PAGE No.
DATE 04/12/23

CH-1

→ ① Translation: ② Translation software ③ What are the language translators?

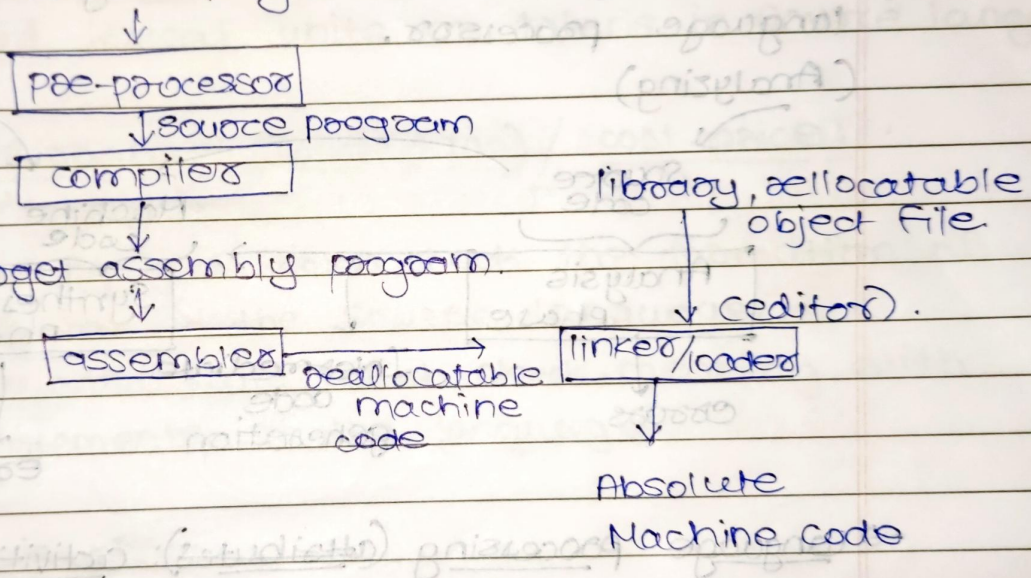
⇒ translation: is more than just changing the word from one language to another. So translation gives bridge b/w the languages.

⇒ translation software: translate whole program to another language (low level).

→ language translators: ① compilers ② interpreters ③ assemblers

* 1) Language processing system:

→ skeletal source program.



→ pre-processor: ① macro-processing
② file inclusion
③ relational pre-processor
④ language extensions

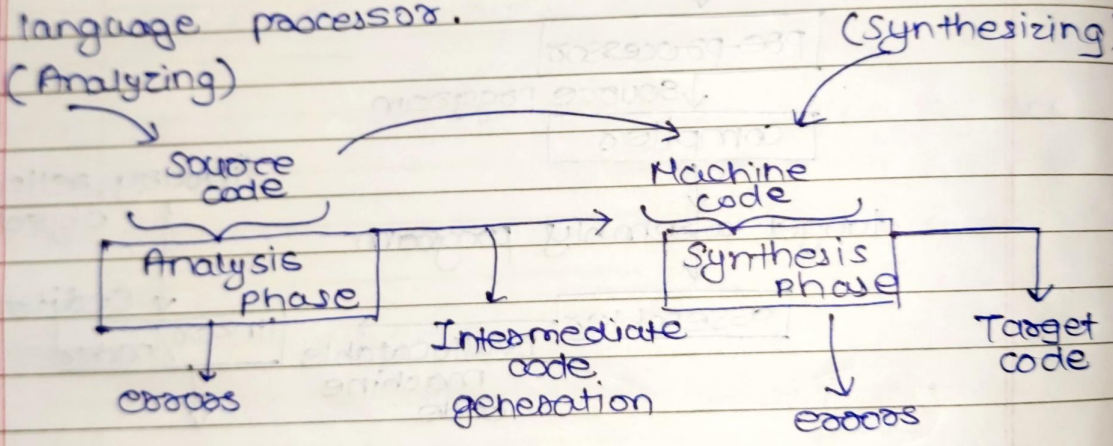
⇒ ① Assembler ② Compiler ③ interpretter ④ editor ⑤ loader ⑥ linker ⑦ debugger ⑧ macro ⑨ Operating System ⑩ Device Driver.

→ Assembler: used to translate assembly lang. prog in into machine code.

* Fundamental of Language Processing:

→ Language processing or: It converts source code into machine code

→ Language processing: The collection of language processor components engaging the analysis of source program constitutes analysis phase of language processor.



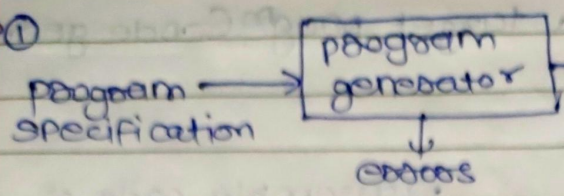
→ Language processing (~~attributes~~): activities:

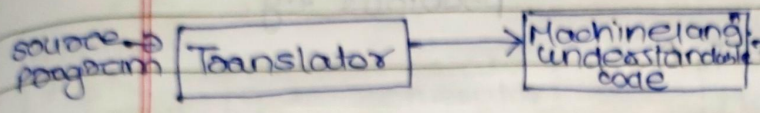
① Program generation activity.
② " execution activity. → ① program translation
② program interpretation

PAGE NO. DATE 06/12

SSD

→ ①





→ program Interpretation: Ex: .java → .class

steps: ① Fetch ② Decode ③ Execute.

⇒ Analysis phase: The specification consist of 3 components. ① lexical ~~groups~~ rules: which check the valid lexical units or tokens in source lang.

eg. percent_profit := (profit * 100) / cost_price;

↳ tokens.

→ ② Syntax Analysis: Which check the formation of valid syntax in the source language.

→ ③ Schemantic rules: which check, meaning with valid statements of the language.

⇒ Synthesis phase:

→ The synthesis phase is concerned with the construction of target language statements. which have the same meaning as a source statement.

→ It peroforms 2 main activities.

