- Aho, Lam, Sethi, Ullman

22.1.25

written in 'C'

Compiler Design

Ch. 1 → Ch-3

(Lexical Analysis)

↳ divides input stream of characters into meaningful chunks called 'lexemes' & issues tokens for valid lexemes

↓

Implemented through NFA

var var var

e.g. y = a b +

relop arith

Token : <name, value>

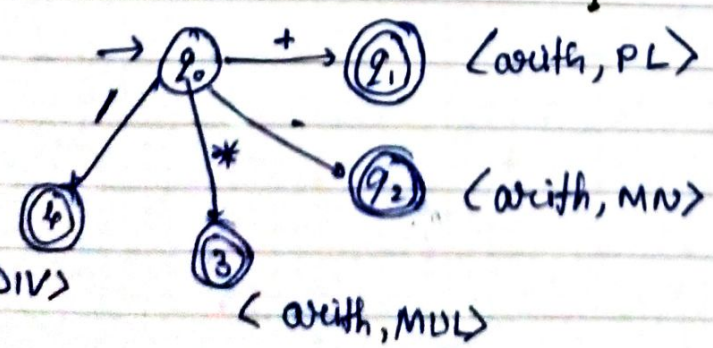
⇒ Implement NFA for following

	CATEGORY	VALID LEXEMES	TOKEN ISSUED
①	Keywords		
②	Variables		
③	Numbers		
④	Arithmetic	+, −, *, /	e.g. <arith, PL> (Plus)
⑤	Relational Operators		

Therefore, each lexeme belongs to a ~~color~~ category which is its name.

token issued

e.g.



NFA for arithmetic category.

→ Design NFA

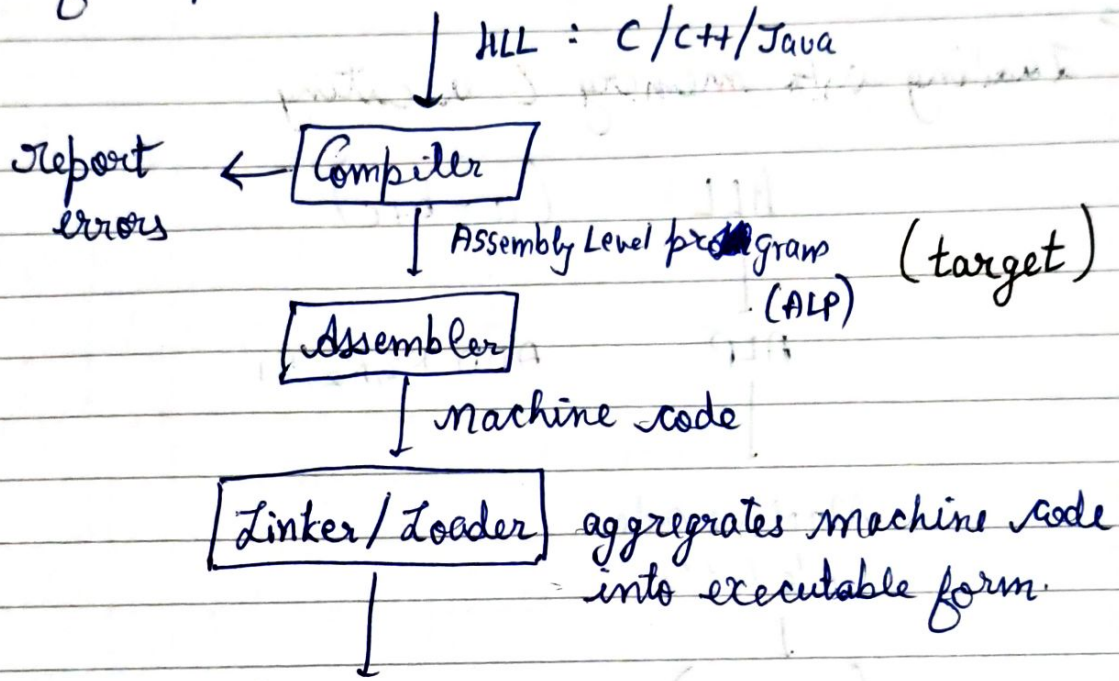
1. user enters string
2. Call function NFA
3. Return back a signal to main

↳ In case of multiple NFAs, the order in which they are call
is important

Introduction

→ Compiler : a program written in C that translates a high level program to assembly level program
(i) translates a high level program to assembly level program
(ii) reports any errors

