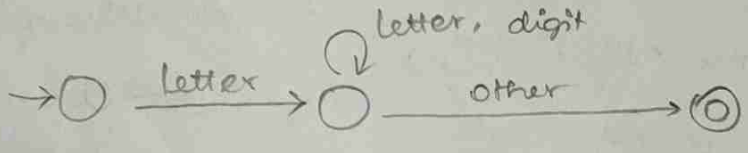
return (kw, "enum")

Recognizer for operators



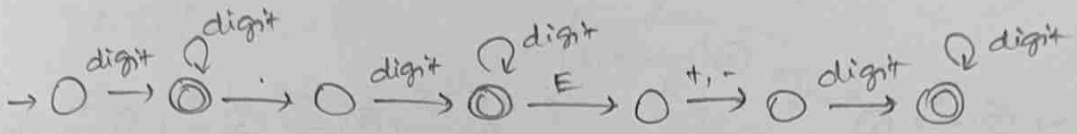
Recognizer for Identifiers

id → letter (letter | digit) *



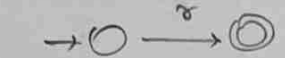
Recognizer for Literals

num → [0-9]+ (. [0-9]+)? (E (+|-)? [0-9]+)?

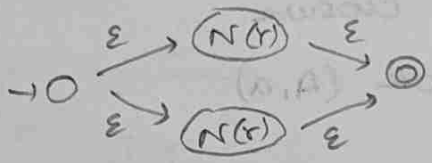


Regular Expression - NFA

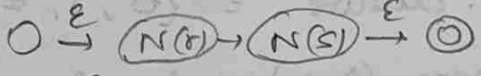
Let r be r.e



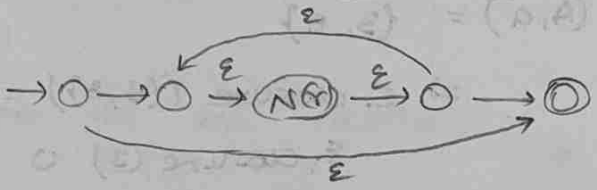
Let r|s be r.e



Let r.s be r.e

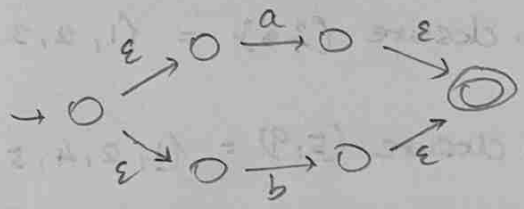


r*

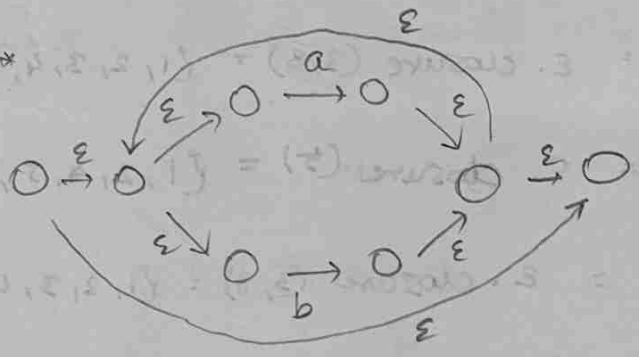


Q) Obtain the DFA equivalent from the given RE, (a|b)* abb

a|b

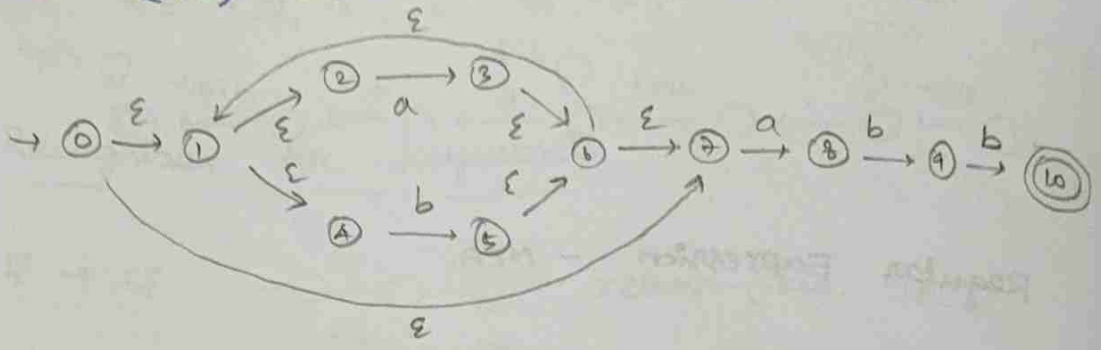


(a|b)*



(a|b)* abb

9) Obtain the DFA equivalent from the given RE (a|b)* abb.



1. ε closure
2. move (A, a)

ε. closure = {0, 1, 2, 4, 7} – (A)

move (A, a) = {3, 8}

⇒ ε. closure ({3,8}) = {1, 2, 3, 4, 6, 7, 8}

ε closure (3) ∪ ε closure (8) – (B)

move (A, b) = {5}

⇒ ε. closure({5}) = {1, 2, 4, 5, 6, 7} – (C)

move (B, a) = ε. closure {3 8} = {1, 2, 3, 4, 6, 7, 8} – (B)

move (B, b) = ε. closure (5, 9) = {1, 2, 4, 5, 6, 7, 9} – (D)

move (C, a) = ε. closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} – (B)

move (C, b) = ε. closure (5) = {1, 2, 4, 5, 6, 7} – (C)

move (D, a) = ε. closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} – (B)

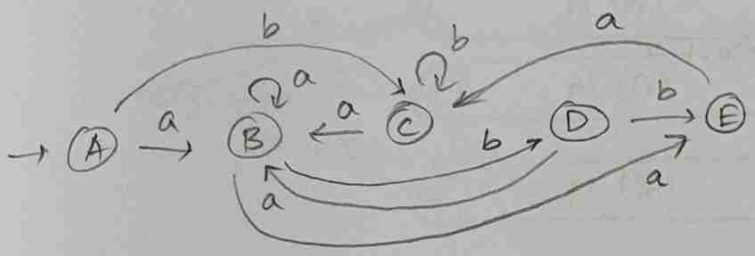
move (D, b) = ε. closure (5, 10) = {1, 2, 4, 5, 6, 7, 10} – (E)

move (E, a) = ε. closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} – (B)

move (E, b) = ε. closure (5) = {1, 2, 4, 5, 6, 7} – (C)