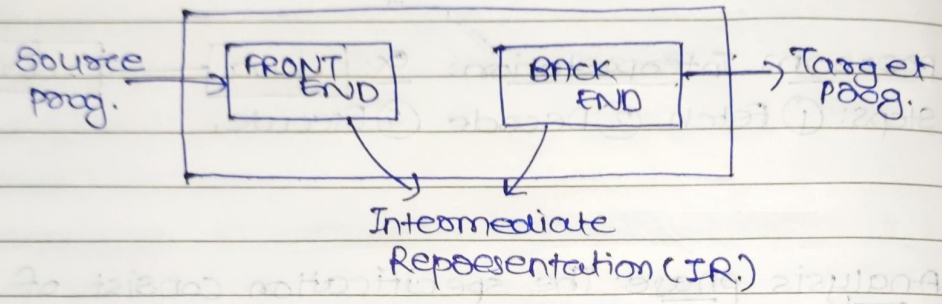
① creation of data structure in the target progm. (memory allocation).

⑧ generation of a target progm (Code generation)

eg. MOV A, PROFIT
MUL A, PROFIT, 100
DIV A, COST
MOV PERCENT_PROFIT, A

⇒ assembly code of previous eg.

⇒



* cousins of the compiler:

→ The i/p to a compiler will be produced by one or more pre-processors and further processing of the compiler's outcome may be needed before running machine code is obtained.

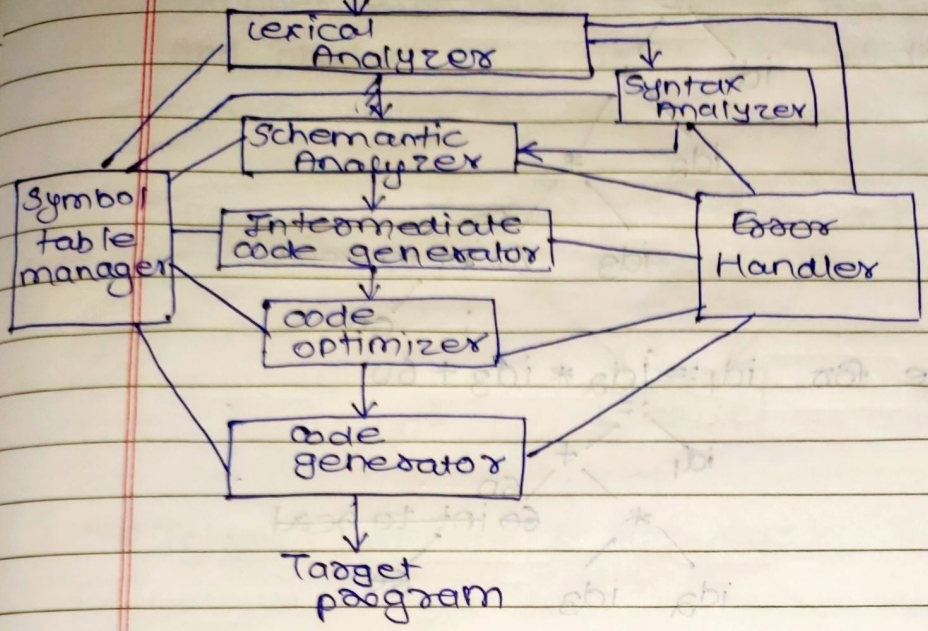
⇒ pre-processors → source code → target code
↓
IR.

① Macro-processing ② File inclusion ③ Rational preprocessor ④ language extensions.

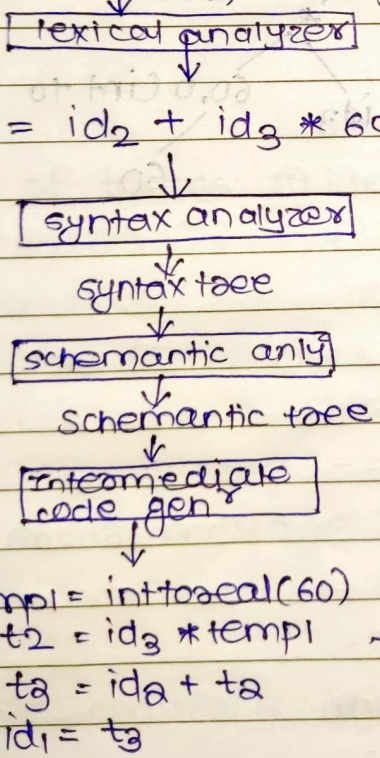
eg. equel lang
↓
(to include extensions).

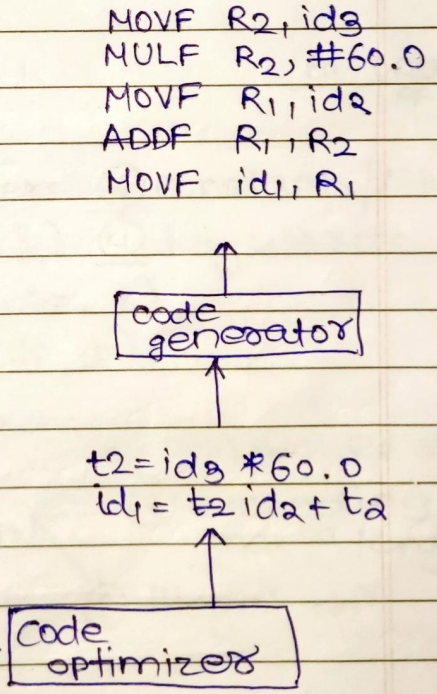
Rational → (do minor changes in older languages to make progm faster)

⇒ source-program

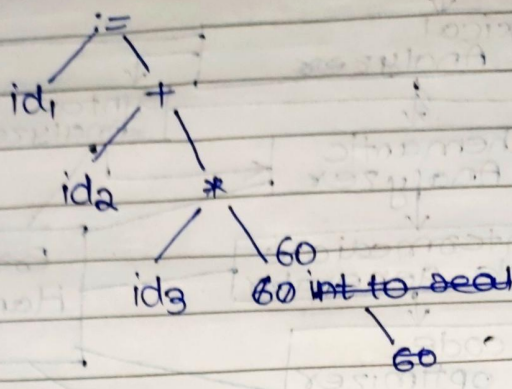


Eg: ① position := initial + rate * 60
idf op idf op idf op const
(id_1) (id_2) (id_3)

