অনেকগুলো object file থাকলে linker সবগুলা executable file create করে।

- .c
- .exe → executable file
- .o → like an interpreter (object file)

* Define grammar
* Build the front-end compiler for source code
* Build the back end code generator

- compiler → language processor

*whole program at a time (faster)

source code → [compiler] → target program / intermediate / middle language

- compiler can detect errors. (during translation process)
- i/p → [target program] → o/p
  (.exe file)

*step 2

- source program → [interpreter] → o/p
  input →

*statement by statement পড়ে;

hello.py → source code

hello.pyc → interpreted to bytecode (executable file)

*error দিলে চলে যায়

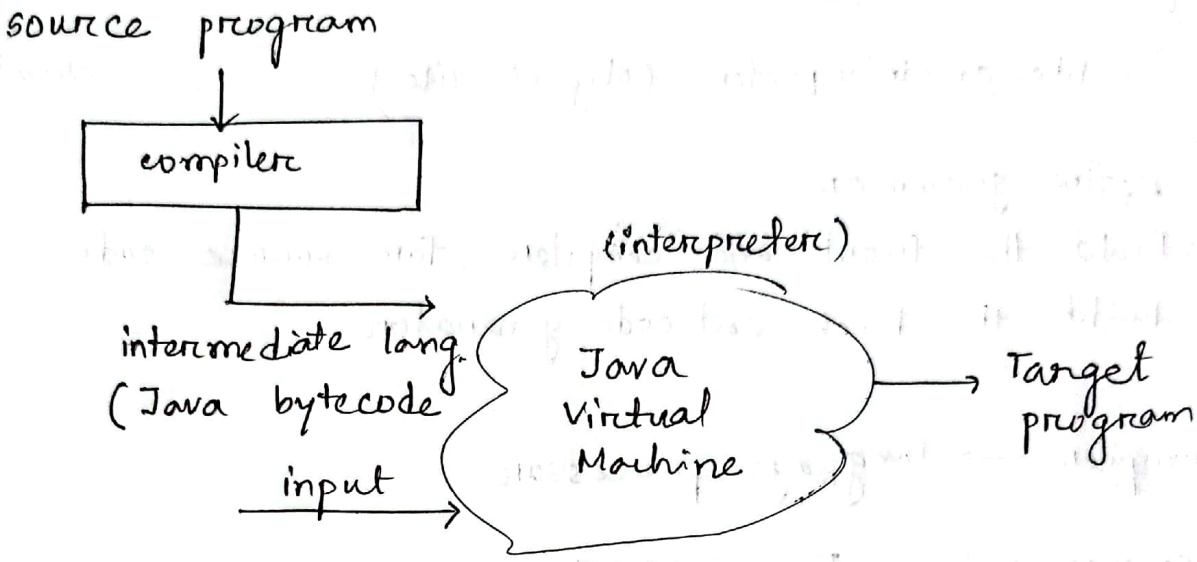
- python doesn't use compilers

*step 1

Imp : compiler vs Interpreter

25.4.24

Imp : Java → hybrid compiler → Justify it.



Hybrid compilation

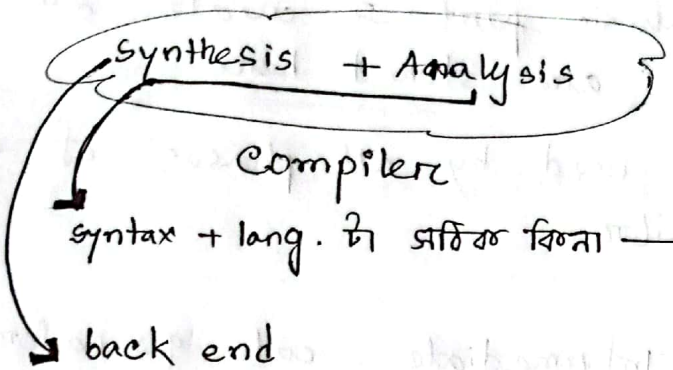
* main benefit of Java hybrid compiler
* assembler → machine dependent
  intel → 8086 ...

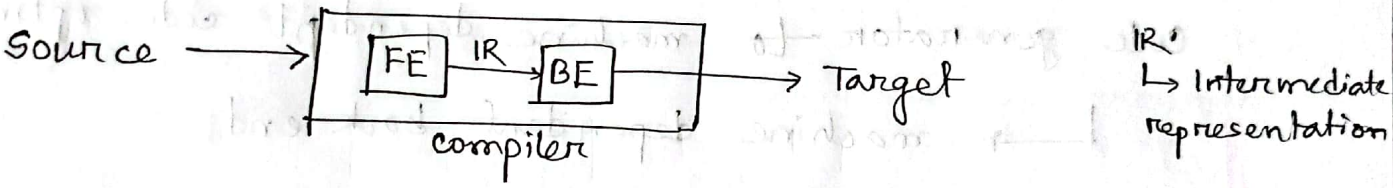
& object files

Linker → all library files কে link করে executable file create করবে।

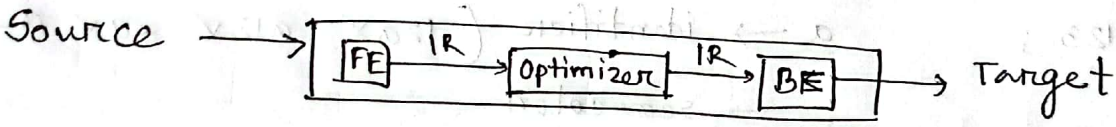
Loader → executable file কে memory তে create করে

Linker & loader এর functions → definition





↳ Two ~~One~~ pass compiler



↳ Single Pass compiler

* symbol table ⟶ analysis part ↴ create হয় & data are stored here

⟶ it's used by all phases of a compiler

* Lexical Analyzer to Intermediate code generator

⟶ Machine independent front end (analysis)

code generator to machine dependent code optimer

⟶ machine dependent back end

* int a ;
a = 123 ;

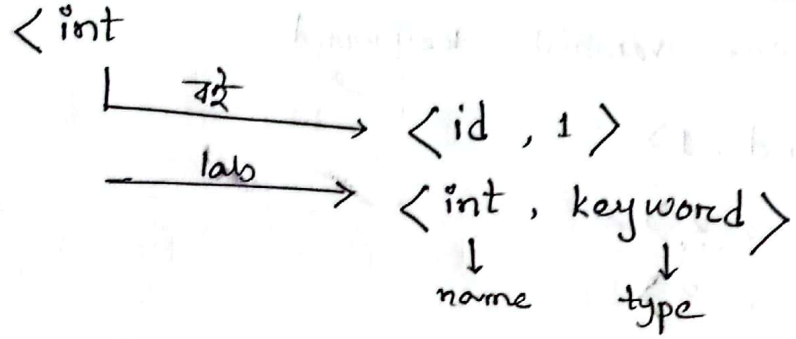
int ⟶ keyword
a ⟶ identifier (12aX a12✓ _a✓ a_✓)
; ⟶ semicolon
= ⟶ assignment operator
== ⟶ relational "
123 ⟶ number

name | type

lexical analyzer ⟶ scanner ⟶ symbol table এর scan করে

⟶ token stream generate করে

how token streams are generated :



