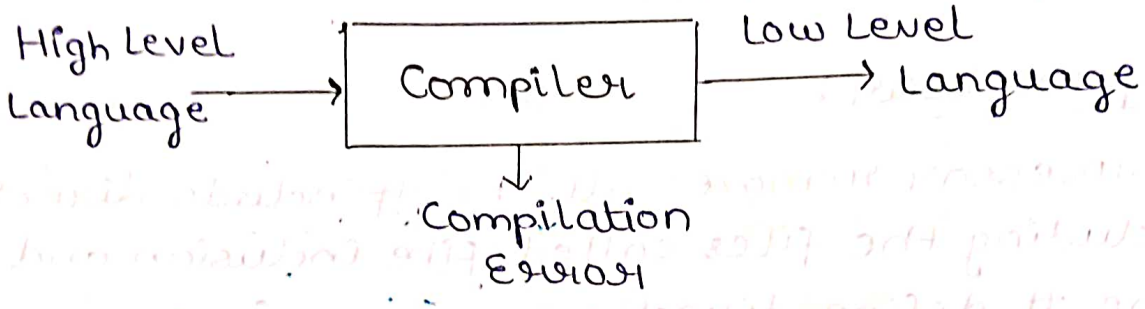
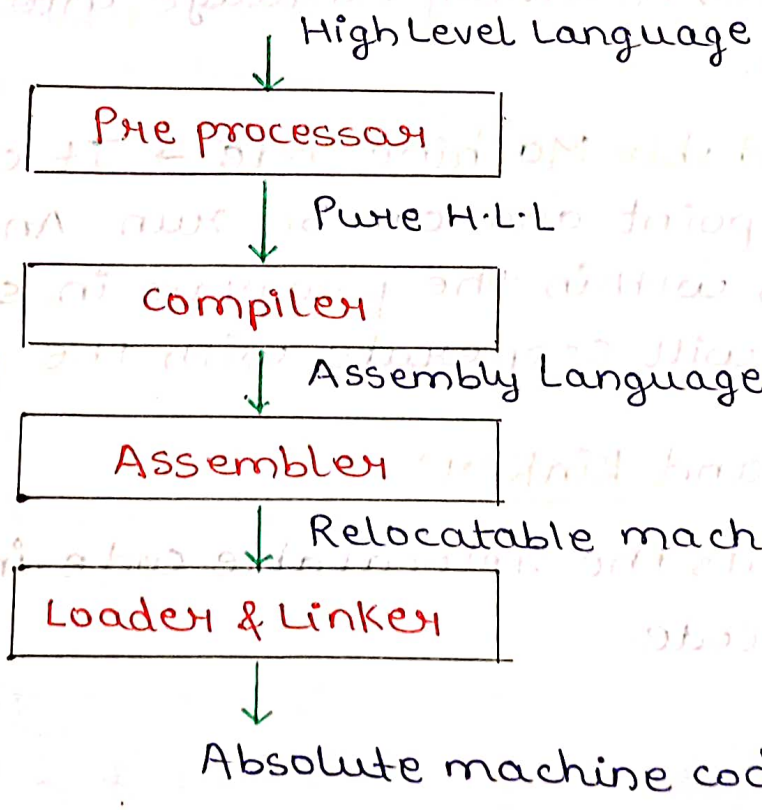
Compiler Design

UNIT 1

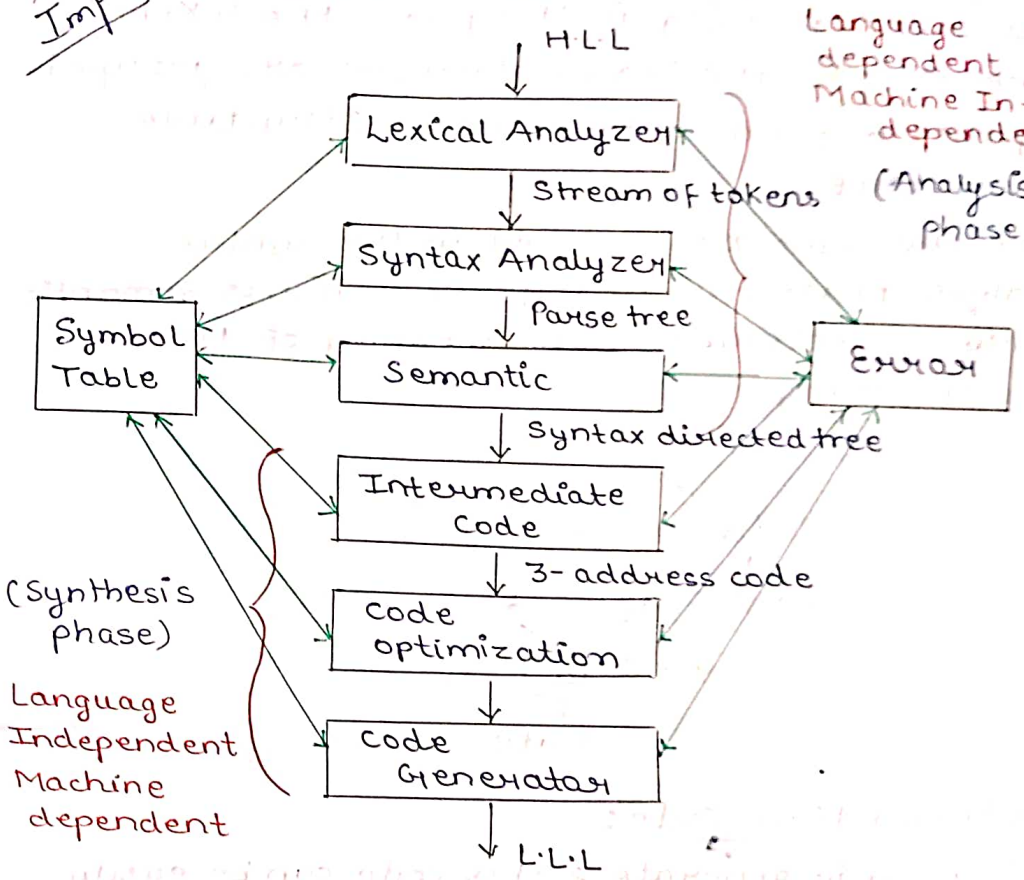


Compiler is a software that converts a program written in High Level Language into Low Level Language.

Language Processing Data:



Imp Phases of Compiler:



Lexical Analyzer:

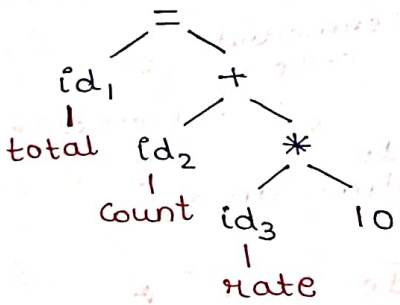
Lexical Analyzer also called Scanner. It is the phase of Compiler in which the complete source code is scanned and source program is break up into groups of tokens.

Ex: total = count + rate * 10;

$id_1 = id_2 + id_3 * 10$;

Syntax Analyzer: The Syntax Analyzer is also called parsing, in this phase the tokens generated by the Lexical Analyzer are grouped together to form a hierarchial structure called tree.

Once, the syntax is checked in the syntax Analyser phase, the next phase which is Semantic Analysis, determines the meaning of the Source string.



Intermediate Code:

It is easy to generate & this code can be easily onverted into target code.

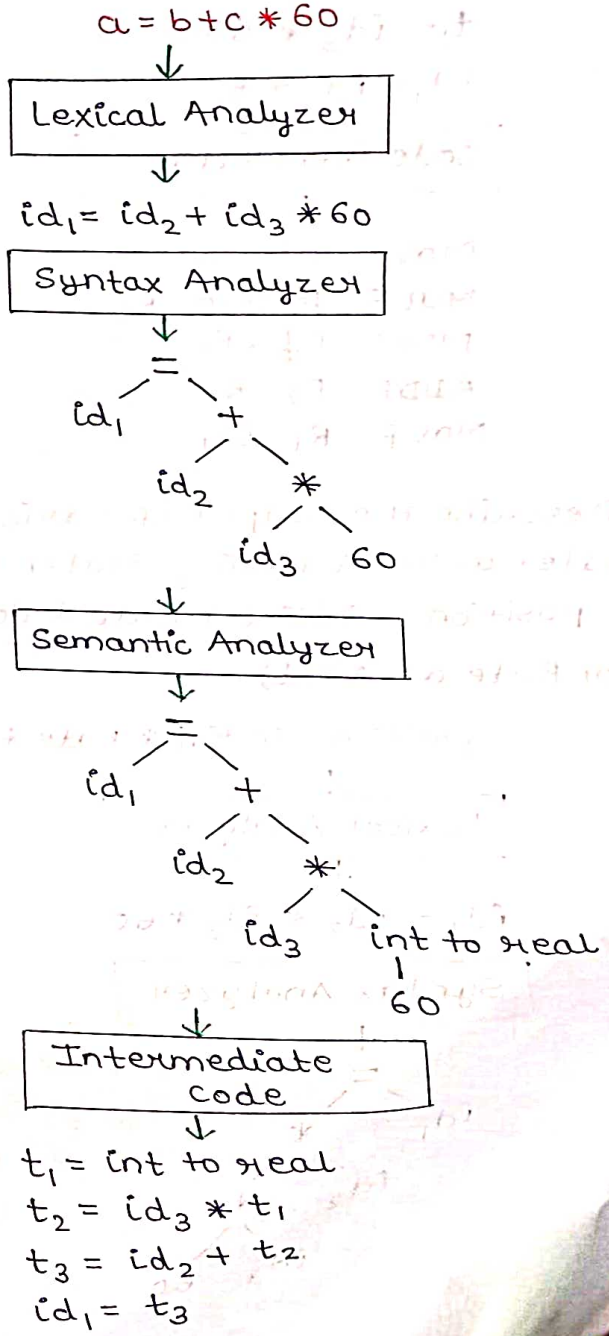
Code optimization:

he code optimization attempts to improve the ntermediate code because it is necessary to ave a faster executing code & less onsumption of memory.

Code Generation: In the code generation phase, the target code get generated the intermediate code instructions are translated into sequence of machine instructions.

