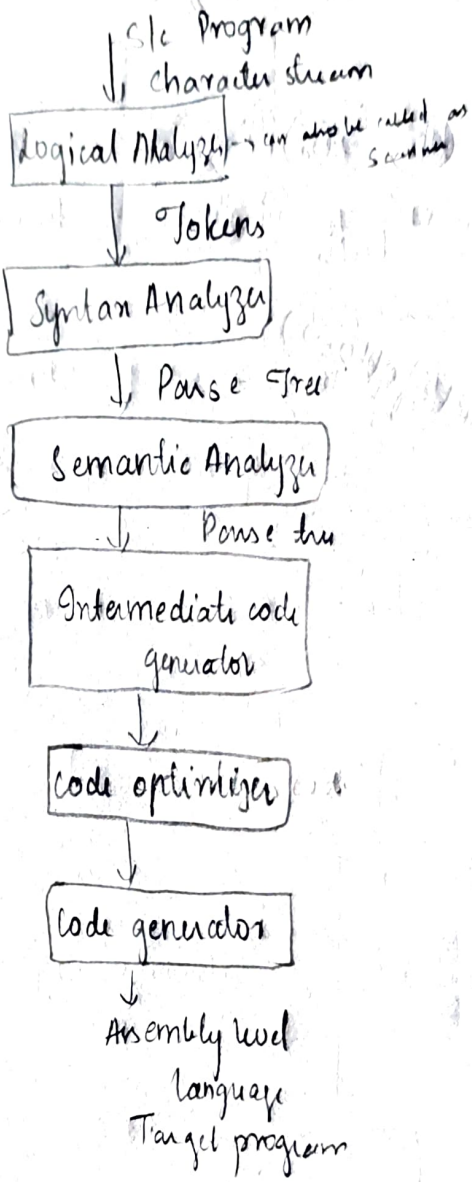
Phases of Compiler:

1) Analysis phase
2) Synthesis phase



Position = initial + rate * 60

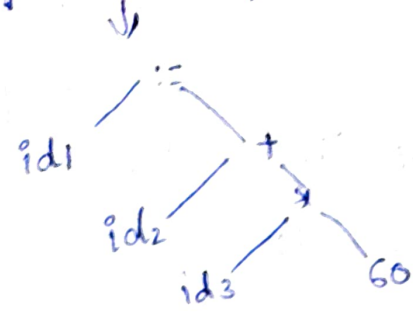
P = i + r * 60

$id_1 := id_2 + id_3 * 60$

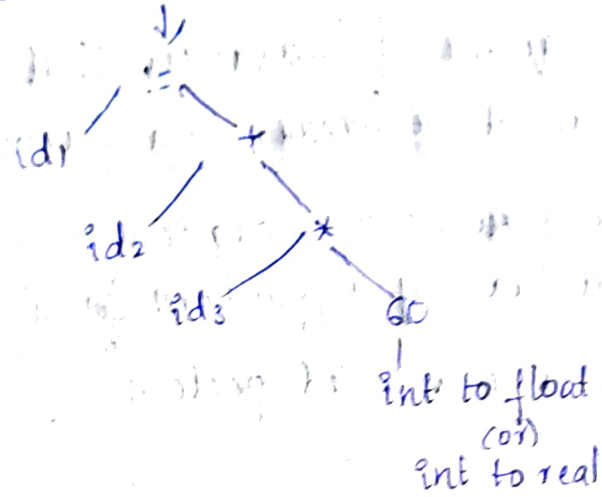
↓ lexical analyzer

↓ id1, :=, id2, +, id3, *, 60

Syntax analyzer



Semantic analyzer:



t_1 = inttofloat(60)

t_2 = id_3 * t_1

t_3 = id_2 + t_2

id_1 = t_3

↓

code optimizer

t_1 = id_3 * 60.0

id_1 = id_2 + t_1

or

t_1 = id_3 * 60.0

t_2 = id_2 + t_1

id_1 = t_2

Code generator

↓

MOVF id3, R2 // R2 = id3

MULF #600, R2 // R2 = R2 * 600

movf id3 R1 // R1 = id2

addf R1 id1 // id = R1

Token is a sequence of characters that are treated as unit as it cannot be break down further

Lexeme is a sequence of characters in source program that is matched by a pattern for a token.