Thus, the DFA is defined by

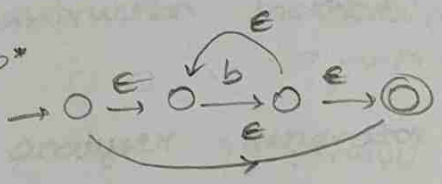
	Pres. state	I/P = a	I/P = b
A	{0, 1, 2, 4, 7}	B	C
B	{1, 2, 3, 4, 6, 7, 8}	B	D
C	{1, 2, 4, 5, 6, 7}	B	C
D	{1, 2, 4, 5, 6, 7, 9}	B	E
E	{1, 2, 3, 4, 6, 7, 10}	B	C



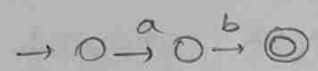
6) Construct DFA for RE (ab*ab)

a

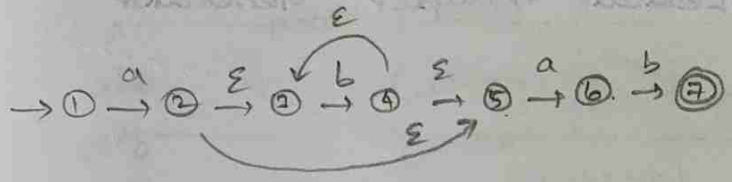
b*



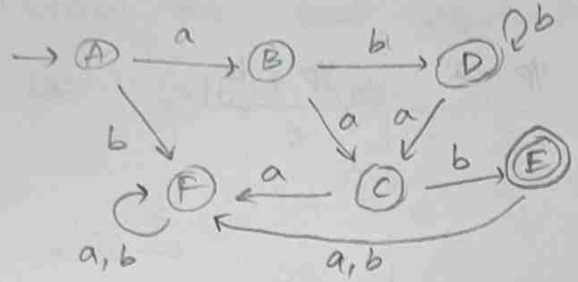
ab



(ab* ab)



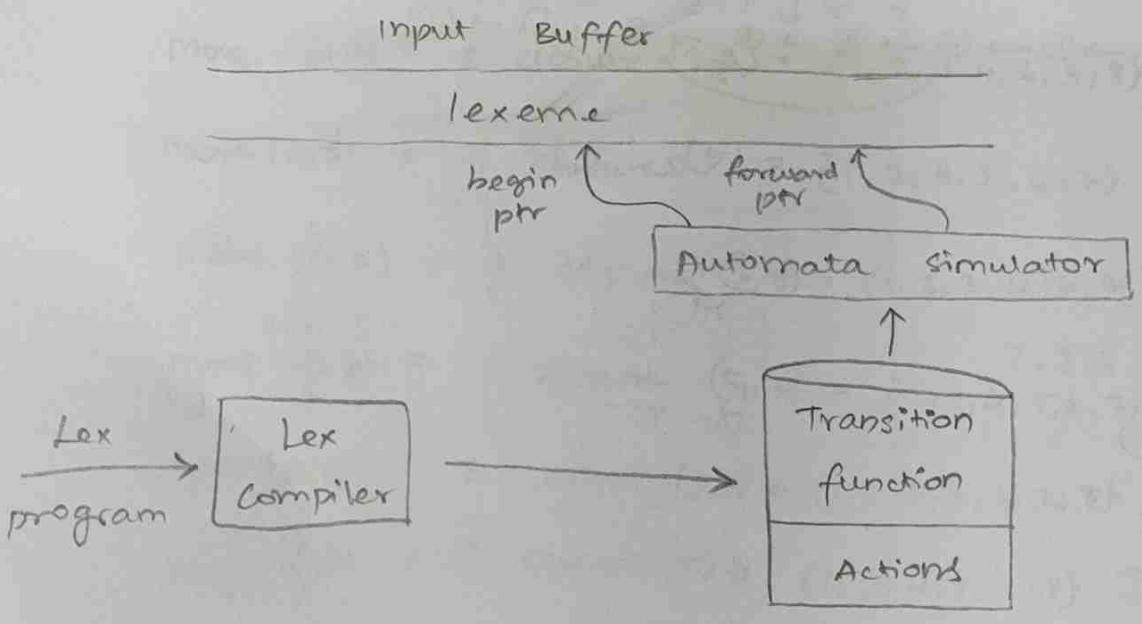
	a	b
→ (A) 1	2, 3, 5 (B)	ϕ (F)
(B) 2, 3, 5	6 (C)	3, 4, 5 (D)
(C) 6	ϕ (F)	7 (E)
(D) 3, 4, 5	6 (C)	3, 4, 5 (D)
* (E) 7	ϕ (F)	ϕ (F)
(F) ϕ	ϕ (F)	ϕ (F)



Lexical Analyser Generator (LEX)

```
%{
#include <stdio.h>
%}
%%
%%
int main(){
    yylex();
    return 0;
}
void yywrap(){
    return 1;
}
```