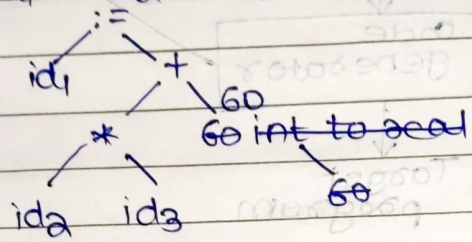




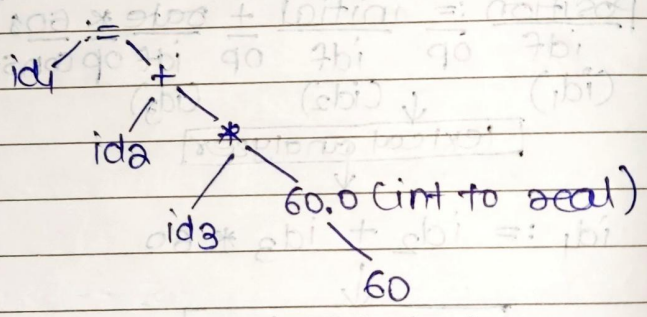
syntax tree: $id_1 := id_2 + id_3 * 60;$



Eg: syntax tree for $id_1 := id_2 * id_3 + 60$



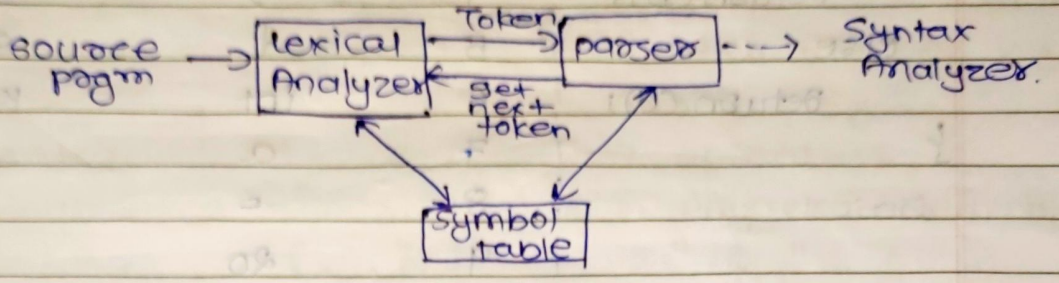
schemantic tree: $id_1 := id_2 + id_3 * 60.0$



CH-2

PAGE NO. DATE 7/12/23

→ The Role of the lexical analyzer: 1st phase of compiler. Main task is the to the read the i/p chrs and produce as a o/p, a seqn of tokens that the parser uses for syntax Analysis.



→ Tokens, patterns & Lexemes:

Eg: int a := 5 ;
lex1 lex2 lex3 lex4 lex5

main ( )
lex6 lex7 lex8

{
9

}
10

Eg: TRUE, FALSE

→ Types of tokens: ① identifiers ② constants/literals ③ seperators (,, ;, {, (,), }) ④ keywords ⑤ operators (+, -, :=, <=, >=, *). ⑥ special characters (&, #, $).

⇒ Gives error message when ① unmatched string ② exceeding maxm length ③ illegal chr

→ Eliminate comments and white space.

Eg =>

```
int main()
{
  int a=20,b=30;
  if(a<b)
    return(b);
  else
    return(a);
}
```

patterns → write patterns accordingly

no	lexeme	tokens
1	int	keyword
2	main	keyword
3	(	sepr
4	)	sepr
5	{	sepr
6	int	keyword
7	a	idf
8	=	opr
9	20	const.
10	,	sepr
11	b	idf
12	=	opr
13	30	const.
14	;	sepr
15	if	keyword
16	(	sepr
17	a	idf
18	<	opr
19	b	idf
20	)	sepr
21	return	keyword
22	(	sepr
23	b	idf
24	)	sepr
25	;	sepr
26	else	keyword
27	return	keyword
28	(	sepr
29	a	idf
30	)	sepr
31	;	"
32	}	"

PAGE No.
DATE 8/12/23

⇒

Token	Sample lexems	Informal Description of pattern
Constant	12, 15, ...	→ Any numeric number
Keyword	if, do, ...	→ keywords
Relational opr	<, >, <=, ...	→ compaaision symbols
identifier	a, b, ...	→ letter followed by letter / digit.
operator	+, −, *, /	→ operators
literals	"....."	→ Any character b/w " & ".
separators	(,), {, ;	→ , OR ; OR (OR {

#⇒ Lexical Errors: ① exceeding length of identifier OR numeric constant.
eg. int a = 9234567890;
② Appearance of illegal chrs. ⇒ eg. printf("Hi"); #
③ Unmatched string. → eg. main(){
/* .... comment not closed.
}

④ Spelling error: It occurs when identifier Rule not satisfied.
eg. int 7b = 134;
⑤ Replacing a chr with an incorrect chr.
eg. int x = 12#3;
⑥ Removal of chr that should be present.
eg. #includ <conio.h>
⑦ Transposition of 2 chrs.
eg. int mian().
⑧ Repeatation of chrs
eg. int a = 7..5;

Lexical error: when lexical analyzer is not able to identify valid lexemes / words which generate the lexical error.

→ RE → Input Buffer

↓

DFA → ? (error).

Eg: ① Int x, y; ⇒ Not generate lexical error (∵ Int is compile time error).

② int x,, y; →

↳ lexical errors.

③ ~~char~~ int A = #123; → Replacing a chr with incorrect chr.

④ String S = "6th sem; ⇒ Unmatched string.