Token stream → < token_name , attribute_value >

* position = intial + rate * 60 → make a token stream for this source program

position → <ID, 1>

= → <=>

initial → <ID, 2>

+ → <+>

rate → <ID, 3>

* → <*>

60 → <60>

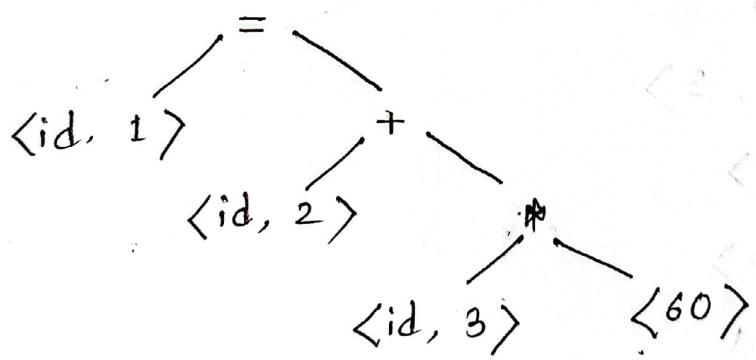
→ syntax analyzer → this
parser

id → function_name, variable, keyword.

int → < keyword, 1 >
float → < " , 2 >
double → < " , 3 >

Lexical analyzer → token → syntax analyzer
↓
syntax tree generate করতে হবে।
↓
semantic analyzer

syntax tree :
operator → parent/root



* ';' (semicolon) won't be included in syntax tree

* Semantic analyzer :
↳ type checking করে conversion করে।

Ex : int a = 1;
int b = 3;
float sum = a + b;
4.0

float a = 1.6
float b = 3.2
int sum = a + b;
4

3-address code ⟶ 3 টার বেশি operand থাকা যাবে না।
⟶ প্রতি assignment operation এ right side এ 1 টা (atmost) operator থাকবে।

Ex : Suppose, rate, initial, position $\xrightarrow{\text{datatype}}$ float.

rate * 60

↓

let, rate = 40.0

60 will be converted into 60.0, (semantic analyzer.

So, int to float (60)

- Semantic analyzer ⟶ intermediate code generator
  ↳ যে code generate হয় তাকে 3 address code বলে।

```
t1 = int tofloat (60)
t2 = <id, 3> * t1 ;
t3 = <id, 2> + t2
<id, 1> = t3
```

} intermediate representation

- int. code gen ⟶ code optimizer (optional)

```
t1 = id 3 * 60.0
id1 = id2 + t1
```

} code optimization (Faster & shorter)

Machine code generate :

machine dependent

assembler uses register.

(Mid level lang).

LOAD
ADD
SUB
MUL
DIV
STORE
MOV
(DEL)

```
LOAD    R2, id3
MUL     R2, 60.0
LOAD    R1, id2
ADD     R1, R2
STORE   id1, R1
```

#60.0 → 60 is a constant.

assembler uses # to identify constant.

(scanner) lexical → token
(parser) syntax → tree
semantic → type check
inter. CG → 3 address code.

28.4.24

scanner → lexical analyzer

Symbol Table

name, type, scope, function এর info.

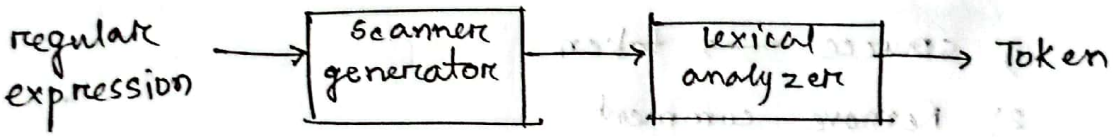
Grouping of phases into passes :

- One pass / Front End Pass ⟶ Analysis
- Back end Pass ⟶ synthesis

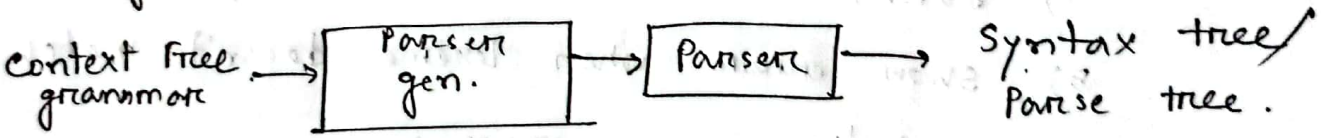
* Code optimizer ⟶ Optional pass

Tool for compiler :

* Scanner generators :



* Parser generator :



* Syntax Directed Translation Engine
* Code generator generators
* Data Flow Analysis Engine
  ↳ calculation এর flow নিয়ে বিশ্লে
* Compiler construction Tool Kits
  ↳ 6 টা phase একসাথে integrated.

5. 5. 24

Chap 3

Lexical Analysis

position = initial + rate * 60

- character stream / lexeme is generated using lexical analyzer
- stream of tokens are sent to parser (synb analyzer)

types of generated token:

position ⟶ <id, 1>

or, < position, identifier, 1 >

better → or, < identifier, symbol table entry for Position,

Type checking ⟶ semantic analyzer

ROLE OF LEXICAL ANALYZER :

1) source ⟶ token
2) Remove comment
3) keyword, id, operator, constant, separator → recognize
4) blank, newline, blank ⟶ strip out
5) show error when lexeme doesn't match
6) correlating error message

⟶ Scanning
⟶ Lexical analysis

Why separated from Syntax analyzer :

- Simplicity → easier to maintain, test
- Efficiency
- Portability

lexeme → seq. of characters
pattern → description

Token :

if, else, comparison, id, number, literal

comparison → <=, !=

literal → surrounded by " "

number → 60, 3.1416, pi (constants)

```
if (a == b)
    printf ("it's true") ;
```

if
a → <id, 1>
== → <==> or,
<comparison, 1>, or
<comparison, symbol table entry for ==>

```
printf("Total = %d", score);
```

printf → printf (unique keyword like if)
"..." → literal
, → comma
score → <id, symbol table entry for score>
; → semicolon

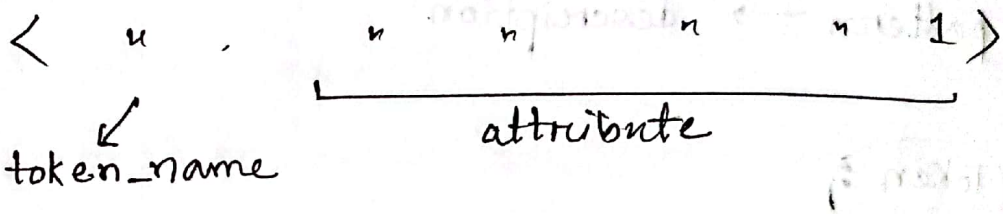
Attribute for token :

* mutiple lexeme → same pattern

  ↳ then attribute for token is must.

0
1 > both number

So, <number, symbol table entry for 0>



<hello world"> or < literal, symbol table entry for "hello world" >

7.5.2024

Attribute for token :

int result ;

appropriate attribute value

<result, identifier, pointer to the symbol table entry for result>

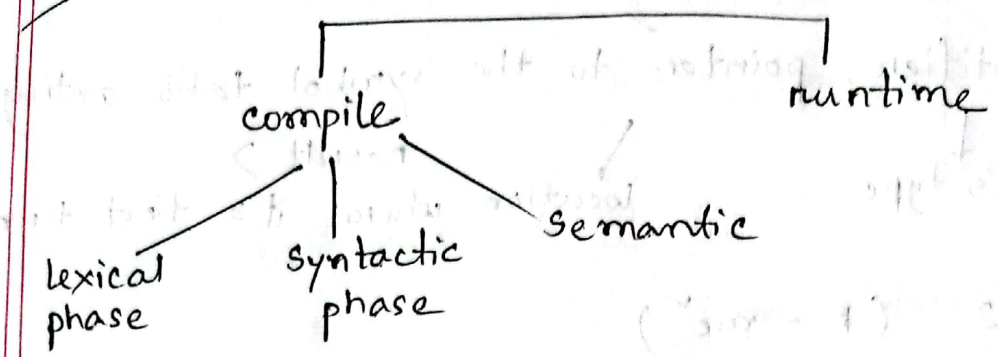
result → lexeme
identifier → it's type
pointer → location where it is first found

E = m * c ** 2 ($E = mc^2$)

E	< identifier, pointer t. t. s. t. e. f. E >
=	< assign-op >
m	< identifier, pointer ---------- M >
*	< mul-op >, or < * >
c	< identifier, pointer ---------- c >
**	< exp-op >
2	< number, pointer ---------- 2 > or < number, integer value 2 >

op, punctuation, keyword → no need of attribute value

Errors :



lexical error :

fi (a = = f(x))

↳ misspelling of keyword if

→ cannot detect the error ; it'll be used as id.