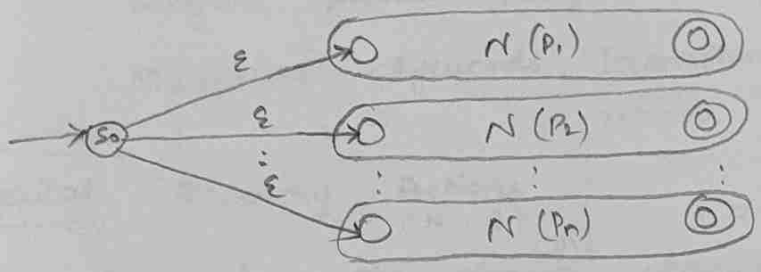
Design of Lexical Analyser Generator



Steps

1. For each pattern P_i (in lex prg), convert it to NFA N_i, $i = 1, 2, \ldots n$

2. Combine all the NFA's by introducing a common start state, create a ϵ-transition to the start state for each N_i

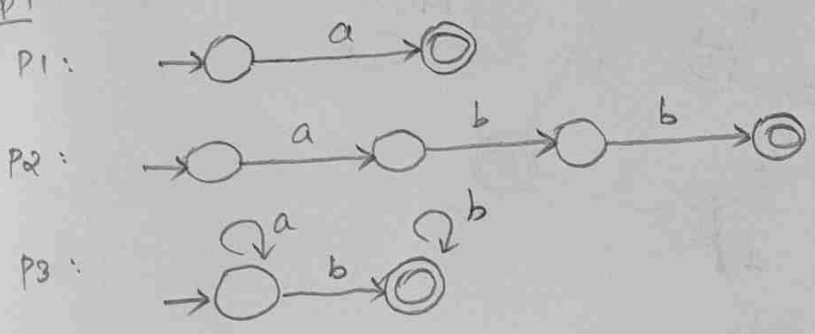


3. Convert the NFA into its equivalent DFA by using Subset construction method.

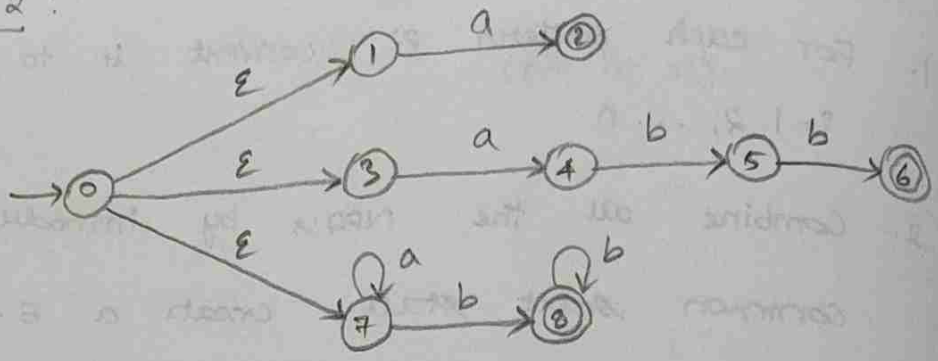
Qn) Design a lexical analyser generator for the following segment of .lex program.

```
:
%%
a        printf {action A1 for pattern P1}
abb      {action A2 for pattern P2}
a*b+     {action A3 for pattern P3}
%%
:
```

Step 1 :

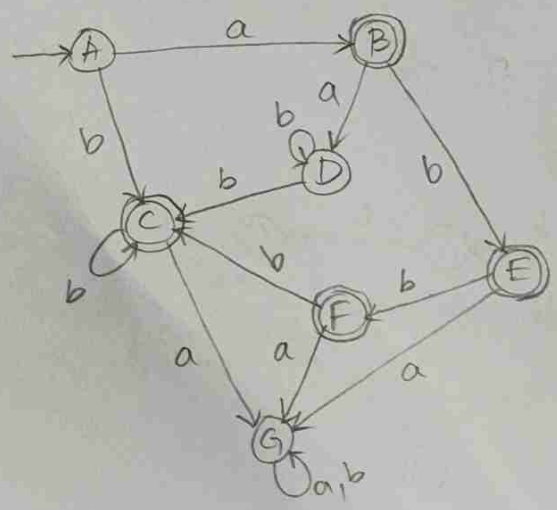


Step 2 :



Step 3 :

Present States	a	b
→ (A) {0, 1, 3, 7}	(B) {2, 4, 7}	(C) {8}
* (B) {2, 4, 7}	(D) {7}	(E) {5, 8}
* (C) {8}	(G) ∅	(C) {8}
(D) {7}	(D) {7}	(C) {8}
* (E) {5, 8}	(G) ∅	(F) {6, 8}
* (F) {6, 8}	(G) ∅	(C) {8}
(G) ∅	(G) ∅	(G) ∅



Lexical Analysis and symbol Table mgmt

Make an entry into S T

Lexical Analysis and Error Handling

Lexical Errors

Strange char

Longest quoted string

Misspelled keywords, Identifier

Lexical - Recovery Actions

Removing the strange character

Add the missing character

Transposing characters

parameter passing Mechanism

Call - by - Value

Call - by - reference

Copy - Restore calling to called — copy
called to calling — ref.

call - by - name

Optimization of DFA based Pattern Matchers

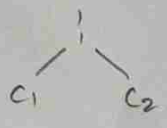
Steps:

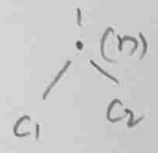
1. Concatenate given R.E r with right end marker #
2. Build a syntax Tree for r.

   Leaves - Operands (corresponds to)

   Intermediate node - or-node (|), cat-node (.) } binary; star-node (*) } unary
3. Compute Nullable(), Firstpos(), Lastpos() for every node in T
4. Compute Followpos()
5. DState & DTran

Algorithm: Nullable, Firstpos(), Lastpos()

Node in T	Nullable	Firstpos()	Lastpos()
→ Leaf node (ϵ)	True	∅	∅
→ Leaf node with operand at position 'i'	False	{i}	{i}
→ or-node: n 	Nullable(c_1) or Nullable(c_2)	Firstpos(n) = Firstpos(c_1) ∪ Firstpos(c_2)	Lastpos(n) = Lastpos(a) ∪ Lastpos(c_2)

→ Cat-node: n 	Nullable(c_1) and Nullable(c_2)	If (c_1 is Nullable) { Firstpos (n) = Firstpos (c_1) ∪ Firstpos (c_2) } else { Firstpos (c_1) }	If (c_2 is Nullable) { Lastpos (n) = Lastpos (c_1) ∪ Lastpos (c_2) } else { Lastpos (c_2) }
→ Star-node: n 	True	Firstpos(n) = Firstpos (c_1)	Lastpos(n) = Lastpos (c_1)

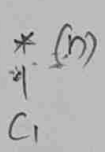
Followpos(i) - only for 2 nodes

1. Cat node

For every i in Lastpos (c_1),

followpos (i) = followpos (i) ∪ Firstpos (c_2)

2. Star node



For every i in Lastpos (n),

followpos (i) = followpos (i) ∪ Firstpos (n)

Q) Construct optimised DFA from R.E

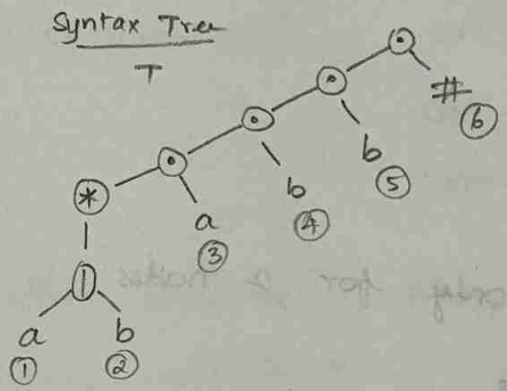
$r = (a|b)^* abb$

Soln,

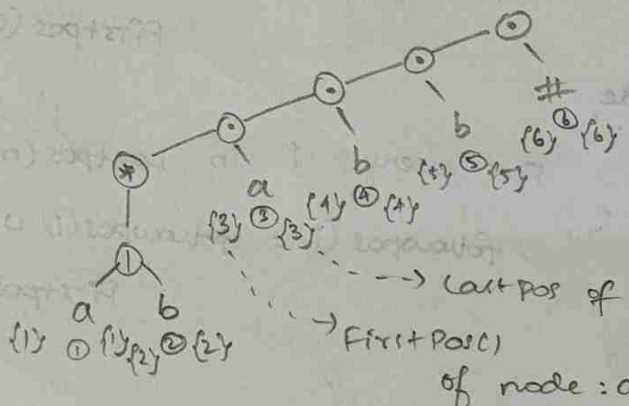
Step 1 :