Attributes for the token:

eg: token and associated attr value for the following statement;

E = M * C ** 2 ; // It is valid

↓

⇒ <id, pointer to symbol table entry for E>

<assignment_op, =>

<id, pointer to symb table entry for M>

<mult_op, *>

<id, pointer to symb table entry for C>

<~~mult_op, *~~> } <exp_op, **>

<~~mult_op, *~~>

<const, 2>

PAGE No. DATE 11/12/23

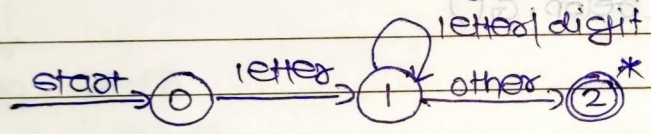
* Recognization of token by transition diagram:

1) Recognization of identifiers
2) Recog^{n} of delimeters
3) Recog^{n} of Relational op^{s}
4) Recog^{n} of keywords (such as 'if', 'else', 'for').
5) Recog^{n} of numbers (int, float, exp^{n}).

① Recog^{n} of identifiers:

→ letter → a|...|z|A|...|Z
digit → 0|1|...|9

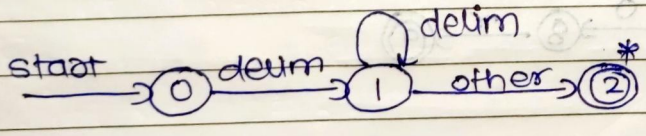
id → letter (letter | digit)* ⇒ Transition diagram:



return (gettoken(), install_id())

② Recog^{n} of delimeters:

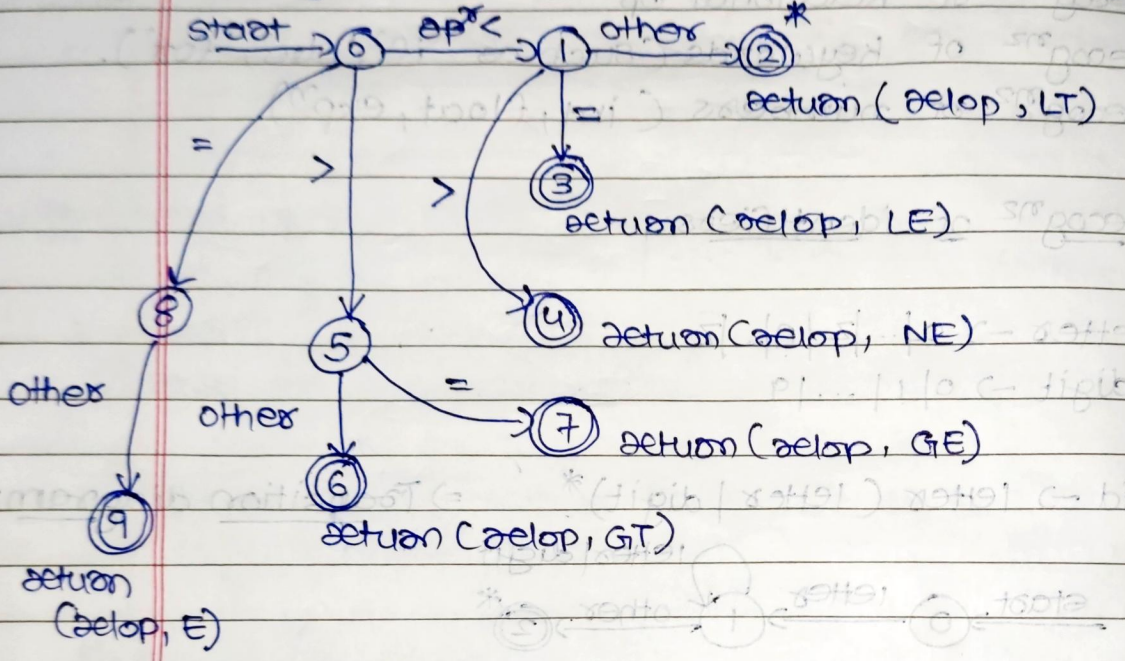
→ delim → blank | tab | newline
ws → delim (delim)* (; ws → whitespace).



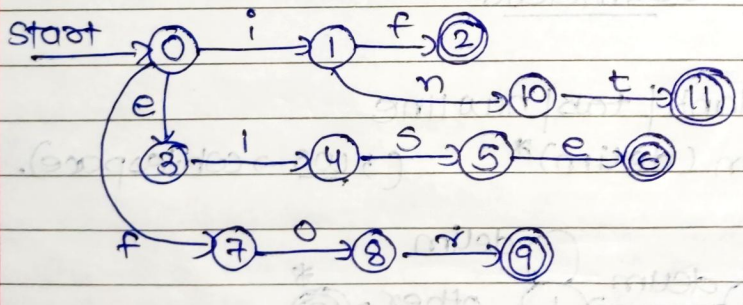
return(gettoken(), install_id()).

③ Recognition of Relational operator:

>| >=| <| <=| =| <> → NE
<> → not equal to.

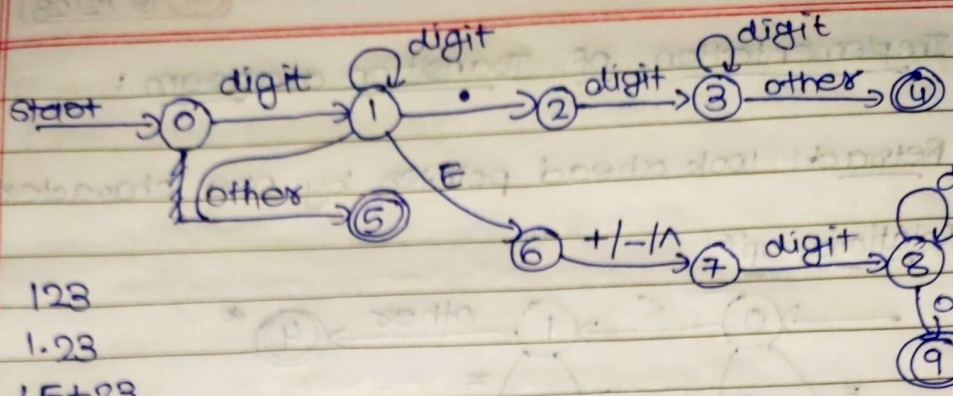


④ Recognition of Keywords:



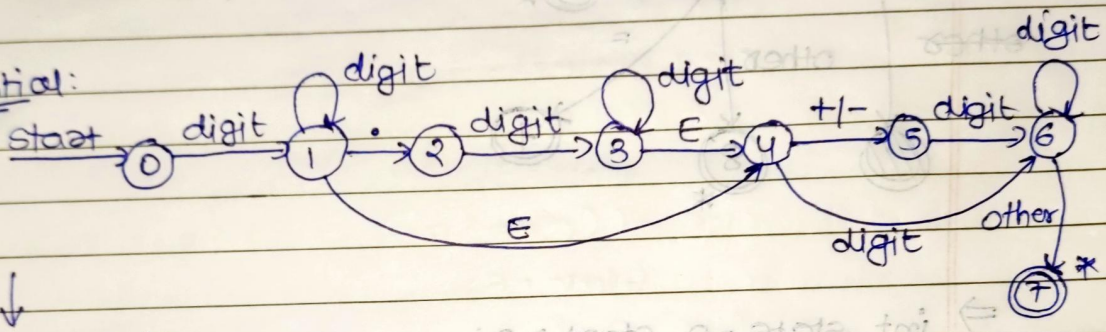
⑤ Recognition of numbers:

num → digit$^+$ (.digit$^+$)? (E (+|-)?digit$^+$)?



Eg: 123
1.23
1E+23
1E-23
1E23

exponential:
⇒



↓

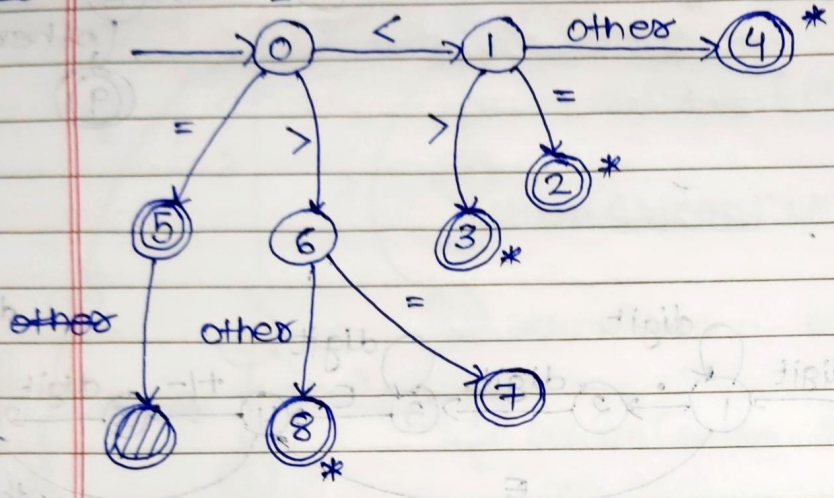
eg. 2E+5
2.7E-5
2.7E52
2E4

PAGE No.
DATE 18/12/23

* Implementation of Transition diagram :

⇒ Retract: look ahead pointer by one character.

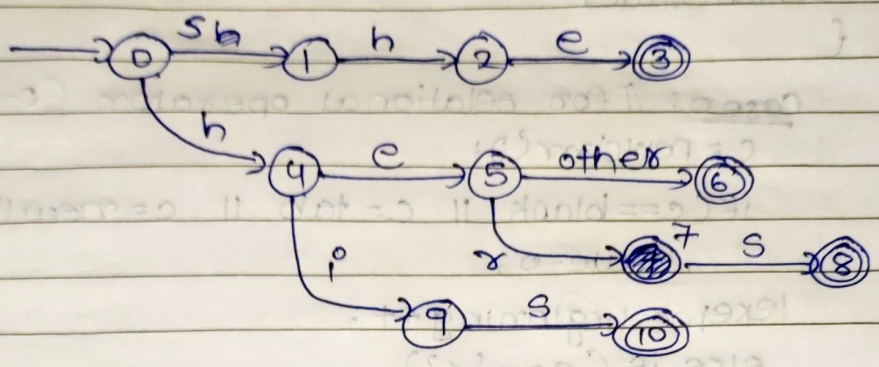
eg. Relational op^{n}:



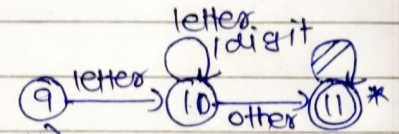
⇒
```
int state = 0, start = 0;
int lexical_value;
int fail(){
    forward = token_beginning;
    switch (start)
    {
        case 0:
```

PAGE No.
DATE 14/12/23

eg * Draw transition diagram for the set of keywords {he, she, his, hers}



eg. token nexttoken(){
while(1)
switch(state)
{
case 0: // for relational operator (0-8)
c = nextchar();
if(c==blank || c= tab || c=newline)
state=0;
lexeme_beginning++;
else if (c=='<')
state=1;
else if(c == '=')
state=5;
else if(c=='>')
state=6;
else fail();
break;
case 9: // identifier (9-11)



c = nextchar();
// if(c== blank || c=tabs || c=newline)
// state=0;
lexeme_beginning++;
else if(c==is letter)(c))
state=10;
else if (c==is letter (c) || c==is digit (c))
state=10;
else fail();
break;

```
case 10:
    c=nextchar();
    if(isletter(c)|| isdigit(c))
        state=10;
    else fail();
    break;
case 11:
    retract(1);
    install_id();
    return(gettoken());
```

* Implement transition diagram for exponential:

```
token nexttoken(){
    while(1)
    switch(state)
    {
        case 0:
            c=nextchar();
            if(isdigit(c))
                state=1;
            else fail();
            break;
        case 1:
            c=nextchar();
            if(isdigit(c))
                state=1;
            else if(c=='.')
                state=2;
            else if(c=='E')
                state=4;
            else fail(); break;
```

```
case 2:
    c=nextChar();
    if(isdigit(c))
        state=3;
    else fail();
    break;
case 3:
    c=nextchar();
    if(isdigit(c))
        state=3;
    else if(c=="E")
        state=4;
    else fail();
    break;
case 4:
    c=nextchar();
    if(c=="+" || c=="-")
        state=5;
    else if(isdigit(c))
        state=6;
    else fail();
    break;
case 5:
    .
    .
    .
case 8:
    retract(1);
    get install_id();
    return(gettoken());
```

14/12/23

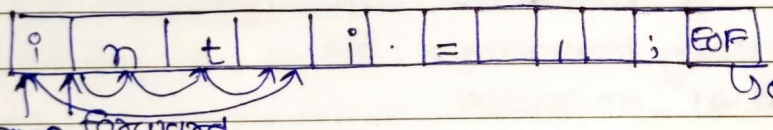
⇒ lex source pgm (lex.l) → lex compiler → lex.yy.c → C compiler → a.out

i/p stream → a.out → seqn of tokens.

