21/2/22.

Compiler Design.

Unit - 1.

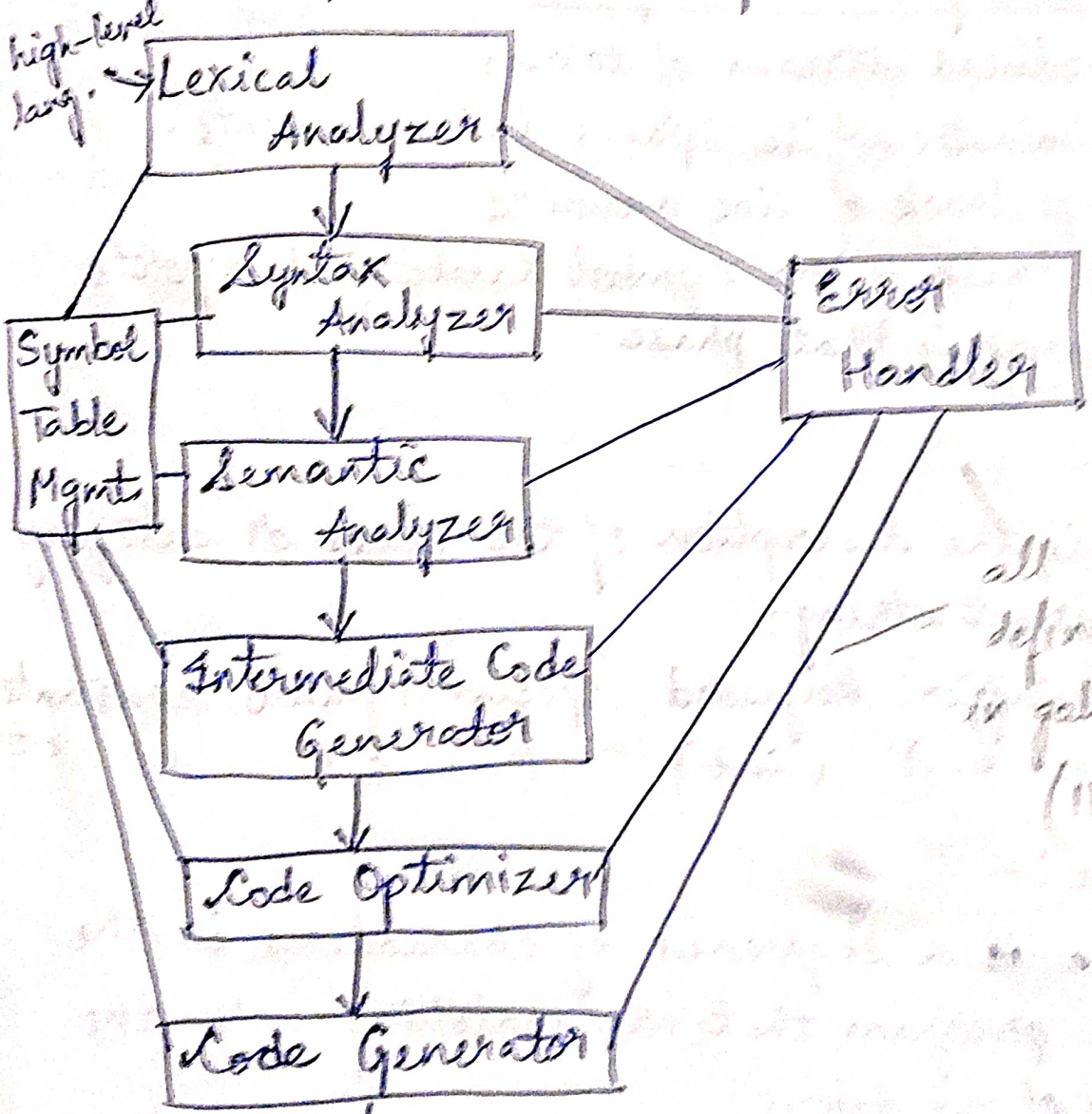
Translator:-

Converts source language to target language.

Compiler:-

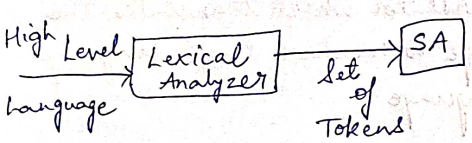
Compiler is a translator which translates the high level language to machine level language or low-level language.