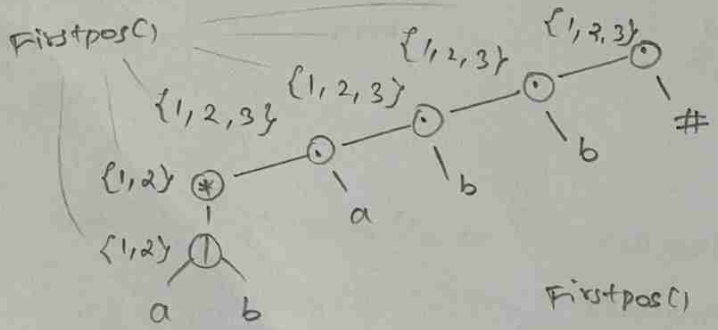
$r = (a|b)^* abb\ \#$

Step 2 :



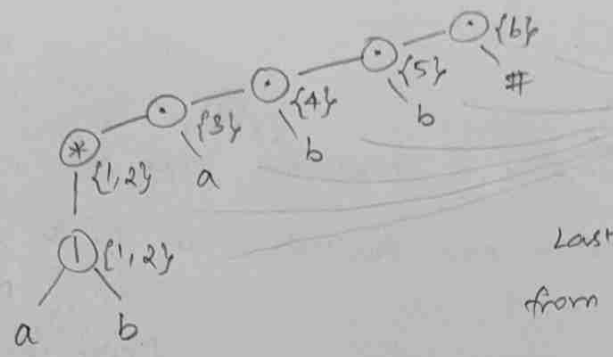
Step 3 : (From table)





(* - is nullable)

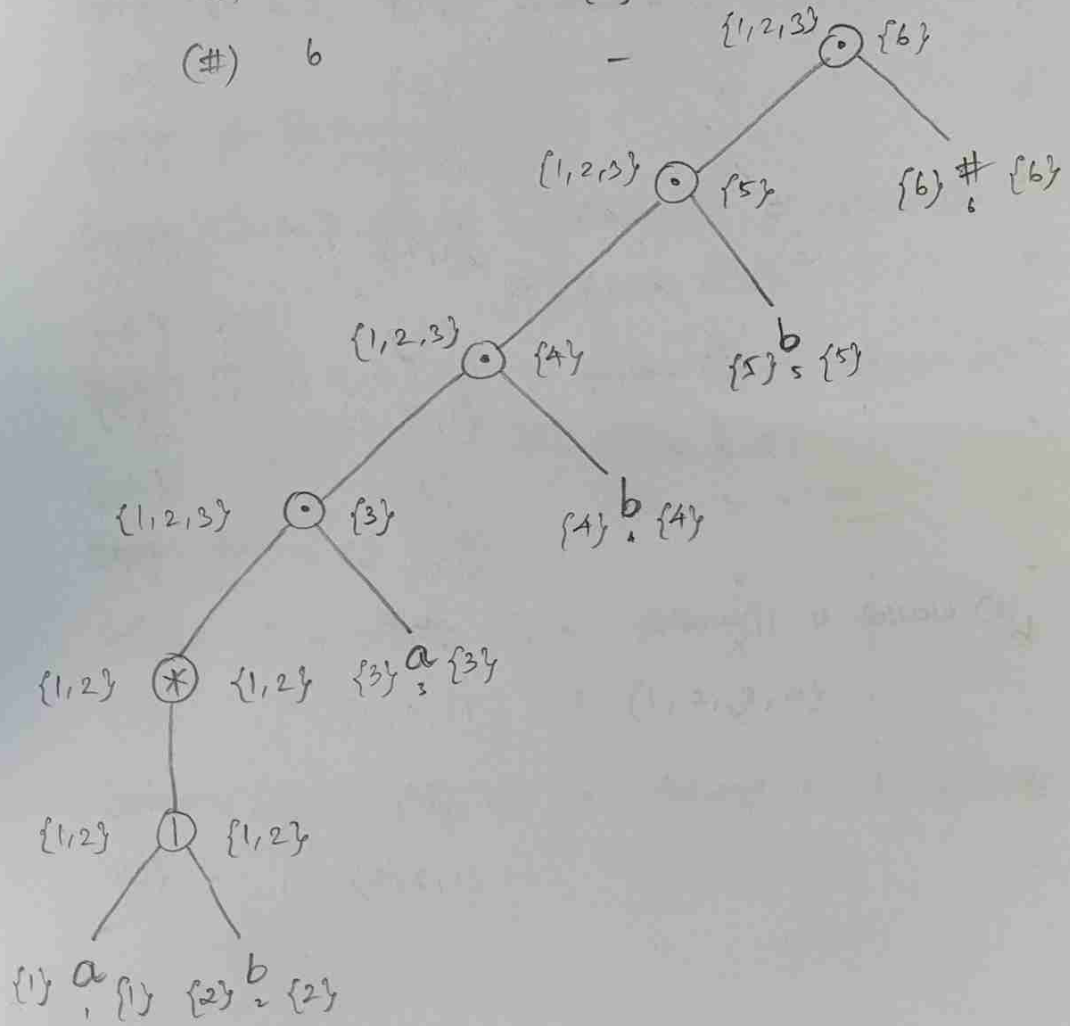
Firstpos() is calculated from table & above tree



Lastpos() is calculated from table & above tree

Step 4:

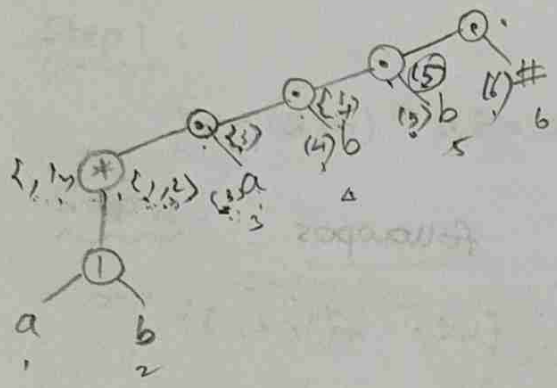
Position		followpos
(a)	1	~~{1,2}~~ {1, 2, 3}
(b)	2	~~{1,2}~~ {1, 2, 3}
(a)	3	{4}
(b)	4	{5}
(b)	5	{6}
(#)	6	–