⇒ declaration: eg.
```
%{
    int count=0;
%}
```

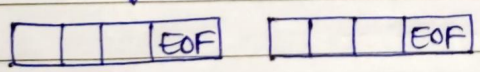
⇒ rules: eg.
```
%%
pattern.
%%
```

i/p buffering:

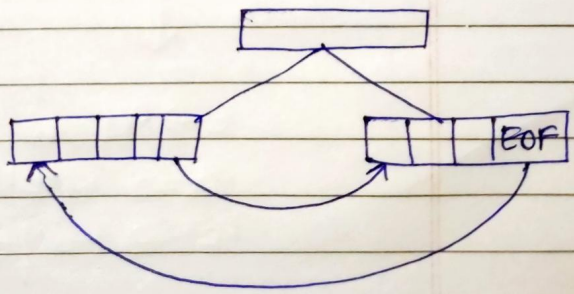


⇒ whenever forward pointer detects 'space' it will consider whole lexemes as one token and then forward pointer and lexeme_begin will move to next lexeme.

⇒ sentinal:



⇒ buffer pair:



→ specification of tokens

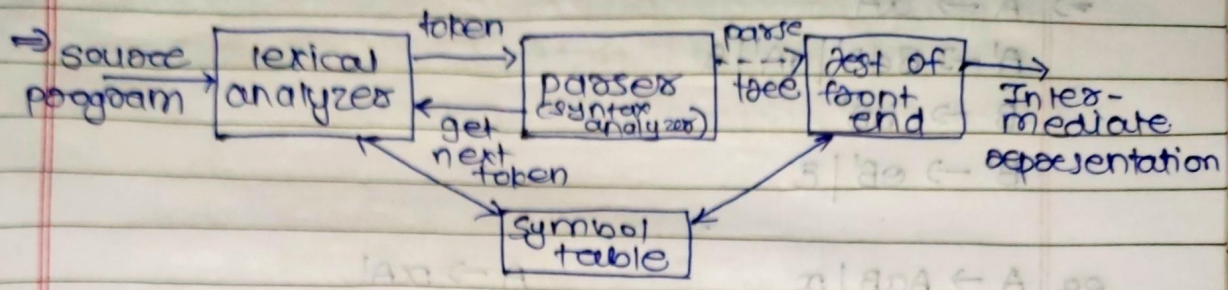
① prefix of 's' : a string obtained by removing 0 or more trailing symbols of a string.
eg. s = banana.
↳ prefix

→ suffix of 's' : a string obtained by removing 0 or more leading symbols of a string.
eg. s = banana
↳ suffix

② Substring : a string obtained by deleting a prefix and suffix from 's'.
eg. s = banana
↳ prefix ↓ substring ↳ suffix

Syntax Analyzer

DATE 14/12/23



15/12

⇒ Context free grammer:

→ (N, T, P, S) → starting symbol.
- N → set of non terminals
- T → set of terminals
- P → production Rules

eg ⇒ $(0+1)^*$

$S \rightarrow 0S \mid 1S \mid \wedge$

* Left recursion: i/p : Grammar 'g'

o/p : equivalent grammar with no recursion. ~~method~~.

method: if we have left recursive pair of production $[A \rightarrow A\alpha \mid \beta]$ where, β does not begin with 'A' then we can eliminate left recursion by replacing this pair of production with

$$\begin{pmatrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{pmatrix}$$

eg: $A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

→ eliminate the left recursion.

$\Rightarrow$ $A \to aA'$

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

$B \to bB'$

$B' \to eB' \mid \varepsilon$

eg. $A \to AaB \mid \alpha$

$B \to BCb \mid y$

$C \to Cc \mid \varepsilon$

$\Rightarrow$

$A \to \alpha A'$

$A' \to aBA' \mid \varepsilon$

$B \to yB'$

$B' \to CbB' \mid \varepsilon$

$C \to C'$

$C' \to cC' \mid \varepsilon.$

eg. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$\Rightarrow$

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

eg. $S \to Aa \mid b$

$A \to Ac \mid Sd \mid \varepsilon$

$\Rightarrow$ $\left.\begin{array}{l} S \to Aa \mid b \\ A \to SdA' \mid A' \\ A' \to cA' \mid \varepsilon \end{array}\right\}$ X

$S \to Aa \mid b$

$A \to Ac \mid Aad \mid bd \mid \varepsilon$

$\Downarrow$

$S \to Aa \mid b$

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

eg. $S \to (L) | a \Rightarrow S \to (L) | a$
$L \to L,S | S \quad L \to L,(L) | L,a | (L) | a$

Ans. $\Rightarrow S \to (L) | a$
$L \to aL' | (L)L'$
$L' \to ,(L)L' | ,aL' | \cancel{(L)L'} | \varepsilon$

* Error Recovery strategies:

→ ① Panic mode recovery:
eg. int oc, ~~x2y~~, ~~#z~~;
→ it will identify by taking/checking one token at a time, whenever it detects syntax error it will delete each token/lexeme until it finds separator like ',', ';' etc.
⇒ whenever there are minimum (small) no. of errors in the statement then only it is used.

② Phrase level recovery:
int a, b,
int a=5;
↓
(;) replaced by semi-colon.
because ';' is expected.