Ques: How an input a = b + c * 60 get translated by using compiler? Show the output of each stage of compiler? (Consider c as real)

⇒



Code optimization

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

Code Generator

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

ues2 : Describe the output of Various phases of compiler w.r.t following statement -

position = initial * Rate * 60

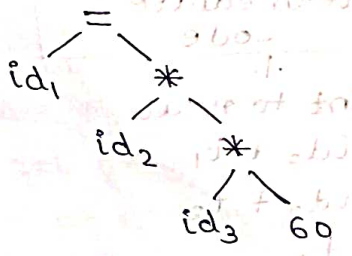
(consider Rate as real).

position = initial * Rate * 60

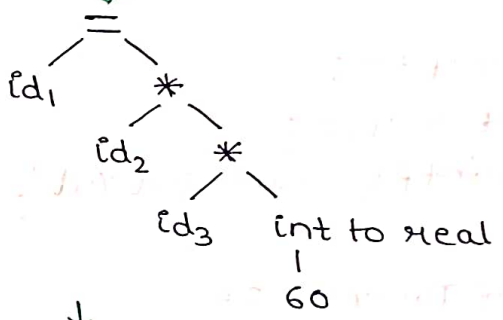
Lexical Analyzer

$id_1 = id_2 * id_3 * 60$

Syntax Analyzer



Semantic Analyzer



Intermediate Code

$t_1 = int\ to\ real$

$t_2 = id_3 * t_1$

$t_3 = id_2 * t_2$

$id_1 = t_3$

Code optimization

$t_1 = id_3 * 60.0$

$id_1 = id_2 * t_1$

Code Generator

```
MOV  R1, id3
MUL  R1, #60.0
MOV  R2, id2
MUL  R1, R2
MOV  id1, R1
```

OR

```
MOVF  id3, R1
MULF  #60.0, R1
MOVF  id2, R2
MULF  R2, R1
MOVF  R1, id1
```

Ques: Find Number of
c code:

1.
```
int main ) (
}
  x = y + z;
  int x, y, z;
  printf (" sum %d %d ", x);
{
```

⇒ No. of Tokens = 26

Lexical Analyzer task is to convert stream of character into stream of Tokens.

2.
```
main ( )
{
  a = b + + + - - - + + + = = ;
  printf (" %d %d, a, b);
}
```

⇒ No. of Tokens = 17

3.
```
main ( ) {
int a = 10 ;
char b = " abc " ;
int c = 30;
char d = " xyz " ;
}
```

⇒ No. of Tokens = 27

4.
```
int main ( )
{
  /* find Max of a and b */
  int a = 20, b = 30;
  if (a < b)
  return (b);
  else
  return (a);
}
```

⇒ No. of Tokens = 32

Types of Tokens:

There are 5 types of tokens:

1. Identifier
2. Keywords
3. Separator
4. Operator
5. Constant

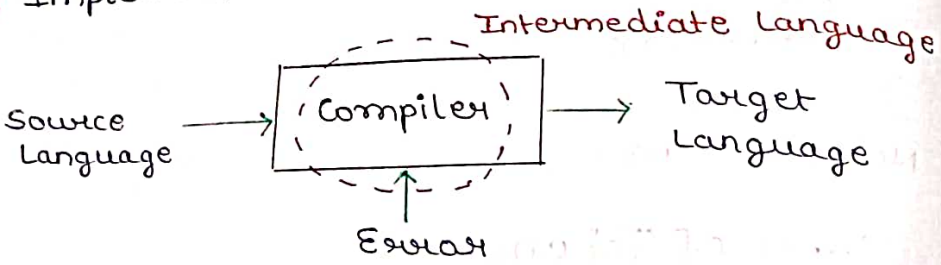
Bootstrapping:

Bootstrapping is widely used in compiler. It is a process by which simple Language is used to translate more complicated program which is in term may handle more complicated programs.

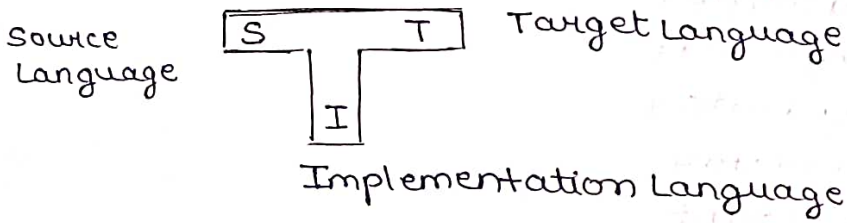
It is a type of compiler that can compile its own source code and known as self Hosting compiler.
It is used to produce a self Hosting Compiler.
It is represented by T-diagram.

Compiler can be characterized by three Language-

(a) Source Language
(b) Target Language
(c) Implementation Language

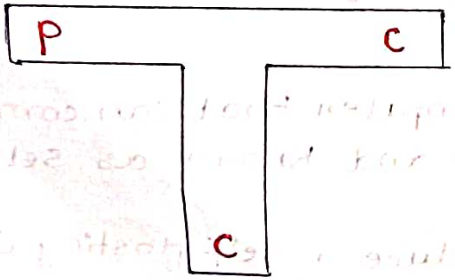


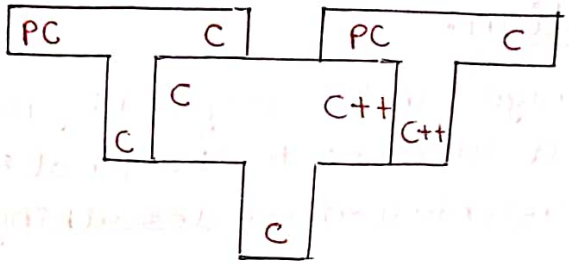
T-Diagram:



Example: Pascal Translator -

Source Language : Pascal
Target Language : C
Implementation Language : C



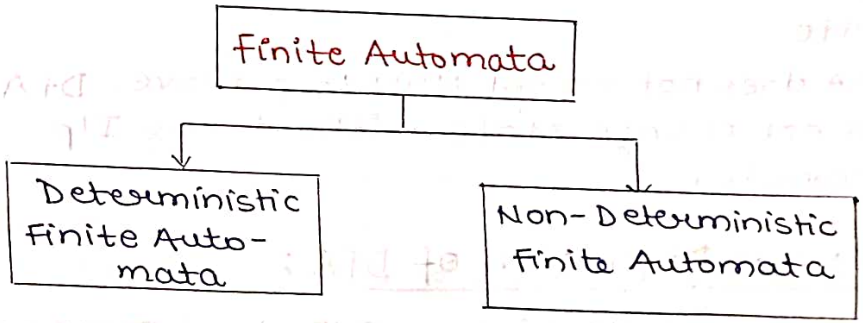


... Finite Automata:

A Finite Automata is collection of 5 Tuple $(Q, \Sigma, \delta, q_0, F)$

Q = A finite set of states
Σ = Input Alphabets indicates input set.
δ = Transition Mapping function
q_0 = Initial state (q_0 is in Q, $q_0 \in Q$)
F = Set of Final states ($F \subset Q$)

... Type of finite Automata:



Representation:

A string & language can be accepted by finite Automata when it reaches to the final state. There are 2 representation for describing Automata -

1. Transition Diagram
2. Table form or Tabular form.

Deterministic Finite Automata:

1. The Finite Automata are called DFA if the machine is read an input string one symbol at a time.
2. Deterministic refers to the uniqueness of computation.
3. In DFA, there is only one path for specific input from the current state to the next state.
4. DFA does not accept NULL ~~toop~~. move. DFA can not change state without any I/p character.

Formal Definition of DFA:

A DFA is a collection of 5 Tuples (same as Finite Automata)

Q = Finite set of states

Σ = Finite set of input

q_0 = Initial state

F = Final state

δ = Transition function ($Q \times \Sigma \to Q$)

Graphical Representation of DFA:

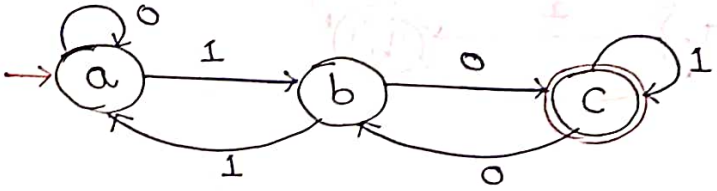
- The state is represented by vertex and the arc labled with an input character show the transition.
- The Initial state is marked with an arrow.
- The final state is denoted by double circle.

Let a DFA be -

$Q = \{a, b, c\}$

$\Sigma = \{0, 1\}$

$q_0 = \{a\}$ $F = c$



Q \ Σ	0	1
→a	a	b
b	c	a
(c)	b	c

Ques: Construct a DFA accept string starting with 'a'. [$\Sigma = \{a,b\}$]

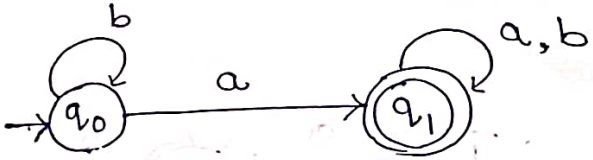
Ques: construct a DFA which accepts all strings.
(i) containing a [$\Sigma = \{a,b\}$]
(ii) End with a

Ques: Design a finite Automata which accepts ~~0 and 1~~ 101.

Ques: Design a DFA which checks that given binary number is even.

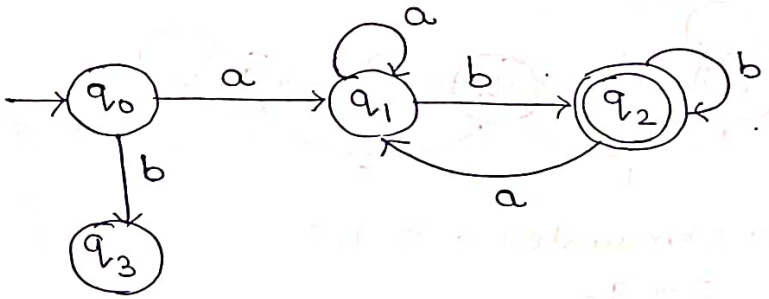
Ques: Construct the finite state machine

1.

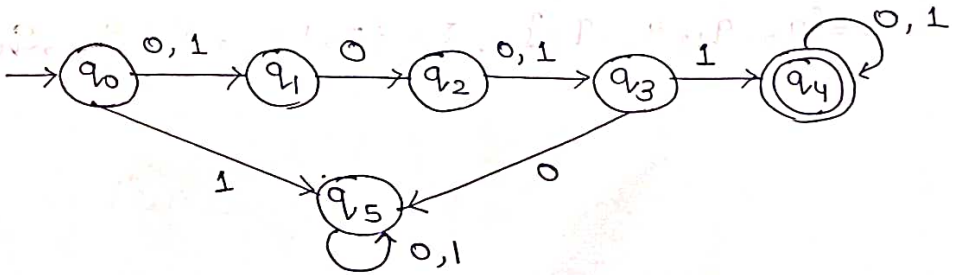


2.

Ques: Construct a DFA which accepts a Language of all string starting with 'a' and ending with 'b'.



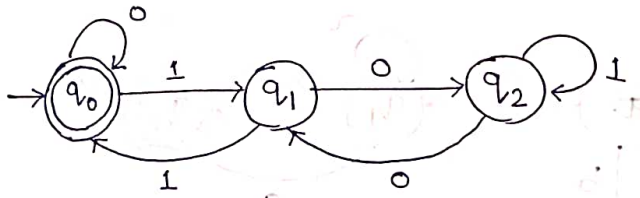
Ques: Design a DFA which accepts all strings in which 2nd symbol is 0 and 4th symbol is 1.



Ques: Construct a DFA ... of all strings not starting with 'a' and not ending with 'b'.