* exceeding length of id / numeric constant.
* appearance of illegal chars.
* unmatched string / pattern

Ex : printf ("Hello World"); $

↖ illegal char (lexical error ধরতে পারবে)

panic mode → illegal char কে skip করবে as এটা recognizable না।

- Synchronizing token are delimiters → ; or }

error - recovery actions :

1. deleting extraneous character

   Ex : printfx ⟶ printf কে symbol table এ entry করে deleting the char "x"

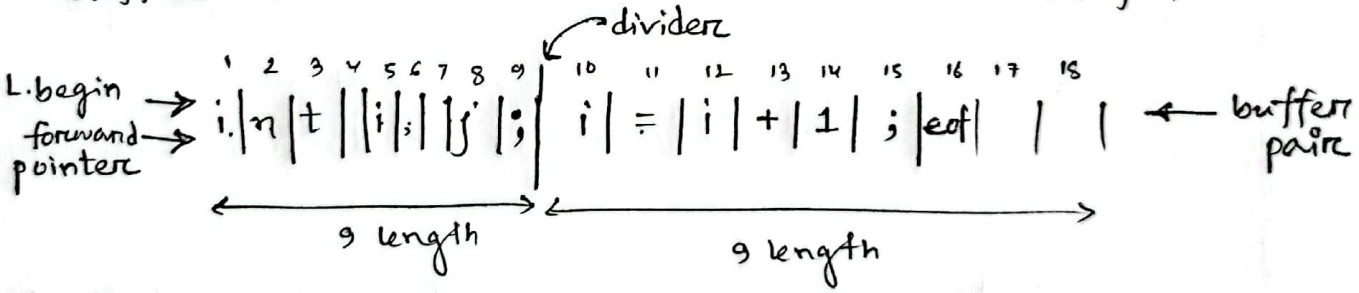
2. Inserting a missing character.
3. replacing an incorrect char by correct char.
4. transposing two adjacent characters.

5test ⟶ 5 এর পর ভুলে "test" suffix ⟶ lexical error.

How can we speed up the reading

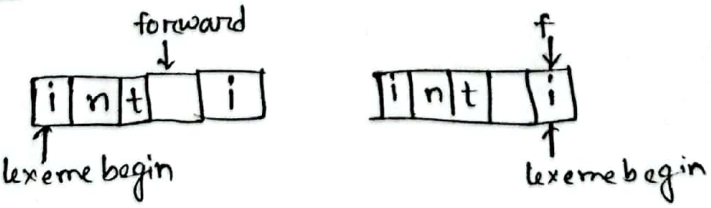
Input buffering :

buffer is divided into 2 N characters half.



source code শেষে ⟶ eof দিয়ে দিবো buffer এ

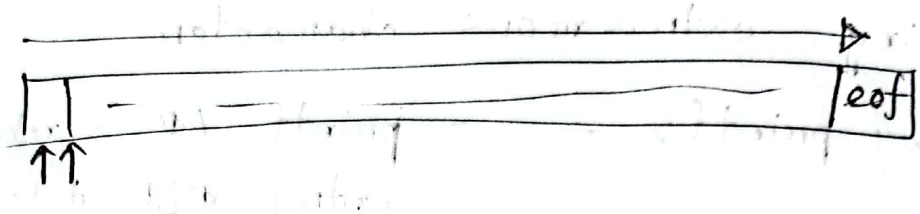
- lexeme begin & forward initially points to the first character of input string.
- forward ⟶ moves ahead to search end of lexeme



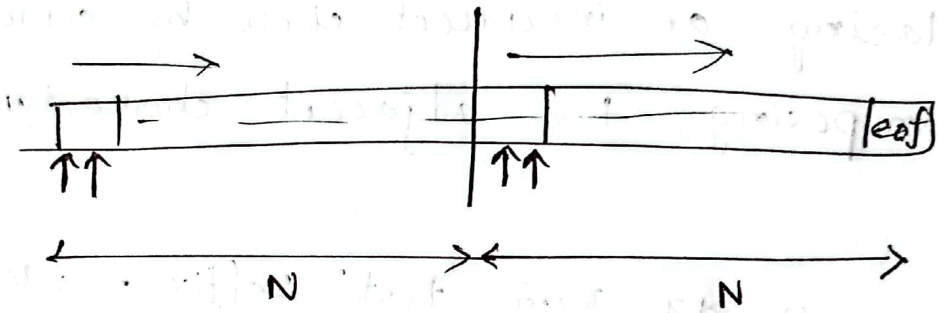
tab/↵/blank space ⟶ lexeme generate হবে না

Question } Input buffering for ?
One scheme buffering का function
Two " " " "

one ~~stream~~ scheme buffer :



to speed up → two ~~scheme~~ buffer



* 2 part मध्ये lexeme begin & forward pointer याचा
* first & second buffer are scanned alternately.

8. 5. 2024

Sentinel → প্রহরা or guard

→ next char পড়ার আগে we will check 1st or 2nd half এর end এ চলে এসেছি কিনা।

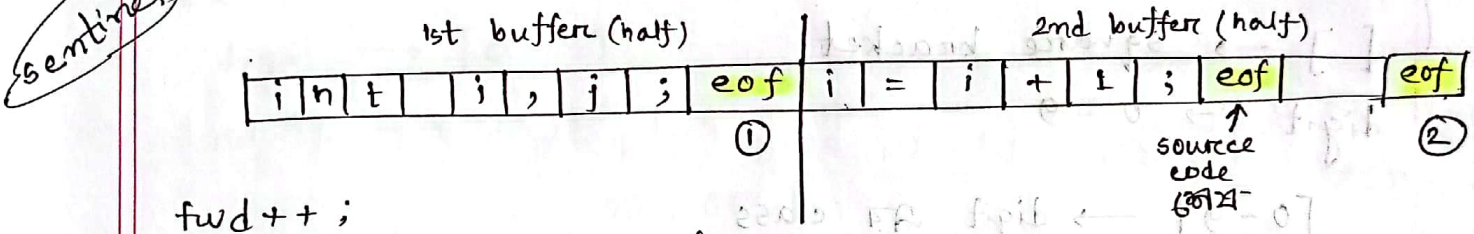
1st half এর শেষে
→ reload 2nd half
→ fwd কে 2nd half এর first এ দিব

2nd half শেষ,
→ reload 1st half
→ fwd কে 1st half এর শুরুতে দিব।

soln

time consuming to take two tests always.

sentinel



fwd++;
checks if fwd == eof.

① & ② → guards/sentinel

↳ optimizes the code by reducing number of tests

Recognition of tokens :

stmt → if expr (is true) then stmt |
if expr then stmt else stmt | ∈

expr → term relop term | term

term → id | number

relop → relational operator (>, <, >=, <=, ==, !=)
সেকোনো comparison op. is a relational operator.

id → letters followed by some digits or letters

Note
1) '|' → or
2) number is constant
3) term should be id or numb

S → stmt
E → expr
T → term

S → if E then S | if E then S else S | ϵ

E → T relop T | T

T → id | number

Primitives :

. → any char except newline

\n → newline

() → parentheses

{} → curley braces

[] → square bracket

digit → 0 - 9

[0-9] → digit এর class

└→ 0-9 এর মধ্যে যেকোনো একটা digit

* → Ex : 0^* → either 0 or no zero

(যার) ←

└→ zero or more copies of preceeding expression

(ϕ, 0, 0000 ....)