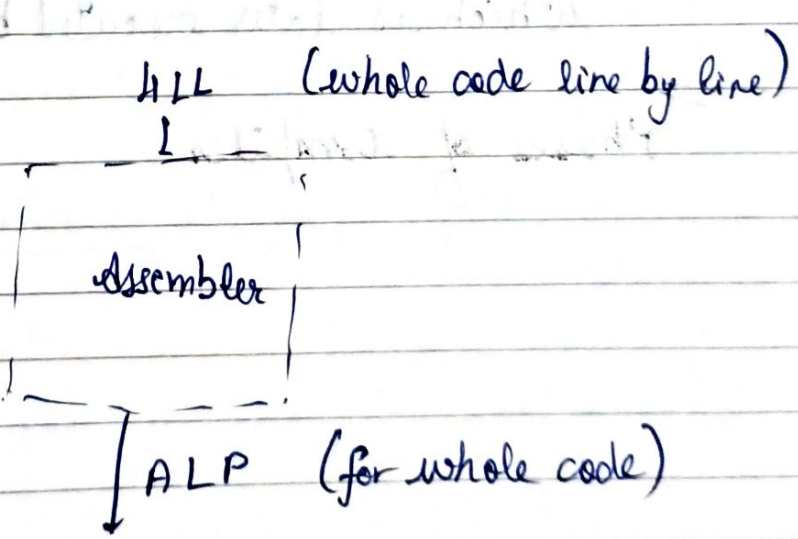
→ Process of compiler :-



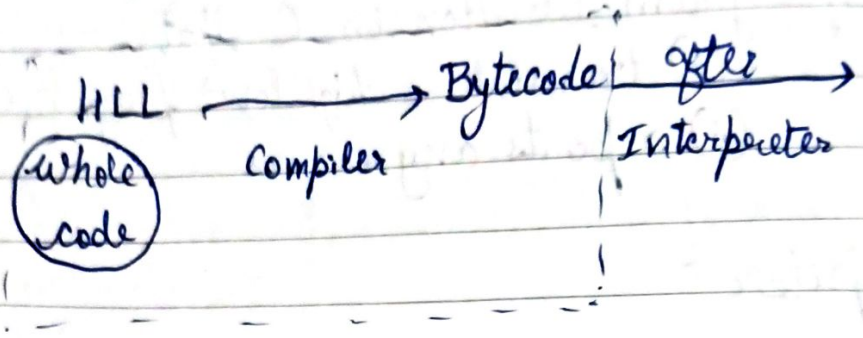
⇒ ~~Java~~ Python :-

Interpreter

line-by-line execution



⇒ Java : Hybrid of compiler and ~~the~~ interpreter



= Loading into memory & executing

HLL (a = b+c)

⋮

ALP ADD R_3, R_2, R_1

Machine code
0's & 1's
↓
(memory)

↳ Data as well as program are stored in memory, which is later executed by processor.

= Phases of compiler

Target : LD R_1, id2
LD R_2, id3
ADD R_3, R_1, R_2
ST id1, R_3

a = b + c

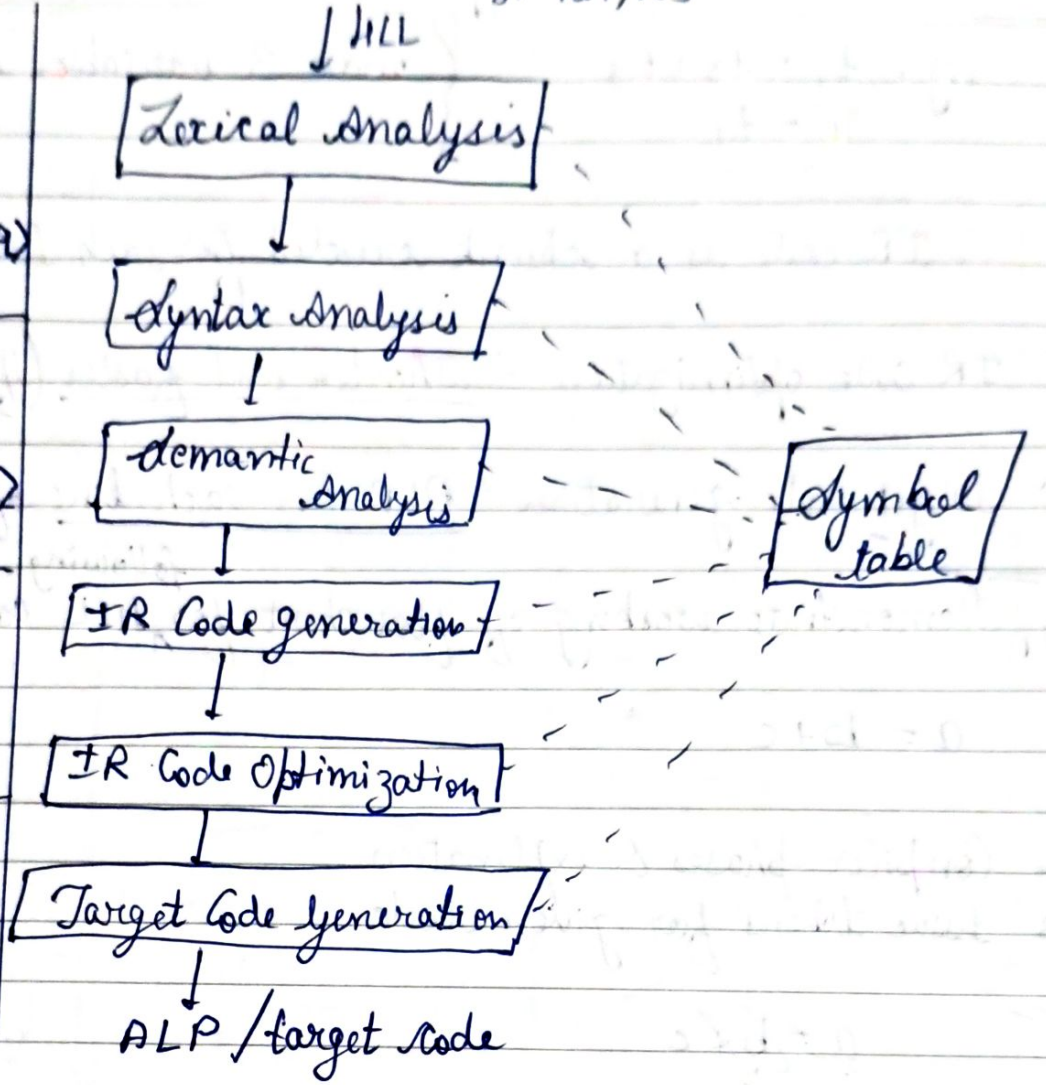
tokens : <id, 1>
<assign op, EQ>
<id, 2>, <arith, PL>
<id, 3>

tree :


tree : -- ditto --

$t_1 = id2$
$t_2 = id3$
$t_3 = t_1 + t_2$
$id = t_3$

$t_1 = id_2$
$t_2 = id_3$
$id_1 = t_1 + t_2$



① Lexical analysis : divides the input stream of characters into meaningful chunks (lexemes). For each valid lexeme, issue tokens : <name, value>

② Syntax analysis : checks the arrangement of tokens

③ Semantic analysis : type checking → try type conversion if possible
→ or, declare type mismatch error

e.g. 'int' → 'float'

④ IR code generation
↳ Intermediate representation = abstract code = 3 address code

④ Symbol table : stores info on variables

value, type, scope, size

e.g. $t_1 = t_2 + t_3$
$a = t_1$

} max 3 variables in a statement

∴ IR code is a chunk created for each line of HLL.