Describing a rule is called pattern

Token	Lexeme	Informal description of pattern
const	const	const
if	if	if
relation	<, >, =, <=, >=	<, >=, <=, >
id	pi, count, D5	Letter followed by a letter digit
num	3.14, 15	any numeric constant

<id1, point to the symbol table for c>
<assign-operator>
<id2, point to symbol table for value m>
<mul-op>
<id3, point to the symbol table for value of c>
<exp-op>
<id3, point <num>

Term	Definition
1. prefix	a string obtained by removing zero or more trading symbols of string ex: abc {ϵ, a, ab, abc}
2. sufix	a string form by deleting zero or more leading symbols of string ex: abc {abc, bc, c, ϵ}
3. substring of s	a string obtained by deleting a prefix & suffix from given string ex: abc {a, b, c, ab, bc, abc, ϵ}

Note: LUD
↓ ↓
Letters digits

1) LUD is the set of letters and digits
2) LD is the set of string consisting of a letter followed by a digit.
3) L^4, L is the set of 4 letter strings
4) L^* is the set of all string of letters including ϵ

5) $L(L \cup D)^*$ is the set of all strings of letters & digits beginning with letter

6) D^+ is the set of all strings of one or more digits

Recognition of tokens :

consider the following grammar

stmt → if exp then stmt - |
if exp then stmt else stmt | ε

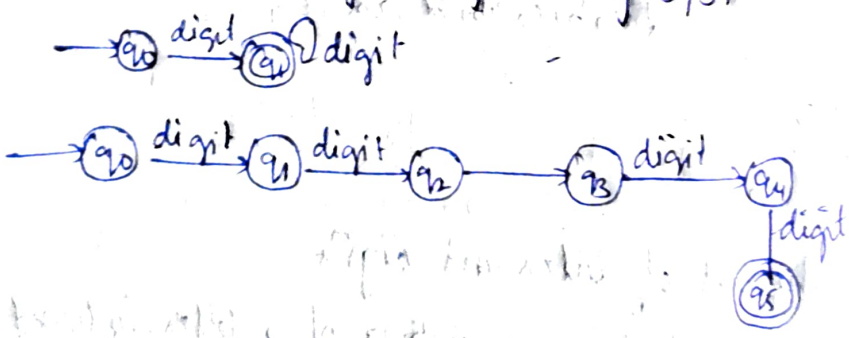
expr → term relop term | term

term → id | num

In the above grammar the tokens are if, then, else, relop, id, num, digit

if → if
then → then
else → else
relop → <, >, <=, >=, <>
id → letter (letter | digit)*
num → digit+ (. digit+)?
digit → 0 - 9

Draw transition diagram for 5280 & 3937



22/9/23

Lexical Analysis:

Lexical analyzer eliminates the byte spaces, blank spaces, tab spaces and new line characters in the source program. It also identifies the keywords, identifiers, constants & punctuation marks.

Functions of lexical analyzer:

1) Read i/p characters from source program.
2) Dividing the program into valid tokens.
3) Removes byte spau characters
4) Remove comments.
5) Helps to identify tokens into the symbol table.
6) It generates lexical errors.

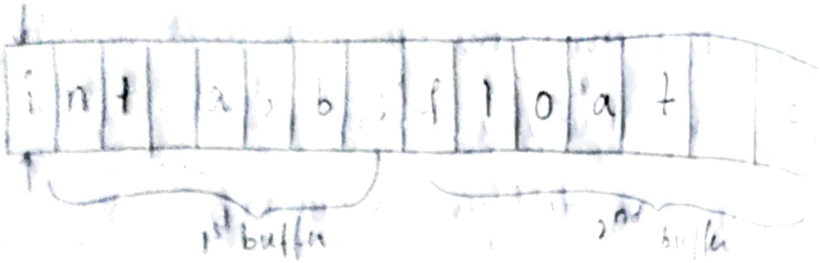
Lexical error: A character sequence which is not possible to scan any valid token is a lexical error.