+ → one or more copies of preceeding expression

Ex : 0^+ → 0, 00, 000, ...

? → zero or one copy of

Ex : 0? → 0 or ϕ

Note :

1) whitespace is a charact

Ex :

```
if(c==' ')
{
;
```

ϵ → empty string

^ → beginning of line

$ → end of line

a|b → a or b

$(ab)^+$ → one or more copy of 'ab' group)

Ex : ab✓, abab✓, ababab✓, aabX .

$(ab)^*$ → ϕ, ab, abab, ...

"a+b" → literal |"a+b"

Ex :

digit → [0-9]

digits → $digit^+$

Ex : 00, 012 ...

number → digits (. digits)? (E [+-]? digits)?

token — pattern

generated examples are lexeme

Ex : 12.35 (float number)

↳ 12 (. digits)?

optional

12, 12.0, 0.35, 12.35783 70, ...

12E45✓, 3.5EX, 3.5E24✓, 35E+24✓,

0.35E-24✓, 12.34E-34✓, 12.E34X

letter → [A-Z a-z]

or [A-Z] | [a-z]

id → letters (letters | digits)*

Ex : a✓, abb✓ a123✓, a123u✓, _ty X

letters → letter ✗

CT → sentinel পর্যন্ত

If, letter → [A-Za-z] | [_ | $ | #]

then, id → Ex: #112, A#12, all are valid as I've redefined letter.

if → if
then → then
else → else
break → break
} pattern

relop → < | > | <= | >= | = | <> → not equal (!= can also be used)

token

Note:

neg. numbers এর জন্য,

number → [+-]? digits (. digits)? (E [+-]? digits)?

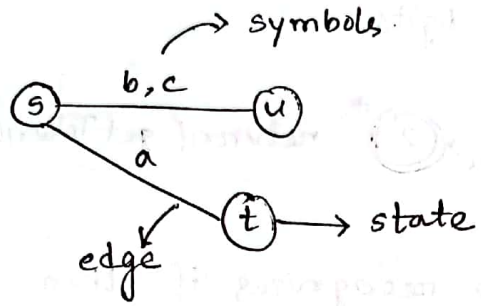
ws → (blank | tab | newline)+

↳ whitespace strip out করতে define করবো, we don't return it to the parser

9.5.2024

Transition Diagrams :

→ collection of nodes/circles called states.

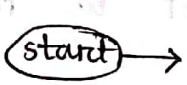


symbol "a" হলে s → t যাবে

→ deterministic (not more than one edge out of a given state for a given symbol)

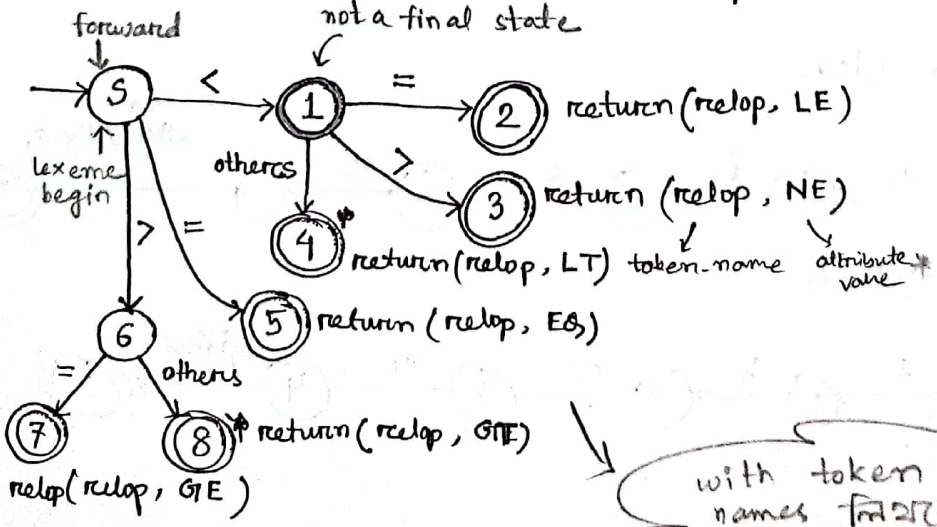
→ accepting / final state → double circle. 

→ যখন বলানো situation when I need to step back my forward pointer → I will use "*" in my accepting state.

→ Start state →  → begins from state "start".

Example of transition diagram :

- relop → < | > | <= | >= | = | <>

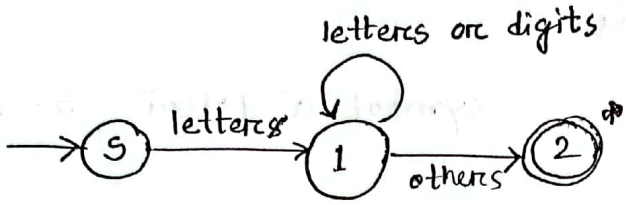


<others → FS 4
<= → FS 2
<> → FS 3
=, > এরকমগুলো দিব না, cz we need to know for which lexeme I moved to final state

with token names লিখতে হবে

Recognition of reserved words or identifiers :

- id → letter (letters | digits)*



return(getToken(), installID())

↳ this diagram also recognizes if, then, else --- etc keywords.

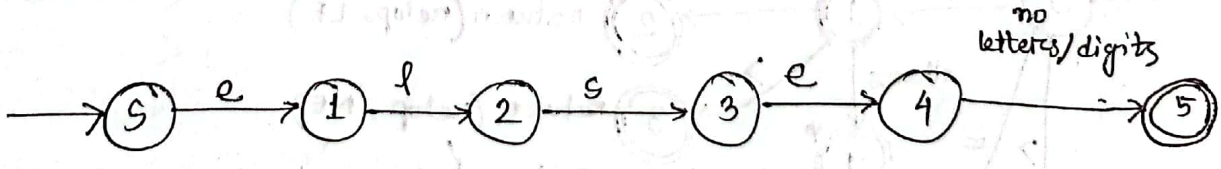
↳ 32 টা transition diagram for keywords বানাবো (symbol table এ আগেই দিয়ে রাখবো)

installID() → checks if the lexeme is already in the table. If it's not present, → it's id.

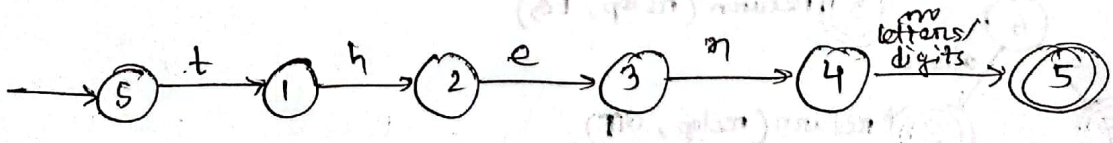
Any identifier not in the symbol table during lexical analysis cannot be reserved words, so token is id.

getToken() → returns the token name; either id or a name of reserved words.