(*) - Nullable

(|) - any one child is Nullable then it is Nullable

(.) - both children are Nullable then it is Nullable



$f(5) = \phi \cup 4$

$f(1) = 1,2$

$f(2) = 1,2$

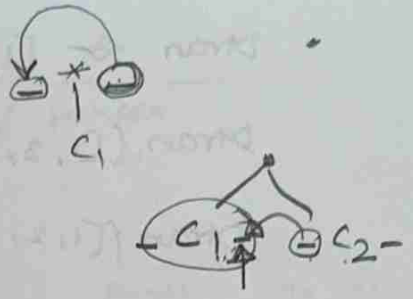
$f(1) = (1,2) \cup (3)$

$f(3) =$

Nodes in T	Nullable	Firstpos	Lastpos
a	×	{1}	{1}
b	×	{2}	{2}
|	×	{1,2}	{1,2}
*	✓	{1,2}	{1,2}
a	×	{3}	{3}
.	×	{1,2,3}	{3}
b	×	{4}	{4}
.	×	{1,2,3}	{4}
b	×	{5}	{5}
.	×	{1,2,3}	{5}
#	×	{6}	{6}
.	×	{1,2,3}	{6}

Followpos

a	1	{1,2}	{1, 2, 3}
b	2	{1,2}	{1, 2, 3}
a	3	{4}	
b	4	{5}	
b	5	{6}	
#	6	—	



Step 5 :

Dstate & Dtran

$a^* = \epsilon, a, aa, aaa$

Dstate = {[1, 2, 3]} (Firstpos of root node)

Dtran for [1, 2, 3]

Dtran ([1, 2, 3], a) = follow (1) ∪ follow (3)

= {1, 2, 3, 4} ↳ new state

1,3 are taken coz they rep 'a' in followpos

Dtran ([1, 2, 3], b) = follow (2) = {1, 2, 3}

2 is taken coz it rep 'b' in followpos

Dtran for [1, 2, 3, 4]

Dtran ([1, 2, 3, 4], a) = follow (1) ∪ follow (3)

= {1, 2, 3, 4}

Dtran ([1, 2, 3, 4], b) = follow (2) ∪ follow (4)

= {1, 2, 3, 5}

↳ new state

Dtran for [1, 2, 3, 5]

Dtran ([1, 2, 3, 5], a) = follow(1) ∪ follow (3)

= {1, 2, 3, 4}

Dtran ([1, 2, 3, 5], b) = follow (2) ∪ follow(5)

= {1, 2, 3, 6}

↳ new state

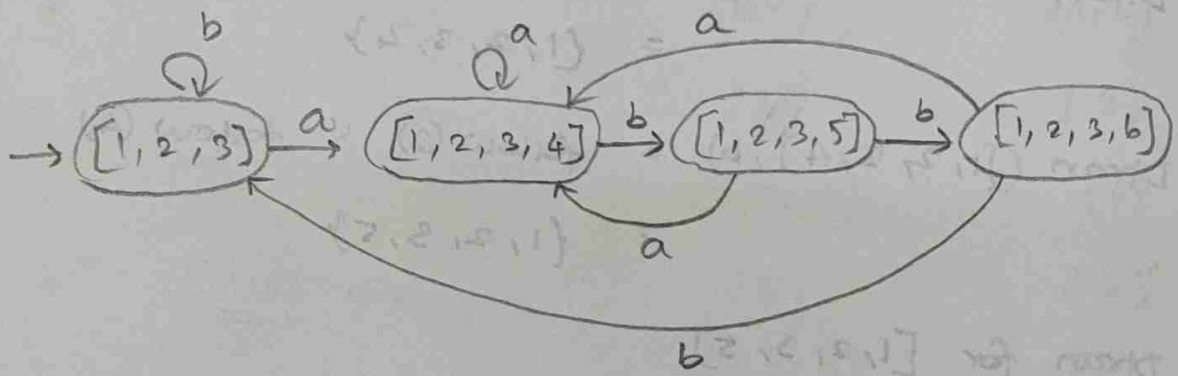
Dtran for [1, 2, 3, 6]

Dtran ([1, 2, 3, 6], a) = {1, 2, 3, 4}

Dtran ([1, 2, 3, 6], b) = {1, 2, 3}

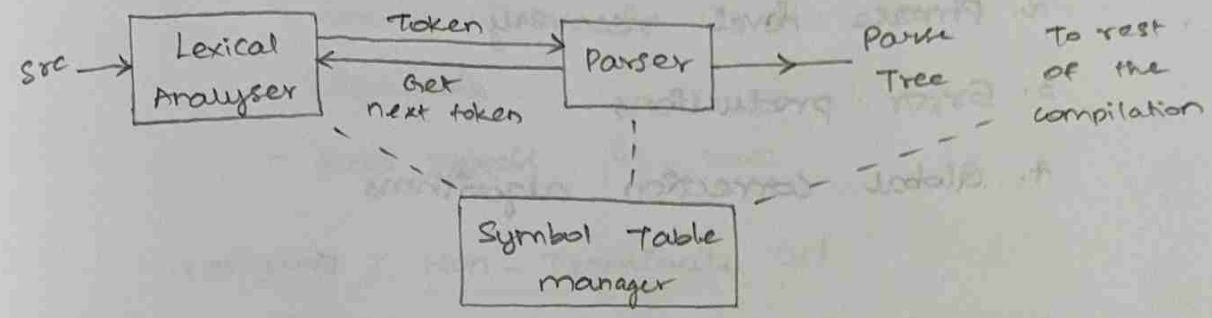
Optimized DFA

State	a	b
[1, 2, 3]	[1, 2, 3, 4]	[1, 2, 3]
[1, 2, 3, 4]	[1, 2, 3, 4]	[1, 2, 3, 5]
[1, 2, 3, 5]	[1, 2, 3, 4]	[1, 2, 3, 6]
[1, 2, 3, 6]	[1, 2, 3, 4]	[1, 2, 3]



2. Syntax Analysis

Role of syntax Analyser / Parser



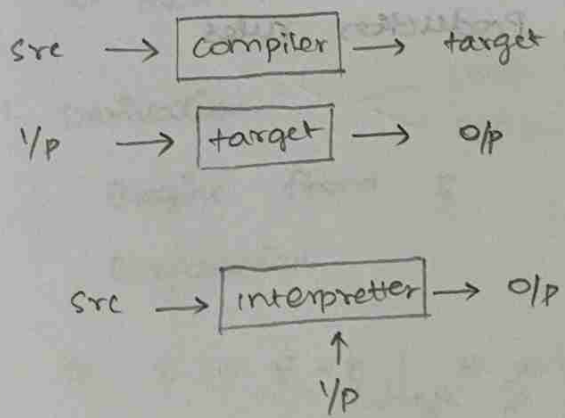
Parser & STM :

Ensure the entry for every symbol

Parser & Error handling :

Levels of Errors

1. Lexical errors
2. Syntatic error
3. Semantic error
4. Logical error



Compiler reads the full src even though there are error present it recovers them to move forward

Interpreter reads the full src but if error found then it recovers and stops. inbetween.