Ques: Construct a DFA which accepts Language of all strings (binary) divisible by 3.
[$\Sigma = \{0, 1\}$]

⇒



Possible remainder → 0, 1, 2

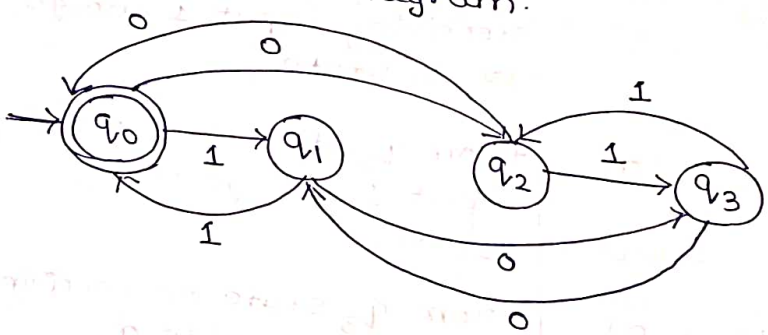
$0 \rightarrow q_0$
$1 \rightarrow q_1$
$2 \rightarrow q_2$

Ques: Consider a finite state machine in which $S = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = q_0$, $S_0 = q_0$

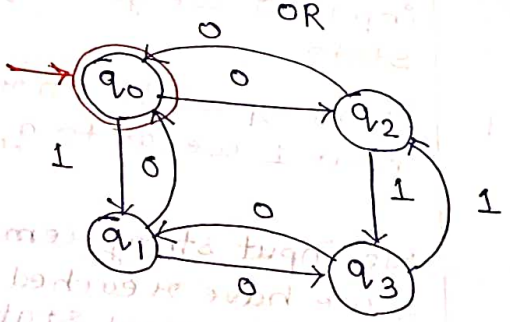
State	0	1
→ q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

(i) Draw Transition Diagram.
(ii) Give the entire sequence of input string 101101
(iii) Find out the string among the following strings which are accepted by finite automata or machine-
(a) 101101 (b) 11111 (c) 000000

⇒ (i) Transition Diagram.



OR



(ii) The entire sequence for 101101 is denoted by '⊢' Left to ⊢ symbol is state in which the input is read and right to ⊢ is the input to be processed.

Step	Processing	Meaning
1.	$q_0 \vdash 101101$	Initially be there in q_0 state read input 1 From q_0 with input 1. we go to state q_1.
2.	$1q_0 \vdash 01101$	From q_1 state on receiving input 0 we go to q_3 state.
3.	$10q_3 \vdash 1101$	From q_3 state on receiving input 1 we go to q_2 State.
4.	$101q_2 \vdash 101$	From q_2 state on receiving input 1 we go to q_3 state.
5.	$1011q_3 \vdash 01$	From q_3 state on receiving input 0 we go to q_1 state.
6.	$10110q_1 \vdash 1$	From q_1 state on receiving input 1 we go to q_0 state
7.	$101101q_0 \vdash$	No input string remaining & we have reached q_0 which is a final state. Hence, it is a valid string.

Ques: Find the no. of Tokens-

```
printf(" i=%d, si=%x ", i, si );
```

⇒ No. of Tokens = 9

Regular Language:

- Regular Expression is a method to represent a language. A language accepted by finite Automata can easily described by simple expression called Regular Expression.
- It is most effective way to represent any language. The language accepted by some R.E are referred to as Regular Language.
- A Regular Expression can also be described as sequence of pattern that defines a string.
- For Instance: In R.E x^* means zero or more occurence of x.
  x^*, we call it x closure or Kleen closure.

Operation on R.E:

Union: If 'L' and 'M' are two Regular Languages, L∪M is also Regular Language.

Intersection: If 'L' and 'M' are two Regular Languages then L∩M is also Regular.

Ques: Construct DFA for

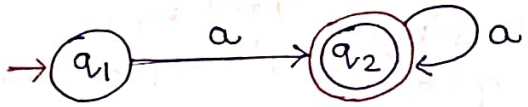
$a^* = \{\epsilon, a, aa, aaa, \ldots\}$

⇒

Ques: Construct DFA for R.E = a^+

$a^+ = \{a, aa, aaa, \ldots\}$

⇒



Methods of Generating a Language :

Let 'R' be a Regular Expression over alphabet Σ. If R is -

1. ∈ is Regular Expression denoting by the Set {∈}.
2. ϕ is Regular Expression denoting by the empty set { }.
3. Each symbol a∈Σ, a is R.E Denoting by {a}
4. Union of R.E is also regular.
5. Concatenation of two (R.E) expression is also Regular.
6. Kleen closure of R.E is also regular.

Ques: Σ = (a, b)

(1.) Length 0 (2.) Length 1 (3) Length 2
(4.) Length 3 (5) At most 1 (6) At most 2

⇒ 1) Length 0 = {∈}

2) Length 1 = {a, b}

3) Length 2 = {aa, ab, ba, bb}
= aa + ab + ba + bb
= a(a+b) + b(a+b)
= (a+b)(a+b)

4) Length 3 = (a+b)(a+b)(a+b)

5) At most 1 = (∈ + a+b)

6) At most 2 = (∈+a+b)(∈+a+b)

Ques: Consider Σ(a,b) -

1) All string having a single 'b'.
2) All string having atleast 'b'
3) All String having 'bbbb'.
4) All String end with 'ab'
5) All String begining and end with 'a'.
6) All String start with 'ba'
7) All String containing 'a'.
8) All Strings starting and ending with diff. Symbol.
9) All Strings having two 'b'.

⇒ i) R·E = a^*ba^*

ii) R·E = $(a+b)^*b(a+b)^*$

iii) R·E = $(a+b)^*bbbb(a+b)^*$

(iv) R·E = $(a+b)^*ab$

(vi) R·E = $ba(a+b)^*$

(v) R·E = $a(a+b)^*a$

(vii) R·E = $(a+b)^*a(a+b)^*$

(viii) R·E = $a(a+b)^*b + b(a+b)^*a$

9. R·E = $a^*ba^*ba^*$

Ques: Write the R·E for a language containing strings which end with abb. over $\Sigma = \{a, b\}$

⇒ R·E = $(a+b)^*abb$

Ques: Write a R·E to denote a language 'L' such that 3rd character from right end of the string is always 'a'. $\Sigma = \{a, b\}$

⇒ ~~R·E = $(a+b)^*abb$~~

R·E = $(a+b)^*a(a+b)(a+b)$

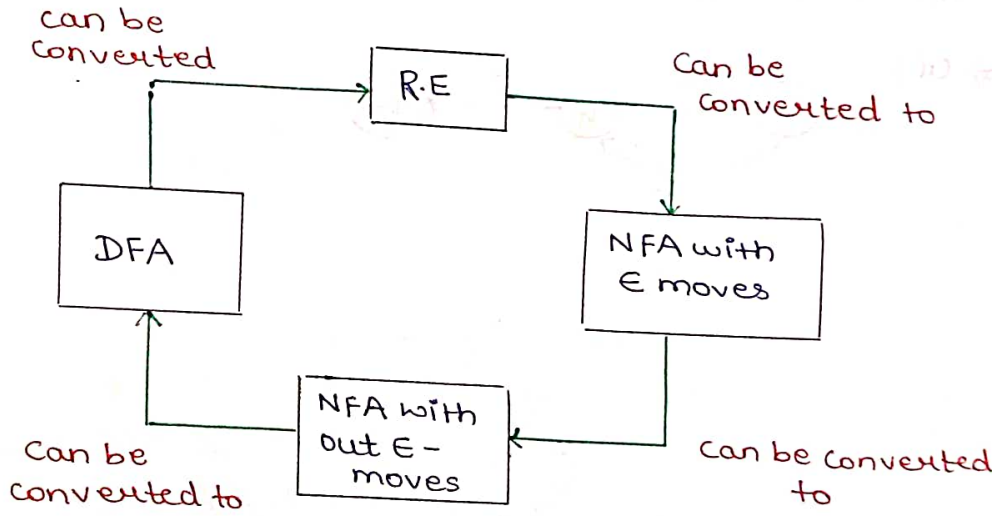
Conversion of R·E into finite Automata

1. To convert R·E to Finite Automata we use a method called subset method.

Step 1: Design a transition diagram for a given R·E using NFA with ∈ moves.

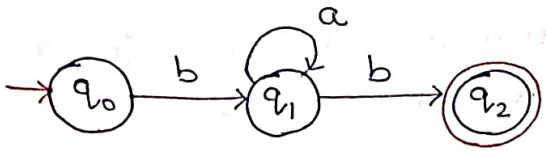
Step 2: Convert NFA with ∈ moves to NFA without ∈-moves.

Step 3: Convert the obtained NFA into equivalent DFA.



Ques: Convert the following Regular Expression to their equivalent finite Automata-

(i) ba^*b



(ii) $(a+b)c$

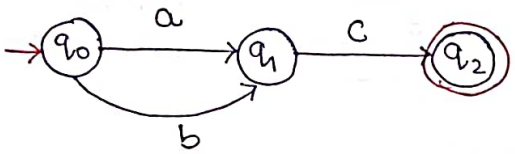
(iii) $a(bc)^*$

(iv) $10+(0+11)0^*1$

(v) $1(1^*01^*01^*)^*$

(vi) 0^*1+10

⇒ (ii)



Cross Compiler:

It is a type of compiler that generate machine code targeted to run on the different system.

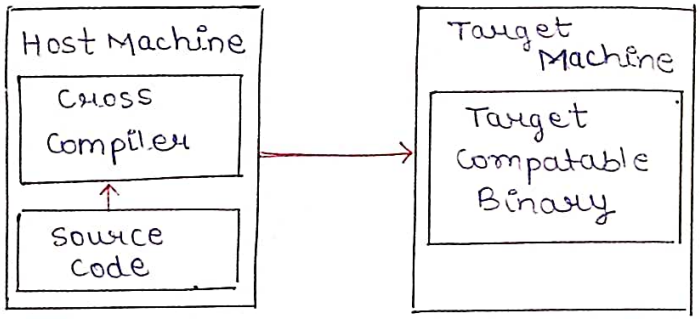
For Ex: A compiler that runs on Window platform also generate a code that runs on Linux platform.

The process of generating executable code for different machine is called Retargeting code.

Hence, the Cross compiler is also known as Retargetable compiler.

GNU, GCC are the example of Cross compiler.

Cross compiler operation

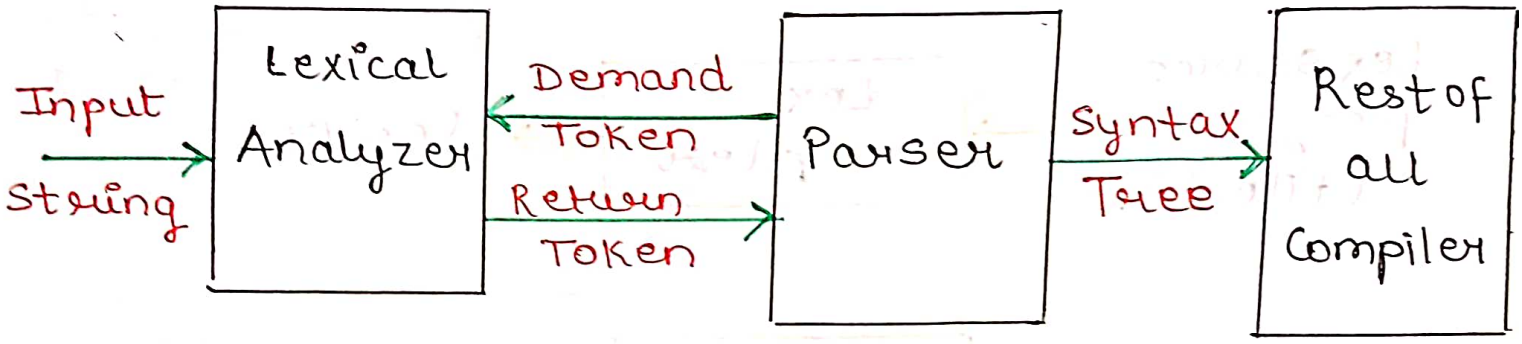


Implementation of Lexical Analyzer:

Lexical Analyzer is the first phase of compiler. The Lexical Analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens, each token is a single lexical cohesive unit such as identifiers, Keywords, operators etc.

The parser to determine the Syntax of source program can use these tokens.

The Role of Lexical Analyzer in the process of Compilation is shown below-



The task of Lexical Analyzer apart from Token generation as follows-

- It eleminates Blanks and comments.
- It generates Symbol Table, which store the information about identifiers, constant
- It keeps track of line number
- It reports the error en-counter by generating the tokens.

Lexical Analyzer Generator:

The efficient design of compiler various tools are used to automate the phases of compiler.

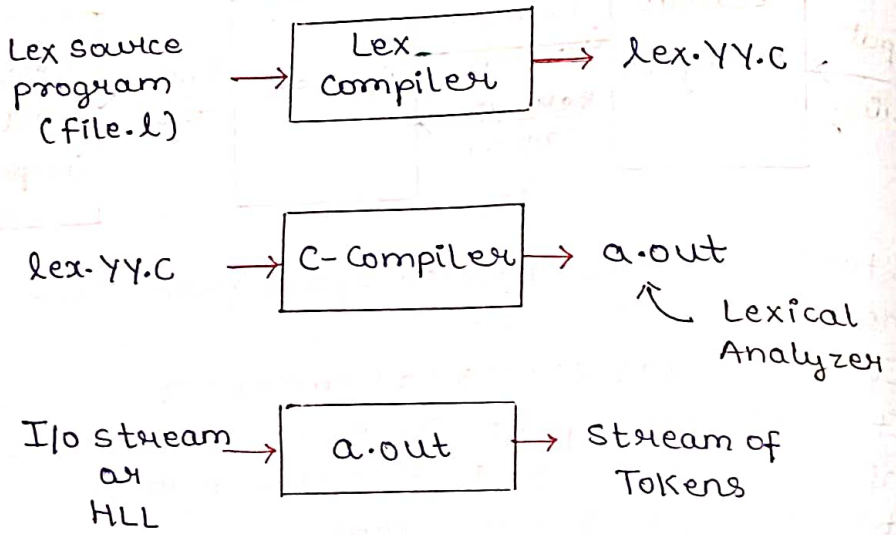
The Lexical Analyzer can be automate by using Lex tool.

Lex is a unix utility which generate lexical analyzer.

Lexical analyzer is generate with the help of R.E.

... LEX Tool in Compiler Design:

LEX is a tool which generates Lexical Analyzer. It will transfer input stream of character into Transition diagram and generate code in a file called lex.YY.C.



... Function of Lex:

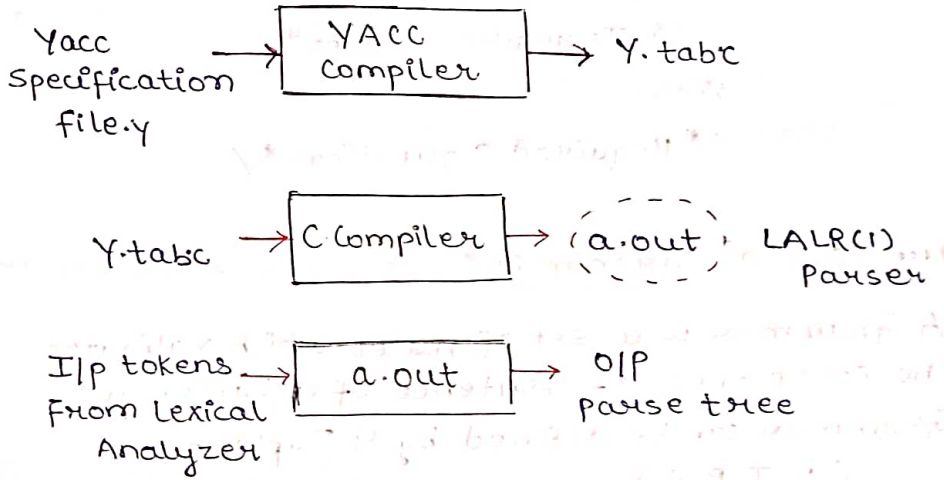
1. Source code is in Lex Language with file name (file.l). It is given to Lex Compiler which is a Lex tool which generates lex.YY.C, [it is a C program.]
2. The C Compiler runs this code (lex.YY.C) and produce an output a.out.
3. a.out transfers streams of character into stream of tokens.

... YACC: YACC stand for 'Yet Another Compiler Compiler'. It is a tool for generating Left to right Lookahead [LALR(1)].

- It takes input from Lexical Analyzer and generates parse Tree.

Syntax Analyzer / Parser:

It is second phase of compiler which takes input as token and generate parse tree.



- Input to YACC compiler will be compiled with .y extension. It will contains desired program in YACC format.
- YACC compiler will convert it into a C code & the o/p is Y.tab.c
- This Y.tab.c is given as input to C Compiler & the output will be LALR(1) parser.
- Token generated by Lexical Analyzer, using the Lex tool will be given as a.out

and we get parse

Specification of YACC:

1. Declaration section
2. Translation tools
3. Auxiliary function

Syntax:

```
%{ /* Declaration */
%}
   %%
     /* Transition rules */
   %%
   /* Required C function */
```

Formal Grammar:

A Grammar is a set of rules which validates the correctness of Sentence of a Language.
Grammar can be defined by 4 Tuples -

(V, T, P, S)

V = Variable or Non-Terminal
T = Terminal Symbol (Token)
P = Production rule
S = Start Symbol

CFG and its Application:

CFG is used to define the Syntatic structure of a programming language. It contains a set of rules called production rules.
CFG is used to check the Syntax of programming language. (Algebric Expression, if-else statement, for loop, while loop, Array representation etc.)

Ex: In English language Syntatic structure
I am going.

Ques: A CFG is given as -

$E \rightarrow E+E / E-E / E*E / E/E / (E) / id$

check whether the given sentence is a part of this grammar.

(i) id + id * id (valid)

(ii) id + + id * id (Invalid)

$E \rightarrow E+E \;(P1) \;/\; E-E \;(P2) \;/\; E*E \;(P3) \;/\; E/E \;(P4) \;/\; (E) \;(P5) \;/\; id \;(P6)$

(i) ~~$E \rightarrow id$ P6~~

$E \rightarrow E+E$ P1

$E \rightarrow E+E*E$ P3

$E \rightarrow E+E*id$ P6

$E \rightarrow E+id*id$ P6

$E \rightarrow id+id*id$ P6

∴ It is a valid Sentence

If else Grammar :

→ if expression then, statement

→ if expression then, statement
else statement

(i) Example:
Sentence -

* if (a<b) then printf("Yes") else printf("No")

S → if expression then statement else statement

→ if (a<b) then statement else statement

→ if (a<b) then printf("Yes") else ~~printf("No")~~ statement

→ if (a<b) then printf("Yes") else printf("No")

BNF Notation : (Backus-Naur Form)

Developed by John Backus and peter Naur.
It is a Notation Technique for CFG, this notation is useful for specifying the syntax of the Language. BNF specification is given below -

<Symbol> := $Exp_1 | Exp_2 | Exp_3$ - - - -

where,
<Symbol> is a non-terminal and
$Exp_1 | Exp_2 | Exp_3$ ---- and so on are the Sequence of Symbol.

This symbol can be combination of Terminal and Non-terminal.

For Example :

<Address> := <Full Name> "," <Street> "," <Zip code>

<full Name> := <first Name> "," <Middle Name> "," <Sur Name>

Commonly used Notations in BNF:

1. Optional Symbols are written within Square Brackets.
2. For repeating the Symbol for 0 or more number of time asterick (*) can used as-

$(name)^*$, $ab^* = \{a, ab, abb - -\}$

3. For repeating the symbol for atleast 1 or more occurence.

$a(bc)^+ = \{ abc, abcbc, abcbcbc - - \}$

4. The Alternative rules are separated by Vertical Bar.
5. The group of items must be enclosed within Brackets.

... Ambiguity:

A Grammar is said to be ambiguous grammar if it can produce either more than one left most derivation or more than one right most derivation.

LMD: In LMD replace the leftmost Non-terminal with a suitable production rule starting with start symbol.

Ex: $E \to E+E / E*E / id$
P1 P2 P3

Sentence: id + id * id

⇒ LMD 1
$E \to \underline{E}*E$ P2
$E \to E+E*E$ P1
$E \to id+E*E$ P3
$E \to id+id*E$ P3
$E \to id+id*id$ P3

LMD2 is ~~not~~ possible for this Grammar Therefore, the grammar that is given to us is ~~un~~ ambiguous.

LMD2
$E \to E+E$ P1
$E \to id+E$ P3
$E \to id+E*E$ P2
$E \to id+id*E$ P3
$E \to id+id*id$ P3

RMD: In RMD, replace the rightmost non-terminal with a suitable production rule starting with start symbol.

Ex: $E \to E+E / E*E / id$
P1 P2 P3

RMD 1
$E \to E+E$ P1
$E \to E+E*E$ P2
$E \to E+E*id$ P3
$E \to E+id*id$ P3
$E \to id+id*id$ P3

RMD2
$E \to E*E$ P2
$E \to E*id$ P3
$E \to E+E*id$ P1
$E \to E+id*id$ P3
$E \to id+id*id$ P3

Check whether the Grammar is Ambiguous or not:

1. G_1: $E \to E+E / E*E / id$
Sentence: ~~E+E~~
id + id * id

⇒ Solution is explained above-

∴ Given grammar is Ambiguous.

Elimination of Ambiguity:

Ambiguity can be eliminated by the following method -

1. By Defining Precedence Level of Arithmetic operation * is having more/higher Precedence than +.
2. By defining Associativity of Arithmetic operations.
   * & + both are Left associative.
   ^ is Right Associative.

Ques: check the following Grammar is Ambiguous or not.

G1: $S \rightarrow aS \mid Sa \mid a$

Sentence : aa

⇒ LMD1

	$S \rightarrow aS$	P1
	$S \rightarrow aa$	P3

LMD2

	$S \rightarrow Sa$	P2
	$S \rightarrow aa$	P3

Since, two Left most Derivation are possible for this grammar, therefore the grammar is ambiguous.

Ques: $E \to E+E \mid E-E \mid id$

Sentence: id+id-id

⇒ LMD 1

$E \to E+E$ P1

$E \to id+E$ P3

$E \to id+E-E$ P2

$E \to id+id-E$ P3

$E \to id+id-id$ P3.

LMD 2

$E \to E-E$ P2

$E \to E+E-E$ P1

$E \to id+E-E$ P3

$E \to id+id-E$ P3

$E \to id+id-id$ P3

Since, there is two LMD possible, therefore the grammar is ambiguous.

Ques: $S \to aSb \mid SS$, $S \to \epsilon$

P1 P2 P3

Sentence: aabb

⇒ LMD1

$S \to aSb$ P1

$S \to aaSbb$ P1

$S \to aa\epsilon bb$ P3

$S \to aabb$

LMD 2

$S \to SS$ P2

$S \to aSbS$ P1

$S \to aaSbbS$ P1

$S \to aa\epsilon bbS$ P3

$S \to aabb\epsilon$ P3

$S \to aabb$

Since, there exist two LMD therefore, the grammar is ambiguous.

Ques: $S \to SaS$

$S \to b$

⇒ Let Sentence: babab

LMD 1

$S \to SaS$ P1

$S \to SaSaS$ P1

$S \to baSaS$ P2

$S \to babaS$ P2

$S \to babab$ P2

LMD 2

$S \to SaS$ P1

$S \to baS$ P2

$S \to baSaS$ P1

$S \to babaS$ P2

$S \to babab$ P2

Since, there exists two LMD for same sentence therefore, the grammar is ambiguous.

.....Ambiguous to Unambiguous

We can make a grammar unambiguous if we take care of two things-

1. Associativity
2. Precedence

Ex: G: E+E / E*E / id

This grammar is both Left and right associative because we can draw our tree in any direction.

If we want to achieve Left Associativity then we have to restrict our tree to grow in Left direction only.

So, if we want operator to be Left Associative we have to make that grammar is Left recursive always.

"For the Left Associative Recursion, Left most symbol of RHS must be equal to LHS."

For Precedence problem-

We should take care that highest precedence ~~pro~~ operator should be at least Level.

So, we will introduce several different symbols like -

Once we have reached T we can not generate +

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to id$$

Ques: Change according to precedence & Associativity rule - and generate Grammar-

bExp → bExp OR bExp / Exp AND b Exp / NOT b Exp / True / False

⇒ NOT > AND > OR

$$E \to E \text{ OR } F$$
$$F \to F \text{ AND } G$$
$$G \to \text{NOT } G \mid \text{True} \mid \text{False}$$

Ques:

$$A \to A\$B \mid B$$
$$B \to B\#C \mid C$$
$$C \to C@D \mid D$$
$$D \to id$$

@ > # > $

Find Associativity and precedence of Given grammar-

⇒ Left Associative and @ has higher precedence since, it is far away from starting symbol A.

Ques

$$E \to E+E \mid E*E \mid F$$
$$F \to F-G \mid G$$
$$G \to id$$

⇒

Recursive Grammar:

A Grammar is said to be recursive if it produce itself again & again. Recursion can be of two types-

- Direct
- Indirect

Left Recursion:

A production of grammar is said to have Left recursion if the Left most variable of its RHS is same as variable of its LHS.

Ex: 1. $A \rightarrow A\alpha \mid \beta$ /* Direct Left Recursive

2. $S \rightarrow Aa$
   $A \rightarrow Sm$ /* Indirect

Ques: Why it is important to remove Left Recursion?

⇒ It is because top-down parser can not accept the grammar having Left recursion. Hence, they do not allow Left recursive grammar that's why we have to remove left recursion from the grammar.

Conversion of Left Recursive Grammar:

Ques: Eliminate Left Recursion from the given grammar -

1. $A \rightarrow A\alpha \mid \beta$ [Format 1]

⇒ $A \rightarrow \beta A'$ [Right Recursive] $= \beta\alpha^*$

$A' \rightarrow \alpha A' \mid \epsilon$

This right recursive grammar is equivalent to given Left recursive grammar.

2. $E \rightarrow E+T \mid T$

⇒ $E \rightarrow T \cdot E'$

$E' \rightarrow +T \cdot E' / \epsilon$

3. $S \rightarrow SOS1O / 01$

⇒ $S \rightarrow 01S'$

$S' \rightarrow OS1OS' \mid \epsilon$

4. $S \rightarrow (L) / x$

$L \rightarrow L,S \mid S$

⇒ $S \rightarrow (L) / x$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

$\underset{A}{L} \rightarrow \underset{A}{L},\underset{\alpha}{S} \mid \underset{\beta}{S}$

5. $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \beta_m$ [Format 2]

⇒ $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \dots \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \dots \alpha_n A' \mid \epsilon$

6. $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

$E \to \underset{A}{E}+\underset{\alpha}{T} \mid \underset{\beta}{T}$

$T \to \underset{A}{T}*\underset{\alpha}{F} \mid \underset{\beta}{F}$

$\Rightarrow E \to TE'$
$E' \to +TE' \mid \in$
$T \to FT'$
$T' \to *FT' \mid \in$
$F \to id$

Ques: Consider the following Grammar -

$A \to ABd \mid Aa \mid a$
$B \to Be \mid b$

Remove Left Recursion

$\Rightarrow A \to ABd \mid a$
$A \to Aa \mid a$
$B \to Be \mid b$

$\therefore A \to aA'$
$A' \to BdA' \mid \in$
$B \to bB'$
$B' \to eB' \mid \in$
$A' \to aA' \mid \in$

$A \to \underset{A}{A}\underset{\alpha}{Bd} \mid \underset{\beta}{a}$

$B \to \underset{A}{B}\underset{\alpha}{e} \mid \underset{\beta}{b}$

$A \to \underset{A}{A}\underset{\alpha}{a} \mid \underset{\beta}{a} \leftarrow \underset{A}{A}$

(a) $A \to AaA \mid b$

$\Rightarrow A \to bA'$
$A' \to aAA' \mid \in$

$A \to \underset{A}{A}\underset{\alpha}{aA} \mid \underset{\beta}{b}$

(b) $S \to SSS \mid 0$

$\Rightarrow S \to 0S'$
$S' \to SSS' \mid \in$

$S \to \underset{A}{S}\underset{\alpha}{SS} \mid \underset{\beta}{0}$

(c) $S \to SOS1 \mid 0 \mid 1$

$\Rightarrow S \to 0S' \mid 1S'$
$S' \to 0S1S' \mid \in$

$S \to \underset{A}{S}\underset{\alpha}{OS1} \mid \underset{\beta_1}{0} \mid \underset{\beta_2}{1}$

$A \to \beta_1 A' \mid \beta_2 A' \ldots$
$A' \to \alpha_1 A' \mid \alpha_2 A' \ldots$

(d)
$S \to AaB$
$A \to Aa \mid bB \mid a$
$B \to bB \mid c$

$\Rightarrow$ Removing ~~the~~ Left recursion from production 2

$A \to Aa \mid bB \mid a$

which can be written as

$A \to Aa \mid a$
$A \to bB \mid a$

$\therefore A \to Aa \mid a$
$A \to aA'$
$A' \to aA' \mid \in$

$A \to \underset{A}{A}\underset{\alpha}{a} \mid \underset{\beta}{a}$

$\therefore$ Production will be -

$S \to AaB$
$A \to bB \mid a$
$A \to aA'$
$A' \to aA' \mid \in$ and $B \to bB \mid c$

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

(f) $R \to R^*/RR/(R)/id$

(g) $S \to (L)/a$
$L \to L,S/S$

Sol(e) $T \to T*F/F$

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

$\therefore$ Final production will be-

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

Sol(g)

$S \to (L)/a$
$L \to SL'$
$L' \to ,SL'/\epsilon$

Sol(f) This grammar can be written as-

$R \to R^*/id$
$R \to RR/id$
$R \to (R)/id$

$T \to T*F/F$
A A α β

$\Rightarrow R \to idR'$
$R' \to *R'/\epsilon$
$R \to idR'$
$R' \to RR'/\epsilon$
$R \to (R)/id$

$\therefore$ Final production will be-

$R \to idR'$
$R' \to *R'/\epsilon$
$R' \to RR'/\epsilon$
$R \to (R)/id$

Assignment

Ques1: What is the language generated by the following grammar?

$S \to OS1/01$
$S \to +SS/-SS/a$
$S \to S(S)S/\epsilon$

Justify your answer in each case-

Ques2: Write short note on context free Grammar. Give example.
(b) Parse tree., Give example.

Ques3: How is finite Automata useful for Lexical Analyser. Construct NFA and DFA for following Regular Expression-

$(a+b)^*abb$

Ques4: Write short note on Cross Compiler.

Ques5: Discuss the implementation of Look Ahead operator while doing the Lexical Analyser.

Ques6: Explain all necessary phases and passes of compiler Design.

Ques7: What is Bootstrapping? How does bootstrapping is done on more than one machine?

Ques8: Why translators are needed?

Left Factoring:

(Deterministic & Non-Deterministic Grammar)

If a grammar can not decide a production by using single input it is non-deterministic grammar else Deterministic grammar.

Ex 1: $A \to \alpha | \beta | \gamma$

$A \to \alpha$
$A \to \beta$ (Deterministic)
$A \to \gamma$

Ex 2: $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \alpha\beta_4$ ----

input $\alpha\beta_3$ (Non-deterministic)
Back tracking is required.

Steps to Make Non-Deterministic Grammar to Deterministic Grammar:

1. Make one production for common prefix.
2. One production for remaining part of production.

1. $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \alpha\beta_4$ [format]

Convert the given Non-Deterministic grammar into deterministic.

$\Rightarrow$ $A \to \alpha A'$
$A' \to \beta_1 | \beta_2 | \beta_3 | \beta_4$

2. $A \to aB | abc$

$\Rightarrow$ $A \to aA'$
$A' \to B | bc$

Process of making Non-Deterministic Grammar to Deterministic Grammar is called Left Factoring

Ex 1: $S \to iEtS | iEtSeS | a$
$E \to b$

$\Rightarrow$ $S \to iEtsS' | a$
$S' \to \epsilon | eS$
$E \to b$

Ex 2: $S \to aSSbS | aSaSb | abb | b$

$\Rightarrow$ $S \to aSS' | b$ $\quad$ $S \to aB' | b$
$S' \to bS | asb$ $\quad$ $B' \to SbS | sasb | bb$

$\Rightarrow$ $S \rightarrow aS \mid b$

$S' \rightarrow SS'' \mid bb$

$S'' \rightarrow SbS \mid asb$

$\beta_1 = SSbS$

$\beta_2 = SasS$

$\beta_3 = bb$

$A = S$

For production 3

$A = S'$

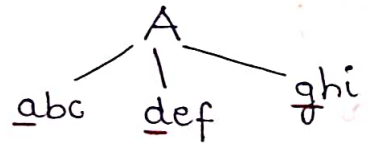
$\alpha = S$

$\beta_1 = SbS$

$\beta_2 = asb$

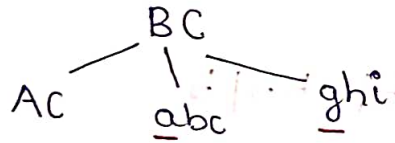
... Finding First Function:

1. $A \rightarrow abc \mid def \mid ghi$



first(A) = { a, d, g }

2. $B \rightarrow AC \mid abc \mid ghi$



first(B) = {a, g}

= { first(A), a, g }

first(A) is set of terminal symbols that are first symbol appears at LHS in derivation of A. If $A \rightarrow \epsilon$ then,

first(A) = ϵ

Ques: find first function in given Grammar:

(i) $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (F)$

$\Rightarrow$ first(E) = first(T) = first(F) = {id, (}

First(E') = {+, ϵ}

first(T) = first(F) = {id, (}

first(T') = { *, ϵ}

first(F) = {id, (}

(ii) $S \rightarrow ABC \mid ghi \mid jKl$

$A \rightarrow a \mid b \mid c$

$B \rightarrow b$

$D \rightarrow d$

$\Rightarrow$ first(S) = {first(A), g, j} = { a, b, c, g, j}

first(A) = {a, b, c}

first(B) = {b}

first(D) = {d}

(iii) $S \to ABC$
$A \to a \mid b \mid \epsilon$
$B \to c \mid d \mid \epsilon$
$C \to e \mid f \mid \epsilon$

$\Rightarrow$ first(S) = first(A) = {a, b, ε, c, d, e, f}
first(A) = {a, b, ε}
first(B) = {c, d, ε}
first(C) = {e, f, ε}

(iv) $A \to BCD$
$B \to bcf \mid \epsilon$
$C \to ceg \mid \epsilon$
$D \to d$

first(D) = {d}
first(C) = {c, ε}
first(B) = {b, ε}
first(A) = {b, c, d, $}

Finding Follow:

Follow(A) contains set of all terminals present immediate in right of A.

Rule1:
Follow of start symbol is $

Rule2:
Follow never contain ε

(i) $S \to ACD$
$C \to a \mid d \mid \epsilon$
$D \to d \mid \epsilon$

$\Rightarrow$ follow of (S) = {$}
follow of (A) = {a, d, $}
follow(D) = follow(S) = {$}
follow(C) = First(D) = {d, $}

Ques: Find first and follow of the following-

1) $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

$\Rightarrow$ first of B = {ε}
first of A = {ε}
first of S = {a, b}

follow(S) = {$}
follow(A) = {a, b}
follow(B) = {a, b}

2) $S \to ABCDE$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$C \to c$
$D \to d \mid \epsilon$
$E \to e \mid \epsilon$

$\Rightarrow$ first(E) = {e, ε}
first(D) = {d, ε}
first(C) = {c}
first(B) = {b, ε}
first(A) = {a, ε}
first(S) = {a, b, c}

follow(S) = { $ }
follow(A) = { b, c }
follow(B) = { c }
follow(D) = { e, $ }
follow(C) = { d, e, $ }
follow(E) = { $ }

3) $S \to Bb \mid Cd$
$B \to ab \mid \epsilon$
$C \to cC \mid \epsilon$

⇒ first(S) = { a, b, c, d }
first(B) = { a, ϵ }
first(C) = { c, ϵ }

follow(S) = { $ }
follow(B) = { b }
follow(C) = { d }

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

⇒ first(F) = { id, (}
First(T') = { *, ϵ }
first(T) = { id, (}
first(E') = { +, ϵ }
first(E) = { id, (}

follow(E) = { $,) }
follow(T) = { +, $,) }
Follow(E') = { $,) }
follow(T') = { +, $,) }
follow(F) = { *, $, +,) }

Ques: Find first & follow of all Non-Terminals-

$S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

⇒

First

first(C) = { h, ϵ }
first(B) = { g, ϵ }
first(A) = { d, g, h, ϵ }
first(S) = { d, g, h, ϵ, b, a }

follow

follow(S) = { $ }
follow(A) = { h, g, $ }
follow(B) = { $, g, a, h }
follow(C) = { g, $, b, h }

Ques: $S \to aBDh$
$B \to cC$
$C \to bC \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

⇒ first(F) = {f, ε} follow(S) = {$}
first(E) = {g, ε} follow(B) = {g, f, h}
first(D) = {g, f, ε} follow(C) = {g, f, h}
first(C) = {b, ε} follow(D) = {h}
first(B) = {c} follow(E) = {f, h}
first(S) = {a} follow(F) = {h}

Imp
Find out whether the given grammar is LL(1) or not:

1. $S \to aSbS \mid bSaS \mid \epsilon$

⇒ first(S) = {a, b, ε}
follow(S) = {$, b, a}

LL(1) Parsing Table

	a	b	$
S	S → aSbS S → ε	S → bSaS S → ε	S → ε

This grammar is not LL(1) grammar.

2. $S \to (S) \mid \epsilon$

⇒ first(S) = {(, ε}
follow(S) = {$,)}

	(	)	$
S	S → (S)	S → ε	S → ε

As each cell in the above table contains unique entry, the given grammar is LL(1) grammar.

3. $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

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

follow(S) = {$}
follow(A) = {a, b}
follow(B) = {b, a}

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

$first(\alpha_1) \cap first(\alpha_2) = \phi$

OR

$first(\alpha_1) \cap first(\alpha_2) = \phi$

$first(\alpha_2) \cap first(\alpha_3) = \phi$

4. $S \to aB \mid \epsilon$
   $B \to bC \mid \epsilon$
   $C \to cS \mid \epsilon$

⇒ $first(C) = \{c, \epsilon\}$
$first(B) = \{b, \epsilon\}$
$first(S) = \{a, \epsilon\}$

$follow(S) = \{\$\}$
$follow(B) = \{\$\}$
$follow(C) = \{\$\}$

	a	b	c	$
S	S→aB			S→ϵ
B		B→bC		B→ϵ
C			C→cS	C→ϵ

Since, each cell of each column has a single entry so, this grammar is LL(1) grammar.

5. $S \to AB$
   $A \to a \mid \epsilon$
   $B \to b \mid \epsilon$

⇒ $first(B) = \{b, \epsilon\}$
$first(A) = \{a, \epsilon\}$
$first(S) = \{a, b, \epsilon\}$

$follow(S) = \{\$\}$
$follow(A) = \{b, \$\}$
$follow(B) = \{\$\}$

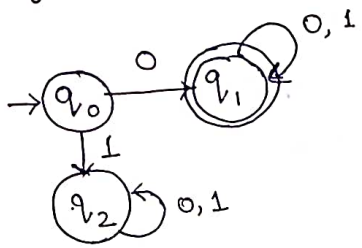
	a	b	$
S	S→AB	S→AB	S→AB
A	A→a	A→ϵ	A→ϵ
B		B→b	B→ϵ

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

7. $S \to aAa \mid \epsilon$
   $A \to abS \mid \epsilon$

Ques: DFA with $\Sigma = \{0,1\}$ starting with 0.

⇒



Smallest string = 0

$Q = \{q_0, q_1, q_2\}$

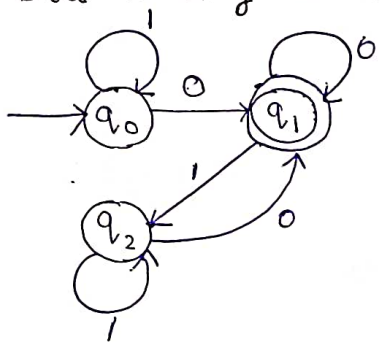
$\Sigma = \{0,1\}$

$q_0 = \{q_0\}$

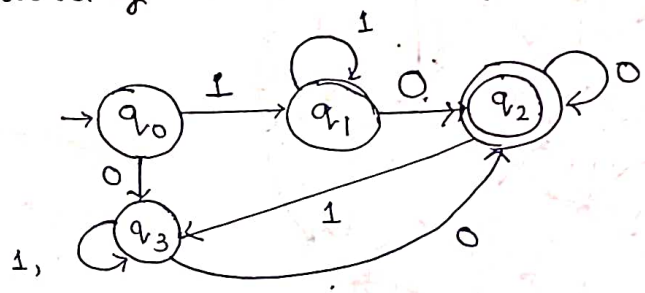
$F = \{q_1\}$

Ques: DFA with $\Sigma = \{0,1\}$ accepts all strings ~~sta~~ ending with 0.

⇒



Ques: DFA with $\Sigma = \{0,1\}$ accepts all strings starting with 1, ending with 0.

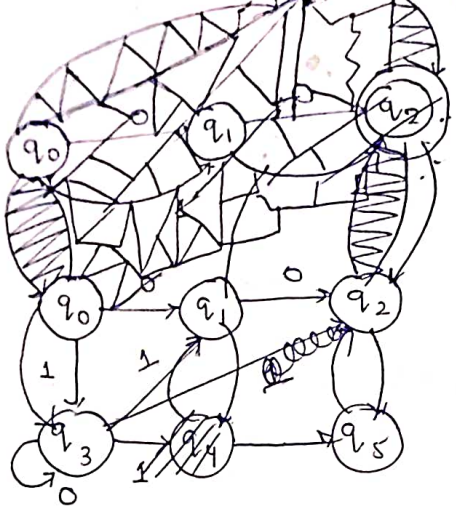


$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{0,1\}$

$q_0 = \{q_0\}$

$F = \{q_2\}$



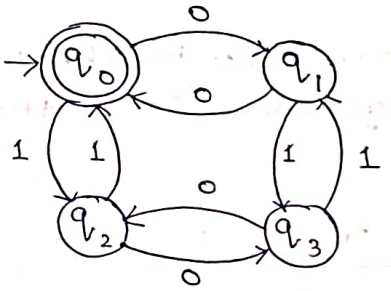
~~oo1oo1o~~

11100100

10101010

Ques: DFA with $\Sigma = \{0,1\}$ accepts all strings starting with 1 and ending with 0.

Ques: DFA with $\Sigma = \{0,1\}$ accepts all strings with even no. of 0 and 1.



$Q = \{q_0, q_1, q_2, q_3\}$

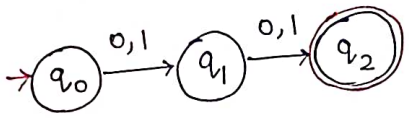
$\Sigma = \{0, 1\}$

$q_0 = \{q_0\}$

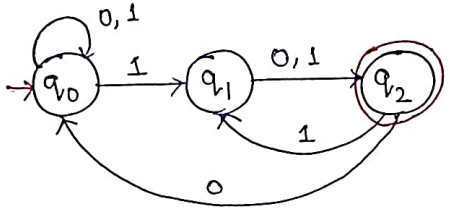
$F = \{q_0\}$

Ques: Draw a NFA ... all strings of length atleast 2.

⇒ $L = \{00, 11, 01, 10\}$



Ques: Draw a NFA in which second last bit is 1, with $\Sigma = \{0,1\}$



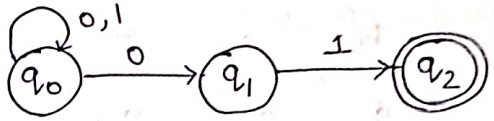
$L = \{010, 111, 10, 11, 110 ---\}$

Ques: Conversion from NFA to DFA :
{set of all strings over (0,1) that ends with '01'}

⇒ NFA :

	0	1
→ q_0	$\{q_0, q_1\}$	q_0
q_1	ϕ	q_2
(q_2)	ϕ	ϕ

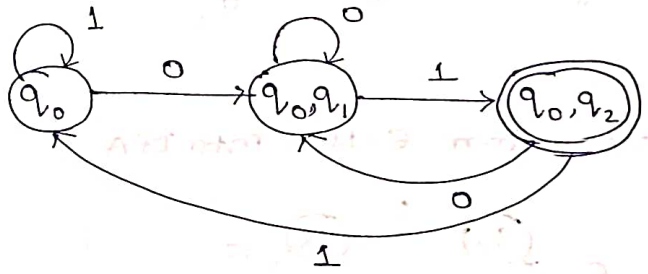
NFA.



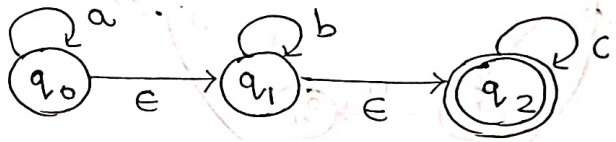
DFA Transition Table-

	0	1
→ q_0	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
($\{q_0, q_2\}$)	$\{q_0 q_1\}$	q_0

Transition diagram:



Ques: Conversion from ∈-NFA to DFA.



⇒ ∈-closure $(q_0) = \{q_0, q_1, q_2\}$

∈-closure $(q_1) = \{q_1, q_2\}$

∈-closure $(q_2) = \{q_2\}$

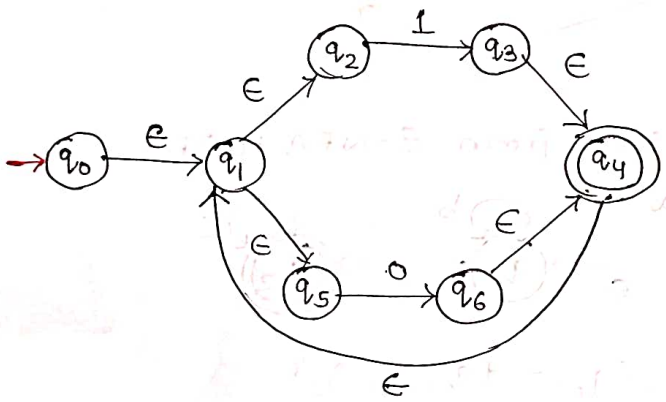
ϵ-NFA

	a		ϵ
q_0	q_0	ϕ	q_1
q_1	ϕ	q_1	q_2
q_2	ϕ	ϕ	q_2

DFA.

	a	b	c
→ $\{q_0,q_1,q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_1,q_2\}$	q_2
$\{q_1,q_2\}$	ϕ	$\{q_1,q_2\}$	q_2
$\{q_2\}$	ϕ	ϕ	q_2

Ques Conversion from ϵ-NFA into DFA



⇒ ϵ-closure(q_0) = $\{q_0, q_1, q_2, q_5\}$

ϵ-closure(q_1) = $\{q_1, q_2, q_5\}$

ϵ-closure(q_2) = $\{q_2\}$

ϵ-closure(q_3) = $\{q_3, q_4, q_1, q_2, q_5\}$

ϵ-closure(q_4) = $\{q_4, q_1, q_2, q_5\}$

ϵ-closure(q_5) = $\{q_5\}$

ϵ-closure(q_6) = $\{q_6, q_4, q_1, q_2, q_5\}$

ϵ-NFA

	0	1	ϵ
q_0	ϕ	ϕ	q_1
q_1			
q_2			
q_3			
q_4			
q_5			
q_6			

Ques: Remove left Recursion :

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

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

$E \rightarrow \underbrace{E}_{A} \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

$\therefore$ $E \rightarrow TE'$

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

$T \rightarrow T*F \mid F$

$T \rightarrow \underbrace{T}_{A} \underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$

$\therefore$ $T \rightarrow FT'$

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

$\therefore$ Production will be -

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

(ii) $A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

$A \rightarrow \underbrace{A}_{A} \underbrace{Bd}_{A\alpha_1} \mid \underbrace{Aa}_{A\alpha_2} \mid \underbrace{a}_{\beta_1}$

$\Rightarrow$ $A \rightarrow aA'$

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

$B \rightarrow bB'$

$B' \rightarrow eB' \mid \epsilon$

LL(1) parser

LL(1) → Look ahead symbol

- Left to right
- Leftmost derivative

1.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

First

First(E) = {id, (}

First(E') = {+, ϵ}

First(T) = {id, (}

First(T') = {*, ϵ}

First(F) = {(, id}

Follow

Follow(E) = { $,) }

Follow(E') = { $,) }

Follow(T) = { +, $,) }

Follow(T') = { +, $,) }

Follow(F) = { *, +, $,) }

Production	+	*	(	)	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			$F \to (E)$		$F \to id$	

2. $A \to AcB \mid cC \mid C$

$B \to bB \mid id$

$C \to CaB \mid BbB \mid B$

$A \to cCA' \mid CA'$

$A' \to cBA' \mid \epsilon$

$B \to bB \mid id$

$C \to BbBC' \mid BC'$

$C' \to aBC' \mid \epsilon$

First

First

$A \to AcB \mid cC$

$A \to AcB \mid C$

Bottom up Parsing: (... Pruning)

1. $S \to aABe$
   $A \to Abc \mid b$
   $B \to d$

Input string - abbcde

abbcde (RMD) in reverse order
↓
aAbcde $A \to b$
↓
aAde $A \to Abc$
↓
aABe $B \to d$
↓
S $S \to aABe$

Shift Reduce Parser:

1. $E \to E+E$
   $E \to E*E$
   $E \to id$

Input string : $id_1 + id_2 + id_3$

⇒

Stack	Input Buffer	Action
$	$id_1+id_2+id_3\$$	shift
$ id_1	$+id_2+id_3\$$	Replace by $E \to id$
$ E	$+id_2+id_3\$$	shift
$ E+	$id_2+id_3\$$	shift
$ E+id_2	$+id_3\$$	Replace by $E \to id$
$ E+E	$+id_3\$$	shift
$ E+E+	$id_3\$$	shift
$ E+E+id_3	$	Replace by $E \to id$
$ E+E+E	$	$E \to E+E$
$ E+E	$	$E \to E+E$
$ E	$	Accept

2. $E \to E+T \mid T$
   $T \to T*F \mid F$
   $F \to (E) \mid id$
   Input string : id * id

Operator Precedence Parser

Rule1: In RHS, there should not present ∈

Rule2: And No two Non-terminals must be together at RHS.

Ques: E → E+E | E*E | id

[Terminals have high prority then- operator then- $]

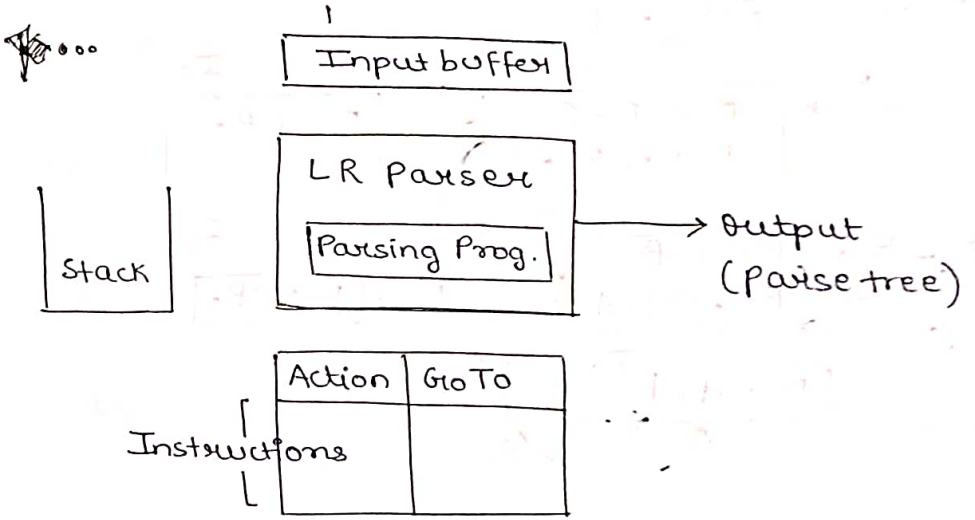
	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	>	>
$	<	<	<	Accept

Start	Relation	Input String	operation
$	<	id+id*id$	Shift id
$id	>	+id*id$	Reduce E→id
$E	<	+id*id$	shift +
$E+	<	id*id$	shift id
$E+id	>	*id$	Reduce E→id
$E+E	<	*id$	Shift *
$E+E*	<	id$	shift id
$E+E*id	>	$	Reduce E→id
$E+E*E	>	$	E→E+E
$E	—	$	Accept

LR Parser:

L R → Right most derivative

Left to Right



Ques: E → BB
B → cB | d
Input string: ccdd$

⇒ Step1: Augment the grammar
fake.

E' → E
E → BB
B → cB | d

{ if we have grammars then, S' → S (Augment)

Step2: Draw canonical item:

I_0:

P1 $E' \to \cdot E$

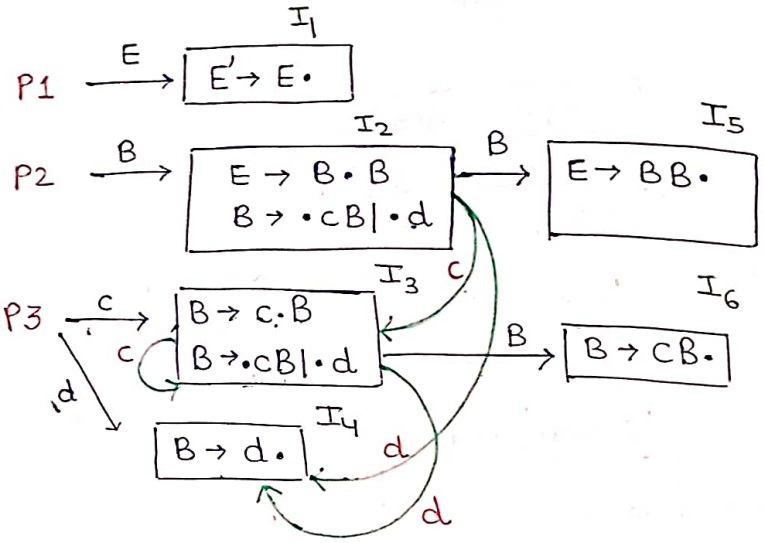
P2 $E \to \cdot BB$

P3 $B \to \cdot cB \mid \cdot d$

] canonical form

$E' \to E$

$E' \to \cdot E$ } LR(0)



Step3: Name the production

$E \to BB$ (1)

$B \to cB$ (2) | d (3)

Step4: Create the parsing table

States	Action			GOTO	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

Step5: Stack implementation

Stack	Input	Action
$0	ccdd$	Shift c and goto statement 3
$0c3	cdd$	Shift c and goto statement 3
$0c3c3	ddd$	Shift d → 4
$0c3c3d4	dd$	reduced B → d
$0c3c3B6	dd$	reduced B → cB
$0c3B6	dd$	reduced B → cB
$0B2	dd$	Shift d → 4
$0B2d4	d$	reduced B → d

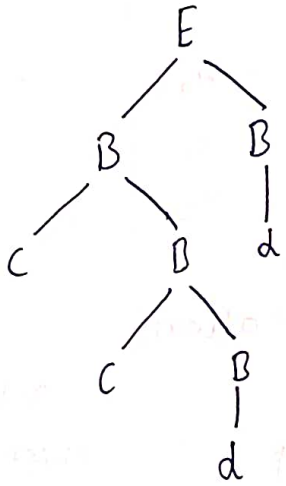
$ | reduced E→BB

$ | Accept

$0B2B5

$0E1

Step 6: Parse tree



SLR Parser:

Step 4 Parsing table

→ follow of canonical collection

→ Then write producting name as $r_1, r_2 \ldots$

Ques: $E \to T+E \mid T$

$T \to id$

⇒ Step 1: Augment the grammar

$E' \to E$

$E \to T+E \mid T$

$T \to id$

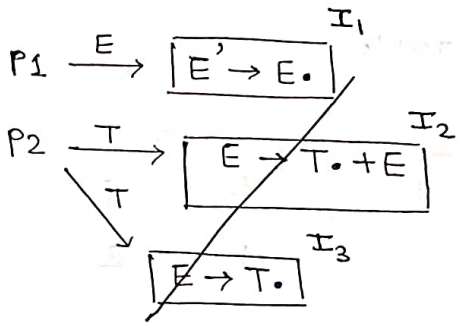
Step 2: Draw canonical collection of LR(0) items:

I_0

P1 $E' \to .E$

P2 $E \to .T+E \mid .T$

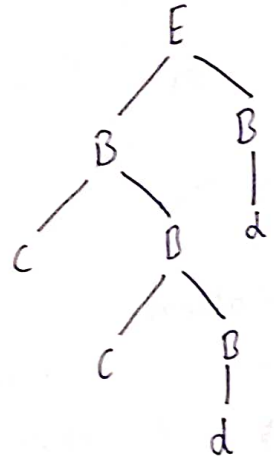
P3 $T \to .id$



⇒

$	$	reduced $E \to E$
$0B2B5	$	Accept
$0E1		

Step 6: Parse tree



SLR Parser:

Step 4 Parsing table

→ follow of canonical collection

→ Then write producting name as $r_1, r_2 \ldots$

Ques: $E \to T+E \mid T$

$T \to id$

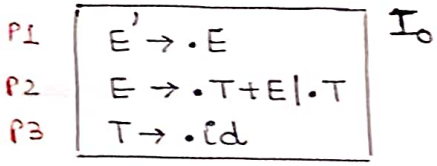
⇒ Step 1: Augment the grammar

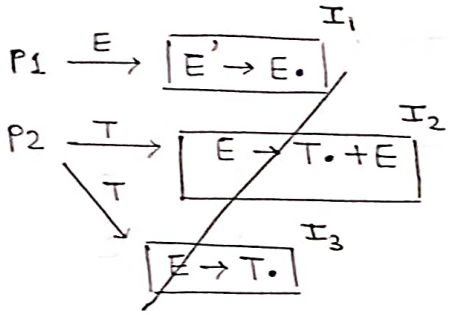
$E' \to E$

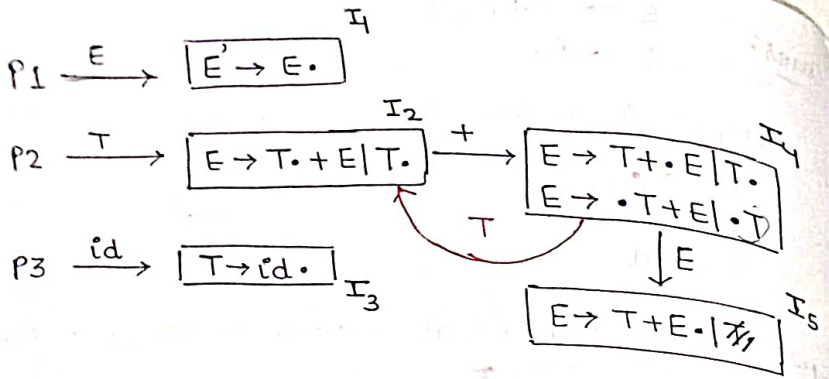
$E \to T+E \mid T$

$T \to id$

Step 2: Draw canonical collection of LR(0) items:







Step3: Name the production.

$E \to T+E^{1} \mid T^{2}$

$T \to id^{3}$

Step4: Create the parsing table (LR(0))

States	Action			GO TO	
	+	id	$	E	T
I_0		S_3		1	2
I_1					
I_2	S_4				
I_3	R_3	R_3	R_3		
I_4					
I_5	R_1	R_1	R_1	5	2

SLR parser:

first(E) = {id}
first(T) = {id}

follow(E) = {$}
follow(T) = {+, $}

CLR Parser:

Ques: $S \to CC$
$C \to cC$
$C \to d$

$S \to C^1C$
$C \to cC^2$
$C \to d^3$

⇒ Step1: Augment the grammar -

$S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

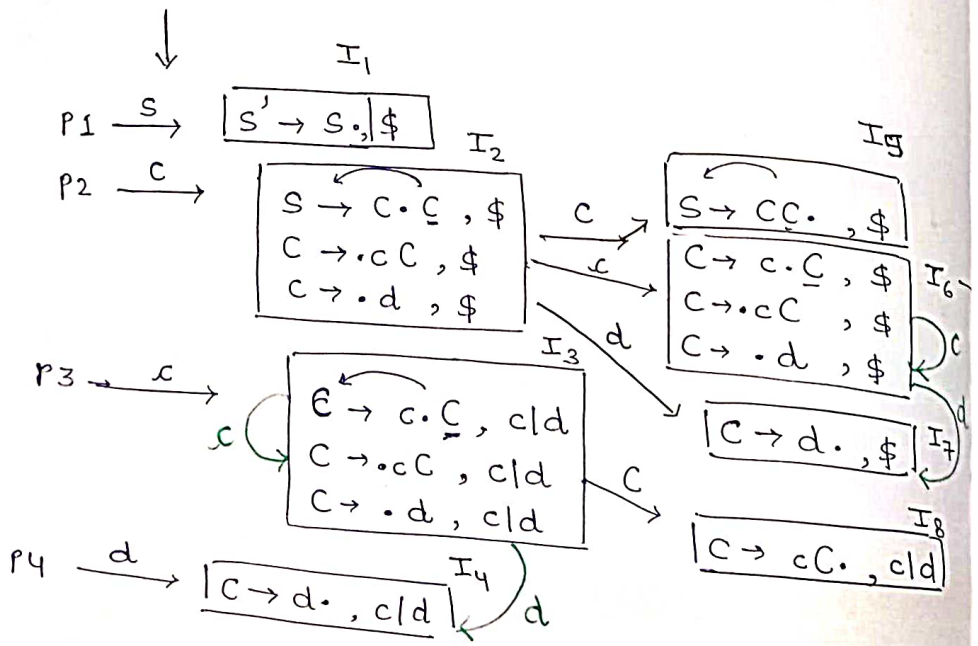
[Jiski vja se nxt production aayegh, sirf uska follow niklega]

Step2: Draw canonical collection

I_0

P1	$S' \to .S$, \$
P2	$S \to .CC$, \$
P3	$C \to .cC$, c/d
P4	$C \to .d$, c/d

follow(S') = \$
follow(S) = \$
follow(C) = c, d



State	Action c	Action d	Action $	GoTo S	GoTo C
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Stack Implementation:

Stack	Input	Action

... LALR parser:

Ques: $S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$

... SDT: (Syntax Directed Translation):

CFG + Semantic rules = SDT

... Types of Attribute:

1. Synthesized attribute
2. Inherited attribute

- If a node takes values from child node then it is synthesized attribute.

SDD:

Ques: Construct parse Tree / Syntax tree and Annotated parse tree for the input string

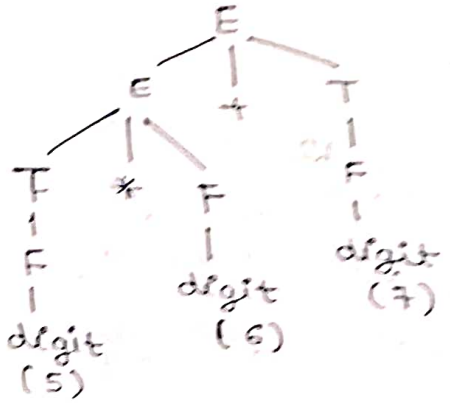
5 * 6 + 7;

$S \to EN$	print(E.val)
$E \to E+T$	E.val = E.val + T.val
$E \to E-T$	E.val = E.val − T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to T/F$	T.val = T.val / F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val

$F \to digit$ F.val = digit.lexval

$N \to ;$ can be ignored by lexical analyzer

Ans

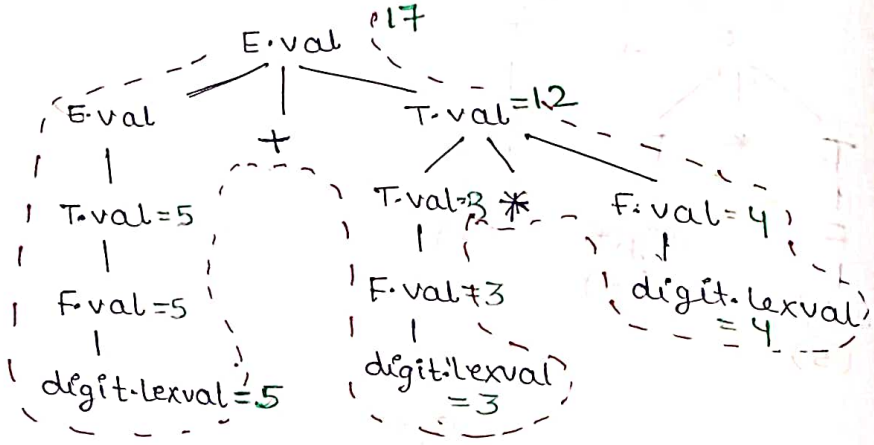


S.O-

Ques: Annotated parse tree for
5 + 3 * 4

Production Rule	Semantic Actions
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit.lexval

⇒



Dependency Graph:

Ques: String int a,b,c

Production Rule	Semantic Actions
S → T L	L.in = T.type
T → int	T.type = integer
T → float	T.type = float
T → char	T.type = char
T → double	T.type = double
T → L.id	
L → id