⑤ IR code optimization : shorter and faster (if possible)

⑥ Target code generation : ALP for each line of IR code

---

*Q1 Demonstrate working of flowchart for following HLL code

$a = b + c$

Q2 Compiler phases & explanation

Q3 Issue tokens for given code.

---

a = b + c

↓

tokens

29.1.25

implemented by nfa

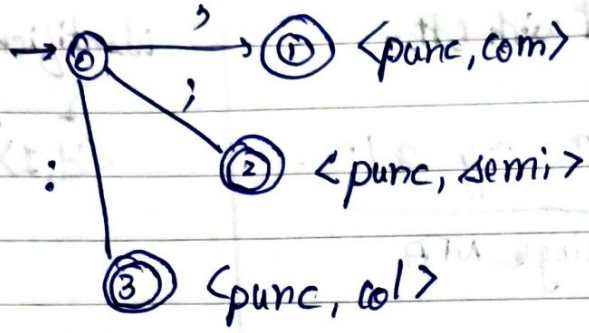
Lexical Analysis → meaningful chunk

Lexical categories	Valid lexemes	Tokens issued
1. Keywords	if, while etc. (NFA1, NFA2)	<if>, <while> (only kw)
2. Variables	x xy x12 .... (must start with letter; single NFA)	<id,1>, <id,2>, ... (identifier; order in which it comes)
3. Numbers	1 1000 ...	<num,1>, <num,1000>
4. Relational	>, >=, =, <, <=, <>	<relop,GT>, <relop,GE>, <relop,EQ>, <relop,LT>, <relop,LE>, <relop,NE>
5. Arithmetic	+, -, *, /	<arith,PL>, <arith,MN>, <arith,MUL>, <arith,DIV>
6. Punctuation	, ; : #	<punc,com>, <punc,semi>, <punc,col>
7. Parenthesis	( ) { } [ ]	<par,S-op>, <par,S-cl>, <par,C-op>, <par,C-cl>, <par,b-op>, <par,b-cl>
8. White space	blank, newline, tab	<> (empty token)

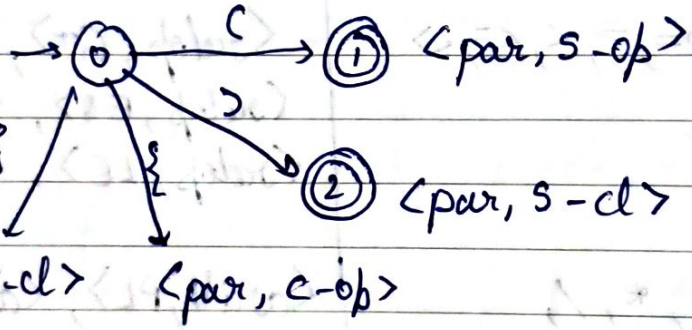
ple because they are single character

→ Task is to issue tokens
→ nfa issues tokens.
→ our task is to design nfas which issue tokens

e.g. punctuation nfa :-



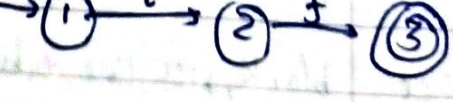
e.g. parenthesis



=

NFA for keyword & variables

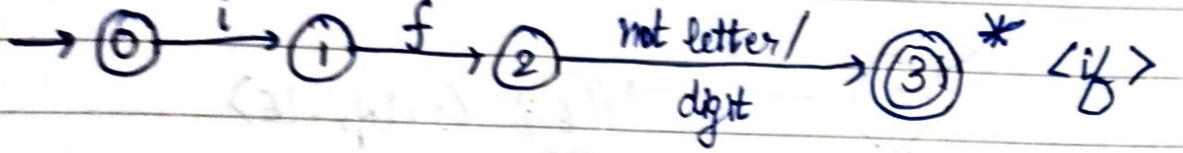
e.g. Keyword "if" Variable iff, if123

⇒ if we design nfa as  then even 'iff' will emit <if>

⇒ Therefore, we design nfa for each category, and when nfa accepts then break out.

∴ order of keyword → variable → number → relop is important.

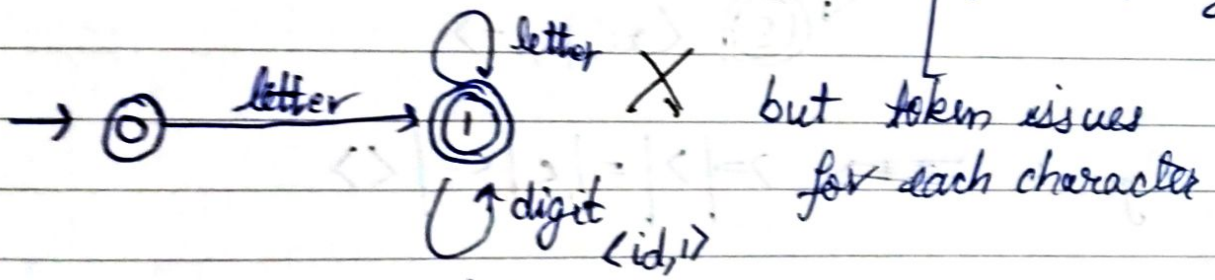
⇒ but if source is "iff" we dont want keyword nfa to accept it