- else → else

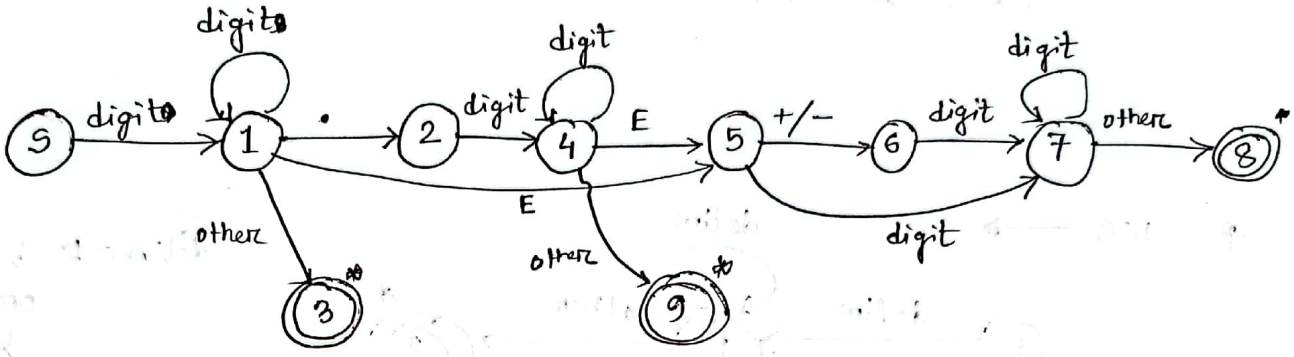


- then → then



digit → [0-9]
digits → [digit]+

- number → digits (.digits)? (E [+-]? digits)?



·12E· → it's not a lexeme of token 'number'.

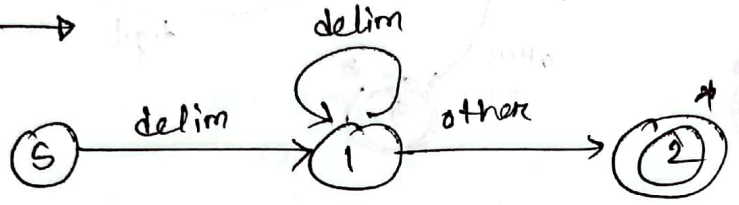
(Q) If it's a lexeme or not, explain the reason.

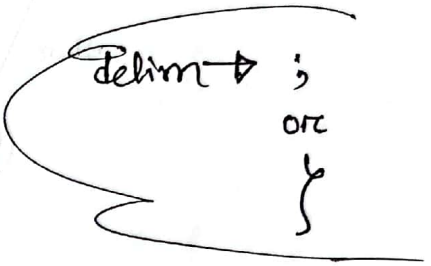
15. 5. 2024

H.W complex number এর token & ~~pattern~~ lexeme use করে pattern লিখে আনবো।

Ex : $a \pm bi$

WS →





Note : No action done in the accepting state.

বই / slide এর state no. follow করতে হবে

term paper on number or id

Architecture of Transition Diagram :

State → case

retract() → 1 step back এ যাওয়া

fail() → not an error case

Ex : 123.E

error case → সবগুলা

Token_name
attribute_value

TOKEN ⟶ class

retToken ⟶ relational operator এর একটা Instance new(RELOP) দিয়ে create করছি।

state ⟶ 0 (start)

যত state ⟶ তত case in the switch.

state 0 ⟶ তে next char read করে c তে রাখবো

(case 0)
- < ⟶ 1 state
- = ⟶ 5 "
- > ⟶ 6 "

else fail() ⟶ lexeme is not relop.

case 1 : c = nextchar();

if(c == '=') ⟶ state=2;

else if(c == '>') ⟶ state=3;

else state=4;

break }

retToken এর name সবার জন্য relop or RELOP.

case 4 : retract();

retToken.attribut_value = LT

return(retToken);

break;

```
case 2 :
        retToken.attribute = LE ;
        return ( retToken ) ;
        break ;

case 3 :  retToken.attribute = NE
        return (retToken) ;
        break ;

case 5 :  retToken.attribute = EQ ;
        return (retToken) ;
        break ;

case 6 :
        c = nextChar() ;
        if (c == '=')  state = 7
        else  state = 8 ;
        break ;

case 7 :  retToken.attribute = GE ;
        return (retToken) ;
        break ;

case 8 :  retract() ;
        retToken.attribute = GT ;
        return (retToken) ;
        break ;
```

mid + final ⟶ (imp)

(9) Write a C++ function whose job is to stimulate transition diagram for relop / id / number / complex

(Note : Token, pattern & transition diagram might be given)

- token
- accepted
- fail case

Lexical analyzer ⟶ lex ⟶ mid + Final ⟶ (imp).

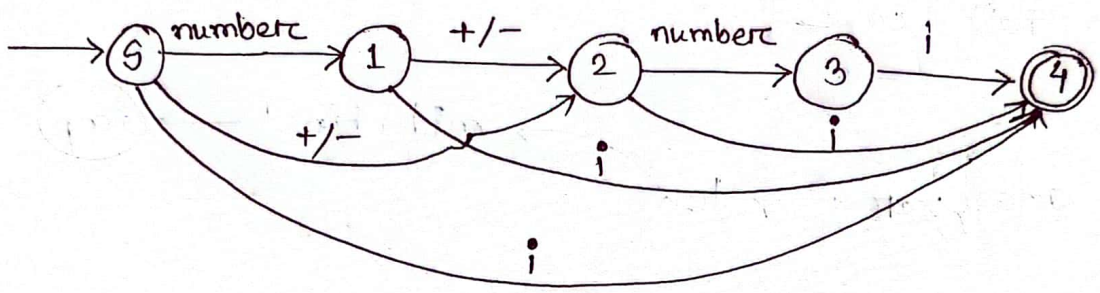
16.5.24

Transition Diagram of complex numbers :

complex number ⟶ a ± bi

CN ⟶ number [+/-] (number) ? i |

[+/-] ? (number) ? i



a+bi
a-bi
+bi
-bi
-i
bi
2i
i
-2i
bi ✗
+bi ✓
i ✓
a real
b imaginary

29. 5. 2024

V/p buffering
sentinel
panic mode
One or two scheme
lexeme, pattern, transition diag.
code given → token generation এর জন্য pattern লিখতে হবে।

```
int main()
{
    char ch;
    ch = 'a';
    printf("%c", ch);
}
```

1) keyword
2) id
} sequence must be accurate

→ pattern generate করতে হবে. int, char, id, semi, comma, LC, RC, LP, RP, string

SYNTAX ANALYSIS

→ CHAPTER 4 ←

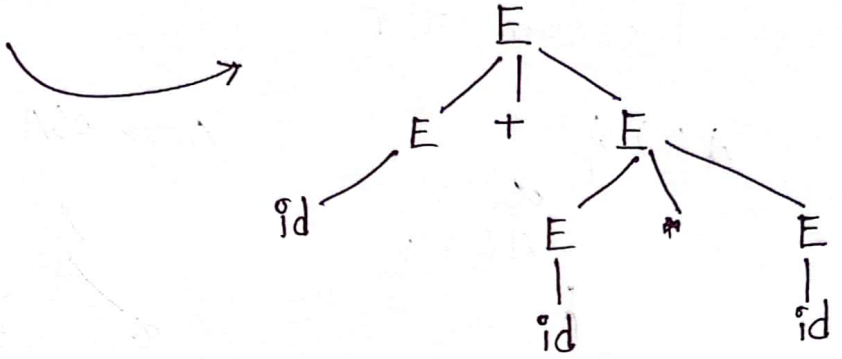
যার বেশি grammar generate হয় → non terminal (usually capital letters)

যা না → terminal

Ex : (,), *, +, id

- $E \longrightarrow E+E \mid E*E \mid (E) \mid id$

$a + b * c$



→ top-down (root to leaves)

- top-down
- bottom-top
- universal

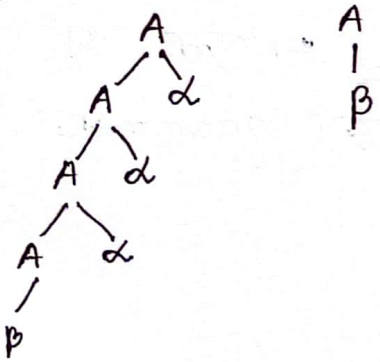
} CFG to parse tree

- Left recursion → terminal পড়ে
  → the funtion is being called first

$A \rightarrow A\alpha \mid \beta$

```
A() { A() ;
     α ;
}
```

→ A is calling itself at first.