③ Error productions: eg. $S \to X$
$X \to aX | bX | \varepsilon$
$Y \to cd.$
$\Rightarrow abcd.$

A

④ global corrections: We can correct grammar, sta anything to correct errors. It is not possible in Real-time. Physically not possible.

* Elimination of Left Factoring:

→ i/p: Grammer 'G'
o/p: equivalent non left factoring grammar.
method: for each NT 'A' find the prefix 'α', comm two or more of its alternatives. Replace one of th 'A' productions.

eg. $A \to \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$ (left factoring grammar).

; γ → represents all alternatives that do not begin with 'α'.

o/p: $A \to \alpha A' | \gamma$

$A' \to \beta_1 | \beta_2 | \ldots | \beta_n$

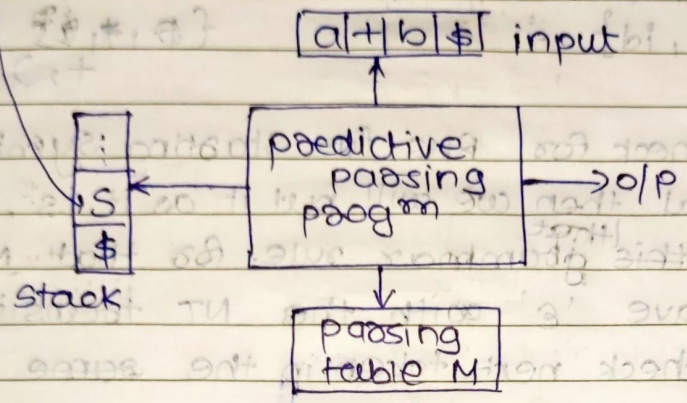
→ Repeatedly, apply their transformation until tha two alternatives for a common prefix.

LT

PAGE No. DATE 22/12/23

* Non-recursive predictive parser / LL(1) parser / predictive parser / table-driven predictive parser:

⇒ S → a+b



⇒ for LL(1) → it uses only one i/p symbol at a time to predict the parsing process.
- the i/p is scanned from left to right
- it uses left most derivation for i/p string.

→ data structure uses ① stack, ② parsing table ③ i/p buffer.

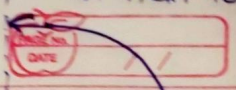
⇒ stack: initially the stack contains the start symbol of the grammar on top of '$' symbol.

⇒ parsing table: parser construct the table each time by taking the parsing actions.

⇒ i/p buffer: it is used to store the i/p tokens.

Eg: * E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Rules: ① put '$' in starting.
② find starting symbol in every transitions. If the next token is terminal then put as it is.
③ If there is nothing in next token



Starting Symbol	First	Follow
E	→ {(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{$, +, ε}
T'	{*, ε}	{$, +, ε}
F	{(, id}	{$, *, ε, +,)}

Rules: ① We will check for RHS for starting symbol, if it is Terminal then we will put it as it is. If NT then Go to that grammar rule for that NT.
② if we have 'ε' with the NT transition then we have check next token in the production rules for that NT.

Eg: S → AB
A → aA | ε
B → BaAC|ε B → CB' B' → aACB' | ε
C → b|ε

Starting Symbol	First	Follow
S	{ε, b, a, c}	{$}
A	{ε, b, a}	{$, a, c}
B	{ε, c}	{$}
C	{ε, b}	{a}
B'	{ε, a}	{$}

Eg: S → AcB | CbB | Ba
A → da | BC
B → g | ε
C → h | ε

start symb.	first	follow
S	{d, g, e, h, c, a}	{$}
A	{d, g, e, h}	{h, c}
B	{ε, g}	{$, a, h, c}
C	{ε, h}	{h, g, f} {c}

eg: S → (L) | a
L → SL'
L' → ,SL' | ε

start symb	first	follow
S	{(, a}	{$, , ,)}
L	{(, a}	{),)}
L'	{, , ε}	{),)}

eg: S → aBDh
B → cC
C → bc | ε
D → EF
E → g | ε
F → f | ε

start symb	first	follow
S	{a}	{$}
B	{c}	{h, g, f}
C	{b, ε}	{h, g, f}
D	{g, ε, f}	{h, h}
E	{g, ε}	{h, f, h}
F	{f, ε}	{$, h}

Eg. $S \to ABC$ $B \to c|d|\varepsilon$

$A \to a|b|\varepsilon$ $C \to e|f|\varepsilon$

	first	follow
S	{a, ε, b, c, d, e, f}	{$}
A	{a, b, ε}	{$, c, d, e, f}
B	{c, d, ε}	{$, e, f}
C	{e, f, ε}	{$}

Rules for follow:

Do not write 'ε' in follow set of any symbol.

① $ at starting of ~~every~~ set first

② Find starting symbol in all RHS production Rules.

③ check for the next token in that rule.

④ If terminal then put as it is. in set.

⑤ If there is nothing then check for starting symbol of that production Rule and put that rule's follow in this follow.

⑥ If NT then write the first of that token into follow of this symbol.

if there is 'ε' in first then see the next token and follow all the rules. again.

PAGE No. DATE 27/12

=) steps of LL(1) parser.

① remove left recursion / left factoring.
② find first and follow of the grammer.
③ construct parse table.
④ stack implementation.
⑤ parse tree generation.

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

① remove left recursion →

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

no left recursion
factoring

string: id + id * id

② finding first and follow.

	first	follow
E	{*, ε, (, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ε}	{$, +,)}
F	{(, id}	{*, $, +,)}.

③ construct parse table. (do not put entry for 'ε' in parse table).

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

① See the first of each symbol and write production rule in that.
② If there is two production rule (more than one) then write 1st production rule in that symbol and make entry of null in 'follow' symbols.

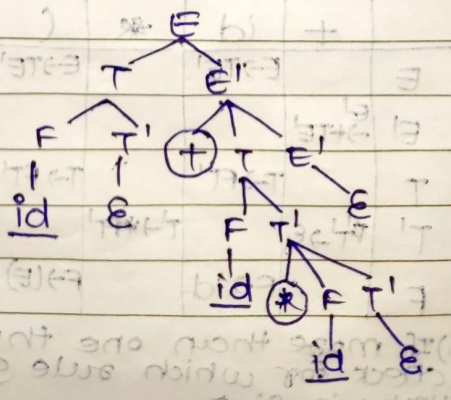
=) If more than one then check for which rule generate symbol in first.