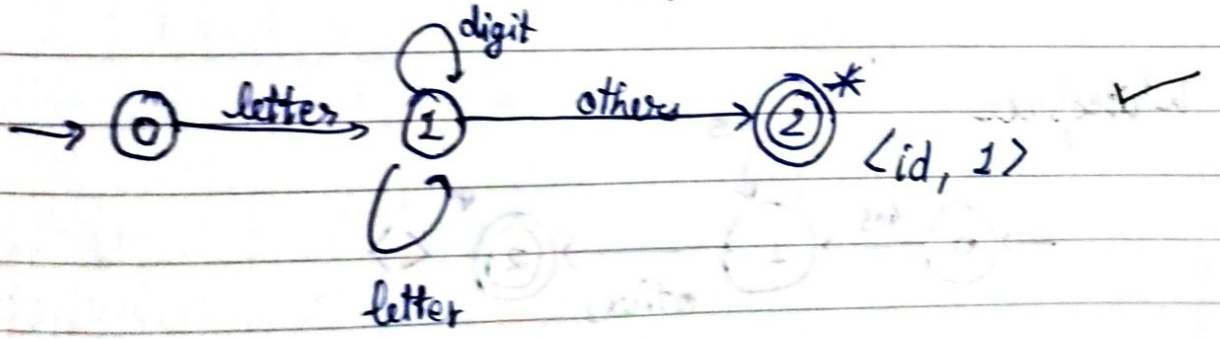
* → retract or go back

→ Variable :-

letter = a|...|z|A|...|Z



but token issues for each character



=

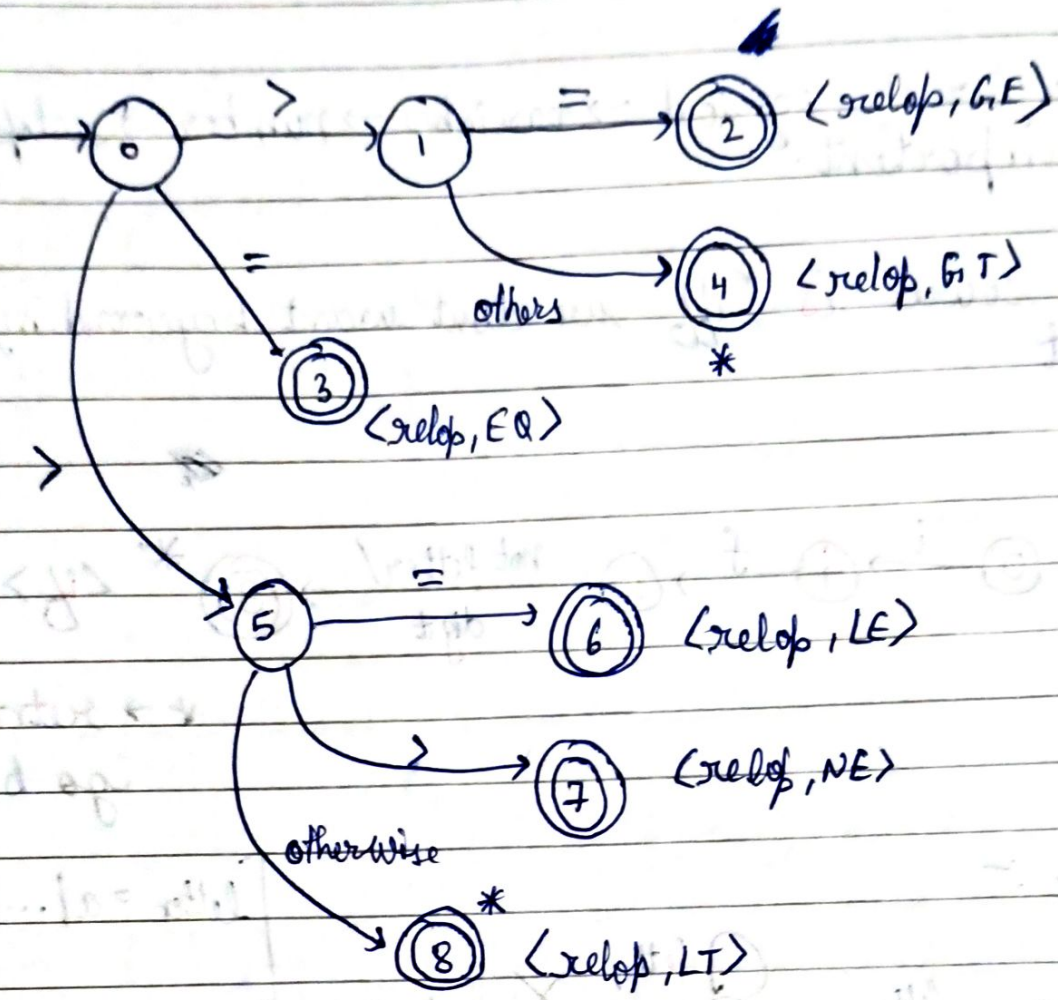
Regular expression : pattern we expect

. → concat
+ → or
* → lots of

<if> : if

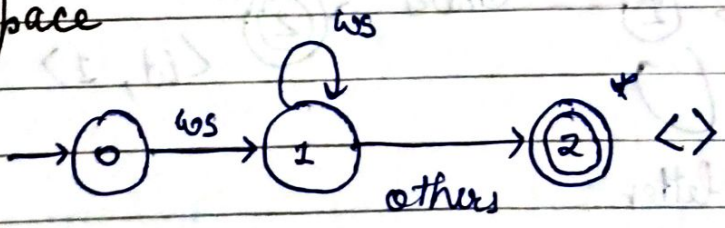
<id, 1> : letter . (letter + digit)*

→ Relational Operators :