$\beta\alpha\alpha\alpha$, β

↓

$\beta\alpha^*$

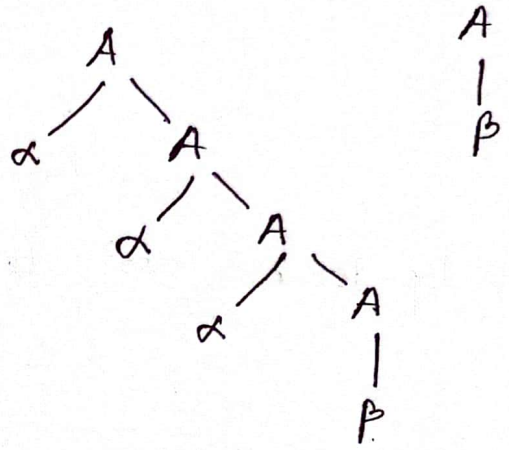
top-down parsing, left recursive grammar accept করে না।
So, we need to eliminate left recursion.

নিজে নিজে call করছি terminal পাওয়ার পরে

- Right recursion
  - ⟶ terminal আগে
  - ⟶ function পরে

```
A() {
    α ;
    A() ;
}
```

$A \rightarrow \alpha A \mid \beta$



$\alpha\alpha\alpha\beta$, β

$\alpha^* \beta$

'*' থাকলে কোনো string এ থাকলে তা modify করতে হবে। এগুলো কোনো grammar accept করবে না।

Left recursion ?

$A \to A\alpha \mid \beta$ ✓

$A \to \alpha A$ ✗

$A \to \alpha$ ✗

$E \to T * T$ ✗

- $A \to A\alpha \mid \beta$ → '|' এর পরের part "β"

  A = recursive part, α = এর পরের part

$A \to \beta A'$ → A' = non terminal এর prime

$A' \to \alpha A' \mid \epsilon$

$A \to A\alpha \mid A\alpha\beta \mid \gamma$

α α β

- $E \to E+T \mid T$

  E+T → α, T → β

$E \to T E'$

$E' \to +T E' \mid \epsilon$

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

  E+E → α1, E*E → α2, (E) → β1, id → β2

$E \to (E)\, E' \mid id\, E'$

$E' \to +E\, E' \mid *E\, E' \mid \epsilon$

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \epsilon$

recursive α β

- $E \rightarrow E(+T) \mid T$ — Non-terminal → E, T

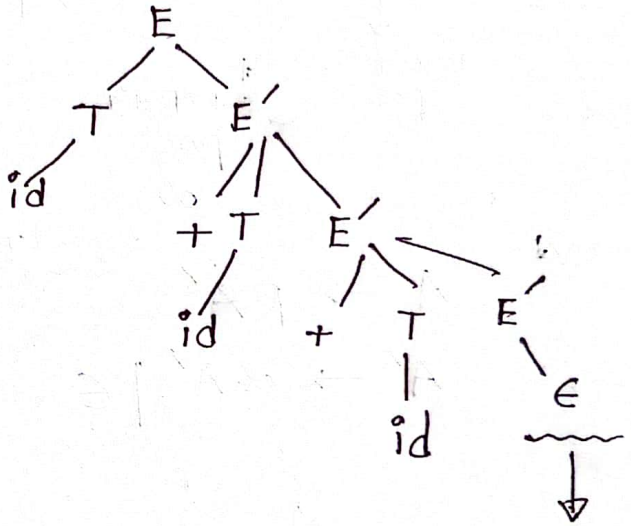
$T \rightarrow id$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow id$

generate id + id + id



(Q) why left recursive grammar goes into loop for top-down parsing ?

→ $A \rightarrow A\alpha \mid \beta$ → we don't known where to stop. A is calling itself again & again

↳ to eliminate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$.

it helps to backtrack

Non-term → E, T, F

- $E \rightarrow \underline{E} + T \mid T$ (α_1 = $+T$, β_1 = T)

  $T \rightarrow \underline{T} * F \mid F$ (α_2 = $*F$, β_2 = F)

  $F \rightarrow (E) \mid id$

  → we didn't replace as nothing matters if I replace or not replace

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \epsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid id$

- $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

  ~~$S \rightarrow Aa \mid b$~~

  ~~$A \rightarrow SdA' \mid A'$~~

  ~~$A' \rightarrow cA' \mid \epsilon$~~

  $S \rightarrow (Aa \mid b)$

  $A \rightarrow Ac \mid (Aa \mid b)d \mid \epsilon$

  $A \rightarrow \underline{A}c \mid \underline{A}ad \mid bd \mid \epsilon$ → S আছে নেই

  $A \rightarrow bdA' \mid A'$

  $A' \rightarrow cA' \mid adA' \mid \epsilon$

যদি উল্টোটা,

$A \rightarrow Ac \mid Sd \mid \epsilon$

$S \rightarrow Aa \mid b$

→

$A \rightarrow \underline{A}c \mid Sd \mid \epsilon$

↳ $A \rightarrow SdA' \mid A'$ ✓ (This 'A' will be used for $S \rightarrow Ab$)

$A' \rightarrow cA'$

$S \rightarrow \underline{S}dA'a \mid A'a \mid b$

↳ $S \rightarrow A'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \epsilon$

1) $S \rightarrow SoSL \mid 01$

2) $L \rightarrow L,S \mid S$

3) $S \rightarrow A$
$A \rightarrow Ad \mid Ae \mid aB \mid a@$
$B \rightarrow bBe \mid f$

4) $A \rightarrow Ba \mid Aa \mid e$
$B \rightarrow Bb \mid Ab \mid d$

5) $S \rightarrow X \mid b$
$X \rightarrow S \mid a$

6) $S \rightarrow xSb \mid b$
$X \rightarrow \epsilon$

(Q) Why 5 & 6 are not left recursive ?

Answer :

1) $S \rightarrow S0S1 \mid 01$

$S \rightarrow 01S'$

$S' \rightarrow 0S1S' \mid \epsilon$

2) $L \rightarrow L, S \mid S$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

3) $S \rightarrow A$

$A \rightarrow aBA' \mid aCA'$

$A' \rightarrow dA' \mid eA' \mid \epsilon$

$B \rightarrow bBe \mid f$

4) $\begin{cases} A \rightarrow Ba A' \mid cA' \\ A' \rightarrow aA' \mid \epsilon \end{cases}$

$B \rightarrow AbB' \mid dB'$

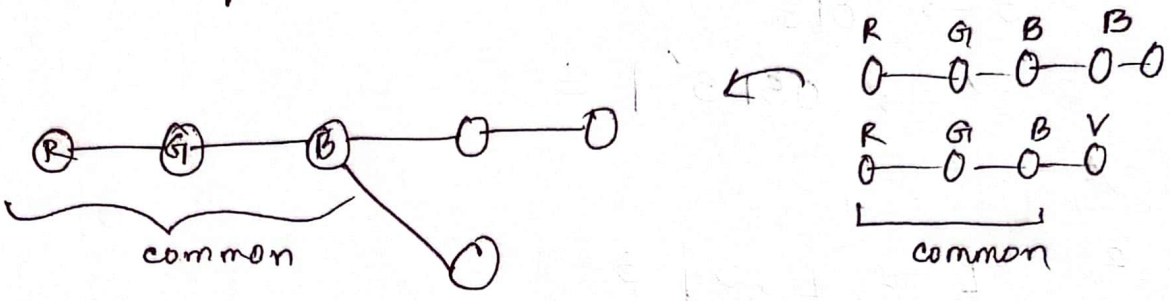
$B' \rightarrow bB' \mid \epsilon$

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

$\begin{cases} B \rightarrow cA'bB' \mid dB' \\ B' \rightarrow bB' \mid aA'bB' \mid \epsilon \end{cases}$