Error Recovery strategies :

1. Panic mode recovery (also followed by Interpreter)
2. Phrase level recovery
3. Error productions
4. Global correction algorithms

Grammar

Used to specify the syntatic rules of the language.

Type 2 / CFG

$G = \{V, T, S, P\}$

T - set of Terminals

V - set of Non Terminals / Variables
(Used to rep. hierarchical structure of the prog. lang.)

S - Start symbol

P - Set of Production rules

Production Rule

Head → Body

- Head → only 1 Symbol → Variable
- Body → any no. of symbols $\{V \cup T\}$

$A \rightarrow \alpha$

CFG - Notations

Terminals (T)

- lower case letters a, b, c, ...
- operators +, -, <, ..
- special char ", (, [, ..
- keywords
- Bold words id, num

Variables / Non-Terminals (V)

- Upper case letters A, B, C
- Upper case late in the alphabet

  X, Y, Z → Grammar symbols (variable, terminal, both)
- Lower case late in the alphabet

  x, y, z → string
- α, β, γ - string of grammar symbols
- Italic words *'expr'*

Generating a string from the grammar -

1. Derivation → LMD / RMD
2. Reduction

1. Derivation
   - LMD (left most derivation)
   - RMD (right " ")

   Begins from S

   Expansion

G: E → E+E | E * E | id

W = id + id * id

V = {E}

T = {id, +, *}

S = E

LMD

E	$E \to E+E$
$E+E$	$E \to id$
$id+E$	$E \to E*E$
$id+E*E$	$E \to id$
$id+id*E$	$E \to id$
$id+id*id$	–

Sentence (only terminals)

Sequential for (both var & terminal)

RMD

E	$E \to E+E$
$E+E$	$E \to E*E$
$E+E*E$	$E \to id$
$E+E*id$	$E \to id$
$E+id*id$	$E \to id$
$id+id*id$	–

Qn) G_1 : $S \to +SS \mid *SS \mid a$

$w = +*aaa$

Qn) G_2 : $S \to S(S)S \mid \epsilon$

$w = (()())$

Qn) G_3 : $S \to S+S \mid SS \mid (S) \mid *S \mid a$

$w = (a+a)*a$

G1 $w = +*aaa$

LMD :

S	$S \to +SS$
$+SS$	$S \to *SS$
$+*SSS$	$S \to a$
$+*aSS$	$S \to a$
$+*aaS$	$S \to a$
$+*aaa$	—

RMD :

S	$S \to +SS$
$+SS$	$S \to a$
$+Sa$	$S \to *SS$
$+*SSa$	$S \to a$
$+*Saa$	$S \to a$
$+*aaa$	—

G2 $w = (()())$

LMD :

S	$S \to S(S)S$
$S(S)S$	$S \to \epsilon$
$(S)S$	$S \to S(S)S$
$(S(S)S)S$	$S \to \epsilon$
$((S)S)S$	$S \to \epsilon$
$(()S)S$	$S \to S(S)S$
$(()S(S)S)S$	$S \to \epsilon$
$(()(S)S)S$	$S \to \epsilon$
$(()()S)S$	$S \to \epsilon$
$(()())S$	$S \to \epsilon$
$(()())$	

RMD :

S	$S \to S(S)S$
$S(S)S$	$S \to \epsilon$
$S(S)$	$S \to S(S)S$
$S(S(S)S)$	$S \to S(S)S$
$S(S(S)S(S)S)$	$S \to \epsilon$
$S(S(S)S(S))$	$S \to \epsilon$
$S(S(S)S())$	$S \to \epsilon$
$S(S(S)())$	$S \to \epsilon$
$S(S()())$	$S \to \epsilon$
$(()())$	$S \to \epsilon$

$S \to \boxed{S}A$

$w = (a+a) * a$

G3:

LMD		RMD	
S	$S \to SS$	S	$S \to SS$
SS	$S \to (S)$	SS	$S \to *S$
$(S)S$	$S \to S+S$	$S*S$	$S \to a$
$(S+S)S$	$S \to a$	$S*a$	$S \to (S)$
$(a+S)S$	$S \to a$	$(S)*a$	$S \to S+S$
$(a+a)S$	$S \to *S$	$(S+S)*a$	$S \to a$
$(a+a)*S$	$S \to a$	$(S+a)*a$	$S \to a$
$(a+a)*a$	-	$(a+a)*a$	-

S
SA
$\to SA$

$S \to Ax$
$A \to Sc$

2. Reduction

Bottom up technique

Handle - a substring that matches with the body of any production rule

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

$w = id + id * id$

Sequential form	Input string	Handle
	id + id * id	
id	+ id * id	'id'
E+	id * id	
E +id	* id	'id'
E+E	* id	'E+E'
E*	id	

E * id		'id'
E * E		'E*E'
(E)		

Qn) Derive the string ((a,a), a, (a)) from the grammar

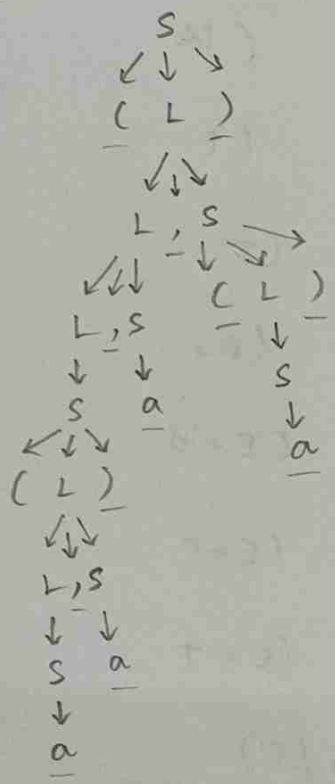
$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

RMD :