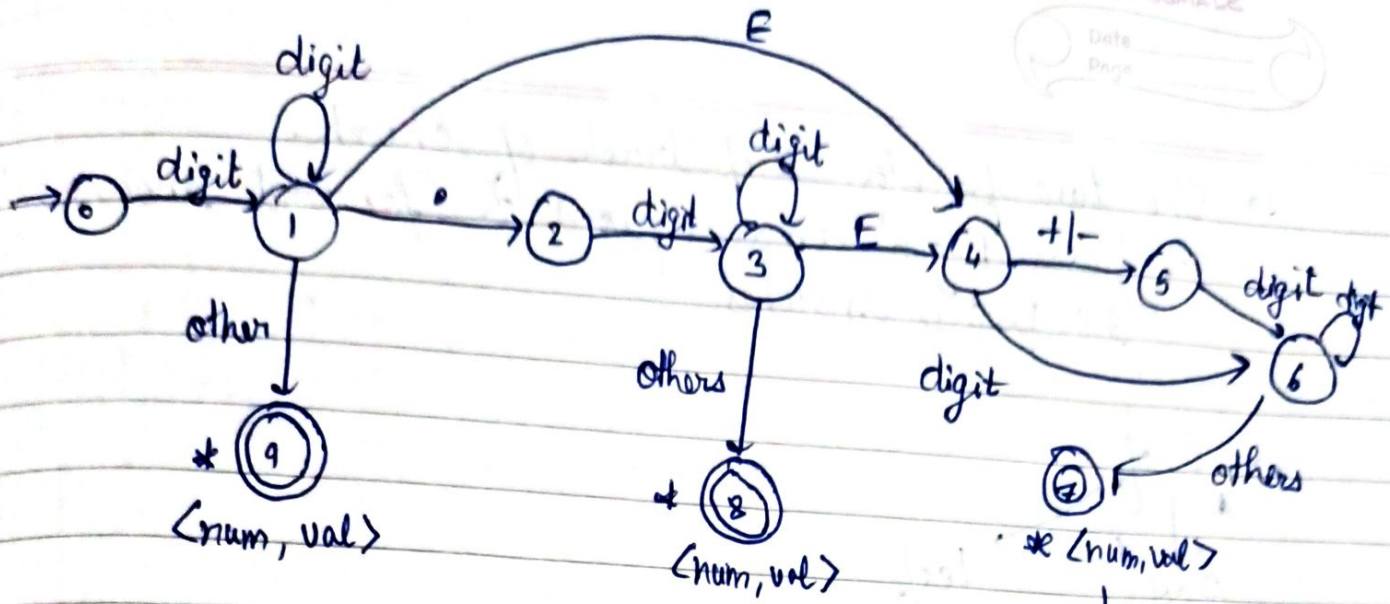


Regex : ⟹ >=|>|=|<|<=|<>

→ Whitespace



→ Numbers



where digit = 0|1|....|9

3
300
300.3
3E+4
3E4
3.3E4

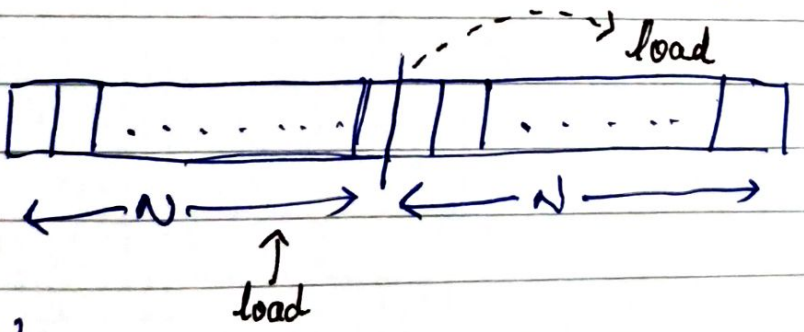
y = -3.3

⟨id, 1⟩ ⟨relop, EQ⟩ ⟨arith, MN⟩, ⟨num, 3.3⟩

Input Buffering

↳ 2 buffers

↳ size N = 4096 bytes i.e. 4096 characters



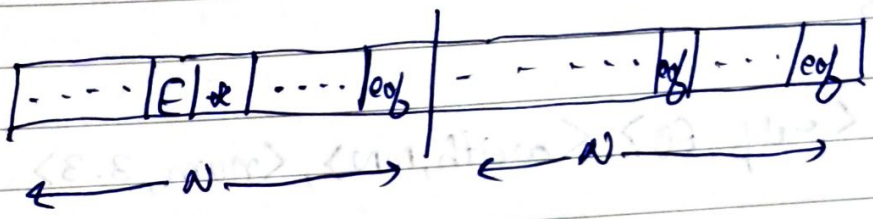
① size N
② alternative loading
③ 2 pointers : lexemeBegin , forward
(Fix) (Moves)
④ Eof

→ The two pointers keep track of chunk.
∴ 'lexemeBegin' remains fixed & 'forward' moves ahead till pattern matches

→ 3 eof

2 eof ⇓ sentinels

1 eof ⇓ actual end of file

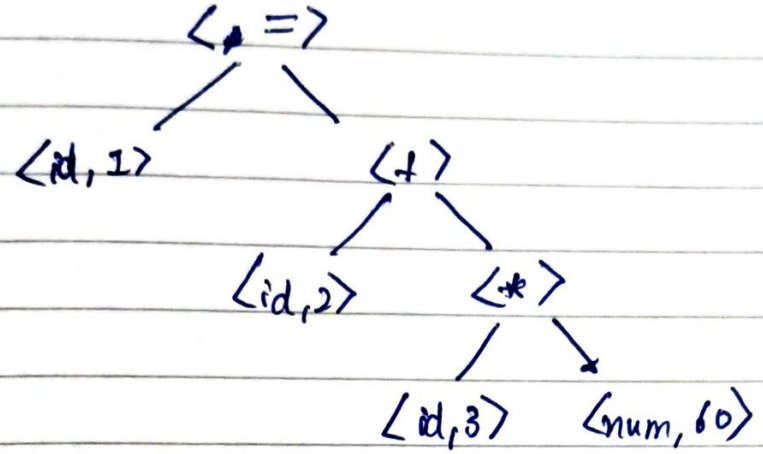


→ When actual eof encountered, the procedure ends.
→ When sentinel is reached, next buffer is loaded and forward continues its journey till pattern matches.