Left factoring :

→ common path একবারই নিবো



Ex :

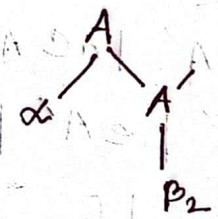
Non-deterministic ← $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ → common 'α' + সাথে A'
" বাদে সব β

deterministic ↓

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$

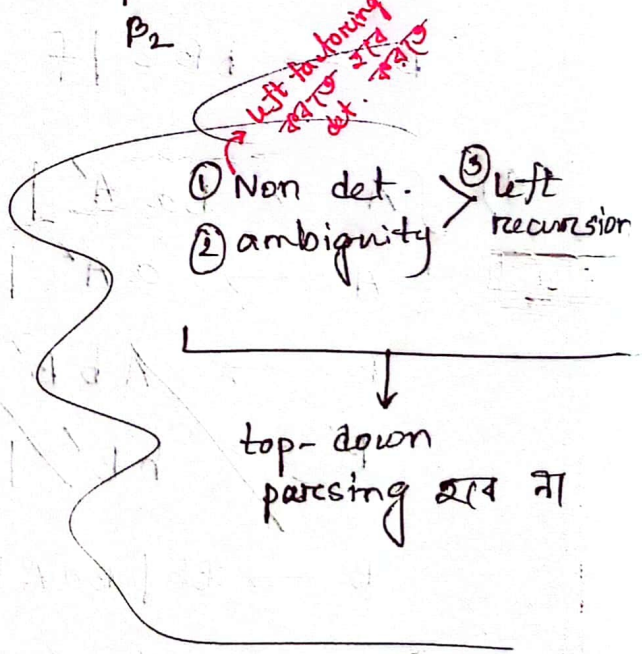
↓
পরের step কোনটা determine করা possible



* $S \to aEd$
$A \to aEf$

string is 'aEf'. So, there is ambiguity.

S থেকে or A থেকে start করে desired string পাবো নাকি বুঝবো না।



* capital letter → Non terminal
small " → terminal

stmt (S) → if expr then stmt else stmt |
if expr then stmt

$S \rightarrow \underbrace{iEtS}_{\alpha}\underbrace{eS}_{w} \mid \underbrace{iEtS}_{\alpha}\underbrace{}_{w}$

$S \rightarrow iEtSS'$
$S' \rightarrow eS \mid \epsilon$

$S \rightarrow \underline{iEtSeS} \mid \underline{iEtS} \mid a \quad , \quad E \rightarrow b$

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

$S \rightarrow \underline{aSSbs} \mid \underline{aSasb}$

$S \rightarrow aSS'$
$S' \rightarrow Sbs \mid asb$

$S \rightarrow \underline{a}SSbs \mid \underline{a}Sasb \mid \underline{a}bb \mid b$

$S \rightarrow aS' \mid b$
$S' \rightarrow \underline{S}Sbs \mid \underline{S}asb \mid bb$

$S' \rightarrow SS'' \mid bb$
$S'' \rightarrow Sbs \mid asb$

So, $S \rightarrow aS' \mid b$
$S' \rightarrow SS'' \mid bb$, $S'' \rightarrow Sbs \mid asb$

$S \rightarrow a\,SSbs \mid aSasb \mid abb \mid b$

$A \rightarrow aB$
$B \rightarrow aB$
} No left factoring

(Same as last one.)

HOME TASK

(i) $S \rightarrow bSSaaS \mid bSSasb \mid bSb \mid a$

(ii) $S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$

(iii) $S \rightarrow a \mid ab \mid abc \mid abcd$

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

$A' \rightarrow AB \mid BC \mid AC$

$E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$

$E \rightarrow TE'$

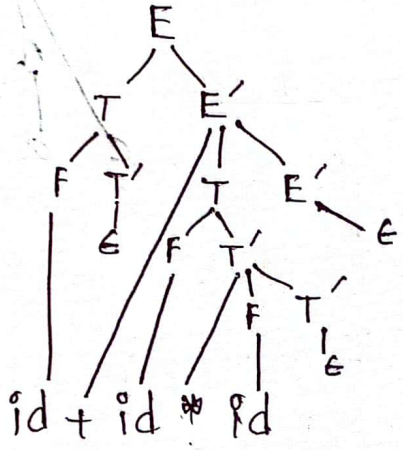
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

id+id*id →



Types of Top-down parsing :

1) Recursive - descent parsing → requires backtracking

2) Predictive parsing

Recursive descent Parsing :

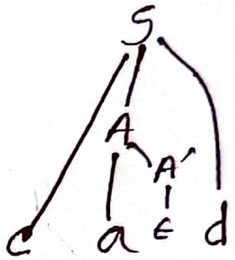
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow b \mid \epsilon$



- $A \rightarrow Ta$
  $T \rightarrow *FT' \mid \epsilon$

  FIRST(A) = FIRST(T) = {*, ϵ}

  $A \rightarrow *a$ $\quad$ $A \rightarrow a$

  So, FIRST(A) = {*, a}

- $A \rightarrow TB$
  $T \rightarrow *FT' \mid \epsilon$
  $B \rightarrow b$

  FIRST(A) = FIRST(T) = {*, ϵ}

  $A \rightarrow *B$ $\quad$ $A \rightarrow B$

  FIRST(B) = {b}

  FIRST(A) = {*, b}

- $S \rightarrow ACB \mid Cb\,B \mid Ba$
  $A \rightarrow da \mid BC$
  $B \rightarrow g \mid \epsilon$
  $C \rightarrow h \mid \epsilon$

First (S) = {d, g, h, ϵ, a, b}

First (A) = {d, g, h, ϵ}

First (B) = {g, ϵ}

First (C) = {h, ϵ}

- $A \rightarrow Ta$
  $T \rightarrow *FT' \mid \epsilon$

  FIRST(A) = FIRST(T) = {*, ε}

  $A \rightarrow *a$ $\quad$ $A \rightarrow a$

  So, FIRST(A) = {*, a}

- $A \rightarrow TB$
  $T \rightarrow *FT' \mid \epsilon$
  $B \rightarrow b$

  FIRST(A) = FIRST(T) = {*, ε}

  $A \rightarrow *B$ $\quad$ $A \rightarrow B$

  FIRST(B) = {b}

  FIRST(A) = {*, b}

- $S \rightarrow ACB \mid Cb\,B \mid Ba$
  $A \rightarrow da \mid BC$
  $B \rightarrow g \mid \epsilon$
  $C \rightarrow h \mid \epsilon$

  First (S) = {d, g, h, ε, a, b}

  First (A) = {d, g, h, ε}

  First (B) = {g, ε}

  First (C) = {h, ε}

bB

gC
C

dCB
gCB
hCB

ga
a

dghba

production rule এর জানে

Rules For FOLLOW :

① প্রথমে যে non terminal এর FOLLOW বের করবো (here S) তার শুরুতে $ বসিয়ে দিব।

- $S \rightarrow aABb$
  $A \rightarrow c \mid \epsilon$
  $B \rightarrow d \mid \epsilon$

FIRST (B) = {d, ϵ}
FIRST (A) = {c, ϵ}
FIRST (S) = {a}

FOLLOW (S) = {$}
FOLLOW (A) = {d, b}
FOLLOW (B) = {b}

② যখন কোনো কিছুকে একটা non-terminal (here B) follow করে এটার FIRST বের করতে হবে।

FIRST (B) = {d, ϵ}
A d b
A b

- $S \rightarrow aAB$
  $A \rightarrow c \mid \epsilon$
  $B \rightarrow d \mid \epsilon$