Ex: fi (a<b)

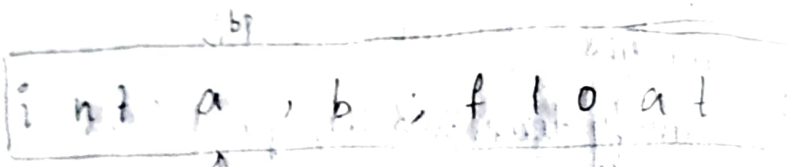
Lexical errors can be

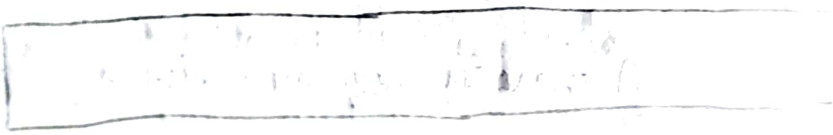
1) spelling error
2) Exceeding length of the error identifier
3) appearence of illegal character.

Algorithm for lexical Analyzer



Begin pointer and forward pointer will be point to the 1st character of lexeme





if (forward at end of the 1st half) then

begin

reload 2nd half;

forward = forward + 1;

else

if (forward at 2nd half) then

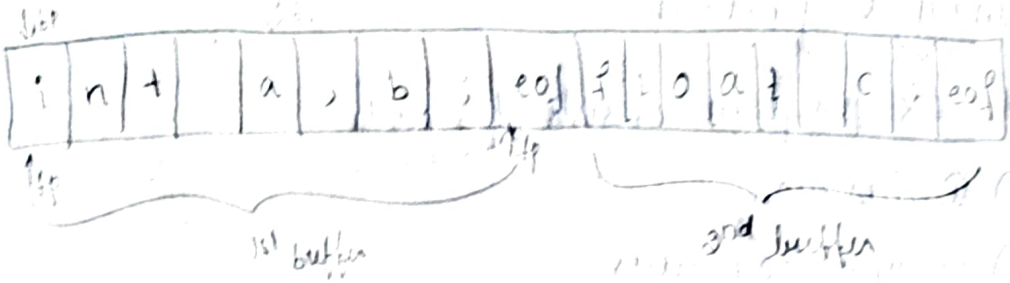
begin

reload 1st half;

move forward pointer to beginning of the 1st half

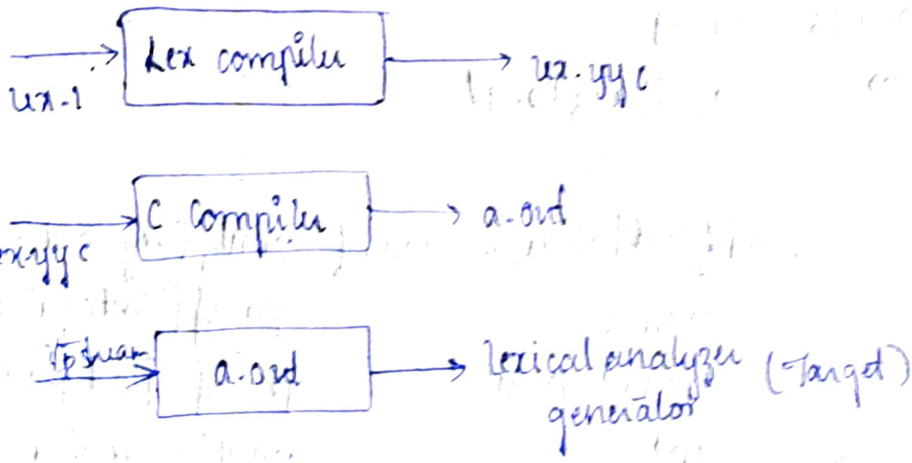
end

Sentinels:




```
if (forward == eof) then
  begin
  if (forward at end of 1st half) then
      begin
        reload 2nd half.
        forward = forward + 1
  else
    if (forward at end of the 2nd half) then
        begin
          reload 1st half
          move forward to begin of 1st half
        end;
  else
  terminate the lexical analysis
```