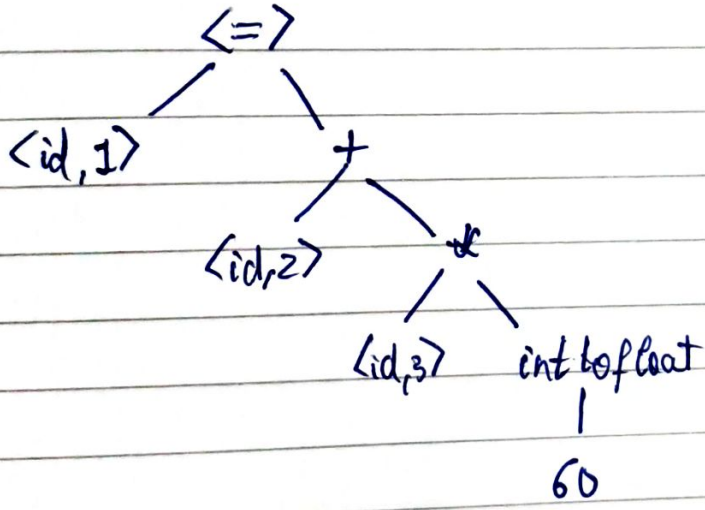
Q pos = in + rate * 60

① <id, 1>, <relop, =>, <id, 2>, <arith, PL>, <id, 3>, <arith, MUL>, <num, 60>

②



③



Bottom ↑ up

④ t_1 = int to float (60)
t_2 = id 3 * t1
t_3 = id 2 + t2
id 1 = t_3

⑤ t_2 = id 3 * int to float (60)
id 1 = id 2 + t_2

⑥ LDF R_1, id3

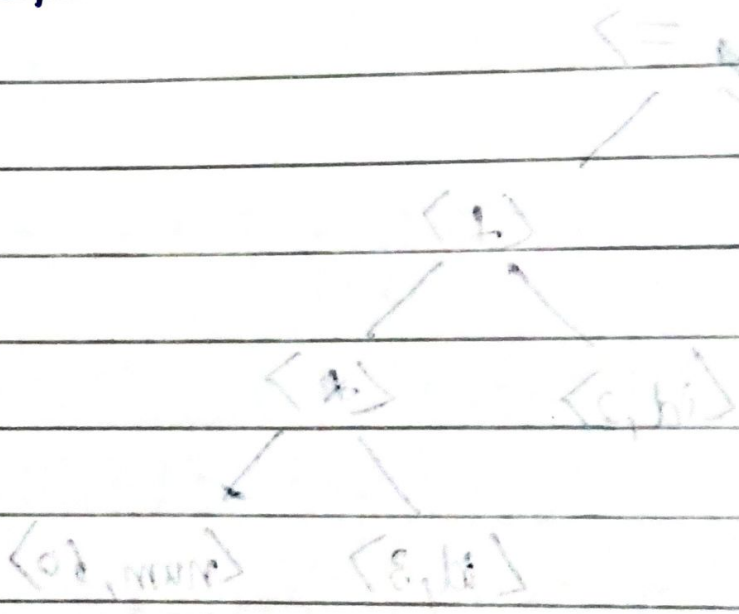
MULF R_2, R_1, #60.0

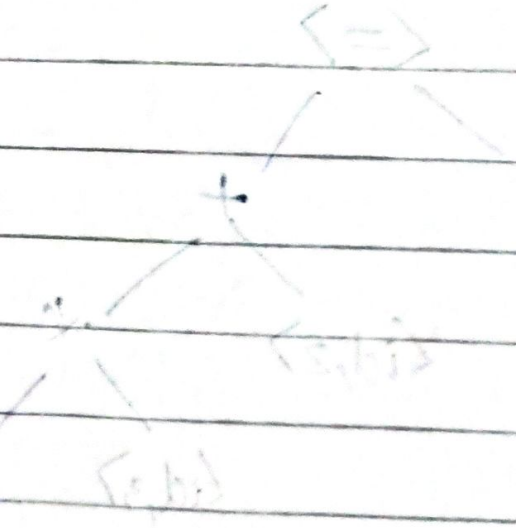
~~ADDF~~

LDF R_3, id2

ADDF R_4, R_3, R_2

STF id1, R_4

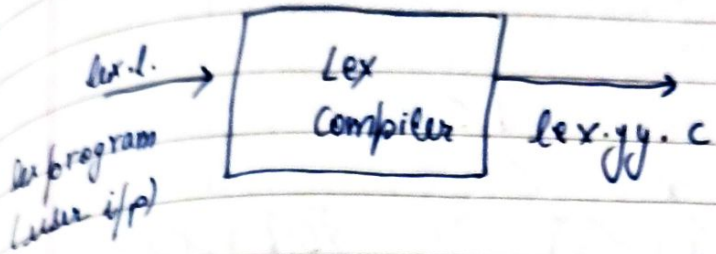


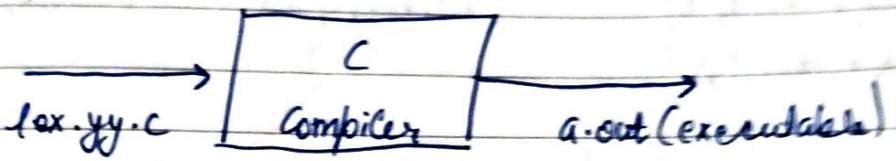


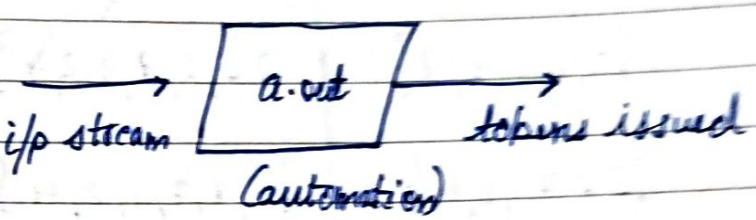
* 1. Lexical categories - NFAs
2. I/p buffering
* 3. Thompson algorithm & subset construction algo
4. Lex