FOLLOW (S) = {$}
FOLLOW (B) = {$}
FOLLOW (A) = {d, $}
↳ aAd, aAϵ
{d, $}

③ যখন কোনো কিছুকে ϵ FOLLOW করবে তখন তার FOLLOW হবে "সে কার কাছ থেকে আসছে" তার FOLLOW।

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

- $E \to TE'$
  $E' \to +TE' \mid \epsilon$
  $T \to FT'$
  $T' \to *FT' \mid \epsilon$
  $F \to (E) \mid id$

FIRST (F) = { (, id }

FIRST (T') = { *, ∈ }

FIRST (T) = { (, id }

FIRST (E') = { +, ∈ }

FIRST (E) = { (, id }

FOLLOW (E) = { $,) }

FOLLOW (E') = { $,) }

∈ ছিল তাই E এর FOLLOW and E'

FOLLOW (T) = E' FOLLOW করছি
↳ non terminal
so, FIRST (E') = { +, ∈ }
↓
T+ , T∈
↓
$ and E এর FOLLOW

= { +, $,) }

FOLLOW (T') = { $, +,) }

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

id + id * id

id FIRST এ আছে ⟶ E, T & F এর
↳ Production rule এ সবার আগে

* Predictive parsing এর ক্ষেত্রে FIRST & FOLLOW বের করা must.
↳ No backtracking

Recursive Parsing

LL(1) ↳ 1 টা input symbol at a time scan করবো।
↙ ↘ leftmost derivation
scanning the input from left to right

M ⟶ table
A ⟶ row ⟶ non terminal
α ⟶ column ⟶ terminal

ϵ লিখবো না

	+	*	id	(	)	$
E			E→TE′	E→TE′		
E′	E′→+TE′				E′→ϵ	E′→ϵ
T			T→FT′	T→FT′		
T′	T′→ϵ	T′→*FT′			T′→ϵ	T′→ϵ
F			F→id	F→(E)		

* ϵ আসলে A এর FOLLOW কপি করে বসাবো।

* A → BT B —FIRST→ b

b
A → BT

1) E → TE′
↳ T এর FIRST (, id

E′ → +TE′ | ϵ
↳ terminal ⟶ E′ এর FOLLOW ⟶ $,)

6, 6, 2024

$S \rightarrow \underbrace{aSbS}_{\alpha} \mid \underbrace{bSaS}_{\beta} \mid \underbrace{\epsilon}_{\gamma}$

(A)

FIRST(S) = { a, b, ϵ }

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

	a	b	$
S	S → aSbS S → ϵ	S → bSaS S → ϵ	S → ϵ

S → α এর FIRST → a
S → β " " → b
S → γ " " → ϵ
} disjoint

S → ϵ এর জন্য S → FIRST { a, b, ϵ }

$S \rightarrow \overbrace{asb}^{\alpha} \mid \overbrace{asc}^{\beta}$ ⟶ LL(1) হবে না।

First same for α, β.

$S \longrightarrow AAab \mid BbBa$

$A \longrightarrow a \mid \epsilon$

$B \longrightarrow b \mid \epsilon$

$A \to a$ ↓ FIRST

$A \to \epsilon$ ↓ FOLLOW of A · a

Same

NOT LL(1).

First (B) = { b, ϵ }

First (A) = { a, ϵ }

FIRST (S) = { a, b }

FOLLOW (S) = { $ }

FOLLOW (A) = { a }

FOLLOW (B) = { b, a }

$S \to iE$ ---- ⟶ NON LL(1) ⟶ HOME TASK.

* left recursion আছে কিনা
* " factor

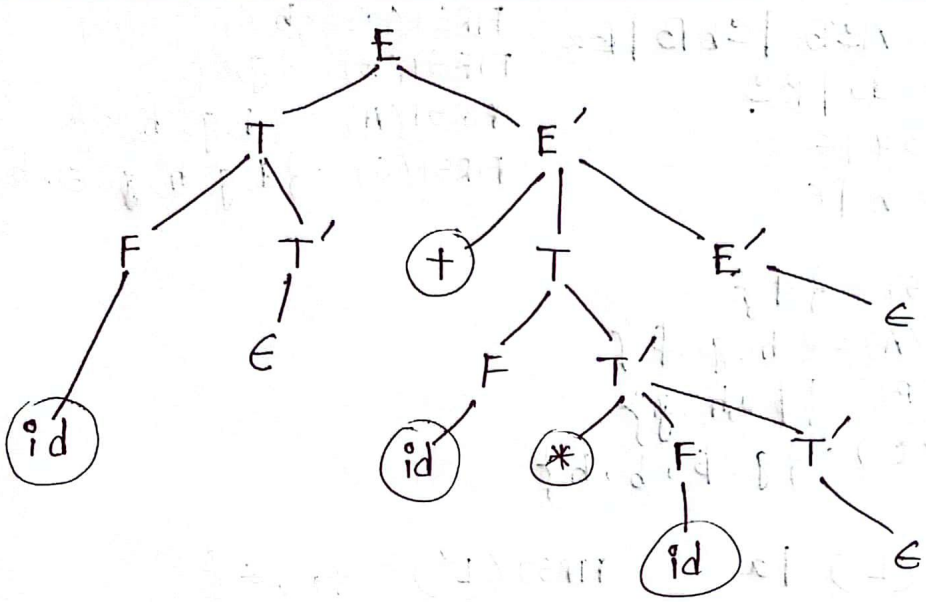
Non recursive predictive parsing

STACK

MATCH	STACK	input	ACTION
	$	id + id * id $	
	E$	id + id * id $	E
	TE'$	id + id * id $	E → TE'
	FT'E'$	id + id * id $	T → FT'
	idT'E'$	id + id * id $	F → id
id	T'E'$	+ id * id $	Matched id
id	E'$	+ id * id $	T' → ∈
id	+TE'$	+ id * id $	E' → +TE'
id+	TE'$	id * id $	matched +
id+	FT'E'$	id * id $	T' → FT'
id+	idT'E'$	id * id $	F → id
id+id	T'E'$	* id $	Matched id
id+id	*FT'E'$	* id $	T' → * FT'
id+id*	FT'E'$	id $	matched *
id+id*	idT'E'$	id $	F → id
id+id*id	T'E'$	$	matched id
id+id*id	E'$	$	T' → ∈
id+id*id	$	$	E' → ∈

input matched !!!

Using Actions:



for final

grammar ⟶ check if left recursive আছে নাকি

Non det হলে left factoring

Non LL(1) হলে / দরকার নাই

To gen. predictive parsing ⟶ det. FIRST & FOLLOW.

Using stack non-recursive pred. parsing.

FIRST & FOLLOW DETERMINE :

① $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

FIRST(C) = {h, ε}
FIRST(B) = {g, ε}
FIRST(A) = {d, g, h, ε}
FIRST(S) = {d, g, h, ε, b, a}

FOLLOW(S) = {$}
FOLLOW(A) = {h, g, $}
FOLLOW(B) = {$, a, h, g}
FOLLOW(C) = {g, $, b, h}

② $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' \mid \epsilon$

FIRST(L') = {, , ε}
FIRST(L) = {(, a}
FIRST(S) = {(, a}

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

③ $S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' \mid \epsilon$
$B \rightarrow b$

FIRST(B) = {b}
FIRST(A') = {d, ε}
FIRST(A) = {a}
FIRST(S) = {a}

FOLLOW(S) = {$}
FOLLOW(A) = {$}
FOLLOW(A') = {$}
FOLLOW(B) = {d, $}