There are 6 phases in a compiler:



The first 3 phases are known as Analysis phase and the last 3 phases are known as Synthesis phase.

Lexical Analyzer:-



Functions of Lexical Analyzer:-

* It produces stream of tokens.
* It eliminates white spaces and comments.
* It keeps track of line numbers.
* Keeps track of the symbol table and reports any errors in that phase.

Token:-

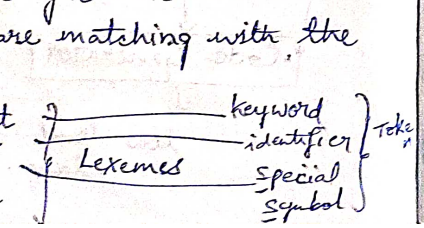
Token is the description of the class or category of the input string.

ex: Identifier, keyword, special symbol, constants etc
(a,b,c,..) (int) (;,$)

Lexeme:-

Lexeme is a sequence of characters in the source program that are matching with the pattern of the token.

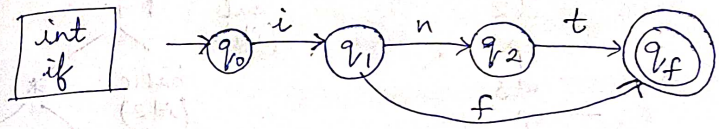
ex: int a,b,c; ⇒



Pattern — It is a set of rules that describe the token.

To design a Lexical Analyzer, we can use Finite State Machine (FSM).

ex:



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.

ex:

① float sal, hra, basic;

Lexemes	Tokens
float	keyword
sal	identifier
,	SS
hra	id
basic	id
;	SS

② sal = basic + hra * 2;

Lexemes	Tokens
sal	id
=	SS (operator)
basic	id
+	SS (operator)
hra	id
*	SS (operator)
2	constant (digit)
;	SS

③ int rno, age;

Lexemes	Tokens
int	Keyword
rno	id
,	SS
age	id
;	SS

④ float cgpa, percentage;

Lexemes	Tokens
float	keyword
cgpa	id
,	SS
percentage	id
;	SS

⑤ cgpa = percentage/10;

Lexemes	Tokens
cgpa	id
=	operator
percentage	id
/	operator
10	constant (digit)
;	SS

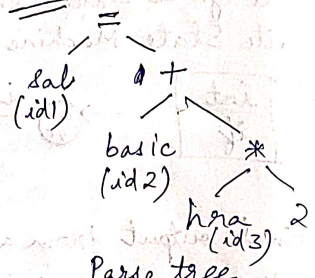
ex: sal = basic + hra * 2;
For this example, in each phase,

LA.

sal — id1
= — op
basic — id2
+ — op
hra — id3
* — op
2 — constant
; — SS

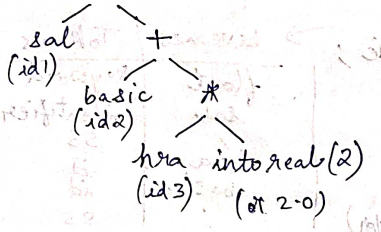
(Assigning no.s to identifiers)

SA.



Parse tree
CFG

Sem A.



ICG

Temp1 = intoreal(2)
Temp2 = hra (id3) * Temp1
Temp3 = basic (id2) + Temp2
sal (id1) = Temp3

CO.

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

CG.

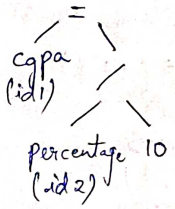
MOV id3, R1	# id3 → R1	
MUL R1, #2.0	# R1 = id3 * 2.0	
MOV id2, R2	# id2 → R2	
ADD R1, R2	# R1 + R2 → R1	(R1 = id2 + (id3 * 2.0))
MOV R1, id1	# R1 → id1	

ex: cgpa = percentage / 10;

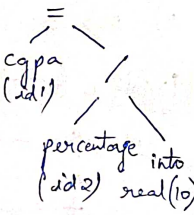
LA.

cgpa — id1
= — op
percentage — id2
/ — op
10 — constant
; — SS

SA.



Sem A.



ICG.

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

CO.

~~Temp1~~ = id2 / intoreal(10)
id1

CG.

MOV id2, R1	# id2 → R1
DIV R1, #10.0	# R1 / 10.0 → R1
MOV R1, id1	# R1 → id1

Q. Try to analyze given statements in each & every phase of compiler.

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

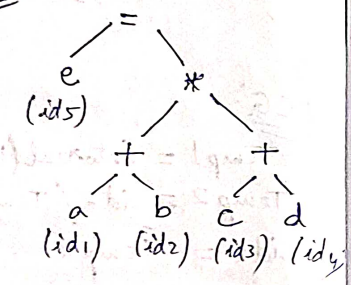
2) pos = initialamount + rate of interest * 6.5

1. soln: (a+b) * (c+d)

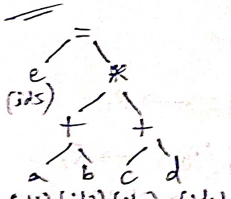
LA.

(	— ss
a	— id1
+	— op
b	— id2
)	— ss
*	— op
c	— id3
d	— id4
e	— id5.

SA.



SemA.



ICG.

```
Temp1 = d
Temp2 = c + Temp1
Temp3 = b
Temp4 = a + Temp3.
Temp5 = Temp4 * Temp2.
e = Temp5
(id5)
```

CO.

```
Temp1 = c+d      = id3+id4
Temp2 = a+b      = id1+id2
id5 = Temp2 * Temp1.
```

CG.

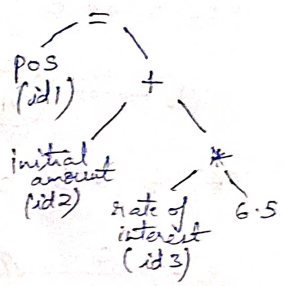
```
MOV  id3, R1      # id3 → R1
MOV  id4, R2      # id4 → R2
ADD  R1, R2       # R1 + R2 → R1
MOV  id1, R3      # id1 → R3
MOV  id2, R4      # id2 → R4
ADD  R3, R4       # R3 + R4 → R3
MUL  R1, R3       # R1 * R3 → R1
MOV  R1, id5      # R1 → id5
```

2. soln: pos = initialamount + rate of interest * 6.5

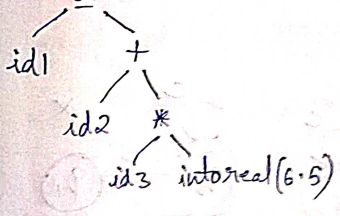
LA.

pos	— id1
=	— op
initialamount	— id2
+	— op
rate of interest	— id3
*	— op
6.5	— constant.

SA.



SemA.



ICG.

```
Temp1 = intoreal(6.5)
Temp2 = id3 * Temp1
Temp3 = id2 + Temp2
pos = Temp3.
(id1)
```

CO.

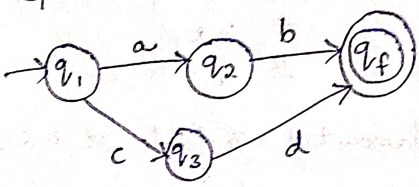
```
Temp1 = id3 * intoreal(6.5)
id1 = id2 + Temp1
```

CG.

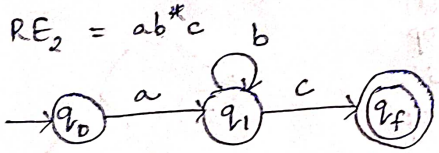
```
MOV  id3, R1      # id3 → R1
MUL  R1, #6.5     # R1 id3*6.5 → R1
MOV  id2, R2      # id2 → R2
ADD  R1, R2       # R1+R2 → R1
MOV  R1, id1      # R1 → id1
```

Convert Regular Expression to Finite Automata

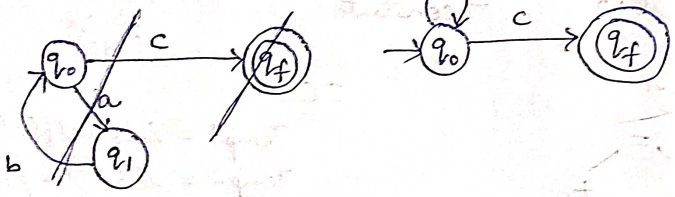
1) $RE_1 = ab + cd$



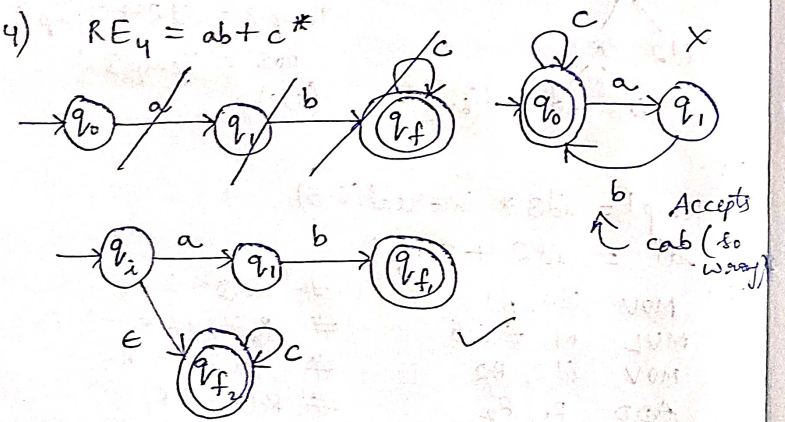
2) $RE_2 = ab^*c$



3) $RE_3 = (a+b)^*c$



4) $RE_4 = ab + c^*$



Input Buffering :-

The main functionality of the LA is to scan the source program. In the input buffering scheme, two pointers are maintained. They are:

bp ⇒ begin pointer

fp ⇒ forward pointer

(A token is checked if its valid or not using the finite automata).

* Initially both the pointers will be pointing to the same character.
* The fp will be incremented until it reaches a ~~white space~~ valid delimiter.
* 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.

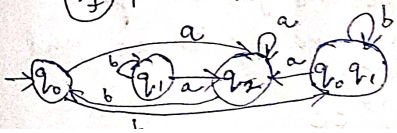
NFA to DFA conversion :-

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	-	—



2)

State	0	1
$\to q_1$	q_2	q_f
q_2	—	q_3
q_3	q_4	q_3
q_4	q_3, q_f	—
(q_f)	—	q_1

$\Rightarrow$

State	0	1
$\to q_1$	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
$\to q_0$	q_1	$q_0 q_2$
q_1	q_2	q_0
(q_2)	q_0	—

$\Rightarrow$

CS	0	1
$\to q_0$	q_1	$[q_0 q_2]$
q_1	q_2	q_0
(q_2)	q_0	—
$[q_0 q_2]$	$[q_1 q_0]$	$[q_0 q_2]$
$[q_1 q_0]$	$[q_1 q_2]$	$[q_0 q_2]$
$[q_1 q_2]$	$[q_0 q_2]$	q_0

Q:- Write Regular Expressions for the foll:

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) $00(11)^*$ — {00, 0011, 001111, ...}

4) Set of all strings over the input alphabet $\Sigma\{0,1\}$ containing exactly one '0'.
   ↳ 1^*01^*

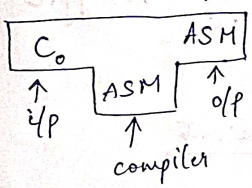
5) $0^i 1^j 2^k$ where $i, j, k \geq 1$
   ↳ $0^+1^+2^+$ (or) $00^*11^*22^*$

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

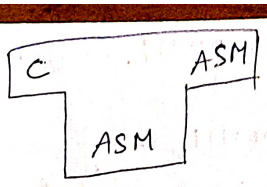
7) $\Sigma(0,1)$ and lang should accept all possible strings of length 4.
   ↳ $(0+1)(0+1)(0+1)(0+1)$

Boot Straping :-

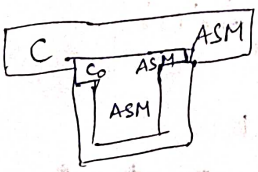
* In order to design compiler for a given high-level lang., a subset of the lang. is selected and compiler is designed for this subset.
* This compiler is used in generating the actual compiler.



C_0 is the subset of lang. C
ASM is the assembly lang. program.



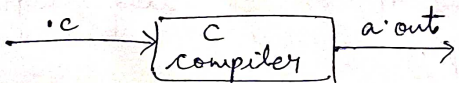
We want to generate compilers for high-level lang like C using assembly lang. compiler.



We are using the compiler for subset of C lang. (C_0) to generate the actual compiler for C.

LEX } Tools → used in Lexical Analysis
YACC } → used in Syntax Analysis.

* The programs in LEX are saved with .l extension.
* The o/p 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 .c file generated by the LEX compiler is lex.yy.c
* 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
```

ex:
```
%{
%}
%%
"Talkative"/
"Naughty" printf("III CSE4");
%%
int main()
{
  yylex();
  ~~int 0;~~ ret 0;   return 0;
}
```

ex: LEX program for counting no. of identifiers.

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

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

1) Explain various phases of a compiler with an example.
2) Explain about the functions of LA.
3) Define RE and FA.
4) Explain about LEX.
5) What is a translator? Explain about diff. translators.

↳ compilers, interpretors (considers source program line by line);
assembler (it takes assembly lang. prog. as i/p & generates the machine code as o/p).

14/3/22:

Unit-2.

Syntax Analysis

* The input to syntax analyzer is set of tokens and the output is parse tree / syntax tree.
* In order to derive the parse tree the SA uses CFG.

Derivation:-

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

CFG:-

It is a ~~portable~~ 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:-

It is the diagrammatic representation of the derivation of the given string using the CFG/ production rules.

Left most derivation:-

In the derivation process, the left most non-terminal is considered first.
Right most derivation - reverse.

Q. For the given CFG, check whether the string "00110101" can be derived or not.

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

Ex:

$S \to 0B$
$\to 00BB$
$\to 001SB$
$\to 0011AB$
$\to 00110SAB$
$\to 00110IAB$
$\to 0011010B$
$\to 00110101$

Hence can be derived.

using LMD.

using RMD.

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

Q. $S \to aSX/b$
$X \to Xb/a$ "aababa".

LMD -

$S \to aSX$
$\to aaSXX$
$\to aabXX$
$\to aabXbX$
$\to aababa$ ✓

RMD -

$S \to aSX$
$\to aSa$
$\to aaSXa$
$\to aaSXba$
$\to aaSaba$
$\to aababa$ ✓

Q. $S \to AA$
$A \to aB$
$B \to bB/\epsilon$ "abba".

LMD -

$S \to AA$
$\to aBA$
$\to abBA$
$\to abbBA$
$\to abb\epsilon A$
$\to abbaB$
$\to abba\epsilon$
$\to abba$ ✓

RMD -

$S \to AA$
$\to AaB$
$\to Aa\epsilon$
$\to aBa$
$\to abBa$
$\to abbBa$
$\to abb\epsilon a$
$\to abba$ ✓

Q. $S \to cAd$
$A \to ab/c$ "cabd", "cabdab"

LMD -

$S \to cAd$
$\to cabd$

RMD

$S \to cAd$
$\to cabd$

LMD -

$S \to \bullet cAd$
$\to cabd$
✗ CBD.

Q. $A \to Ba/bc$
$B \to d/eBf$ "edfa"
"edbc".
"eedffa".

LMD - (edfa)

$A \to Ba$
$\to eBfa$
$\to edfa$ ✓
~~$\to eBfa$~~

LMD (edbc)

$A \to Ba$
$\to eBfa$
$\to edfa$
✗ CBD

LMD (eedffa)

$A \to Ba$
$\to eBfa$
$\to eeBffa$
$\to eedffa$ ✓

Parsing Techniques

- Top Down
  - Back Tracking
  - Predictive Parsing
    - Recursive Descent Parsing
    - LL Parsing
- Bottom Up

Back Tracking :-

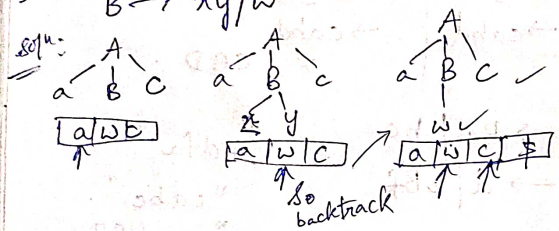
Based on Trial and Error procedure, the given input string is checked for acceptance.

Look Ahead Pointer — A pointer named LAP is used to point the next character in the given input string. Whenever the LAP reaches the end of the string ($) then the string is said to be derivable from the grammar.

ex: $A \rightarrow aBc$

$B \rightarrow xy/w$

"awc"



Adv:

Easy to implement

Dis-adv:

Not at all efficient.

Recursive Descent Parsing :-

In RDP, procedures for the non-terminal symbols that appear in the CFG are written.

```
A()
{
Choose any production
        A → X1 X2 . . . Xk
for (i=1 to k)
{ if Xk is a Non-terminal, 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);
}
```

Steps for construction of a RDP :-

1) If the input symbol is a non-terminal, then a call to the corresponding non-terminal is made.
2) If the input symbol is a terminal, then it is checked for a match with the Look Ahead from the input.
3) If the production rule has many alternatives, then they all should be combined into a single procedure.
4) The parser should be activated by a procedure corresponding to the start symbol.

Adv:

* Simple and easy to implement.

Dis-adv:

* Time taking and may enter into infinite loop.

HDFS Commands :-

1. Version check → version details of Hadoop.
2. List command →

Q. E → num T
T → * num T / ε

Generate RDP for the given grammar.

Soln:

```
Procedure E()
{
 if la = num
 {
  match(num);
   T();
 }
 else
   error();
```

```
 if la = $
   print("succes");
 else
   error();
}
```

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

Q. Check whether given strings are accepted or not using RDP technique for the above grammar.

(i) 3*4$
(ii) 3*4*$
(iii) 3*4*5

Soln: (i) 3*4$

```
Procedure E()
{ if la = 3
  {
   match(3) ✓
   T() → la = *
         { match(*)
           if la = 4
           { match(4) → T() → else NULL ($)
                                  ↳ success.
```

(ii) 3*4*$

```
Procedure E()
{ if la = 3
  { match(3) ✓
    T() → la = *
          { match(*)
            if la = 4
            { match(4)
              T() → la = *
                    if la = num ✓
                    ↳ else error
```

Hence rejected

Q. S → cAd
A → ab/c

Generate RDP for given grammar.

Soln:

```
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();
}
```

Q. A → Ba / bc
B → d / e Bf

Generate RDP for given grammar and check for "edfa" and "edbc".

Soln:

```
Procedure A()
{
 ~~if la =~~
   B();
 if la = a
   match(a);
 else if la = b
 { match(b)
   if la = c
     match(c)
 }
```

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

```
if la=$
  print("success");
else
  error();
}
```

edfa.
↳ accepted.

edbc.
↳ Not accepted ∵ since it doesn't reach b.

LL(1) Parser :-

- LMD
- Left to Right the i/p string is processed
- only 1 char of i/p string is being parsed

Production Rules
⇓
1) First
2) Follow
⟹ LL(1) Table
⇓
I/p string is accepted / rejected.

Rules for "First" :-

1) First(a) = {a}
2) First(ϵ) = {ϵ}
3) $X \to x_1 x_2 x_3 \ldots x_n$
   First(X) = First(x_1)

ex: A → Bad
First(A) = First(B)

A → bad
First(A) = First(b) = {b}

1) First of any terminal symbol is the terminal symbol itself.
2) First of an empty symbol Epsilon is Epsilon.
3) If the production is of the form $X \to x_1 x_2 \ldots x_n$ then, First of x_i is included in First of X.

① S → aAB | bA | ϵ
A → aAb | ϵ
B → bB | ϵ

sol^n: First(S) = {a, b, ϵ}
First(A) = {a, ϵ}
First(B) = {b, ϵ}
} Elaborate in exam.

② S → iCtSA | a
A → eS | ϵ
C → b

sol^n: First(S) = {i, a}
First(A) = {e, ϵ}
First(C) = {b}

③ S → ABC
A → a | ϵb | ϵ
B → c | dA | ϵ
C → e | f

sol^n:
First(S) = First(ABC)
= First(A)
= First(a) ∪ First(c) ∪ First(ϵ)
= {a} ∪ {e, f} ∪ {ϵ}
First(S) = {a, e, f, ϵ}.

First(A) = {a} ∪ {e, f} ∪ {ϵ}
= {a, e, f, ϵ}.
First(B) = {c, d, ϵ}.
First(C) = {e, f}.

(4) $S \longrightarrow A \mid BCD$
$A \longrightarrow BBA \mid EB$
$B \longrightarrow bEc \mid bc \mid bDc \mid \epsilon$
$C \longrightarrow c \mid \epsilon$
$D \longrightarrow a \mid BDb$
$E \longrightarrow a \mid bE \mid \epsilon$

Sol^n: First(S) = First(A) ∪ First(B)
= First(B) ∪ First(E) ∪ First(B)
= {b, b, b, ∈} ∪ {a, b, ∈}.
= {b, a, ∈}.

First(A) = First(B) ∪ First(E)
= {a, b, ∈}.

First(B) = {b, ∈}.
First(C) = {c, ∈}.
First(D) = {a} ∪ First(B)
= {a} ∪ {b, b, b, ∈}
= {a, b, ∈}.
First(E) = {a, b, ∈}.

(5) $S \longrightarrow (L) \mid a$
$L \longrightarrow L,S \mid S$.

Sol^n: First(S) = First(L) ∪ {a}
= First(L) ∪ First(S) ∪ {a}
First(L) ∪ First(S)
: repetition. So can't find.

Since the given grammar is left recursive, eliminate it.

$A \longrightarrow Aa \mid b \Longrightarrow LR$.
$A \longrightarrow bA' \Rightarrow$ Elimination of LR
$A' \longrightarrow aA' \mid \epsilon$].

So now the given grammar is,
$S \longrightarrow (L) \mid a$
$L \longrightarrow SL'$
$L' \longrightarrow ,SL' \mid \epsilon$

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

(6) $E \longrightarrow E+T \mid T$
$T \longrightarrow T*F \mid F$
$F \longrightarrow (E) \mid id$

Sol^n: Eliminate LR,
$E \longrightarrow TE'$
$E' \longrightarrow +E' \mid \epsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \epsilon$
$F \longrightarrow (E) \mid id$.

First(F) = {(, id}
First(E) = First(T) = First(F) = {(, id}
First(E') = {+, ∈}
First(T) = First(F) = {(, id}
First(T') = {*, ∈}.

Rules for "Follow" :-

1) "$" must be included Follow(S)
2) If $A \to \alpha B \beta$, then First(β) should be included in Follow(B) except ϵ.
3) If $A \to \alpha B$, then Follow(A) is included in Follow(B)

(or) $A \to \alpha B \beta$ and $\beta \to \epsilon$

Q.① $S \to aAB \mid bA \mid \epsilon$
$A \to aAb \mid \epsilon$
$B \to bB \mid \epsilon$

Sol":
First(B) = {b, ε}
Follow(S) = {$}
Follow(A) = {b, $}
Follow(B) = {$}

② $S \to iCtSA \mid a$
$A \to eS \mid \epsilon$
$C \to b$

Sol": i C t S A

Follow(S) = {$,
Follow(A) = {
Follow(C) = {t

iCtSA
Follow(S) = {$, e}.
Follow(A) = {
Follow(C) = {

iCtSA
Follow(S) = {$
Follow(A) = {$, e}.
Follow(C) =

∴ Finally → Follow(S) = {$, e}.
Follow(A) = {$, e}.
Follow(C) = {t}.

Rules for construction of LL(1) Table :-

For $A \to \alpha$
All the terminals are columns & non-terminals are rows.

1) For each 'a' in First(α)
$M[A, a] = A \to \alpha$
2) If First(α) = ε, for each 'b' in Follow(A)
$M[A, b] = A \to \alpha$
3) other entries are marked as "Error".
(if $ is included in set of columns).

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

Q.① $S \to aAB \mid bA \mid \epsilon$
$A \to aAb \mid \epsilon$
$B \to bB \mid \epsilon$

Sol":

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

∴ LL(1) grammar

First(α) = ε then Follow(S) = {$}
ie, $S \to \epsilon$
First(α) = ε then Follow(A) = {B, $}.
First(α) = ε then Follow(B) = {$}.

② $S \to iCtSA \mid a$
$A \to eS \mid \epsilon$
$C \to b$

Sol":

	i	t	a	e	b	$
S	S → iCtSA		S → a			
A				A → eS A → ε		A → ε
C					C → b	

Not LL(1) ← grammar.

(3) $E \to E+T / T$
$T \to T*F / F$
$F \to (E) / id$

$\longrightarrow$

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

sol":
Follow(E) = {$,)}
Follow(E') = {$,)}
Follow(T) = {+, $,)}
Follow(T') = {+, $,)}
Follow(F) = {*, +, $,)}

$E \to TE'$ and $E' \to \epsilon$ (or rule).
$\alpha B \beta$
So include Follow(E) in Follow(T).
also follow(E) in Follow(E')
My for $T \to FT'$

LL(1) Table:

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

Q. Test whether the given string id + id * id is accepted or not by the grammar.
$E \to E+T / T$
$T \to T*F / F$
$F \to (E) / id$
using LL(1) parsing technique.

Stack ⊐ (symbol)

sol":

Stack		Input	Action
$E\$$	$E \to TE'$	id + id * id $	push TE' $(E \to TE')$
$TE'\$$	$T \to FT'$	id + id * id $	push FT' $(T \to FT')$
$FT'E'\$$	$F \to id$	id + id * id $	push id
$idT'E'\$$		id + id * id $	Match of id & id
$T'E'\$$		+ id * id $	Advance look ahead pointer
$E'\$$	$E' \to +TE'$	+ id * id $	Pop T' $(T' \to \epsilon)$
$+TE'\$$		+ id * id $	Push $+TE'$
$TE'\$$	$T \to FT'$	id * id $	Advance Look Ahead
$FT'E'\$$	$F \to id$	id * id $	push FT' $(T \to FT')$
$idT'E'\$$		id * id $	Push id $(F \to id)$.
$T'E'\$$	$T' \to *FT'$	* id $	Advance Look Ahead
$*FT'E'\$$		* id $	push $*FT'$ $(T' \to *FT')$
$FT'E'\$$	$F \to id$	id $	Advance Look Ahead

id T' E' $	id $	Push id (F → id)
T' E' $	$	Advance Look Ahead
E' $	$	pop (T' → ∈)
$	$	pop (E' → ∈)
		Accept . ✓

Q. check if "aabb" is accepted or not for :

S → aAB | bA | ∈
A → aAb | ∈
B → bB | ∈

Stack		Input	Action
S $	S → aAB	a a b b $	push aAB (S → aAB)
a A B $		a a b b $	Match of a & a.
A B $	A → aAb	a b b $	Advance Look Ahead.
a A b B $		a b b $	Push aAb (A → aAb)
$ A b B $	A → ∈	b b $	Match of a & a. Advance Look Ahead
b B $		b b $	Pop (A → ∈)
B $	B → bB	b $	Advance Look Ahead
b B $		b $	~~Advance Look Ahead~~ Push bB (B → bB)
B $	B → ∈	$	Advance Look Ahead
$		$	Pop (B → ∈)
$		$	Accept . ✓

Bottom-Up Parsing Techniques :-

Shift Reduce | Operator Precedence | LR parsing .

In this technique there are 4 diff. 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 i/p buffer contains $ only then accept the string.
4) Error — A condition where the parser cannot perform any of the above 3 actions.

Q. E → E - E
E → E * E
E → id
" id - id * id " .

Stack	I/p buffer	Action
$	id - id * id $	
id $	- id * id $	Shift "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$	$	Shift "id"
E*E$	$	Reduce (E → id)
E$	$	Reduce (E → E*E)
E$	$	Accept.

Q. S → cAd
A → a|b|c
"ccd"

Sol":

Stack	I/P buffer	Action
$	ccd$	
c$	cd$	Shift "c"
A$	cd$	Reduce (A → c)
cA$	d$	Shift "c"
AA$	d$	Reduce (A → c)
dAA$	$	Shift "d"
dAA$	$	Error.

Hence the given string "ccd" is not accepted.

LR parsing technique:-

It is Left to Right scanning technique. It is RMD.

Steps followed in LR parsing:

1) Formation of augmented grammar (G').
2) Compute closure and GOTO functions.
3) Finding First and Follow for all non-terminals.
4) Construction of the Parse table.

→ Augmented grammar can be formed by adding an additional production for the given grammar. S' → S.

→ closure: Initially every item will be in I_0 and it will be in closure(I_0).
If there is a production of the form $A \to \alpha \cdot B\beta$ in closure(I) and $B \to \gamma$ is a production, then $B \to \cdot\gamma$ is also added to closure(I).

- GOTO(I, X) will be:
If there is any production $A \to \alpha \cdot X\beta$ in I then, $A \to \alpha X \cdot \beta$ should be in GOTO(I, X)