6.2.25

Lex : Automatic lexical analyser generator







Thompson algo & subset construction

r.e. → nfa nfa → dfa

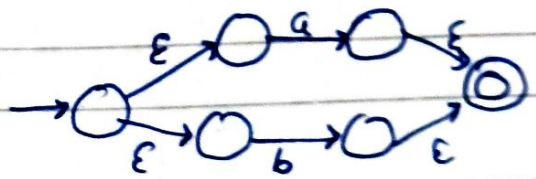
Q (i) Construct nfa from r.e. using Thompson
(ii) convert that nfa to dfa using subset construction

r.e. : $(a/b)^*$

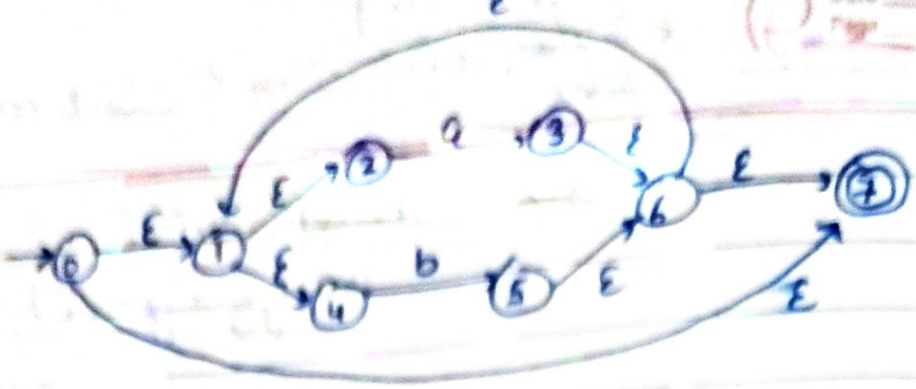
a :

b :

a/b :



$(a|b)^*$:



P.S.

N.S.

a

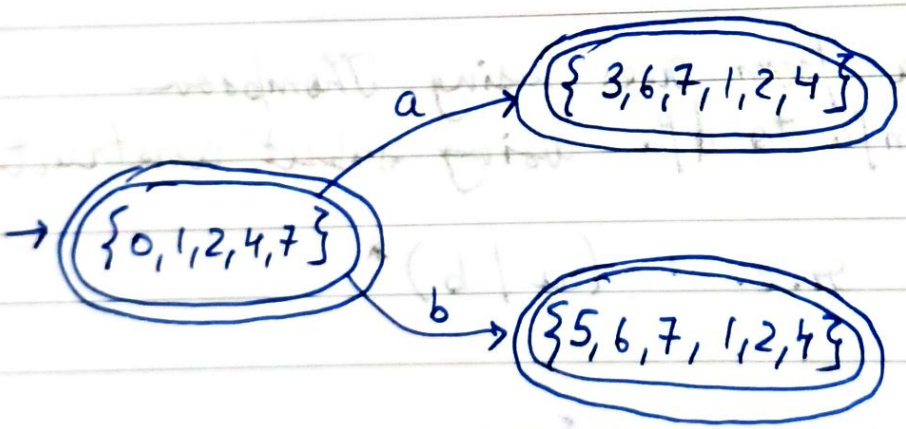
b

{0, 1, 2, 4, 7}

{4, 6, 7, 1, 2, 3}

{5, 6, 7, 1, 2, 3}

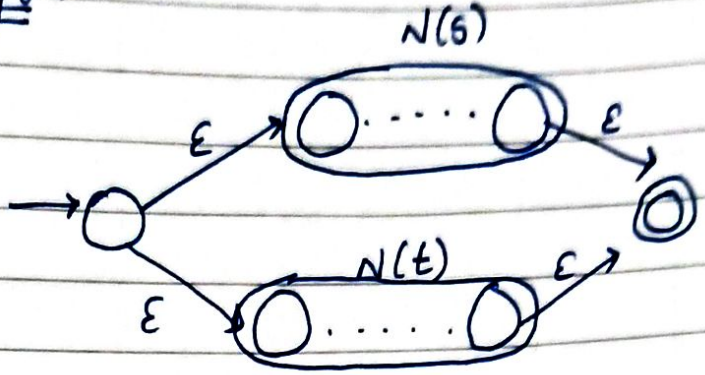
P.S.	N.S.	
	a	b
{0, 1, 2, 4, 7}	{3, 6, 7, 1, 2, 4}	{5, 6, 7, 1, 2, 4}
{3, 6, 7, 1, 2, 4}	{3, 6, 7, 1, 2, 4}	{5, 6, 7, 1, 2, 4}
{5, 6, 7, 1, 2, 4}	{3, 6, 7, 1, 2, 4}	{5, 6, 7, 1, 2, 4}



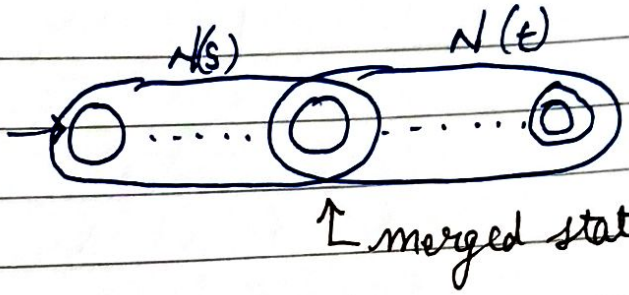
★ Must draw rules before answering

Rules for Thompson algo :-

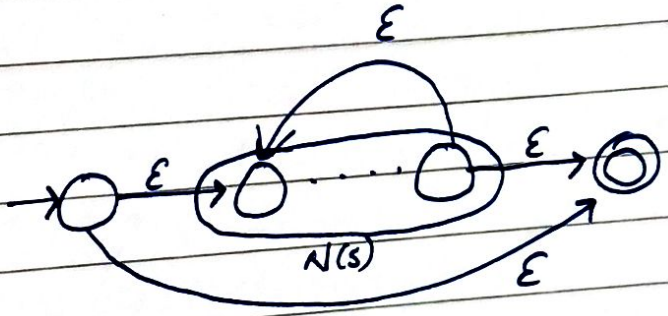
(1) $s|t$:



(2) $s.t$:



(3) s^*



→ Transition table of ~~DFA~~ NFA : (i) no blanks
(ii) one entry per cell

★ LL(1) parser

classmate
Date 13.1.25
Page

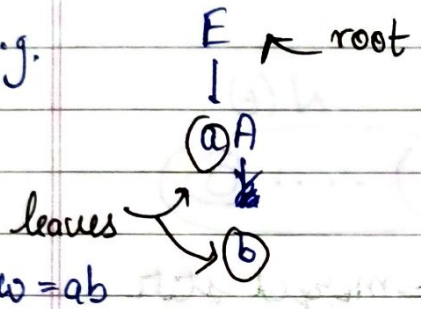
Syntax Analysis (arrangement of tokens)

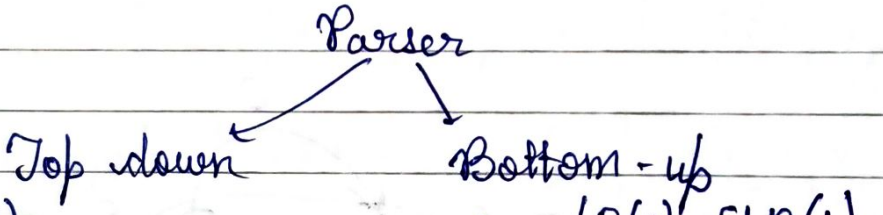
→ Parser : program which says yes/no for a stream.

Q Design a parser given grammar G & string ω

↳ if parser gives green flag, we can make tree.

e.g.





e.g. LL(1)

e.g. LR(0), SLR(1)

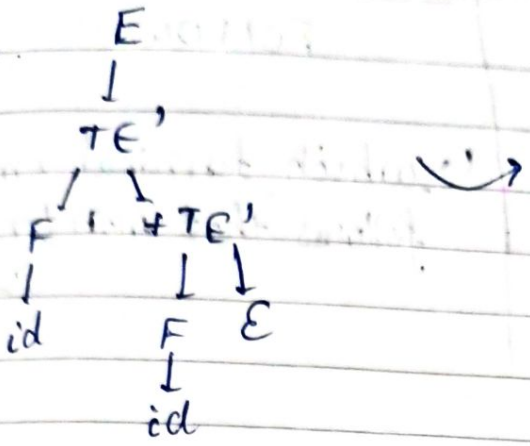
→ Designing LL(1) parser :-
↳ given G & ω

e.g. G : $E \to TE'$ (6 productions)
$E' \to +TE' \mid \varepsilon$
$T \to F$
$F \to (E) \mid id$

∴ terminals : $\{id, (,), +\}$
non terminal : $\{E, T, E', F\}$
start : E

↳ info about identifier is stored in symbol table

e.g.



w = id + id $

↑ EOF (mandatory)

Step - 1 : parsing table (from grammar)
Step - 2 : table of sequence of moves (from table & i/p string)

both are sets

FIRST	FOLLOW
↳ first terminal you see from L to R (ε allowed)	↳ which terminal immediately follows that non terminal. (ε not allowed)

① Let $X \rightarrow Y_1, Y_2 \ldots Y_K$

↳ then, FIRST(X) = Y_1, if Y_1 is terminal
= FIRST(Y_1) otherwise

↳ if FIRST(Y_1) = ε, then, FIRST(X) = FIRST(Y_2)
.....

if all are ε, then FIRST(X) = {ε}

② FOLLOW has 3 rules

(i) FOLLOW(E) = {$, ...}

(ii) for production $X \rightarrow \alpha B \beta$, then
FOLLOW(B) = FIRST(β)

(iii) if FIRST(β) = {ε}, ($\equiv X \rightarrow \alpha B$)
FOLLOW(B) will inherit from FOLLOW(X)

① LL(1) Parsing table :-

	id	(	)	+	$
E	$E \to TE'$	$E \to TE'$	–	–	–
T	$T \to F$	$T \to F$	–	–	–
E'	–	–	$E' \to \varepsilon$	$E' \to +TE'$	$E' \to \varepsilon$
F	$F \to id$	$F \to (E)$	–	–	–

process for designing LL(1):

for each production $A \to \alpha$

1. enter $A \to \alpha$ in row (A) & column : FIRST (α)
2. if FIRST (α) = ε, then enter $A \to \alpha$ in FOLLOW (A) column & row (A)

e.g. $E \to TE'$ ∴ row = E, column = FIRST(TE') = FIRST(T)
= FIRST(F) = {(, id}

e.g. $E' \to \varepsilon$, FOLLOW(E') = FOLLOW(E)
= {$,)}

→ Sequence of moves (ω = id + id $)

Stack	Input	Comments
E$	id + id $	(E, id) call
↑	↑	
TE'$	id + id $	(T, id) call
↑	↑	
FE'$	id + id $	(F, id) call
idE'$	id + id $	MATCH!
E'$	+ id $	(E', +) call
+TE'$	+ id $	~~(E', +) call~~ MATCH!
TE'$	id $	(T, id) call
FE'$	id $	(F, id) call
idE'$	id $	MATCH!
E'$	$	(E', $) call
$	$	YES

Note: in the above procedure, you will get syntax error if

(i) you dont get | $ | $ | ~~Acc~~ YES |

(ii) You reach empty cell

(iii) Mismatch of terminals.