LEX (Lexical analyzer Generator)



LEX is a tool which is used to generate the lexical analyzer generator

A LEX program consists of 3 parts

1) Declaration
2) Translation rules
3) Auxiliary Procedures

- In declaration declare the variables in the LEX program
- Translation rules

```
%    %
P1   {action1}
P2   {action2}
Pn   {actionn}
%    %
```

where each Pn is a regular expression and each action is a program fragment describing what actions can be performed.

- Auxillary procedures

These procedures can be compiled separately & loaded with lexical analyzer

```
DIGITS   [0-9]
ID       [a-z][a-z|0-9]
         %    %
int|float|char|boolean|if    printf("keyword", yytext)
  DIGITS+                    printf("integer", yytext)
  DIGIT+ DIGIT+              printf("float-point", yytext)
  ID+                        printf("identifier", yytext)
```

```
(          printf("open paranthesis", yytext);
<|>|<=|>=|<>   printf("relop", yytext);
)          printf("close paranthesis", yytext);
,|;|"      printf("punctuation marks", yytext);

%%
main()     yywrap()
{          {
  yylex();   return 1;
}          }
```

yytext[0] will hold the 1st char of the lex

yylex() fun makes a call to the yywrap() fun when it encounters the end of the ip. Otherwise it will the prg continously.

```
gedit filename.l
lex filename.l
gcc lex.yy.c
```

```
int keyword
a identifier
, punctuation
b identifier
; punctuation
```

Algorithm : Eliminating LR

Input : Grammer G with no cycles or 'ε' productions

Output : An equivalant grammer with no left recursion

method : Apply the algorithm to grammer G then the resulting non left recursive grammer may have 'ε' productions

i) Arrange the non terminals in order A1 ... An

ii) for (each i from 1 to n) {

iii) {for(each j from 1 to i-1) {

iv) Replace each production of the form $A_i \to A_j \gamma$ by the productions.

$A_i \to \delta_1\gamma | \delta_2\gamma | \ldots | \delta_k\gamma$ where

$A_j \to \delta_1 | \delta_2 | \ldots | \delta_k$ are all current A_j productions

}

v) Eliminate the immidiate LR among the Ai productions

}

Q) $S \to a | \wedge | (T)$

$T \to T, S | S$

Q) $E \to E(T) | T$

$T \to T(F) | F$

$F \to id$

Q) $S \to ABc$

$A \to Aa | Ad | b$

$B \to Bb | e$

$C \to Cc | g$

Q) $A \to ABd | Aa | a$

$B \to Be | b$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow T,S \mid S$ → given grammar is in left

→ remove it

$T \rightarrow ST'$

$T' \rightarrow ,ST' \mid \epsilon$

2) $E \rightarrow E(T) \mid T$ → $E \rightarrow (T)E'$, $E' \rightarrow$

$T \rightarrow T(T) \mid$ $T \rightarrow (B)T'$, $T' \rightarrow ($

F and → is not left recursion

3) $S \rightarrow ABC$ - not in left recursion

$A \rightarrow Aa \mid Ad \mid b$

$B \rightarrow Bb \mid e$ $B \rightarrow BB'$, $B' \rightarrow bB' \mid \epsilon$

$C \rightarrow Ce \mid g$ $C \rightarrow gC'$ $C' \rightarrow eC' \mid \epsilon$

$A \rightarrow Aa \mid Ad \mid b$

$A \rightarrow bA'$

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

4) $A \rightarrow AB1 \mid Aa \mid a$, it is in left rec

$A \rightarrow aA'$

$A' \rightarrow BdA' \mid aA' \mid \epsilon$

$B \rightarrow Be \mid b$

$A \rightarrow bB'$

$A' \rightarrow eB' \mid G$

5) $S \rightarrow \underline{a}\,\underline{S}\,\underline{S}\,\underline{b}\,\underline{S} \mid \underline{a}\,\underline{S}\,\underline{a}\,\underline{S}\,\underline{b} \mid abb \mid b$

(α: a, β: SSbS, α: a, γ: Sa, β: Sb)

not a left recursion it is left factoring

$S \rightarrow aS'$

$S' \rightarrow Sbs \mid asb$

Top Down Parsing

The process of constructing the parse tree which starts from the root & goes down to the leaf is top down parsing

Top down parsers constructs from the grammer which is free from ambiguity & free from left recursion

Top down parsers uses LMD to construct a parse tree

Classification of TDP

- with backtracking
  1) Brutiforce technique
- without Backtracking
  1) RDP
  2) PP
  3) Non recursive

$E \rightarrow TE'$

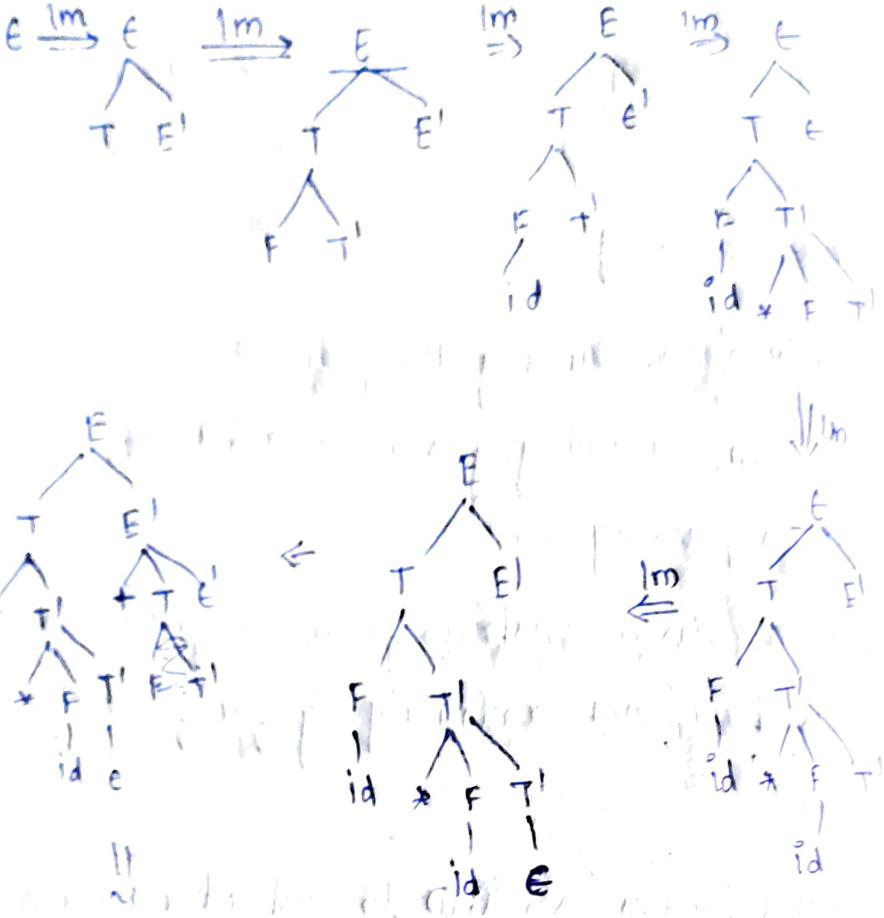
$E' \rightarrow +TE' / E$

$T \rightarrow FT'$

$T' \rightarrow *FT' / E$

$F \rightarrow (E) / id$

id * id + id



RDB With Backtracking (Brute Force Technique)

