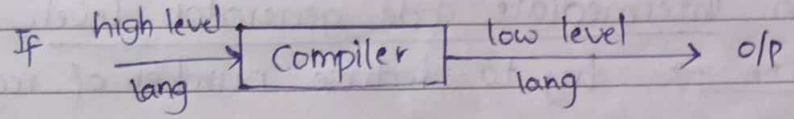
COMPILER DESIGN

DATE: / /
PAGE NO.:

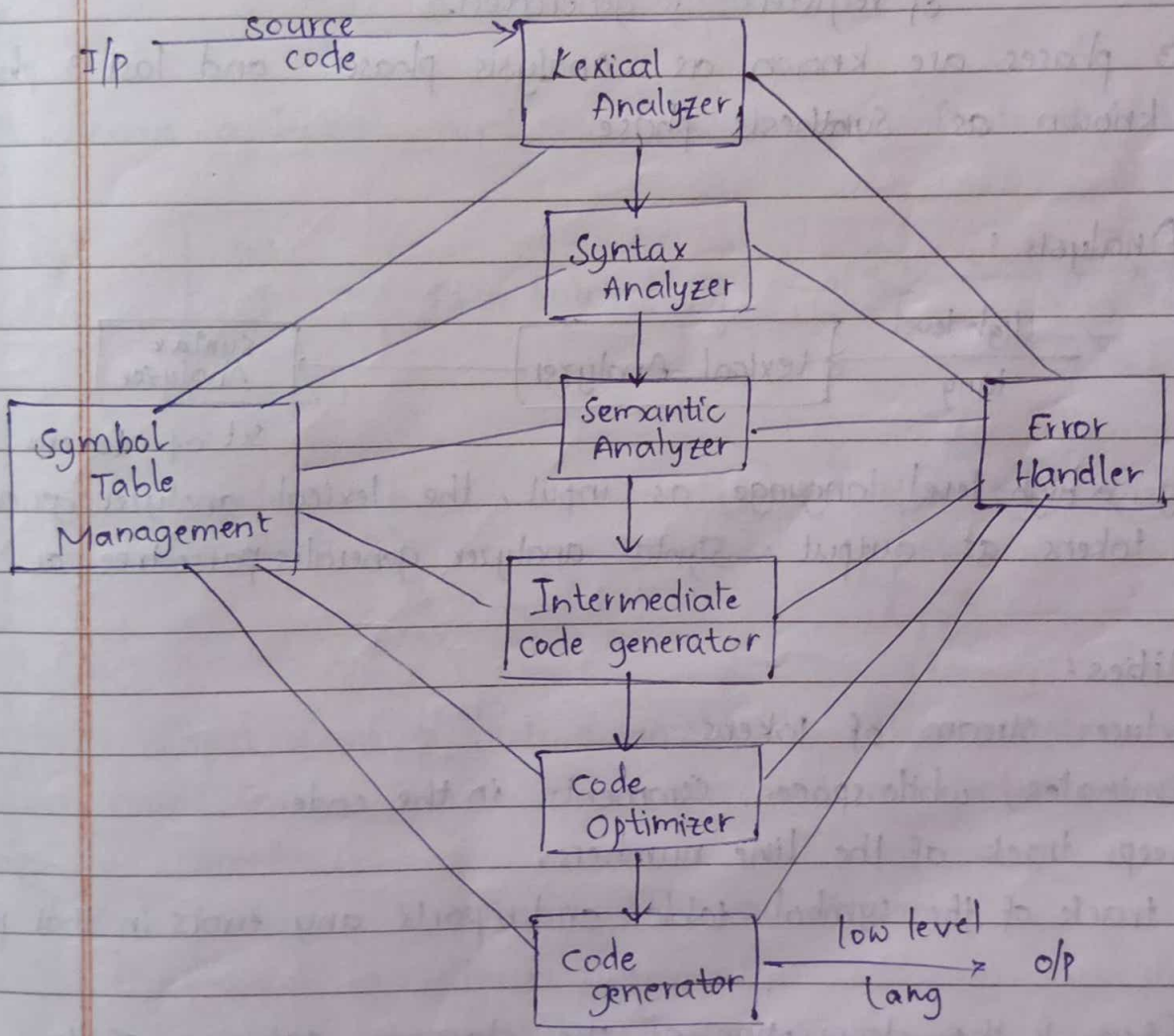
UNIT-I

Input for a compiler - high level language
Output for a compiler - low level language

Translator : converts a source lang to a target language.
Compiler : Compiler is a translator which translates the high level lang to low level language or machine level lang.



6 phases in a compiler



The input to the lexical analyzer is source code. The output from the code generator is low level language.

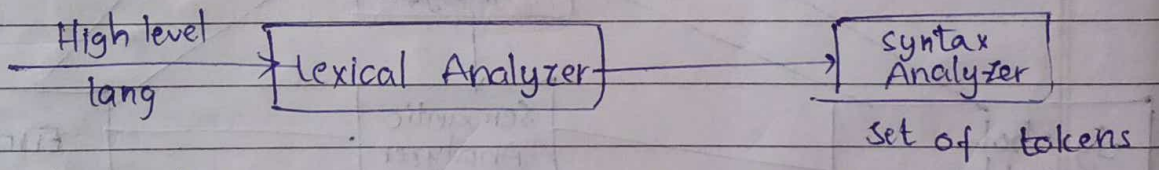
Teacher's Signature

DATE: / /
PAGE NO.:

* Lexical Analyzer converts source code into tokens (smallest units that cannot be further divided.
* Syntax analyzer checks for syntax of the program.
* Semantic Analyzer checks for meaning i.e., int a & a int cannot be detected in above two above levels but in this level.
* A code which is not completely either high level or low level is generated in intermediate code generator level.
* Code Optimizer → try to reduce number of registers required by reducing unnecessary temporary variables.
* Code generator → A complete low level code with required number of registers is generated.

→ First 3 phases are known as Analysis phase and last 3 phases are known as Synthesis phase.

Lexical Analysis:



By taking a high level language as input, the lexical analyzer generates set of tokens as output. Syntax analyzer generates parse tree or

Functionalities:

1. It produces stream of tokens.
2. It eliminates white spaces, comments in the code.
3. It keeps track of the line numbers.
4. Keeps track of the symbol table and reports any errors in that phase.

Token: Token is the description of the class or category of the input string.

Eg: Identifier, keyword, special symbols, constraints, etc

Teacher's Signature

DATE: / /
PAGE NO.:

Lexeme: Lexeme is a sequence of characters in the source program that are matching with the pattern of the token.
Eg: int, num, ;, ', float, a, b, c, char.

Pattern: Pattern is a set of rules that describe the token.
Eg: int a, b, c ;

Token

Lexemes:
- int → Keyword
- a → identifier
- b → identifier
- c → identifier
- , → special symbol
- ; → special symbol.

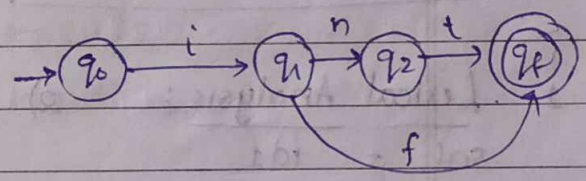
To design a lexical analyzer, we can use finite state Machine (FSM

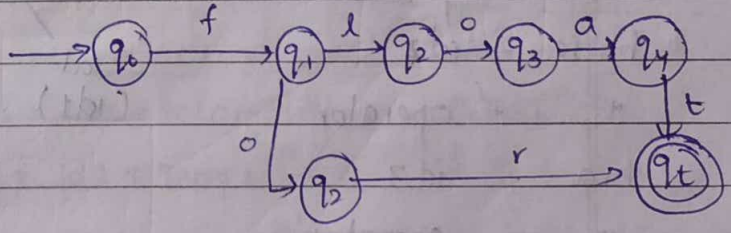
Eg:

int
if
float
for

compiler with 2 keywords, 2 loops.

Finite Automata Eg:





The final output from a finite automata is either accept or reject. So the given lexeme can be either accepted or rejected at this phase of the compiler.

→ Lexical Analyzer is also known as Scanner.

Eg:
1. float-sal, hra, basic ;
2. sal = basic + hra * 2 ;
3. int rno, age ;
4. float cgpa, percentage ;
5. cgpa = percentage / 10 ;

Teacher's Signature

1)

Lexemes	Tokens
float	keyword
sal	identifier
,	special symbol
hra	identifier
basic	identifier
;	ss

2)

Lexemes	Tokens
sal	Identifier
=	ss/operator
basic	identifier
+	Arithmetic Operator
hra	Identifier
*	Arithmetic Operator
2	~~constant~~
;	ss

DATE: / /
PAGE NO.:

3)

Lexemes	Tokens
int	keyword
rno	identifier
,	ss
age	identifier
;	ss

4)

Lexemes	Token
float	keyword
cgpa	identifier
,	ss
percentage	identifier
;	ss

5)

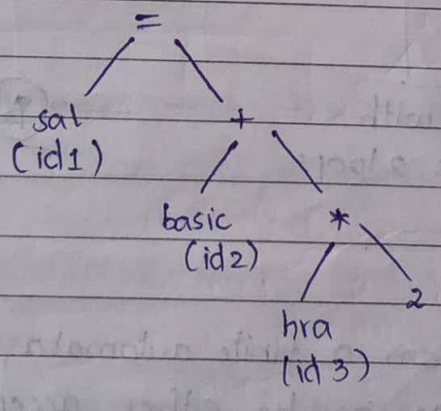
Lexeme	Token
~~float~~ cgpa	identifier
=	operator/ss
percentage	identifier
/	operator
10	constant
;	ss

* sal = basic + hra * 2 ;

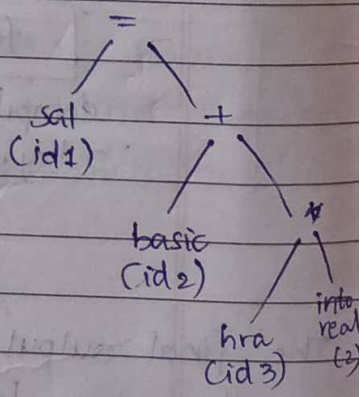
1. Lexical Analysis:

sal - id1
= - operator
basic - id2
+ - operator
hra - id3
* - operator
2 - constant
; - ss

2) Syntax Analysis: (Parse tree CFG)



3) Semantic Analysis:



Intermediate code generator - temporary variables are generated.
In code optimisation, the constraint is satisfied i.e maximum of 3 address locations are ~~taken~~ only used in each statement.

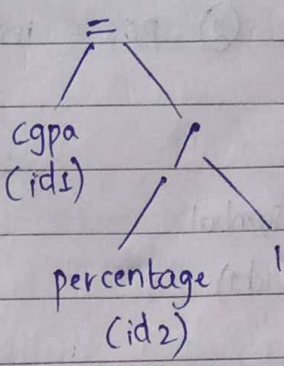
Teacher's Signature

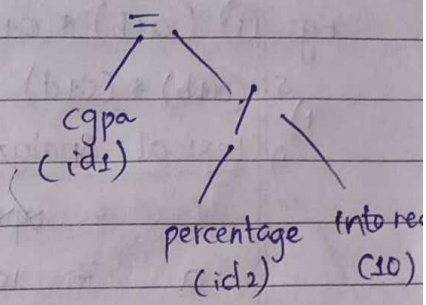
cgpa = percentage / 10 ;

DATE: / /
PAGE NO.:

1) Lexical Analysis 2) Syntax Analysis 3) Semantic Analysis

cgpa - identifier (id1)
= - operator
percentage - identifier (id2)
/ - operator
10 - constant
; - ss





6/02/22 Saturday

4) ICG

Temp1 = intoreal(10)
Temp2 = percentage(id2) / Temp1
cgpa (id1) = Temp2

5) CO

~~Temp1 = percentage / into real (2)~~
(id2)

cgpa (id1) = ~~Temp1~~ percentage (id2) / intoreal (2)

6) Code Generation:

MOV id2, R1 # R1 ← id2
DIV R1, #2.0 # R1 ← R1 / 2.0
MOV R1, id1 # id1 ← R1

sal = basic + hra * 2 ;

4) Intermediate Code Generation

Temp1 = intoreal (2)
Temp2 = (id3) * Temp1
hra
Temp3 = basic + Temp2
(id2)

~~Temp4~~ = sal = Temp3
(id1)

5) Code Optimization

Temp1 = id3 * intoreal (2)
id1 = id2 + Temp1

6) Code Generation

~~MOV id3, R1~~
~~# id3~~
R1 ← id3

6) Code ~~Optimiz~~ Generation:

MOV id3, R1 # R1 ← id3
MUL R1, #2.0 # R1 ← id3 * 2.0
MOV id2, R2 # R2 ← id2
ADD R1, R2 # R1 ← R1 + R2
MOV R1, id1 # id1 ← R1

Any identifier represents address location

Suppose 16 diff operations
⇒ opcode - 4 bits
And 37 registers
⇒ Each register represented with 6 bits
~~id1 size~~

Teacher's Signature

02/03/22

DATE: / /
PAGE NO.:

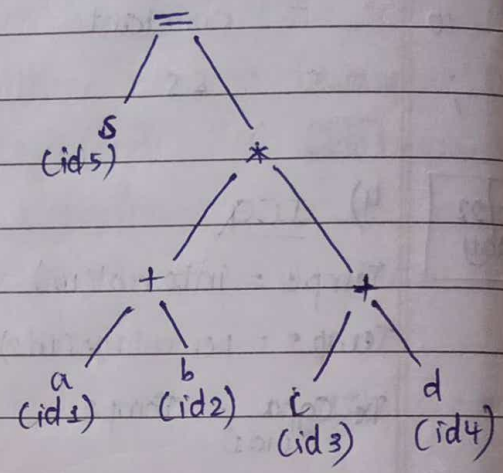
Eg: ① (a+b) *(c+d) ② pos = initial_amount + rate * 6.5

1) s=(a+b) * (c+d)

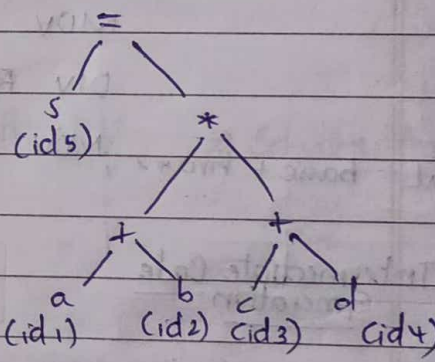
1) Lexical Analyzer

(- ~~super~~ Special Symbol
a - identifier (id1)
+ - operator
b - identifier (id2)
) - ss
* - operator
(- ss
c - identifier (id3)
+ - operator
d - identifier (id4)
) - ~~op~~ ss
s - id5
= - operator

2. Syntax Analysis



3. Semantic Analysis



4. Intermediate Code Generation

Temp1 = a + b
(id1) (id2)

Temp2 = c + d
(id3) (id4)

Temp3 = Temp1 * Temp2

s = Temp3
(id5)

5) Code Optimization

Temp1 = a + b
(id1) (id2)

Temp2 = c + d
(id3) (id4)

~~Temp3~~ = Temp1 * Temp2
s (id5)

6) Code Generation:

- MOV id1, R1 # R1 ← id1	MOV id4, R4 # R4 ← id4
MOV id2, R2 # R2 ← id2	ADD R3, R4 # R3 ← R3+R4
ADD R1, R2 # R1 ← R1+R2	MUL R1, R3 # R1 ← R1 * R3
MOV id3, R3 # R3 ← id3	MOV R1, id5 # id5 ← R1

Teacher's Signature

* Semantic Analysis helps to type cast the constants for correct arithmetic operations.

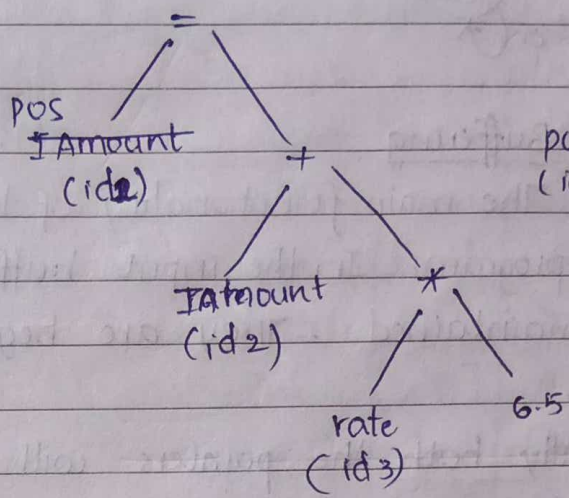
DATE: / /
PAGE NO.:

(2) pos = IAmount + rate * 6.5

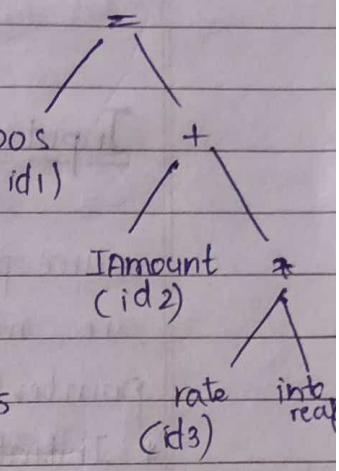
1. Lexical Analyzer

pos - id1
= - operator
IAmount - id2
+ - operator
rate - id3
* - Operator
6.5 - Constant

2. Syntax Analyzer



3. Semantic Analyzer



4. Intermediate Code Generation

Temp1 = intoreal (6.5)
Temp2 = rate (id3) * Temp1
Temp3 = IAmount (id2) + Temp2
pos (id1) = Temp3

5) Code Optimization

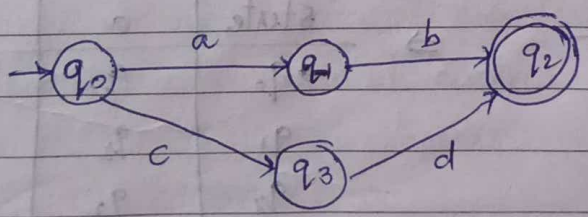
Temp1 = rate (id3) * intoreal(6
~~Temp2~~ = IAmount + Temp1
pos (id1) (id2)

6. Code Generation

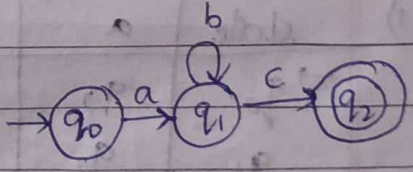
```
MOV  id3, R1    # R1 ← id3
MUL  R1, #6.5   # R1 ← R1 * 6.5
MOV  id2, R2    # R2 ← id2
ADD  R1, R2     #  R1 ← R1 + R2
MOV  R1, id1    #  id1 ← R1
```

05/03/22 Saturday

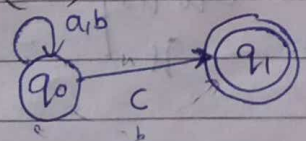
* RE1 = ab + cd



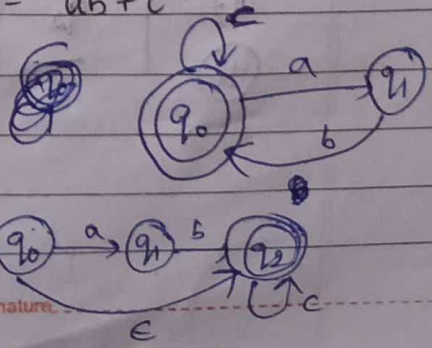
* ab*c



* RE3 = (a+b)*c

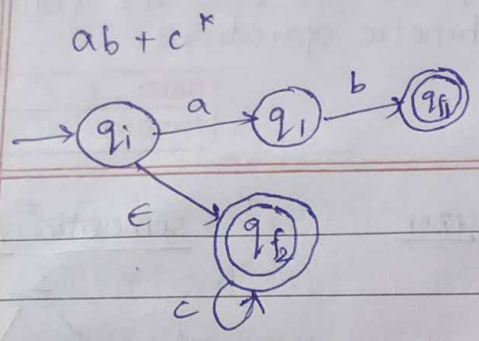


RE4 = ab + c*



cab and acb are accepting here wrong.

Teacher's Signature

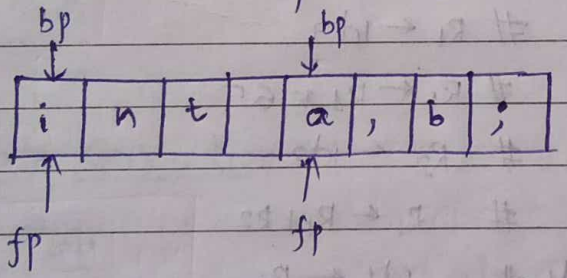


DATE: / /
PAGE NO.:

Input Buffering

The main functionality of lexical analyzer is to scan the source program. In the input buffering scheme, two pointers are maintained. They are begin pointer and forward pointer.

* Initially both the pointers will be pointing to the same character.
* The forward pointer will be incremented until it reaches a white space.
* All the sequence of characters are buffered and they are checked ~~for~~ whether they represent a valid token or not
* Once they are identified to be a valid token, then both the pointers will be pointing to the next valid character



bp - begin pointer
fp - forward pointer

NFA to DFA conversion

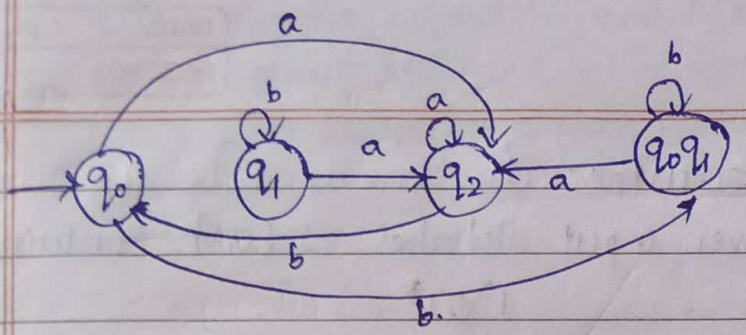
1)

state	a	b
→ q_0	q_2	q_0, q_1
q_1	q_2	q_1
q_2	q_2	q_0
(q_f)	-	-

⇒

state	a	b
→ q_0	q_2	$[q_0 q_1]$
q_1	q_2	q_1
q_2	q_2	q_0
$[q_0 q_1]$	q_2	$[q_0 q_1]$
q_f	-	-

Teacher's Signature



2)

state	0	1
$\rightarrow q_1$	q_2	q_f
q_2	–	q_3
q_3	q_4	q_3
q_4	q_3, q_f	–
(q_f)	–	q_1

⇒

state	0	1
$\rightarrow q_1$	q_2	q_f
q_2	$[q_d]$	q_3
q_3	q_4	q_3
q_4	$[q_3 q_f]$	q_d
$[q_3 q_f]$	q_4	$[q_3 q_1]$
$[q_3 q_1]$	$[q_2 q_4]$	$[q_3 q_f]$
$[q_2 q_4]$	$[q_3 q_f]$	q_3
q_d	–	–
q_f	–	q_1

3)

cs	0	1
$\rightarrow q_0$	q_1	$q_1 q_2$
q_1	q_2	q_0
(q_2)	q_0	–

⇒

cs	0	1
$\rightarrow q_0$	q_1	$[q_0 q_2]$
q_1	q_2	q_0
(q_2)	q_0	–
$[q_1 q_2]$	$[q_1 q_0]$	$[q_0 q_2]$
$[q_1 q_0]$	$[q_1 q_2]$	$[q_0 q_2]$
$[q_1 q_2]$	$[q_0 q_2]$	q_0

Q Write regular expressions for the following:

1. Beginning with c and ending with cc and input alphabet is a, b, c

$$c\,(a+b+c)^*\,cc$$

2. for the set {00, 001, 0011, 00111, ... }

$$001^*$$

3) $0(11)^* \rightarrow \{00, 0011, 001111, \cdots\}$

4) set of all strings over input alphabet $\Sigma\{0,1\}$ containing exactly one zero - i^*0i^*

5) $0^i1^j2^k$ where $i,j,k > 1$ - $0^+1^+2^+$ (or) $00^*11^*22^+$

6) $0^i1^j2^k$ where $i,j \geq 1$, and $k \geq 0$ - $0^+1^+2^*$ (or) $00^*11^*2^*$

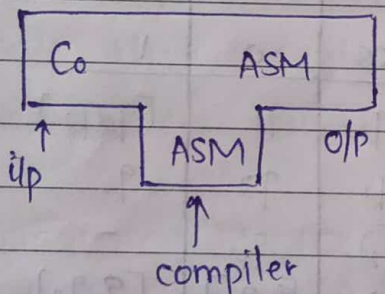
7) $\Sigma(0,1)$ and language should accept all possible strings of length 4.
$(0+1)(0+1)(0+1)(0+1)$

Boot Straping:

→ Inorder to design compiler for a given high-level language, a subset of the language is selected and compiler is designed for this subset.

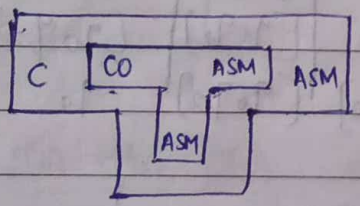
→ This compiler is used for generating the actual compiler.

(1)



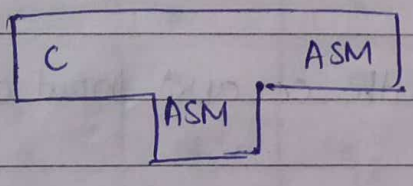
Co is the subset of language c.
ASM is the assembly language program.

(3)



We are using the compiler for subset of C language (Co) to generate the actual compiler for C.

(2)



We want to generate compiler for high level language like c using assembly languag compiler.

Tools:

LEX used in Lexical Analysis

YACC used in syntax Analysis

.....→ continued in next page .....

Teacher's Signature

1. Explain various phases of compiler with example
2. Explain about the functions of Lexical Analyzer
3. Define Regular expression and Finite Automata
4. Explain about Lex.
5. What is a translator. Explain about different translators.

DATE: / /
PAGE NO.:

Translator - 1. Interpreter
2. Compiler
3. Assembler

Translator converts high level language to machine language

Interpreter:
Converts high language into machine language
It considers the source program line by line

Compiler
It considers the entire source program at once.

Assembler
It takes the assembly language program as input and generates the machine code as output.
Assembly lang Eg: MOV R1 R2
The code that is written using Mnemonics is the assembly languag

→ Syntax Analysis takes tokens as input and checks the syntax and generates syntax tree or parse tree
→ It is responsible for identifying the syntax errors in the given source progra

→ Semantic Analysis takes parse tree as input and generates syntax tree with semantics.
→ It is responsible for identifying the semantic errors in the given source program.
→ Implicit type casting is performed in this phase.

→ Intermediate Code Gerator uses temporary variables in prior to generate the machine code.

→ Code optimizer is responsible to reduce or optimize the code generated by the intermediate code generator

Teacher's Signature

→ Also responsible to generate the code strictly following the 3Address Code notation.

→ Code generator is responsible to generate the machine code
→ Actual register assignment, and operations on registers and memory locations is implemented in this phase.

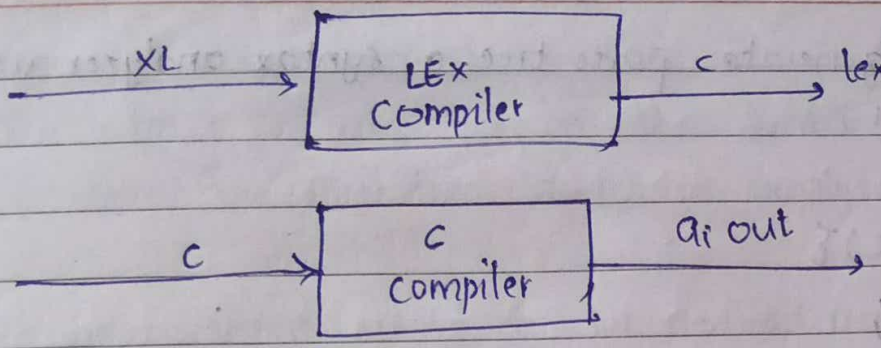
→ Symbol Table Management
→ A data structure known as Location Counter (LC) is used to identify the location of any variable.
→ All the variables are uniquely identified by a number.

→ Error Handler
→ All the errors in various phases of the compiler are recorded by this data structure.

→ .... Continuation ....

* The programs in LEX are saved with L extension.
* The output from a LEX compiler is a C file.
* The C file is given to a C compiler which generates an executable file and it is responsible for generating/identifying the tokens.

→ The general syntax of a LEX file contains 3 sections.
1. Declaration section
2. Rule section
3. Procedure Section




```
%{ Declaration      → syntax of lex program.
%}
%%
   Rules
%%
   Procedures
```

Eg:
```
%{
%}
%%
   "Talkative"
   "Naughty" printf ("iii CSE4");
%%
int main ()
{
  yylex ();
  set 0; return 0;
}
```

LEX program for counting no of identifiers

```
digit [0-9]
letter [A-Z . a-z]
%{
   int count;
%}
%%
{letter} ( {letter}/{digit})*
```

Teacher's Signature

Inorder to generate parse tree, a syntax analyzer uses a CFG

```
    count++;
%%
int main(){
 yylex();
 printf("no of identifiers = %d\n", count);
 return 0;
 }
```

14/08/22 Monday

UNIT - II

SYNTAX ANALYSIS

* The input to syntax analyzer is set of token.
* The output is parse tree or the syntax tree.

→ In order to derive the parse tree, a syntax analyzer uses a parse tree.

The given input string can be derived using the CFG starting with the START symbol

CFG :

It is a 4-tuple (N,T,P,S) where N is set of non-terminals, T is set of terminals, P is set of production rules, and S is start symbol.

Parse Tree :

The diagramatic representation of the derivation of the given string using the CFG or production rules.

Leftmost Derivation :

* In the derivation process, the leftmost non-terminal is considered first

Eg: For the given CFG, $S \rightarrow 0B \mid 1A$

$A \rightarrow 0 \mid 0S \mid 1AA$

$B \rightarrow 1 \mid 1S \mid 0BB$

Check whether the given string can be derived or not. & string: 00110101

Left most derivation:

$S \rightarrow 0B$

$\rightarrow 00BB$

$\rightarrow 001SB$

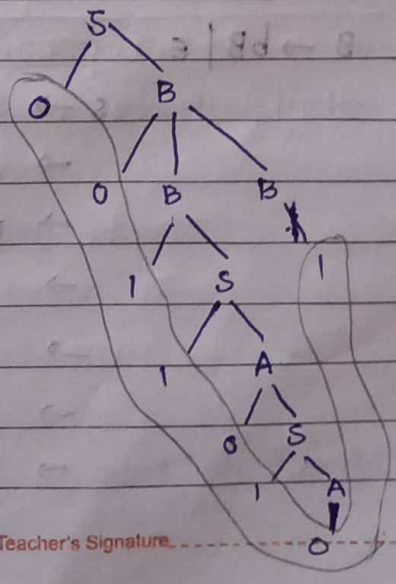
$\rightarrow 0011AB$

$\rightarrow 00110SB$

$\rightarrow 001101AB$

$\rightarrow 0011010B$

$\rightarrow 00110101$



Teacher's Signature

Right most derivation

DATE: / /
PAGE NO.:

00110101

$S \rightarrow 0B$
$\rightarrow 00BB$
$\rightarrow 00B1$
$\rightarrow 001S1$
$\rightarrow 0011A1$
$\rightarrow 00110S1$
$\rightarrow 001101A1$
$\rightarrow 00110101$

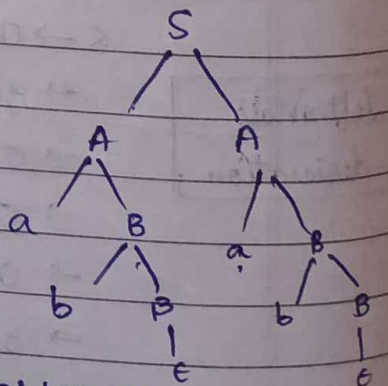
Eg:2 : $S \rightarrow aSX \mid b$
$X \rightarrow Xb \mid a$ string: aababa

$S \rightarrow aSX$
$\rightarrow aaSXX$
$\rightarrow aabXX$
$\rightarrow aabaXbX$
$\rightarrow aababa$

The string cannot be derived

Eg:3 Check whether the string can be derived or not
$S \rightarrow AA$
$A \rightarrow aB$
$B \rightarrow bB \mid \epsilon$ string: abba

$S \rightarrow AA$
$\rightarrow aBA$
$\rightarrow aBaB$
$\rightarrow abBaB$
$\rightarrow abbaB \rightarrow abbBaB$
$\rightarrow abb\epsilon ab \rightarrow abb\epsilon a\epsilon$
$\rightarrow abab\epsilon \rightarrow ab \rightarrow abba$



Teacher's Signature

DATE: / /
PAGE NO.:

Eg:4 : $S \rightarrow cAd$
$A \rightarrow ab|c$

strings: cabd
cabdab

$S \rightarrow cAd$
$\rightarrow cabd$

$S \rightarrow cAd$
$\rightarrow cabd$ (can't derive cabdab)

Eg:5 : $A \rightarrow Ba|bc$
$B \rightarrow d|eBf$

strings: edfa , eebffa
edbc bedffa

$A \rightarrow Ba$
$\rightarrow eBfa$
$\rightarrow edfa$

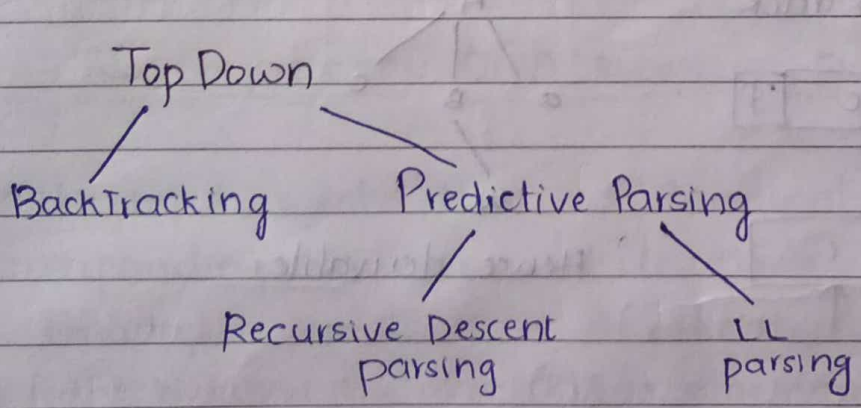
$A \rightarrow Ba$
↓ not ending with c
edbc can't be derived

$A \rightarrow Ba$
$\rightarrow eBfa$
$\rightarrow eeBffa$
$\rightarrow eedffa$

15/08/22 Tuesday

Parsing Techniques:

1. Top down - start the parsing with START symbol
2. Bottom Up



1. BackTracking:

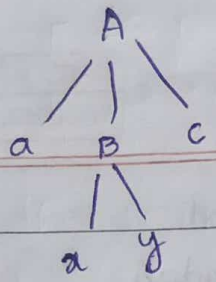
* Here trial and error method is used to check whether the string (given input string) is acceptable or not.

Eg: $A \rightarrow aBc$
$B \rightarrow xy|w$

String: awc

* Any parser start with the START symbol in top down approach

Teacher's Signature

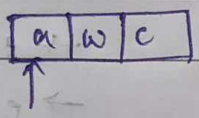


* End of the string represented by "$" symbol

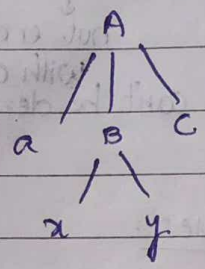
Look Ahead Pointer:

A pointer named look ahead pointer is used to point the next character in the given input string. Whenever the look ahead pointer reaches the end of the string ("$"), then the string is said to be derivable from the grammar.

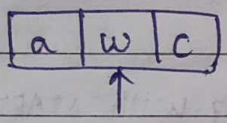
Initially



$a = a$

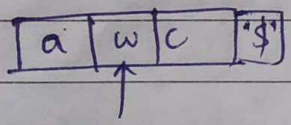


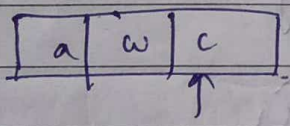
increment pointer

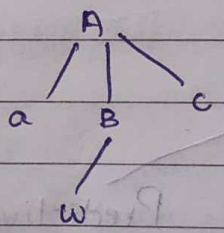


$x \neq w$

Hence Backtrack







Hence derivable

Advantage:
Easy to implement.

Disadvantage
Not at all efficient.

2) Recursive Descent Parsing:

In Recursive descent parsing, procedures for the non-terminals symbols that appear in the CFG are written.

Advantage: can be implemented easily, simple

Disadvantage: It may enter into infinite loops.
Time taking.

Teacher's Signature

DATE: / /
PAGE NO.:

Procedure

```
A()
{
choose any production
    A → x1, x2, x3, ...., xk
for (i=1 to k)
{
if xk is a non-terminal
then call procedure xk();
else if
    xk is a terminal
        call match (xk);
else
    print (error);
}
}
```

```
match(xk)
{
if (la == xk)
    la++;
else
    error();
}
```

```
error()
{
print "error";
}
```

CONDITIONS

STEPS FOR THE CONSTRUCTION OF A RECURSIVE DESCENT PARSING:

Step 1: If the input symbol is a non-terminal, then a call to the corresponding non-terminal is made.

step 2: If the input symbol is a terminal, then it is checked for a match with the Look Ahead pair from the input.

Cond 3: If the production rule has many alternatives, then they all should be combined into a single procedure

Cond 4: The parser should be activated by a procedure corresponding to the start symbol.

Advantages: It can be implemented easily

Disadvantages: 1. Takes more time
2. It may enter into infinite loops.

Teacher's Signature

22/03/22 Tuesday

num → Terminal.

$E \rightarrow num\ T$

$T \rightarrow *num\ T / \epsilon$

DATE: / /
PAGE NO.:

Q. Generate the recursive descent parser for the given grammar.

```
procedure E ()
{
  if la = num
  {
    match (num);
    T();
  }
  else
    error();
  if la = $
  {
    print ("success");
  }
  else
  {
    error();
  }
}
```

```
procedure T()
{
  if la = *
  {
    match(*); if la = num
    {
      match(num);
      T();
    }
    else
      error();
  }
  else
    NULL;
}
```

Match : match(x_k) {
  if (la == x_k)
    la++;
  else
    error }

Error :

```
Error() {
  print ("error");
}
```

2) Check whether the given strings are accepted or not for the above gram using recursive descent parsing technique.

string 1 : 3 * 4 $

string 2 : 3 * 4 + $

String 3 : 3 * 4 * 5

Teacher's Signature

```
procedure E()
{
 if la = 3
  { match (3);
    T();
  }
 else
   error();
 if la = $
   print ("Success");
 else
    error();
}
```

```
procedure T()
{
 if la = *
  { match(*);
    if la = 4
     { match(4);
       T();
     }
    else
      error();
  }
 else
    NULL;
}
```

DATE: / /
PAGE NO.:

Here the string 3*4$ is accepted by using RDP.

3) $S \rightarrow cAd$
$A \rightarrow ab|c$ Generate RDP for given grammar

```
procedure S(){
   if la = c
    { match(c);
      A();
    }
   if la = d
     match(d);
   else
     error();
   if la = $
     print (success);
   else
     error();
```

```
procedure A()
{  if la = a
   { match(a);
     if la = b
     { match(b);
     }
   }
   else if la = c
     match (c);
     else
      error();
}
```

4) $A \rightarrow Ba|bc$
$B \rightarrow d|eBf$ Generate RDP for given grammar and check for "edfa" and "edbc"

```
procedure A(){
   B();
   if la = a {
     match(a);}
   else if la = b{
     match(b);
     if la = c
        match(c);
   }
   if la = $
     print ("success");
   else error();
```

```
procedure B() {
   if la = d {
     match(d);}
   else if la = e
     { match(e);
       B();
     if la = f
     { match (f);
     }
     else error();
   }
```

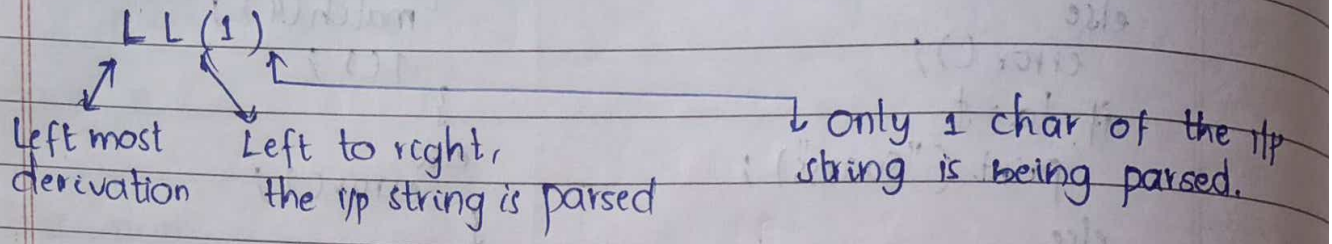
∴ edfa accepted

Teacher's Signature

29/03/22
Tuesday

DATE: / /
PAGE NO.:

LL(1) Parser:

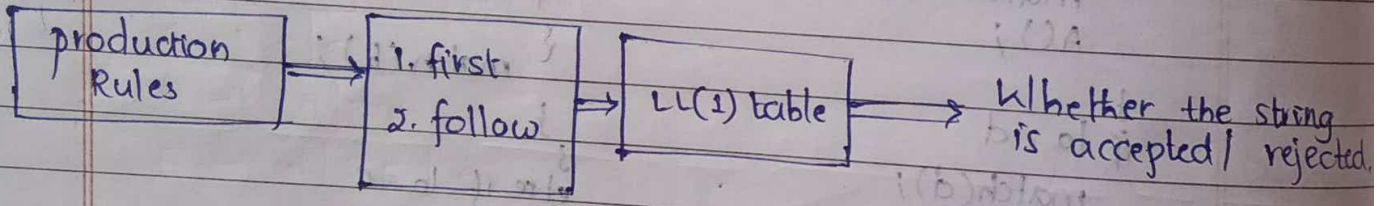


In this particular parsing technique, we need to find certain function values.

* Using the production rules, two functions are applied, From the results, we are constructing or generating the table LL(1) table and from the results of the table, we find whether the i/p string is accepted or rejected.

The two functions are

1. First
2. Follow



Rules for "First" :

1. First(a) = {a}
2. First (ϵ) = {ϵ}
3. X → x1 x2 x3 .... xn
   First (X) = First (x1)

Eg: . A → Bad.
First (A) = First (B) = [redacted]
A → bad
First (A) = First (b)
= {b}

1. First (any terminal symbol) is the terminal symbol itself.
2. First (an empty symbol epsilon) is Epsilon.
3. If the production is of the form
   X → x1 x2 x3 ... xn , then
   First (x1) is included in First (x)

Teacher's Signature

Eg:1 $S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAb \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

DATE: / /
PAGE NO.:

First(S) is the union of first of all productions of S.

For productions of S

$S \rightarrow aAB$ First $(aAB) = \{a\}$

$S \rightarrow bA$ First $(bA) = \{b\}$ First(S) $= \{a, b, \epsilon\}$

$S \rightarrow \epsilon$ First $(\epsilon) = \{\epsilon\}$

For productions of B

$B \rightarrow bB$ First $(bB) =$ First $(b) = \{b\}$ First $(B) = \{b, \epsilon\}$

$B \rightarrow \epsilon$ First $(\epsilon) = \{\epsilon\}$

For productions of A

$A \rightarrow aAb$ First $(aAb) =$ First $(a) = \{a\}$ First $(A) = \{a, \epsilon\}$

$A \rightarrow \epsilon$ First $(\epsilon) = \{\epsilon\}$

First(S) = {a, b, ε}
First (B) = {b, ε}
First (A) = {a, ε}

Eg:2 $S \rightarrow iCtSA \mid a$

$A \rightarrow eS \mid \epsilon$

$C \rightarrow b$

This can be $S \rightarrow iCtSA \mid a$

S → statement; i → if Condition then statement; a → statement

~~A can be~~

A can be $eS \rightarrow$ else statement.

For productions of S

$S \rightarrow iCtSA$ First $(iCtSA) =$ First $(i) = \{i\}$

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

For productions of A

$A \rightarrow eS$ First $(eS) =$ First $(e) = \{e\}$

$A \rightarrow \epsilon$ First $(\epsilon) = \{\epsilon\}$

For production of C

$C \rightarrow b$ First $(b) = \{b\}$

First(S) = {i, a}
First (A) = {e, ε}
First (C) = {b}

Teacher's Signature

Eg: 3) $S \to ABC$
$A \to a \mid Cb \mid \epsilon$
$B \to c \mid dA \mid \epsilon$
$C \to e \mid f$

First(S) = First(ABC)
= First(A)
= First(a) ∪ First(C) ∪ First(ϵ)
= {a} ∪ {e, f} ∪ {ϵ}
= {a, e, f, ϵ}

First(A) = {a, e, f, ϵ}

First(B) = ~~{c, d, ϵ}~~ First(c) ∪ First(dA) ∪ First(ϵ)
= {c} ∪ {d} ∪ {ϵ}
= {c, d, ϵ}

First(C) = {e, f}

First(A) = First(a) ∪ ~~First~~(Cb) ∪ First(ϵ)
= {a} ∪ First(C) ∪ {ϵ}
= {a} ∪ First(e) ∪ First(f) ∪ {ϵ}
= {a, e} ∪ {e} ∪ {f}
= {a, e, f, ϵ}

Eg: 4) $S \to A \mid BCD$
$A \to BBA \mid EB$
$B \to bEc \mid bc \mid bdc \mid \epsilon$
$C \to c \mid \epsilon$
$D \to a \mid BDb$
$E \to a \mid bE \mid \epsilon$

First(S) = First(A) ∪ First(BCD)
First(A) = First(BBA) ∪ First(EB)
= First(B) ∪ First(E)
First(B) = First(bEc) ∪ First(bc) ∪ First(bdc) ~~∪ First(ϵ)~~ ∪ First(ϵ)
= {b} ∪ {b} ∪ {b} ∪ {ϵ}
= {b, ϵ}
First(E) = First(a) ∪ First(bE) ∪ First(ϵ)
= {a, b, ϵ}

⇒ First(A) = First(B) ∪ First(E)
= {b, ϵ} ∪ {a, b, ϵ}
= {a, b, ϵ}

~~firs~~

Teacher's Signature

First (S) = First (A) ∪ First (BCD)
= {a, b, ∈} ∪ First (B)
= {a, b, ∈} ∪ {b, ∈}
= {a, b, ∈}

DATE: / /
PAGE NO.:

First (C) = First (c) ∪ First (∈)
= {c} ∪ {∈}
= {c, ∈}

First (D) = First (a) ∪ first (BDb)
= {a} ∪ First (B)
= {a} ∪ {b, ∈} = {a, b, ∈}

first (E) = {a, b, ∈}

First (A) = First (A) = first (D) = First (E) = {a, b, ∈}
First (B) = {b, ∈}
First (C) = {c, ∈}

04/04/22 Monday

S → (L) | a
L → L, S | S Left Recursive grammar

First (L) = First (L, S) = First (L)
First (L) = First (S) = {(, a}
First (S) = First ((L)) = {(} } {(, a}
First (S) = First (a) = {a}

Cannot be determined.

- Eliminating left ~~grammar~~ recursion

A → Aα | β
A → βA'
A' → αA' | ∈

L → L, S | S
L → SA'
A' → , SA' | ∈

First (L) = First (S) = { (, a }
First (A') = First (, SA') = { , } { , , ∈}
First (A') = First (∈) = {∈}

First (L) = {(,
First (A') = {,
First (S) = {(,

Teacher's Signature

Eg(6)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

It is Left Recursive

DATE: / /
PAGE NO.:

$E \rightarrow E+T \mid T$

$E \rightarrow TE'$

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

First(E) = First(T)

First(E') = First(+TE') = {+}

First(E') = First(ε) = {ε}

$T \rightarrow T*F \mid F$

$T \rightarrow FT'$

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

First(T) = First(F)

First(T') = First(*FT') = {*}

First(T') = First(ε) = {ε}

$F \rightarrow (E) \mid id$

First(F) = First((E)) = {(}

First(F) = First(id) = {id}

First(F) = { (, id }

First(T) = { (, id }

First(E) = { (, id }

First(E') = { +, ε }

First(T') = { *, ε }

07/04/22 Thursday

Rules for Follow

1. "$" must be included follow(s)
2. If $A \rightarrow \alpha B\beta$, then First($\beta$) should be included in Follow(B) except ε
3. If $A \rightarrow \alpha B$, then Follow(A) is included in Follow(B) (or) $A \rightarrow \alpha B\beta$ and $\beta \rightarrow \epsilon$

Eg ①

$S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAb \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

Follow(S) = {$}

Follow(A) = {b, $}

Follow(B) = {$}

$S \rightarrow aAB$

First rule - Include "$" in Follow(s)

Follow(s) = {$}

Teacher's Signature

$S \rightarrow aAB$

It is of the form $A \rightarrow \alpha B \beta$

$\therefore$ ~~Follow(~~ first(β) should be included in Follow of B except ϵ

$\therefore$ first(B) included in Follow(A)

first(B) = b

$\therefore$ Follow(A) = { b

$S \rightarrow \underset{\alpha}{a} \underset{B}{AB}$ also in the form $A \rightarrow \alpha B \Rightarrow$ Follow(A) included in Follow(B)

$\therefore$ Follow(S) included in Follow(B)

Follow(B) = { $

For the production $S \rightarrow \underset{\alpha}{b} \underset{B}{A}$ in the form $A \rightarrow \alpha B$

$\therefore$ Follow(S) included in Follow(A)

Follow(A) = { b, $

For the production $A \rightarrow \underset{\alpha}{a} \underset{B}{A} \underset{\beta}{b}$

first(β) in Follow(A) $\Rightarrow$ first(b) = {b}

Follow(A) = { b, $

For the production $B \rightarrow \underset{\alpha}{b} \underset{B}{B}$

Follow(B) included in Follow(B)

Follow(S) = { $ }
Follow(A) = { b, $ }
Follow(B) = { $ }

Eg:②

$S \rightarrow iCtSA \mid a$

$A \rightarrow eS \mid e$

$C \rightarrow b$

DATE: / /
PAGE NO.:

Follow(S) = {$

$S \rightarrow \underset{\alpha}{i}\underset{B}{C}\underset{\beta}{tSA}$ First (tSA) = First(t) = t

Follow(C) = {t

and

$S \rightarrow \underset{\alpha}{iCt}\underset{B}{S}\underset{\beta}{A}$ First (A) = First (eS) = First(e) = e

Follow(S) = {$, e

$S \rightarrow \underset{\alpha}{iCtS}\underset{B}{A}$ Follow (S) included in Follow(A):

Follow(A) = {$, e

$A \rightarrow \underset{\alpha}{eS}$ Follow (A) included in Follow(S)

Follow (S) = {$, e}

Follow (A) = {$, e}

Follow (C) = {t}

Eg:③

$S \rightarrow ABC$

$A \rightarrow a \mid Cb \mid \epsilon$

$B \rightarrow c \mid dA \mid e$

$C \rightarrow e \mid f$

$A \rightarrow \underset{\alpha}{\epsilon}\underset{B}{C}\underset{\beta}{b}$ First(b) in follow

Follow(S) = {$

$S \rightarrow \underset{\alpha}{A}\underset{B}{B}\underset{\beta}{C}$ First(C) = First(e) ∪ First(f)

Follow(B) = {e, f}

$S \rightarrow \underset{\alpha}{A}\underset{B}{B}\underset{\beta}{C}$ First(β) = First(B) = First(c) ∪ First(dA) = {c, d} included in follow (A)

$S \rightarrow \underset{\alpha}{ABC}$ Follow (S) included in follow(C)

Follow(C) = {$, b}

~~A → Cb~~

$B \rightarrow \underset{\alpha}{dA}$ Follow(B) included in Follow(A)

Follow(A) = {e, f}

Follow(B) = {e, f}

Follow(C) = {$, b}

Follow(A) = {e, f, c, d}

Follow(S) = {$}

Teacher's Signature

08/04/22 Friday

DATE: / /
PAGE NO.:

Rules for Construction of LL(1) Table:

For $A \longrightarrow \alpha$

All the terminals are columns and Non-Terminals are rows

1. For each "a" in First(α)
   $M[A,a] = A \rightarrow \alpha$

2. If First(α) = ϵ, for each "b" in Follow(A)
   $M[A,b] = A \rightarrow \alpha$

3. Other entries are marked as error.
   "$" is included in set of columns

NOTE For any cell value, if we get more than one entry, then it is not LL(1) grammar.

~~a b $ $~~

~~S S→aAB~~

~~A~~

~~B~~

Eg: (1)

$S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAb \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

	a	b	$
S	S→aAB	S→bA	S→ϵ
A	A→aAb	A→ϵ	A→ϵ
B	-	B→bB	B→ϵ

Teacher's Signature