④ Stack implementation:

Note: Here All terminals produce single production rule, so it is a LL(1) Parser.

stack	input	action
$E	id + id * id $	E→TE'
$E'T	id + id * id $	T→FT'
$E'T'F	id + id * id $	F→id
$E'T'id	id + id * id $	Matched
$E'T'	+ id * id $	T'→ε
$E'	+ id * id $	E'→+TE'
$E'T+	+ id * id $	Matched
$E'T	id * id $	T→FT'
$E'T'F	id * id $	F→id
$E'T'id	id * id $	Matched
$E'T'	* id $	T'→*FT'
$E'T'F*	* id $	Matched
$E'T'F	id $	F→id
$E'T'id	id $	Matched
$E'T'	$	T'→ε
$E'	$	E'→ε
$	$	Matched / Accepted.

⑤ Parse tree generation:



Eg. $S \to (L) \mid a$
$L \to SL'$
$L' \to ,SL' \mid \varepsilon$

already ① removed left recursion / factoring.

string: (a,a)

② already found First and Follow.

③ Parse table:

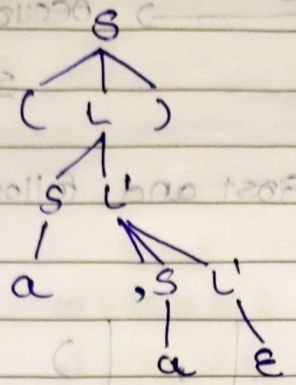
	(	a	,	)	$
S	$S \to (L)$	$S \to a$			
L	$L \to SL'$	$L \to SL'$			
L'			$L' \to ,SL'$	$L' \to \varepsilon$	

④ Stack implementation:

Stack	i/p	action
$S	(a,a)$	$S \to (L)
$)L(	(a,a)$	Matched.
$)L	a,a)$	$L \to SL'
$)L'S	a,a)$	$S \to a
$)L'a	a,a)$	Matched
$)L'	,a)$	$L' \to ,SL'
$)L'S,	,a)$	Matched
$)L'S	a)$	$S \to a
$)L'a	a)$	Matched
$)L'	)$	$L' \to \varepsilon
$)	)$	Matched
$	.$	Accepted.

PAGE NO. DATE 28/12/23

⑤ Parse tree generation.



Eg: S → iEtSeS | iEtS | a (capital (S)) Q. [is it LL(1) parser?]
E → b

↓ removing left factoring.

① S → iEtSS' | a
S' → eS | ε
E → b

② First and Follow.

	First	Follow
S	{i, a}	{e, $}
S'	{e, ε}	{e, $}
E	{b}	{t}

③ Parse table.

	i	a	e	$	b	t
S	S→iEtSS'	S→a				
S'			S'→eS S'→ε	S'→ε		
E					E→b	

so it is not LL(1) parser.

④ Stack implementation:

Eg. $S \to ACB \mid CbB \mid Ba$

$A \to da \mid BC$ → capital.

$B \to g \mid \varepsilon$

$C \to h \mid \varepsilon$

⇒ ① Parse table:

not a LL(1) parser.

	a	d	g	h	c	$
S	S→Ba	S→ACB	S→ACB	S→ACB	S→CbB S→ACB	
A		A→da	A→BC	A→BC		
B	B→ε		B→g	B→ε	B→ε	B→ε
C				C→h	C→ε	

Eg. $S \to S'\#$

$S' \to AB$

$A \to aA'$

$A' \to a \mid \varepsilon$

$B \to b \mid ac$

String: aac#

Q. Does it parse a string?

⇒ First and Follow.

	First	Follow
S	{a, b, a}	{$}
S'	{a}	{#}
A	{a}	{b, a}
A'	{a, ε}	{b, a}
B	{b, a}	{#}

⇒ parse table:

	a	b	$	#
S	S→S'#			
S'	S'→AB			
A	A→aA'			
A'	A'→a / A'→ε	A'→ε		
B	B→ac	B→b		

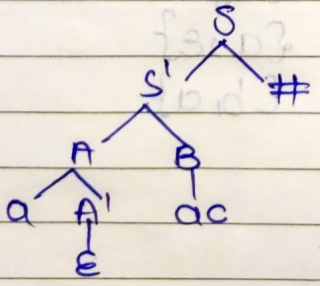
not a LL(1) parser.

⇒ stack implementation:

stack	i/p	action
$S	aac#$	S→S'#
$#S'	aac#$	S'→AB
$#BA	aac#$	A→aA'
$#BA'a	aac#$	Matched
$#BA'	ac#$	A'→ε
$#B	ac#$	B→ac
$#ca	ac#$	Matched
$#c	c#$	Matched
$#	#$	Matched
$	$	Accepted.

But we can't accept string b/c there are two distinct ways to generate a parse string.

⇒ parse tree:



Eg: D → int L;
L → id E
E → E [num] | [num]

String: int a[2];

⇒ ① D → int L;
L → id E
E → [num] E'
E' → [num] E' / ε

② first and follow.

	first	follow
D	{int}	{$}
L	{id}	{;}
E	{[num]}	{;}
E'	{[num], ε}	{;}

③ parse table.

	int	id	[num]	;	$
D	D→int L;				
L		L→id E			
E			E→[num] E'		
E'			E'→[num]E'	E'→ε	

⇒ It is LL(1) parser.

④ Stack implementation:

Stack	i/p	action
$D	int a[2];$	D→ int L;
$;L int	int a[2];$	Matched.

$; L	a[2];$	L → id E
$; E id	a[2];$	Matched
$; E	[2];$	E → [num] E'
$; E' [num]	[num];$	Matched
$; E'	;$	E' → ε
$;	;$	Matched
$	$	Accepted.

⑤ parse tree: