

COMPILER DESIGN

DATE: / /

PAGE NO.:

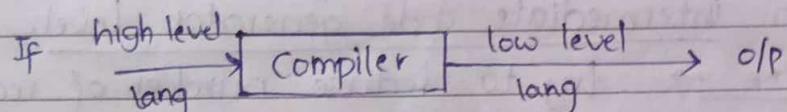
UNIT-1

Input for a compiler - high level language

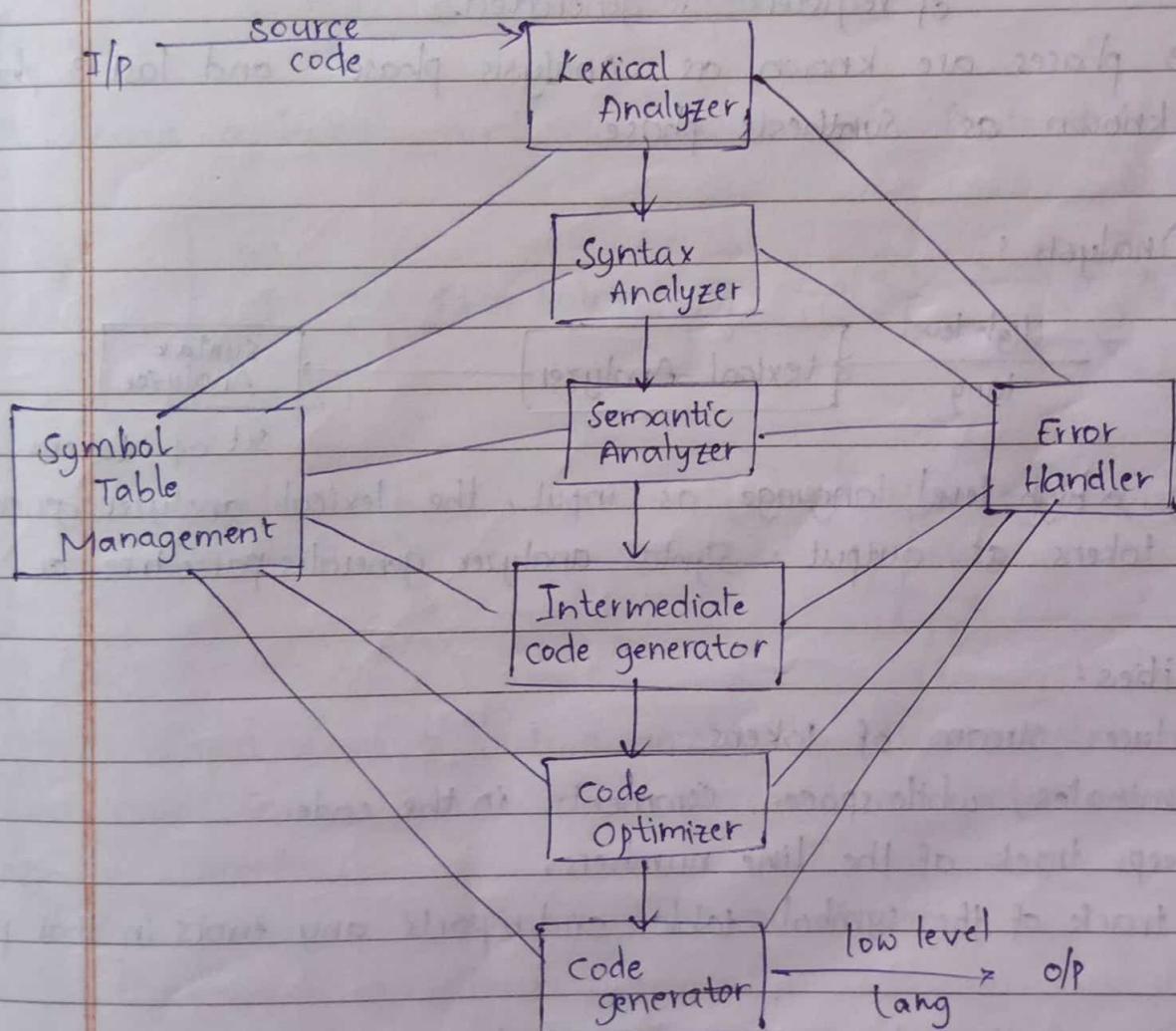
Output for a compiler - low level language

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.



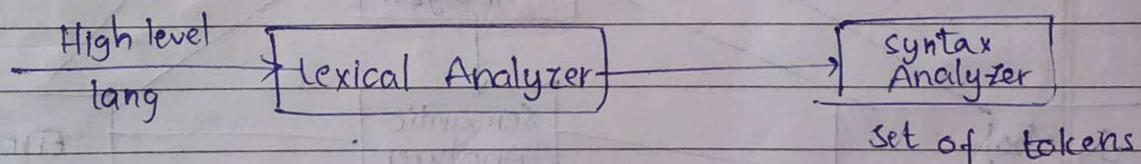
6 phases in a compiler



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

- * 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. if 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 whitespaces, 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.

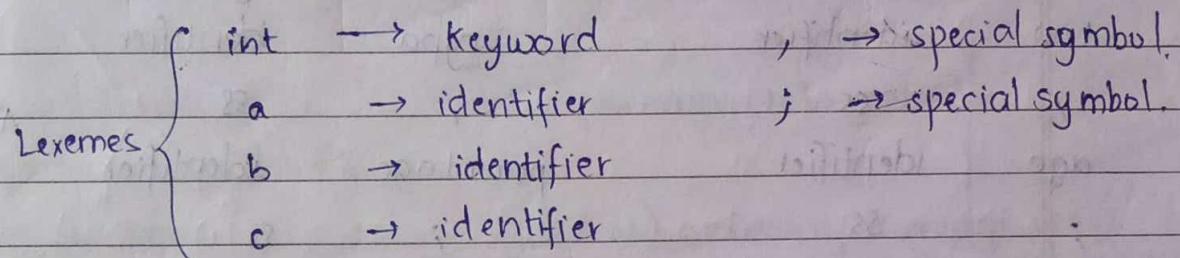
Eg, Identifier, keyword, special symbols, constraints, etc

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 ;



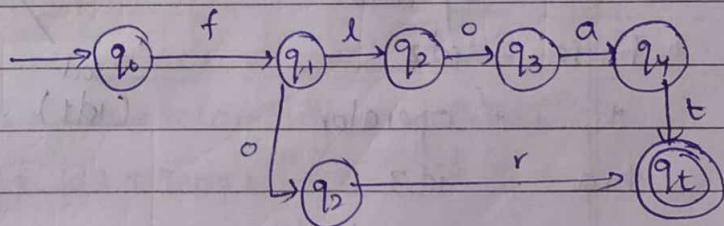
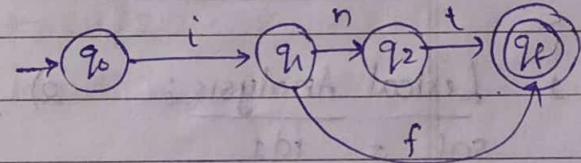
To design a lexical analyzer, we can use finite state machine (FSM).

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 ;



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

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

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

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

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

* $\boxed{\text{sal} = \text{basic} + \text{hra} * 2 ;}$

1) Lexical Analysis:

sal - id₁

= - operator

basic - id₂

+

- operator

hra - id₃

*

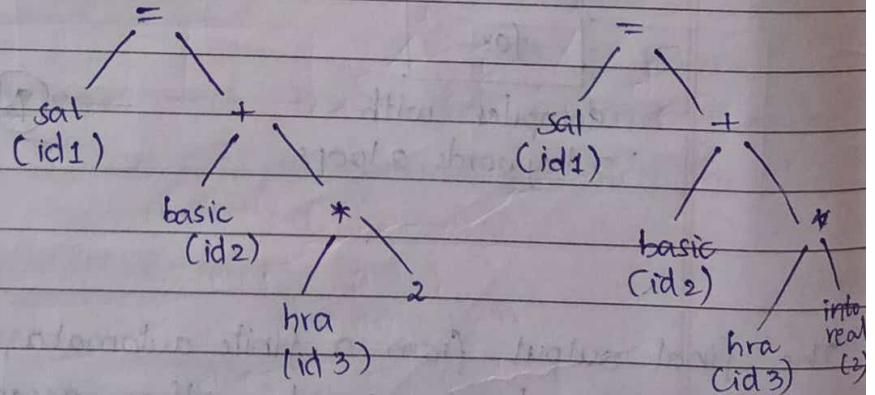
- operator

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.

cgpa = percentage / 10

DATE: / /

PAGE NO.:

1) Lexical Analysis

cgpa
= identifier(id1)
percentage
1 - operator
10 - identifier(id2)
. - operator
ss - constant

2) Syntax Analysis

=
percentage
10.
percentage
ss.

3) Semantic Analysis

=
cgpa
(id1)
percentage
(id2)
into real
(10)

4) ICG

Temp1 = intoreal(10)

Temp2 = percentage(id2) + Temp1

cgpa = Temp2
(id1)

5) CO

Temp1 = percentage / intoreal(2)

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

6) Code Generation:

MOV id2, R1 # R1 \leftarrow id2

DIV R1, #2.0 # R1 \leftarrow R1 / 2.0

MOV R1, id1 # id1 \leftarrow R1
sal = basic + hra * 2;

4) Intermediate Code Generation

Temp1 = intoreal(2)

Temp2 = $\frac{hra}{id3} * Temp1$

Temp3 = basic + Temp2
(id2)

Temp4 = sal = Temp3
(id1)

5) Code Optimization

Temp1 = id3 * intoreal(2)

id1 = id2 + Temp1

6) Code Generation

MOV id3, R1

R1 \leftarrow id3

6) Code Optimization:

MOV id3, R1 # R1 \leftarrow id3

MUL R1, #2.0 # R1 \leftarrow id3 * 2.0

MOV id2, R2 # R2 \leftarrow id2

ADD R1, R2 # R1 \leftarrow R1 + R2

MOV R1, id1 # id1 \leftarrow R1

Suppose 16 diff operations

\Rightarrow opcode = 4 bits

And 37 registers

\Rightarrow Each register represented with

6 bits

size

Any identifier represents address location

Teacher's Signature

Eg: ① $(a+b)*(c+d)$

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

1) Lexical Analyzer

(- ^{a1} open specific symbol

a - identifier (id1)

+ - operator

b - identifier (id2)

) - ss

* - Operator

(- ss

c - identifier (id3)

+ - operator

d - identifier (id4)

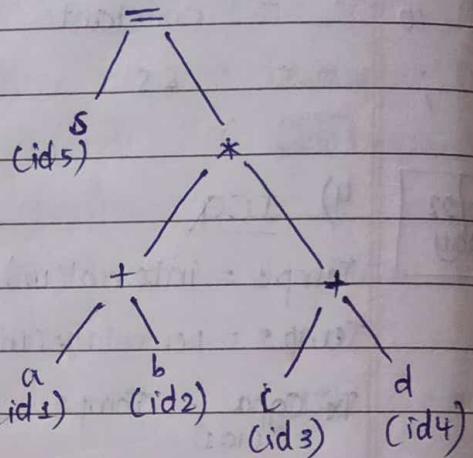
) - op. ss

s - id5

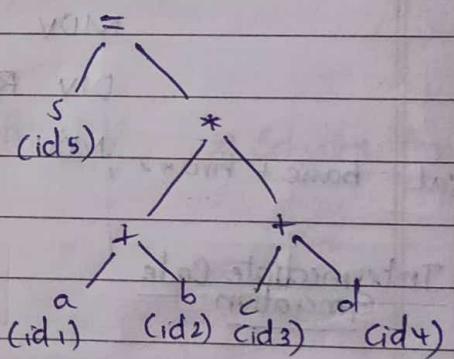
= - operator

② $pos = \text{initial amount} + \text{rate} * 6.5$

2) Syntax Analysis



3) Semantic Analysis



4) Intermediate Code Generation

$$\text{Temp1} = a + b$$

(id1) (id2)

$$\text{Temp2} = c + d$$

(id3) (id4)

$$\text{Temp3} = \text{Temp1} * \text{Temp2}$$

$$s = \text{Temp3}$$

(id5)

5) Code Optimization

$$\text{Temp1} = a + b$$

(id1) (id2)

$$\text{Temp2} = c + d$$

(id3) (id4)

$$\text{Temp3} = \text{Temp1} * \text{Temp2}$$

(id5)

6) Code Generation:

- MOV id1, R1 # $R_1 \leftarrow id_1$

MOV id2, R2 # $R_2 \leftarrow id_2$

ADD R1, R2 # $R_1 \leftarrow R_1 + R_2$

MOV id3, R3 # $R_3 \leftarrow id_3$

MOV id4, R4 # $R_4 \leftarrow id_4$

ADD R3, R4 # $R_3 \leftarrow R_3 + R_4$

MUL R1, R3 # $R_1 \leftarrow R_1 * R_3$

MOV R1, id5 # $id_5 \leftarrow R_1$

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

DATE: / /

PAGE NO.:

$$(2) \text{ pos} = \text{IAmount} + \text{rate} * 6.5$$

1. Lexical Analyzer

pos - id1

= - operator

IAmount - id2

+ - operator

rate - id3

* - Operator

6.5 - Constant

2. Syntax Analyzer

pos
IAmount
(id2)

IAmount
(id2)

rate
(id3)

pos
(id1)

IAmount
(id2)

rate
(id3)
int real

3. Semantic Analyzer

3. Semantic Analyzer

4. Intermediate Code Generation

Temp1 = intoreal(6.5)

Temp2 = rate(id3) * Temp1

Temp3 = IAmount(id2) + Temp2

pos(id1) = Temp3

4) Code Optimization

Temp1 = rate(id3) * intoreal(6.5)

~~Temp1~~ = IAmount + Temp1

pos(id1) = Temp1

5. Code Generation

MOV id3, R1 # R1 \leftarrow id3

MUL R1, #6.5 # R1 \leftarrow R1 * 6.5

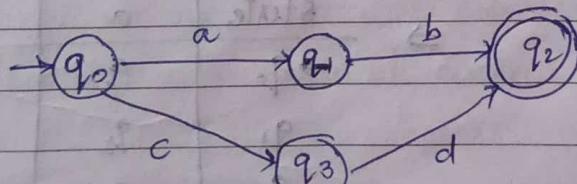
MOV id2, R2 # R2 \leftarrow id2

ADD R1, R2 # R1 \leftarrow R1 + R2

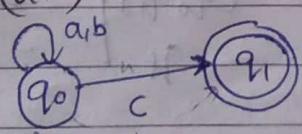
MOV R1, id1 # id1 \leftarrow R1

05/03/22
Saturday

* RE1 = ab + cd

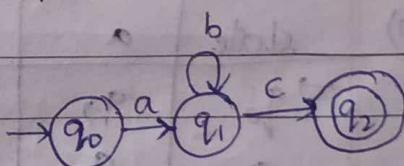


* RE3 = (a+b)*c

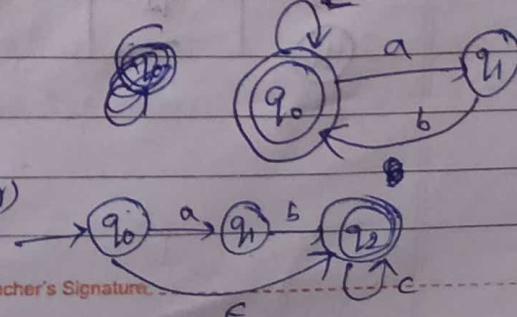


oab
and
acb
are
accepting
here
wrong.

* ab*c

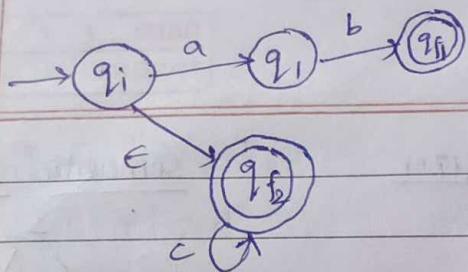


RE4 = ab*c



Teacher's Signature

ab + c*



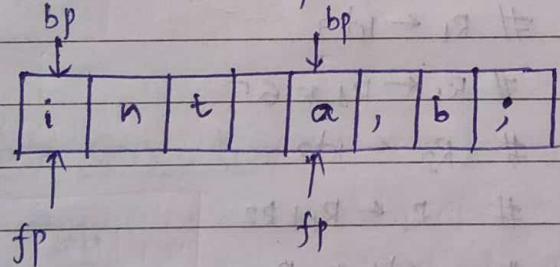
DATE: / /

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.

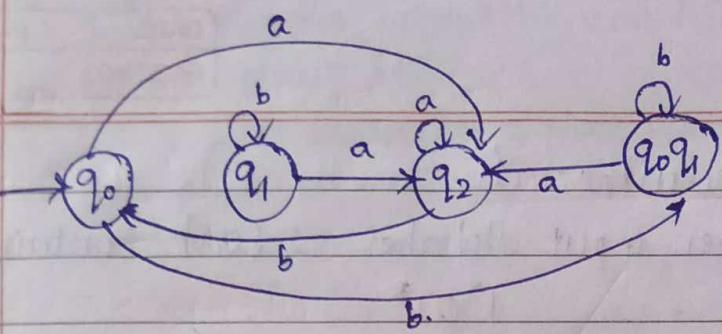


bp - begin pointer

fp - forward pointer

NFA to DFA conversion

state	a	b	state	a	b
$\rightarrow q_0$	q_2	q_0, q_1	$\rightarrow q_0$	q_2	$[q_0 q_1]$
q_1	q_2	q_1	q_1	q_2	q_1
q_2	q_2	q_0	q_2	q_2	q_0
q_f	-	-	q_f	-	-



Q) State transition table:

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

3) State transition table:

cs	cs		cs	0	1
	0	1			
$\rightarrow q_0$	q_1	$q_1 q_2$	$\rightarrow q_0$	q_1	$[q_0 q_2]$
q_1	q_2	q_0	q_1	q_2	q_0
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

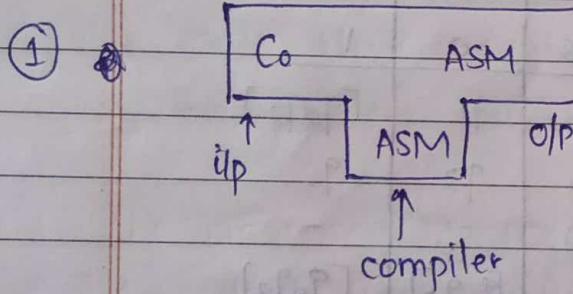
$$c(a+b+c)^*cc$$

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

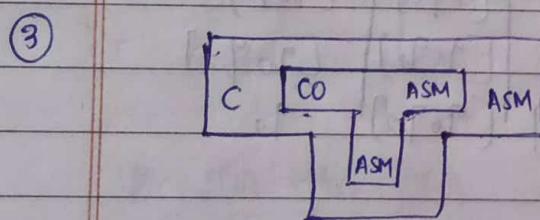
- 3) $0(11)^*$ $\rightarrow \{00, 0011, 001111, \dots\}$
- 4) Set of all strings over input alphabet $\Sigma \{0,1\}$ containing exactly one zero - i^*0i^*
- 5) $0^i1^j2^k$ where $i,j,k \geq 1$ - $0^+1^+2^+$ (or) $00^*11^*22^*$
- 6) $0^i1^j2^k$ where $i,j \geq 1$, and $k \geq 0$ - $0^+1^+2^*$ (or) $00^*11^*2^*$
- 7) $\Sigma \{0,1\}$ and language should accept all possible strings of length 4.
 $(0+1)(0+1)(0+1)(0+1)$

Boot Strapping:

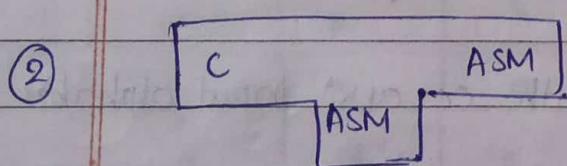
- In order to design compiler for a given high-level language, a subset of the language is selected and compiler is designed for this subset.
- This compiler is used for generating the actual compiler.



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



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



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

Tools :

LEX used in Lexical Analysis

YACC used in syntax Analysis

... → continued in next page ...



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

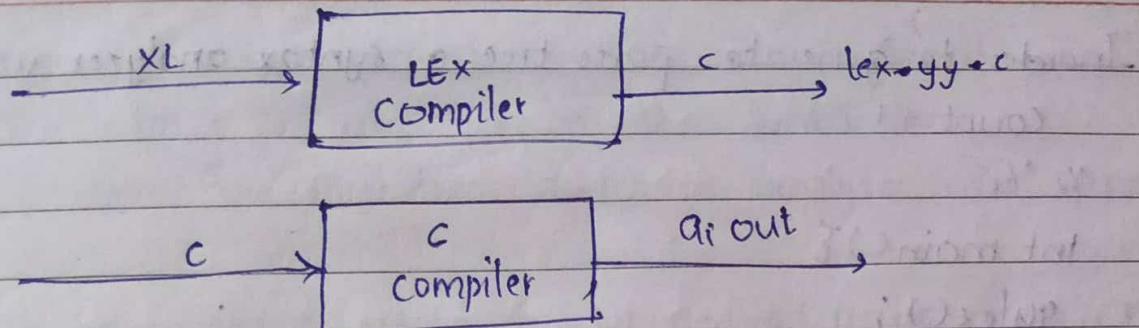
The code that is written using mnemonics is the assembly language

- 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 program
- 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 Generator 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

- 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 } *

Inorder to generate parse tree, a. syntax analyzer uses a CFG
Count ++;

Op %

int main() {

gylex();

printf ("no. of identifiers = %d \n", count);

return 0;

}

14/05/22
Monday

UNIT - II

SYNTAX ANALYSIS

Date: / /
Page No. / /

* 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 diagrammatic 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,

$$S \rightarrow OB / 1A$$

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

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

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

left most derivation:

$$\rightarrow 0OB$$

$$\rightarrow 00SB$$

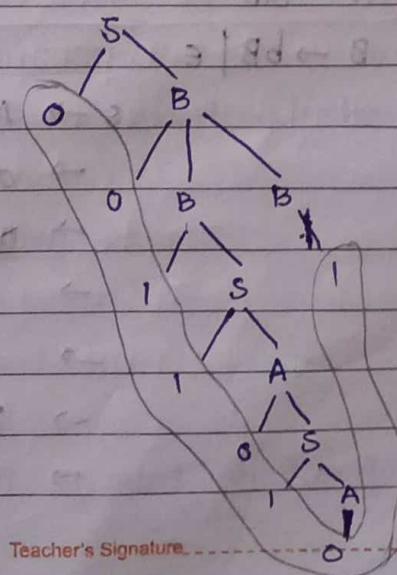
$$\rightarrow 0011AB$$

$$\rightarrow 00110SB$$

$$\rightarrow 001101AB$$

$$\rightarrow 0011010B$$

$$\rightarrow 00110101$$





Right most derivation

DATE: / /

PAGE NO.:

00110101

$S \rightarrow OB$

$\rightarrow ODBB$

$\rightarrow OOB1$

$\rightarrow OOIS1$

$\rightarrow OOIIAI$

$\rightarrow OOIIOS1$

$\rightarrow OOII01AI$

$\rightarrow OOII0101$

Eg: 2 :

$S \rightarrow aSX/b$

$X \rightarrow Xb/a$

String: aababa

$S \rightarrow asX$

$\rightarrow aasXX$

$\rightarrow aa\cancel{b}XX$

$\rightarrow aab\cancel{b}XbX$

$\rightarrow aababa$

The string cannot be derived

Eg: 3 Check whether the string can be derived or not

$S \rightarrow AA$

$A \rightarrow aB$

$B \rightarrow bB/\epsilon$

$S \rightarrow AA$

$\rightarrow aBA$

$\rightarrow ab\cancel{A} aBab$

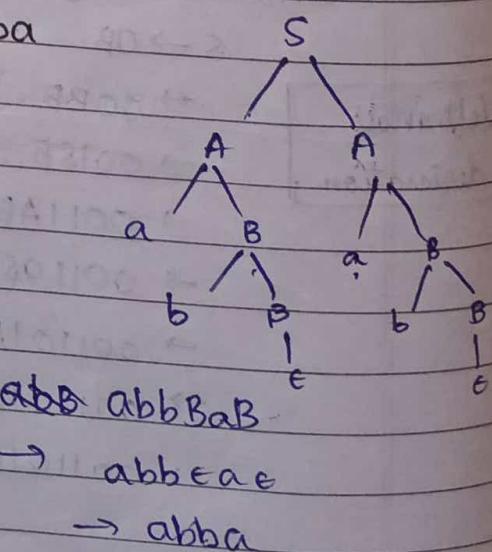
$\rightarrow abBab$

$\rightarrow ab\cancel{B} aB$

$\rightarrow ab\cancel{e} aB$

$\rightarrow ababe \rightarrow ab$

String: abba

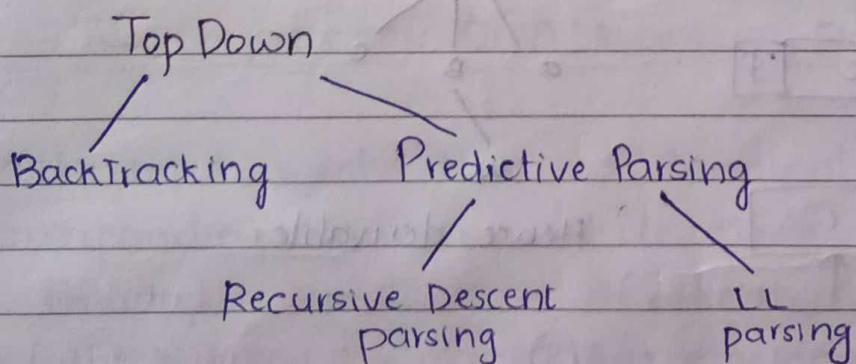


Eg. 4 :	$S \rightarrow cAd$	strings: cabd
	$A \rightarrow ababdab ablc$	cabdab
	$S \rightarrow cAd$	$S \rightarrow cAd$
	$\rightarrow cabd$	$\rightarrow cabd$ (can't derive cabdab)
Eg. 5 :	$A \rightarrow BaBc$	strings: edfa, edfffa
	$B \rightarrow dfeBf$	edbc bedffa
	$A \rightarrow Ba$	$A \rightarrow Ba$
	$\rightarrow eBfa$	\downarrow not ending
	$\rightarrow edfa$	with c edbc can't be derived
		$A \rightarrow Ba$
		$\rightarrow eBfa$
		$\rightarrow eeBffa$
		$\rightarrow eedffa$

15/08/22
Wednesday

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.

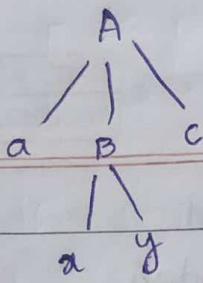
Eg:

$$A \rightarrow aBc$$

String: awc

$$B \rightarrow xy/w$$

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



* End of the string represented by "\$" symbol

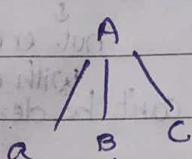
Look Ahead Pointer:

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	w	c
---	---	---

$a = a$



Advantage:

Easy to implement.

increment
pointer

a	w	c
---	---	---

$a \neq w$

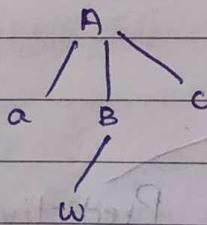
y

Disadvantage

Not at all efficient.

Hence Backtrack

a	w	c	\$
---	---	---	----



a	w	c
---	---	---

Hence derivable

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.

Procedure

A()

{

choose any production

 $A \rightarrow \alpha_1, \alpha_2, \alpha_3, \dots, \alpha_k$ for ($i = 1$ to k)

{

if α_i is a non-terminalthen call procedure $\alpha_i()$;

else if

 α_i is a terminalcall match(α_i) ;

else

print(error) ;

}

}

match(α_k)

{

if ($\alpha_i = \alpha_k$)

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 LookAhead part 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.

22/03/22
Tuesday

num → Terminal

$$E \rightarrow \text{num } T$$
$$T \rightarrow * \text{num } T / \epsilon$$

DATE: / /
PAGE NO.:

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();

}

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 grammar using recursive descent parsing technique.

String 1 : 3*4\$

String 2 : 3*4+5

String 3 : 3*4+5

```

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

```

DATE: / /
PAGE NO.:

```

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

```

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

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

```

procedure SC()
{
    if la = c
    {
        match(c);
        A();
    }
    if la = d
    {
        match(d);
    }
    else
        error();
    if la = $
    {
        print("Success");
    }
    else
        error();
}

```

```

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

```

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

```

procedure A()
{
    BC();
    if la = a
    {
        match(a);
    }
    else if la = b
    {
        match(b);
    }
    if la = c
    {
        match(c);
    }
    if la = $
    {
        print("Success");
    }
    else
        error();
}

```

```

procedure BC()
{
    if la = d
    {
        match(d);
    }
    else if la = e
    {
        match(c);
        BC();
    }
    if la = f
    {
        match(f);
    }
    else
        error();
}

```

Teacher's Signature

- Tedfa accepted

LL(1) Parser:

LL(1)
 ↗
 Left most derivation
 ↘
 Left to right,
 the ip string is parsed

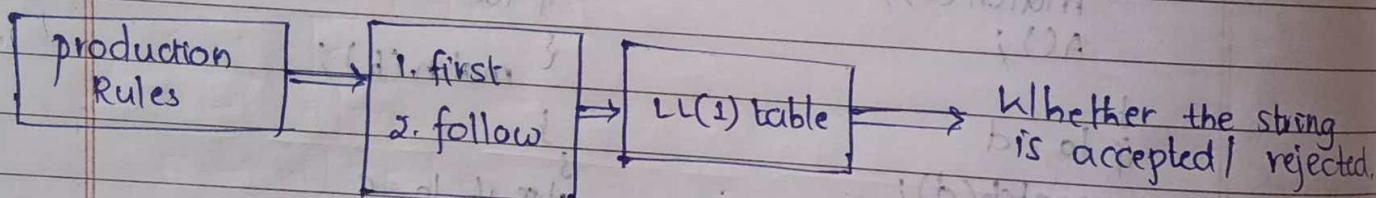
↳ only 1 char of the ip string is being parsed.

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 ip string is accepted or rejected.

The two functions are

1. First
2. Follow



Rules for "first" :

1. $\text{First}(a) = \{a\}$
2. $\text{First}(\epsilon) = \{\epsilon\}$
3. $X \rightarrow x_1 x_2 x_3 \dots x_n$
 $\text{First}(X) = \text{First}(x_1)$

Eg: $A \rightarrow B a d.$

$$\begin{aligned} \text{First}(A) &= \text{First}(B) = \\ A &\rightarrow b a d \end{aligned}$$

$$\begin{aligned} \text{First}(A) &= \text{First}(b) \\ &= \{b\} \end{aligned}$$

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 $X \rightarrow x_1 x_2 x_3 \dots x_n$, then $\text{First}(x_i)$ is included in $\text{First}(X)$

Eg. 1 $S \rightarrow aAB \mid bA \mid \epsilon$
 $A \rightarrow aAb \mid \epsilon$
 $B \rightarrow bB \mid \epsilon$

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

for productions of S

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

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

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

for productions of B

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

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

for productions of A

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

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

$\boxed{\text{First}(S) = \{a, b, \epsilon\}}$

$\boxed{\text{First}(B) = \{b, \epsilon\}}$

$\boxed{\text{First}(A) = \{a, \epsilon\}}$

Eg. 2: $S \rightarrow iCtSA/a$

$A \rightarrow eS/\epsilon$

$C \rightarrow b$

This can be

$s \rightarrow iCtSA/a$

statement if Condition then statement

or $a \rightarrow$ statement

A can be $eS \rightarrow$ else statement.

for productions of S

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

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

for productions of A

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

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

for production of C

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

$\boxed{\text{First}(S) = \{i, a\}}$

$\boxed{\text{First}(A) = \{\epsilon, \epsilon\}}$

$\boxed{\text{First}(C) = \{b\}}$



Eg: 3) $S \rightarrow ABC$

$A \rightarrow a \mid cb \mid c$

$B \rightarrow c \mid dA \mid \epsilon$

$C \rightarrow \epsilon \mid f$

$\text{First}(S) = \text{First}(ABC)$

$= \text{First}(A)$

$= \text{First}(a) \cup \text{First}(C) \cup \text{First}(\epsilon)$

$= \{a\} \cup \{e, f\} \cup \{\epsilon\}$

$= \{a, e, f, \epsilon\}$

$\text{First}(A) = \{a, e, f, \epsilon\}$

$\text{First}(B) = \{c, d, \epsilon\} \cup \text{First}(C) \cup \text{First}(\epsilon)$

$= \{c\} \cup \{d\} \cup \{\epsilon\}$

$= \{c, d, \epsilon\}$

$\text{First}(C) = \{e, f\}$

Eg: 4)

$S \rightarrow A \mid BCD$

$A \rightarrow BBA \mid EB$

$B \rightarrow bEc \mid bc \mid bdc \mid \epsilon$

$C \rightarrow c \mid \epsilon$

$D \rightarrow a \mid BDB$

$E \rightarrow a \mid bE \mid \epsilon$

$\text{First}(S) = \text{First}(A) \cup \text{First}(BCD)$

$\text{First}(A) = \text{First}(BBA) \cup \text{First}(EB)$

$= \text{First}(B) \cup \text{First}(E)$

$\text{First}(B) = \text{First}(bEc) \cup \text{First}(bc) \cup \text{First}(bdc) \cup \text{First}(\epsilon)$

$= \{b\} \cup \{b\} \cup \{b\} \cup \{\epsilon\}$

$= \{b, \epsilon\}$

$\text{First}(E) = \text{First}(a) \cup \text{First}(bE) \cup \text{First}(\epsilon)$

$= \{a, b, \epsilon\}$

$\Rightarrow \text{First}(A) = \text{First}(B) \cup \text{First}(E)$

$= \{b, \epsilon\} \cup \{a, b, \epsilon\}$

$= \{a, b, \epsilon\}$

fixes

$$\begin{aligned}
 \text{First}(A) &= \text{First}(a) \cup \text{First}(C) \cup \text{First}(\epsilon) \\
 &= \{a\} \cup \text{First}(C) \cup \{\epsilon\} \\
 &= \{a\} \cup \text{First}(e) \cup \text{First}(f) \cup \{\epsilon\} \\
 &= \{a, e, f, \epsilon\}
 \end{aligned}$$

$$\begin{aligned} \text{First}(S) &= \text{First}(A) \cup \text{First}(BCD) \\ &= \{a, b, \epsilon\} \cup \text{First}(B) \\ &= \{a, b, \epsilon\} \cup \{b, \epsilon\} \end{aligned}$$

$$= \{a, b, \epsilon\}$$

$$\begin{aligned} \text{First}(C) &= \text{First}(C) \cup \text{First}(\epsilon) \\ &= \{c\} \cup \{\epsilon\} \\ &= \{c, \epsilon\} \end{aligned}$$

$$\begin{aligned} \text{First}(D) &= \text{First}(a) \cup \text{First}(BD) \\ &= \{a\} \cup \text{First}(B) \\ &= \{a\} \cup \{b, \epsilon\} = \{a, b, \epsilon\} \end{aligned}$$

$$\text{First}(E) = \{a, b, \epsilon\}$$

$\text{First}(S) = \text{First}(A) = \text{First}(D) = \text{First}(E) = \{a, b, \epsilon\}$
$\text{First}(B) = \{b, \epsilon\}$
$\text{First}(C) = \{c, \epsilon\}$

04/04/22
Monday

$$S \rightarrow (L) | a$$

left Recursive grammar

$$\& \rightarrow L, S | S$$

$$\text{First}(L) = \text{First}(L, S) = \text{First}(L)$$

$$\text{First}(L) = \text{First}(S) = \{c, a\}$$

$$\text{First}(S) = \text{First}(CL) = \{c\} \cup \{c, a\}$$

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

cannot be determined.

- Eliminating left grammar recursion

$$A \rightarrow A\alpha | \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

$$\text{First}(L) = \text{First}(S) = \{c, a\}$$

$$\text{First}(A') = \text{First}(, SA') = \{, \} \cup \{, \epsilon\}$$

$$\text{First}(A') = \text{First}(e) = \{\epsilon\}$$

$$L \rightarrow L, S | S$$

$$L \rightarrow SA'$$

$$A' \rightarrow , SA' | \epsilon$$

$$\text{First}(L) = \{c, a\}$$

$$\text{First}(A') = \{, \}$$

$$\text{First}(S) = \{c, a\}$$

8/6

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow (E) \mid \text{id}
 \end{array}
 \text{It is Left Recursive}$$

DATE: / /
PAGE NO.:

$$E \rightarrow E + T \mid T$$

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

$$\text{First}(E) = \text{First}(T)$$

$$\text{First}(E') = \text{First}(+TE') = \{+\}$$

$$\text{First}(E') = \text{First}(E) = \{\epsilon\}$$

$$T \rightarrow T + F \mid F$$

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

$$\text{First}(T) = \text{First}(F)$$

$$\text{First}(T') = \text{First}(+FT') = \{+\}$$

$$\text{First}(T') = \text{First}(\epsilon) = \{\epsilon\}$$

$$F \rightarrow (E) \mid \text{id}$$

$\text{First}(F) = \{\text{, id}\}$
$\text{First}(T) = \{\text{, id}\}$
$\text{First}(E) = \{\text{, id}\}$
$\text{First}(E') = \{+, \epsilon\}$
$\text{First}(T') = \{* , \epsilon\}$

$$\text{First}(F) = \text{First}((E)) = \{\epsilon\}$$

$$\text{First}(F) = \text{First}(\text{id}) = \{\text{id}\}$$

07/04/22
Thursday

Rules for Follow

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

Eg. ①

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

$$A \rightarrow aAb \mid \epsilon$$

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

$$\text{Follow}(S) = \{\$\}$$

$$S \rightarrow aAB$$

First rule - Include "\$" in $\text{Follow}(S)$

$$\text{Follow}(A) = \{b\}$$

$$\text{Follow}(B) = \{b\}$$

$$\text{Follow}(S) = \{\$\}$$



$S \rightarrow aAB$

DATE: / /

PAGE NO.:

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

\therefore follow(β) should be included in Follow(B) except

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

First(B) = b

Follow(A) = { b }

$S \xrightarrow{\frac{a}{\alpha B}} aAB$ 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 $S \xrightarrow{\frac{b}{\alpha B}} bA$ in the form $A \rightarrow \alpha B$

\therefore Follow(s) included in Follow(A)

Follow(A) = { $b, \$$ }

For the production $A \rightarrow \alpha Ab$

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

Follow(A) = { $b, \$$ }

For the production $B \xrightarrow{\frac{b}{\alpha B}} bB$

Follow(B) included in Follow(B)

Follow(s) = { $\$, \beta$ }

Follow(A) = { $b, \$$ }

Follow(B) = { $\$$ }

Eg: ②

$$\begin{aligned} S &\rightarrow iCTSA/a \\ A &\rightarrow eS/e \\ C &\rightarrow b \end{aligned}$$