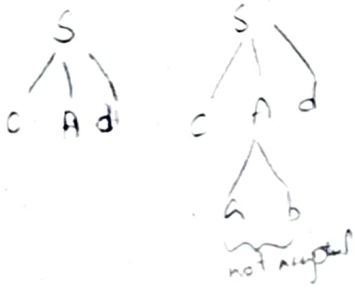
RDB (Recursive Descendent Backtracking) involves backtracking that is if a choice of a production rule doesn't work we backtrack to try other alternatives

In general RDB is not efficient (slower)

```
void A()
{
    choose an A-production:
        A → X1, X2 . . . Xk ;
        for (i=1 to k)
        {
            if (Xi is a non terminal)
                call procedure Xi()
            else
                if (Xi equals the current i/p symbol a)
                    advance the i/p to the next symbol
                else
                    an error has occured;
        }
}
```

S → CAD

A → abla



RDP without backtracking

Consider the given grammer

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Sol Eliminate the left recursion from the given grammar

$E \rightarrow E+T/T$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

$E \rightarrow TE'$

$T \rightarrow FT'$

$FT'E'$

$F \rightarrow id$

$idTE'$

$T' \rightarrow *FT'$

$id*FT'E'$

$F \rightarrow id$

$id*idT'E'$

$T' \rightarrow \epsilon$

$id*idE'$

$E' \rightarrow +TE'$

$id*id+TE'$

$T \rightarrow FT'$

$id*id+FT'E'$

$F \rightarrow (E)$

$id*id+idT'E'$

$T' \rightarrow \epsilon$

$id*id+idE'$

$E' \rightarrow \epsilon$

$id*id+id\epsilon$

FIRST

To compute FIRST(x) for all grammer symbols x apply the following rules until no more terminals or ϵ can be added to any first set

i) If x is a terminal FIRST(x) = {x}

ii) If x is a non terminal i.e $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place 'a' in FIRST(x) if for some 'i', 'a' is in FIRST(Y_i) & ϵ is in all of FIRST(Y_1) . . . . FIRST(Y_{i-1})

If ϵ is in FIRST(Y_j) (for all) $\forall$ $j = 1, 2 \ldots k$ then add ϵ to FIRST(X)

iii) If $X \to \epsilon$ is a production then add ϵ to FIRST(X)

Ex: Compute FIRST for the given grammer

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ϵ}

FIRST(T) = {(, d}

FIRST(T') = {*, ϵ}

FIRST(F) = {(, id}

2) S → ABCDE

A → a|ε

B → b|ε

C → c

D → d|ε

E → e|ε

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

FIRST(A) = {a, ε}

FIRST(B) ⇒ {b, ε}

FIRST(C) = {c}

FIRST(D) = {d, ε}

FIRST(E) = {e, ε}

5|10|25

3) Compute FIRST for the following grammar.

S → ACB | CbB | Ba

A → da | Bc

B → g | ε

C → h | ε

A → da

FIRST(A) = {d}

A → BC (non-terminal)

FIRST(A) FIRST(S)

= {g, ε}

A → B C ⇒ A →

FIRST

FIRST(A) = {d, g, h, ε}

FIRST(B) = {g, ε}

FIRST(C) = {h, ε}

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

4) Compute FIRST for following grammar

S → iEtSS' | a

S' → eS | ε

E → b

FIRST(E) ⇒ {b}

FIRST(S) → {i, a}

FIRST(S') → {e, ε}

★ FOLLOW

To compute FOLLOW(A) for all non terminals A apply the following rules until nothing can be added to any follow set.

Rule 1: Place \$ in FOLLOW(S) where S is the start symbol & \$ is the input right end marker

FOLLOW(S) = \$

Rule 2: If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ϵ is in FOLLOW(B). FOLLOW(B) = FIRST(β) − ϵ

Rule 3: If there is a production $A \rightarrow \alpha B$ (or) $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B)

FOLLOW(B) = FIRST(β) − ϵ ∪ FOLLOW(A)

Compute FOLLOW for the given grammer.

$E \rightarrow TE'$ FIRST(E) = {(, id}

$E' \rightarrow +TE' / \epsilon$ FIRST(E') = {+, ϵ}

$T \rightarrow FT'$ FIRST(T) = {(, id}

$T' \rightarrow *FT'$ FIRST(T') = {*, ϵ}

$F \rightarrow (E) / id$ FIRST(F) = {(, id}

$F \rightarrow (E)$ (A − B, α B β)

Follow(E) = first( )) − ϵ
=) − ϵ
=)

Follow(E')

$E \rightarrow TE'\epsilon$, $E' \rightarrow +TE'$ (A, α B, β)

Follow(E') = {ϵ − ϵ} ∪ Follow(E)
= {\$ −)}

Follow(E') for $E' \rightarrow +TE'\epsilon$ (A, α, B, β)

Follow(E') = {ϵ − ϵ} ∪ Follow(E)
= {\$,)}

FOLLOW(E) = {\$,)}

FOLLOW(E') = {\$,)}

FOLLOW(T) : {+, \$,)}

FOLLOW(T') :

FOLLOW(F) :

FOLLOW(T) = $E \xrightarrow{A\ \alpha\ B} TE'$, $E' \xrightarrow{\alpha\ B} +TE'$,

FOLLOW(T) = { FIRST(E') - ε } ∪ FOLLOW(E)

{ +, ε - ε } ∪ { $,) }

{ +, $,) }

FOLLOW(T) = { +, ε ∈ ε } ∪ { $,) }

{ +, $,) }

FOLLOW(T') = T → FT' , T' → *FT'

Predictive Parsing:

Algorithm: Construction of a predictive parsing table

Input: Grammar G

Output: Parsing table M

Method: For each production $A \to \alpha$ of the Grammar do the following

Step 1: For each terminal 'a' in FIRST(α), add $A \to \alpha$ to M[A, a]

Step 2: If ϵ is in FIRST(α) then for each terminal 'b' in FOLLOW(A) add $A \to \alpha$ to M[A, b]

Step 3: If ϵ is in FIRST(α) & \$ is in FOLLOW(A) add $A \to \alpha$ to M[A, \$]

After performing the above 3 step there is no production all in M[A, a] then set M[A, a] to error

Ex: Construct a predictive parsing table for the grammar

$S \to aBa$

$B \to bB \mid \epsilon$

FIRST(S) $\to \{a\}$ FOLLOW(S) = {\$};

FIRST(B) $\to \{b, \epsilon\}$ FOLLOW(B) = {a}

$S \to aBa$

$B \to bB$

Follow(B) = {a}

	First(α)	PPE
$S \to aBa$	FIRST(aBa) = {a}	M[S, a]
$B \to bB$	FIRST(bB) = {b}	M[B, b]
$B \to \epsilon$	Follow(B) = {a}	M[B, a]

Nonterminal	a	b	$
S	S → aBa		
B	B → ϵ	B → bB	

Construct PPT for gramma

E → TE' ϵ

E' → +TE' / ϵ

T → FT'

T' → *FT' / ϵ

F → (E) / id

FIRST(E) = FIRST(T) ⇒ FIRST(F) = {(, id}

FIRST(E') = {+, ϵ}

FIRST(T) = {(, id}

FIRST(T') = {*, ϵ}

FIRST(F) = {(, id}

FOLLOW(E) = {$,)} Formula

FOLLOW(E') = {$,)}

FOLLOW(T) = ~~{ϵ'~~ = {+, $,)}

E → TE'

FOLLOW(T) = FIRST(E') - ϵ ∪ FOLLOW(E)

= {+, ϵ} - ϵ ∪ {$,)}

= {+, $,)}

E' → +TE'

FOLLOW(T) = FIRST(E') - ϵ ∪ FOLLOW(E')

= {+, ϵ} - ϵ ∪ {$,)}

{+, $,)}

FOLLOW(T')

$T \to F\ T'$ (α = F, B = T', A = ϵ)

FOLLOW(T') = FIRST(ε) − ε ∪ FOLLOW(T)

= ε − ε ∪ {+, $,)}

= {+, $,)}

($T' \to *\ F\ T' \mid \epsilon$)

FOLLOW(F)

$T \to F\ T'$ (A = T, α = F, B = T')

FOLLOW(F) = FIRST(T') − ε ∪ FOLLOW(T)

{*, ε, +} − ε ∪ {+, $,)}

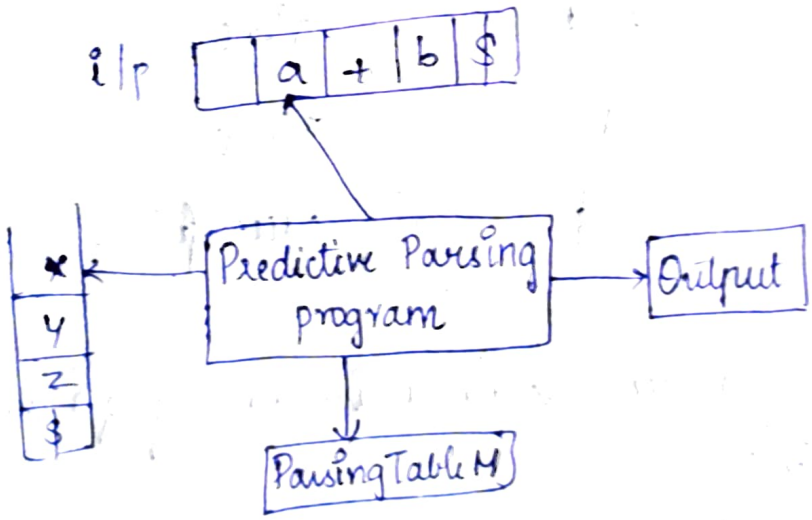
{+, $,)}

terminals	+	*	(	)	$
E			$E \to TE'$		
E'	$E' \to +TE'$				
T			$T \to FT'$		
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	
F			$F \to (E)$		

	First(α)	PPT
$E \to TE'$	First(T) = {(, id}	M[E, (] M[E, id]
$E' \to +TE'$	First(+)	M[E', +]
$E' \to \epsilon$	FOLLOW(E') = {$,)}	M[E', $], M[E', id)]
$T \to FT'$	First(F) = {(, id}	M[T, (], M[T, id)
$T' \to *FT'$	*	M[T', *]
$T' \to \epsilon$	FOLLOW(T') = {+, $,)}	M[T', +], M[T', $], M[T',)]
$F \to (E)$ $F \to id$	FIRST(id) = {id}	M[F, (] M[F, id]

12/10/23

Model of a non-recursive predictive parser



Construct non recursive predictive parser for the following grammer.

$E \to TE'$

$E' \to +TE'/\epsilon$

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to (E)/id$

i/p string id + id * id

Stack	input	o/p
$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 $	T' → ε
$ E'T'	+ id * id $	E' → +TE'
$ E'T+	+ id * id $	T → FT'
$ E'T'F	id * id $	F → id
$ E'T'id	id * id $	T' → *FT'
$ E'T'F*	* id $	F → id
$ E'T'id	id $	T' → ε
$ E'	$	E' → ε
$	$	successful

Construct non recursive predictive parsing for the given

S → a/b/(T)

T → T,S/S

i/p string : (b)

St 1: check whether given grammer is in left recursion or

S → a/b/(T) is not in LR

T → T,S/S is in LR so it be rewritten as A → Aα/β, A → βA', A' → α