S	$S \rightarrow (L)$
(L)	$L \rightarrow L,S$
(L,S)	$S \rightarrow (L)$
(L,(L))	$L \rightarrow S$
(L,(S))	$S \rightarrow a$
(L,(a))	$L \rightarrow L,S$
(L,S,(a))	$S \rightarrow a$
(L, a,(a))	$L \rightarrow S$
(S, a,(a))	$S \rightarrow (L)$
((L), a,(a))	$L \rightarrow L,S$
((L,S), a,(a))	$S \rightarrow a$
((L,a), a,(a))	$L \rightarrow S$
((S,a), a,(a))	$S \rightarrow a$
((a,a), a,(a))	

Parse tree :



Qn) Reduce string $w = (id + id) * id$ from

$E \rightarrow E+T \;/\; T$

$T \rightarrow T*F \;/\; F$

$F \rightarrow (E) \;/\; id$

sequential form	input string	handle
-	(id + id) * id	-
(	id + id) * id	-
(id	+ id) * id	'id'
(F	id) * id	'F'
(T	id) * id	'T'
(E +	id) * id	
(E + id	) * id	'id'
(E + F	) * id	'F'
(E + T	) * id	'E+T'
(E)	* id	'(E)'
F	* id	'F'
T	* id	'T'
T *	id	
T * id	-	'id'
T * F	-	'T*F'
T	-	'T'
E	-	

Left Recursion grammar

(If body begins with same symbol as head)

$A \rightarrow Ax / \alpha$

Elimination of left recursion

$A \rightarrow A\alpha / \beta$ → original grammar

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$ } rewritten

"α, β are string"

Qn) $E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Ans:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

Qn) $A \rightarrow Aa / ad$

$A \rightarrow Abd / c$

Ans:

$A \rightarrow adA' / cA'$

$A' \rightarrow aA' / bdA' / \varepsilon$

$A \rightarrow ad A_1'$

$A_1' \rightarrow aA_1' / \varepsilon$

$A \rightarrow cA_2'$

$A_2' \rightarrow bd A_2' / \varepsilon$

Qn) $S \to (L) / a$

$L \to L.S / S$

Ans:

$L \to S / L'$

$L' \to .SL' / \varepsilon$

Qn) $S \to Sa / Sb / c / d$

Ans:

$S \to cs' / ds'$

$s' \to as' / bs' / \varepsilon$

$S \to S\alpha / \beta$

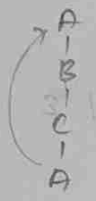
$S \to \beta s'$

$s' \to \alpha s'$

Qn) $A \to Ba$

$B \to Cb$

$C \to Ac / b$



(Indirect left recursion)

Ans:

$V = \{A, B, C\}$

$A_1 \; A_2 \; A_3$

$A_i \to A_j$, $i<j$
else sub variables

$A_1 \to A_2 a$

$A_2 \to A_3 b$

$A_3 \to A_1 c / b \quad (i \neq j)$ $A_3 \to A_2 ac / b \quad (i \neq j)$

$A_3 \to A_3 bac / b \quad (i \neq j)$ (direct left recursion)

$A \to A \alpha / \beta$

$A_3 \to bA'$

$A' \to bacA' / \varepsilon$

$C \to bc'$

$C' \to bacc' / \varepsilon$

$A \to Ba$

$B \to Cb$

$C \to bC'$

$C' \to bacC' / \varepsilon$

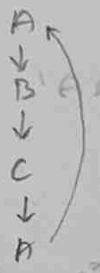
Qn)

$A \to Bab \mid c$

$B \to CD$

$C \to A \mid c$

$D \to b$



Indirect recursion

Ans:

$\{A \quad B \quad C \quad D\}$

$\{A_1 \quad A_2 \quad A_3 \quad A_4\}$

$(A_i \to A_j : i < j)$

✓ $A_1 \to A_2 ab \mid c$

✓ $A_2 \to A_3 A_4$

$A_3 \to A_1 \mid c$ } $i \neq j$

✓ $A_4 \to b$

$A_3 \to A_2 ab \mid c$ } $i \neq j$

$A_3 \to A_3 A_4 ab \mid c$

α

✓ $A_3 \to cA'$

✓ $A' \to A_4 ab A' \mid \varepsilon$

$A \to Bab \mid c$

$B \to CD$

~~$C \to A \mid c$~~

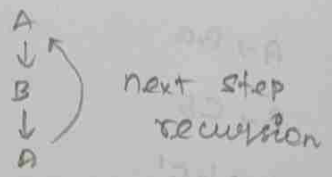
$D \to b$

$C \to cC'$

$C' \to Dab C' \mid \varepsilon$

Qn) $A \to Ba \mid Aa \mid c$

$B \to Bb \mid Ab \mid \alpha$



Ans:

Eliminate direct recursion in A:

$A \to cA' \mid BaA'$

$A' \to aA' \mid \varepsilon$

$A \to A\alpha \mid \beta$ ⇓ $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$; $\beta = c, Ba$; $\alpha = a$

Substitute A in B-production:

$B \to Bb \mid cA'b \mid BaA'b \mid d$

Eliminate direct recursion in B:

$B \to cA'bB' \mid BaA'bB' \mid dB'$

$B' \to bB' \mid \varepsilon$

Final Ans:

$A \to cA' \mid BaA'$

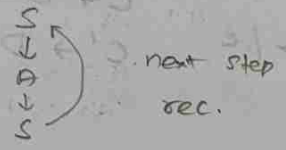
$A' \to aA' \mid \varepsilon$

$B \to cA'bB' \mid BaA'bB' \mid dB'$

$B' \to bB' \mid \varepsilon$

Qn) $S \to Aa \mid a$

$A \to Sb \mid b$



Ans:

$S \to \underset{A}{Sba} \mid \underset{\beta}{ba} \mid \underset{\beta}{a}$ ⇒ $S \to basS' \mid asS'$?

$A \to Sb \mid b$ $S' \to basS' \mid \varepsilon$

$A \to Aab / ab / b$

$A \to abA' / bA'$

$A' \to abA' / \varepsilon$

$S \to Aa / a$

$A \to abA' / bA'$

$A' \to abA' / \varepsilon$

Qn) $A \to Cd$

$B \to Ce$

$C \to A / B / a$

Ans:

$C \to Cd / B / a$

$B \to Ce$

$C \to Cd / Ce / a$

$C \to aC'$

$C' \to dC' / eC' / \varepsilon$

An) $X \to Xsb / Sa / b$

$S \to Sb / Xa / a$

Left factored Grammar

$A \to \alpha\beta_1 / \alpha\beta_2 / \cdots \alpha\beta_n / \gamma$

body has common prefix in a lot of rules (α)

Elimination of LFG

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 / \cdots \beta_n$

$S \to iEtS / iEtSeS / a$ } Syntax of if / if..else in pascal

$E \to b$

$\boxed{iEtS}$ → common prefix

$S \to iEtSS' / a$

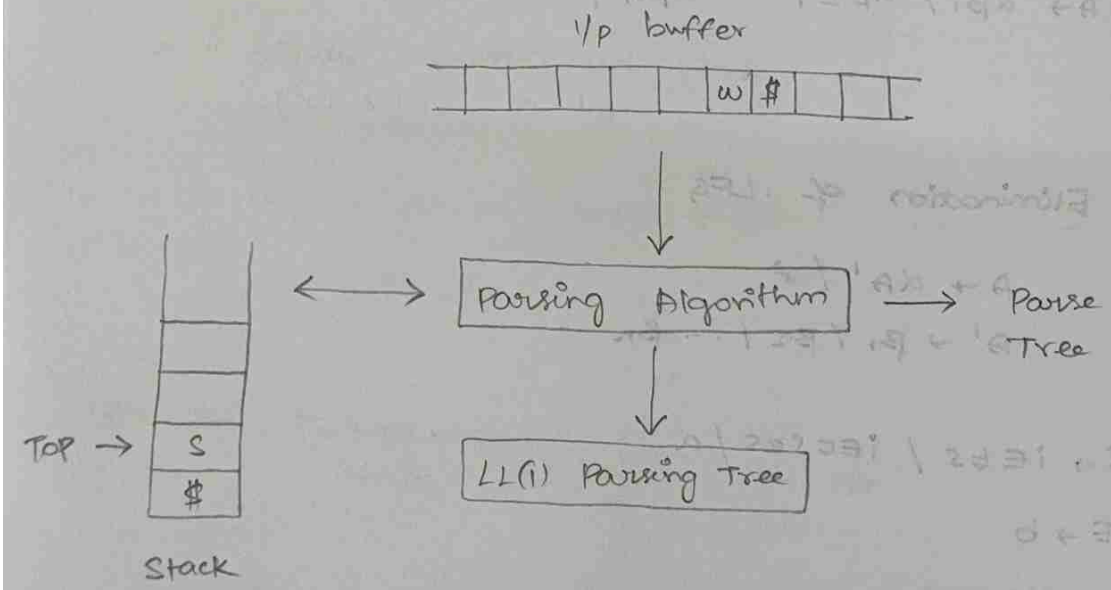
$S' \to eS / \varepsilon$

$E \to b$

Parsing

- Top - Down parsing
  - Parse Tree from Root to Leaf
  - Derivation
  - LL(1) Parser / Non-Recursive Predictive Parser
- Bottom - up parsing
  - Parse Tree from Leaves to the Root
  - Reduction
  - LR Parser
    - SLR (Simple LR) Parser
    - CLR (Canonical LR) Parser
    - LALR (Look Ahead LR) Parser

Predictive Parser



Steps

1. Preprocessing

   Elimination of left recursion

   Elimination of left factor

   Elimination of Ambiguity

2. Computation of FIRST

3. Computation of FOLLOW

4. Construction of parsing tree

5. Parse the i/p string

Computation of FIRST

Algorithm:

R1 : If x is terminal then FIRST $(x) = \{x\}$

R2 : If x is non-terminal, $x \to \epsilon$, then add ϵ into FIRST (x)

R3 : If $x \to \underbrace{y_1, y_2 \cdots y_k}_{\alpha}$ is a production in G

FIRST (x) = FIRST (α) = FIRST (y_1)

If FIRST (y_1) contains ϵ, add FIRST (y_2)

If FIRST (y_2) contains ϵ, add FIRST (y_3)

Add ϵ into FIRST (x) when FIRST $(y_1, y_2 \cdots y_k)$ contains ϵ

Qn) Construct LL(1) parsing table for grammar

$S \rightarrow (L) \mid a$ not left rec.

$L \rightarrow L+S \mid S$ not left factor

not ambiguity grammar

Ans:

Elimination of left recursion

$L \rightarrow L+S \mid S$

$L \rightarrow SL'$

$L' \rightarrow +SL' \mid \varepsilon$

G': $S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow +SL' \mid \varepsilon$

$V = \{S, L, L'\}$

$T = \{a, +, (,), \epsilon\}$

Computation of FIRST

FIRST (S)

$S \rightarrow (L)$, FIRST (S) = FIRST [(L)] = FIRST ((L) = {(}
(y_1, y_2, y_3)

$S \rightarrow a$ FIRST (S) = FIRST (a) = {a}

FIRST (S) = {(, a}

FIRST (L)

$L \rightarrow SL'$ FIRST (L) = FIRST (SL') = FIRST (S) = {(, a}

FIRST (L')

$L' \to +SL'$, FIRST (L') = FIRST (+SL') = FIRST (+) = {+}

$L' \to \epsilon$, FIRST (L') = {ε}

FIRST (L') = {+, ε}

Computation of FOLLOW

Algorithm:

R_1: Add $ (end marker) into FOLLOW (S), where S is the start symbol of G

R2: If any prod. of the form,

$A \to \alpha B \beta$

FOLLOW (B) = FIRST (β) except ε

R3: If a prod. of the form,

$A \to \alpha B$ (or) $A \to \alpha B \beta$ and FIRST (B) includes ε

FOLLOW (B) = FOLLOW (A)

Qn) continue of

$S \to (L) \mid a$

$L \to L+S \mid S$

V = {S, L, L'}

FOLLOW (S)

FOLLOW (S) = {$} ① by R_1

$L \to \underset{\alpha}{} \underset{B}{S} \underset{\beta}{L'}$

FOLLOW (S) = FIRST (L') except ∈

= {+}

FOLLOW(S) = FOLLOW (L) → ①

$L' \rightarrow \underset{\alpha}{+} \underset{B}{S} \underset{\beta}{L'}$

by R2.

FOLLOW(S) = FIRST (L') = {+}

FOLLOW(S) = FOLLOW (L') → ②

FOLLOW (L)

$S \rightarrow \underset{\alpha}{(} \underset{B}{L} \underset{\beta}{)}$

FOLLOW (L) = FIRST ()) = {)}

Sub FOLLOW (L) in ①,

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

FOLLOW (L')

$L \rightarrow \underset{\alpha}{S} \underset{B}{L'}$, FOLLOW(L') = FOLLOW(L) = {)}

$L' \rightarrow \underset{\alpha}{+} \underset{B}{S} L'$, FOLLOW (L') = FOLLOW (L') = {)}

FOLLOW (S) = {$, +,)}
FOLLOW (L) = {)}
FOLLOW (L') = {)}