DATE: / /

PAGE NO.:

$$\text{Follow}(S) = \{\$\}$$

$$S \rightarrow \frac{iCTSA}{\alpha \overline{B} \beta}$$

$$\text{first}(tSA) = \text{first}(t) = t$$

$$\boxed{\text{Follow}(C) = \{t\}}$$

and

$$S \rightarrow \frac{iCTSA}{\alpha \overline{B} \beta}$$

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

$$\boxed{\text{Follow}(S) = \{\$, e\}}$$

$$S \rightarrow \frac{iCTSA}{\alpha \overline{B}}$$

$$\text{Follow}(S) \text{ included in } \text{Follow}(A)$$

$$\boxed{\text{Follow}(A) = \{\$, e\}}$$

$$A \rightarrow \frac{eS}{\alpha \overline{B}}$$

$$\text{Follow}(A) \text{ included in } \text{Follow}(S)$$

$$\text{Follow}(S) = \{\$, e\}$$

$$\text{Follow}(A) = \{\$\}$$

$$\text{Follow}(C) = \{t\}$$

Eg: ③

$$S \rightarrow ABC$$

$$A \rightarrow a|Cb|e$$

$$B \rightarrow c|dA|e$$

$$C \rightarrow .ef$$

$$\text{Follow}(S) = \{\$\}$$

$$S \rightarrow \frac{ABC}{\alpha \overline{B} \beta}$$

$$\text{first}(C) = \text{First}(e) \cup \text{First}(f)$$

$$\text{Follow}(B) = \{e, f\}$$

$$S \rightarrow \frac{ABC}{\alpha \overline{B} \beta}$$

$$\text{first}(\beta) = \text{First}(B) = \text{First}(c) \cup \text{First}(d) = \{c, d\} \text{ included in follow}(A)$$

$$S \rightarrow \frac{ABC}{\alpha \overline{B}}$$

$$\text{Follow}(S) \text{ included in } \text{Follow}(C)$$

$$\text{Follow}(C) = \{\$\}$$

~~ABC~~ / CB

$$B \rightarrow \frac{dA}{\alpha \overline{B}}$$

$$\text{Follow}(B) \text{ included in } \text{Follow}(A)$$

$$\text{Follow}(A) = \{e, f\}$$

$$\text{Follow}(B) = \{e, f\}$$

$$\text{Follow}(C) = \{\$\}, b\}$$

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

$$\text{Follow}(S) = \{\$\}$$

Teacher's Signature

08/04/22
Friday

DATE: / /
PAGE NO.:

Rules for Construction of LL(1) Table:

For $A \rightarrow \alpha$

All the terminals are columns and Non-Terminals are rows.

1. For each "a" in $\text{First}(\alpha)$

$$M[A, a] = A \rightarrow \alpha$$

2. If $\text{First}(\alpha) = \epsilon$, for each "b" in $\text{Follow}(A)$

$$M[A, b] = A \rightarrow \alpha$$

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	$S \rightarrow aAB$			
A		$b \mid (\epsilon)$		
B				
$S \rightarrow aAB \mid bA \mid \epsilon$				
$A \rightarrow aAb \mid bA \mid \epsilon$				
$B \rightarrow bB \mid \epsilon$				

	a	b	\$
S	$S \rightarrow aAB$	$S \rightarrow bA$	$S \rightarrow \epsilon$
A	$A \rightarrow aAb$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B		$B \rightarrow bB$	$B \rightarrow \epsilon$



Eg. ②
 $S \rightarrow iCtSA | a$
 $A \rightarrow eS | e$
 $C \rightarrow b$

DATE: / /
 PAGE NO.:

i	t	a	e	b	\$
S	$S \rightarrow iCtSA$	$S \rightarrow a$			
A			$A \rightarrow eS$	$A \rightarrow e$	
C				$C \rightarrow b$	

Eg. ③
 $E \rightarrow E + T | T$
 $T \rightarrow T * F | F$
 $F \rightarrow (E) | id$

~~Grammatical Error~~

Eliminating Left Recursion

$A \rightarrow A\alpha | \beta$
 $A \rightarrow \beta A'$
 $A' \rightarrow \alpha A' | \epsilon$

$E \rightarrow E + T | T$
 $E \rightarrow TE'$
 $E' \rightarrow +TE' | \epsilon$

$T \rightarrow T * F | F$

$T \rightarrow FT'$
 $T' \rightarrow +FT' | \epsilon$
 $F \rightarrow (E) | id$

and

Follow(E) = { \$ }
 $E \rightarrow \frac{+TE'}{\alpha \beta \bar{\beta}}$

first(E') = first(+TE') = { + } $\cup \{ \epsilon \} = \{ + \}$

Follow(T) = { + }
 $E \rightarrow \frac{TE'}{\alpha \bar{\beta}}$

follow(E) included in follow(E')

Follow(E') = { \$ }
 $E' \rightarrow \frac{+TE'}{\alpha \bar{\beta} \bar{\beta}}$

First(E') = { +, \epsilon } included in Follow(T)

Follow(T) = { + }

$E \rightarrow \frac{+TE'}{\alpha \bar{\beta}}$ Follow(E) included in follow(E')

$E \rightarrow \frac{+TE' + TE'}{\bar{\beta} \bar{\beta}}$ follow(E) included in follow(T)

Follow(T) = { +, \$ }

Top	Input Character	Action to be taken
\$	\$	Accept
a	a	Pop and advance the lookahead pointer
a	A/B	Error
A	a	Refer the LL(1) table entry for [A,a] and if A → PQR, then pop A and push PQR
A	a	Refer the LL(1) table for the entry [A,a] and report entry if the entry is error

12/04/22
Tuesday

Check whether the given string id + id * id is accepted by LL(1) grammar or not

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Stack	Input	Action
E \$	id + id * id \$	push T (E → TE')
TE' \$	id + id * id \$	pop T and push FT'
FTE' \$	id + id * id \$	pop F and push id
idTE' \$	id + id * id \$	pop id and advance the lookahead pointer

Teacher's Signature.



Top	Input Character	Action to be taken
\$	\$	Accept
a	a	Pop and advance the lookahead pointer
a	A/B	Error
A	a	Refer the LL(1) table entry for [A,a] and if $A \rightarrow PQR$, then pop A and push PQR
A	a	Refer the LL(1) table for the entry [A,a] and report entry if the entry is error

12/04/22
Tuesday

Check whether the given string id+id * id is accepted by LL(1) grammar or not

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (\epsilon) \mid id$$

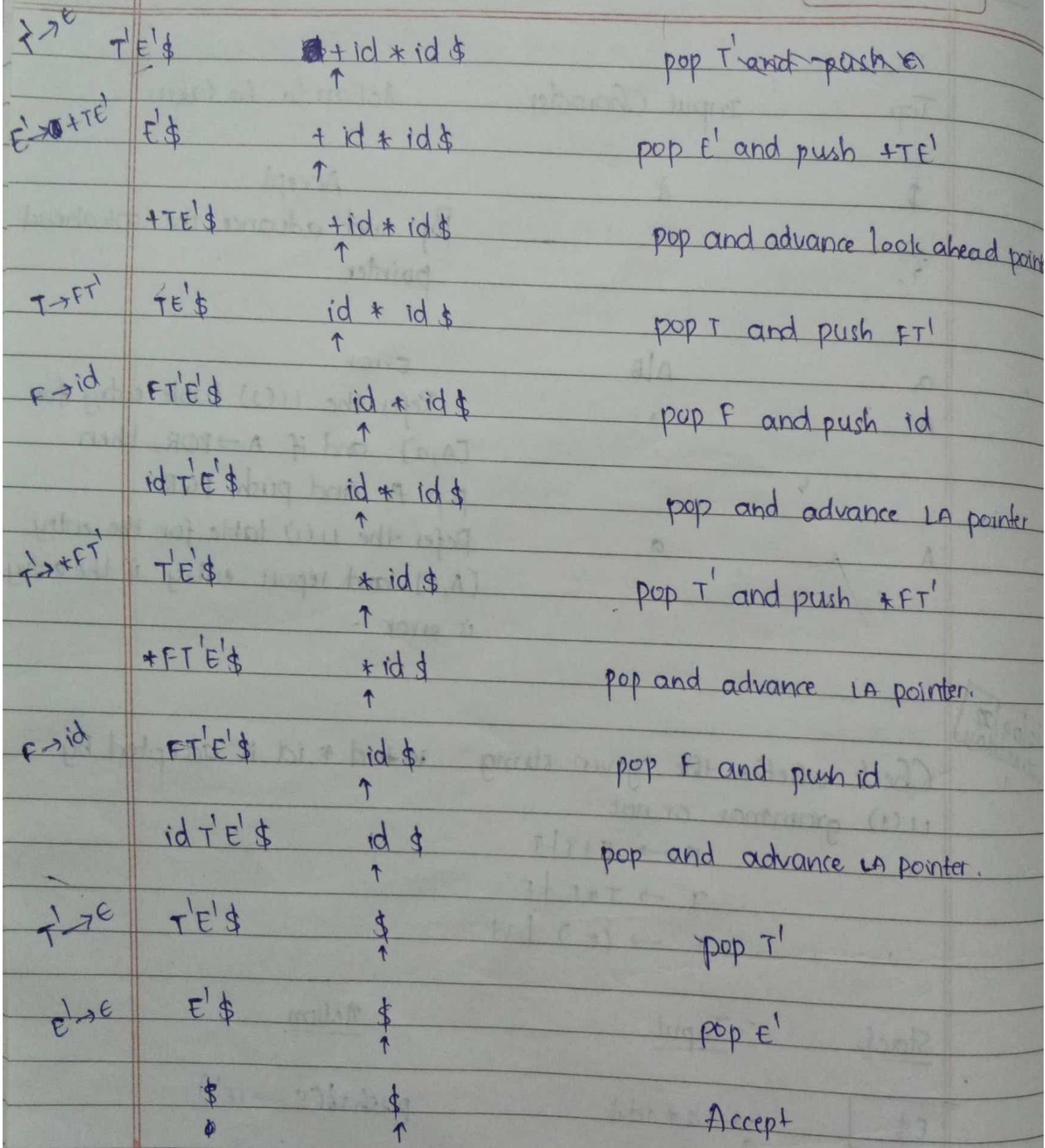
Stack Input Action

E\$ id+id * id\$ push T (E \rightarrow TE')

T \rightarrow FT' TE'\$ id+id * id\$ pop T and push F T'

F \rightarrow id FTE'\$ id+id * id\$ pop F and push id

id T E' \$ id+id * id\$ pop id and advance the lookahead pointer



Q.

aabb is accepted by

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

$$A \rightarrow aAb \mid \epsilon$$

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

DATE: / /

PAGE NO.:

Table: a b \$

S	$S \rightarrow aAB$	$S \rightarrow bA$	$S \rightarrow \epsilon$
A	$A \rightarrow aAb$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B	-	$B \rightarrow bB$	$B \rightarrow \epsilon$

	Top of the Stack	Input	Action
$S \rightarrow aAB$	$S\$$	$aabb \$$	pop 'S' and push aAB
$aAB\$$	$aabb \$$		pop and advance LA pointer.
$AB\$$	$abb \$$		pop A and push aAB
$aABB\$$	$abb \$$		pop and advance LA pointer.
$ABB\$$	$bb \$$		pop A
$B \rightarrow bB$	$BB\$$	$bb \$$	pop B and push bB
$BBB\$$	$bb \$$		pop and advance LA pointer

18/09/22
Monday

BOTTOM UP PARSING TECHNIQUES

DATE: / /
PAGE NO.:

1. Shift Reduce Parsing Technique
2. LR parsing
3. Operator precedence

Shift Reduce Parsing Technique :

* In this technique, there are 4 different actions

1. Shift : Moving the symbols from input buffer onto the stack.
2. Reduce : If the RHS of the rule is on the top of the stack, then reduce it by its corresponding LHS.
3. Accept : If the stack contains start symbol only and the input buffer contains '\$' only, then accept the string
4. Error : A condition where the parser cannot perform any of the above 3 actions.

Eg:

	<u>Stack</u>	<u>Ip Buffer</u>	<u>Action</u>
$E \rightarrow E - E$			
$E \rightarrow E * E$	\$	id - id * id \$	
$E \rightarrow id$	id \$	id - id + rd \$	shift "id"
String: id - id * id	E \$	- id * id \$	Reduce (E → id)
	- E \$	id * id \$	shift "-"
	id - E \$	* id \$	shift "id"
	E - E \$	* id \$	Reduce (E → id)
	E \$	* id \$	Reduce (E → E-E)
	* E \$	id \$	shift "*"
	id * E \$	id \$	shift "id"
	E * E \$	id \$	Reduce (E → id)
	E \$	id \$	Reduce (E → E+E)
	* E \$	\$	shift "*" Accept



$S \rightarrow cAd$
 $A \rightarrow abc$
 Check for the string acceptance of ccd using Shift Reduce
 Parsing technique.

PAGE NO.:

stack	I/P Buffer	Action
\$	ccd\$	Shift "c"
c\$	cd\$	Reduce $(A \rightarrow c)$ shift "c"
A\$	cd\$	Reduce $(A \rightarrow c)$
cA\$	d\$	Shift "c"
dA\$	\$	Shift "d"
dA\$	\$	Reject

2. LR Parsing Technique

- * It is a left to right scanning technique.
- * It is right most derivation.

Steps followed in LR parsing :

Step-1: Formation of Augmented Grammar (G')

Step-2: Compute closure and GOTO functions

Step-3: Finding First and Follow for all the non-Terminals.

Step-4: Construction of the parse table.

* Augmented grammar can be formed by adding an additional production for the given grammar
 $s' \rightarrow s$

* CLOSURE

* Initially every item will be in I_0 and it will be in closure CLOSURE(I_0).

GOTO

2. If there is a production of the form $A \rightarrow \alpha \cdot B\beta$ in closure(I) and $B \rightarrow \gamma$ is a production, then $B \rightarrow \gamma$ is also added to CLOSURE(I)

GOTO

$GOTO(I, X)$ will be :

1. If there is any production $A \rightarrow \alpha \cdot X\beta$ in I , then

DATE: / /
PAGE NO.:

$A \rightarrow \alpha x \cdot \beta$ should be in $\text{GOTO}(I, x)$

* If there is a production