28/08/2023

Compiler: Compiler is a translator that converts high level language into target language (machine level / low level).

» It converts the source code into target code, all at once and creates an Object ~~cta~~ file.

Assembler: Assembler is a translator that converts high level language into machine level language.

Interpreter: Interpreter is a translator that performs line by line execution.

» It is slower when compared to compiler.

→ Compilation can be done in one or many passes.

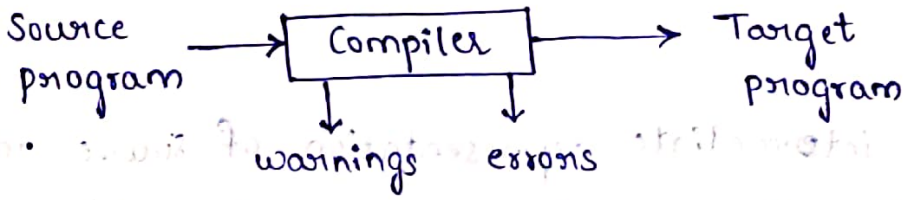
One pass :- HLL → MLL.

more passes :- HLL → LLL , LLL → MLL.

29/08/2023

Compiler: It is a software or a program that converts a program written in high-level language (source code) to a low-level language (object or target language code).

» It also reports errors present in the source program.



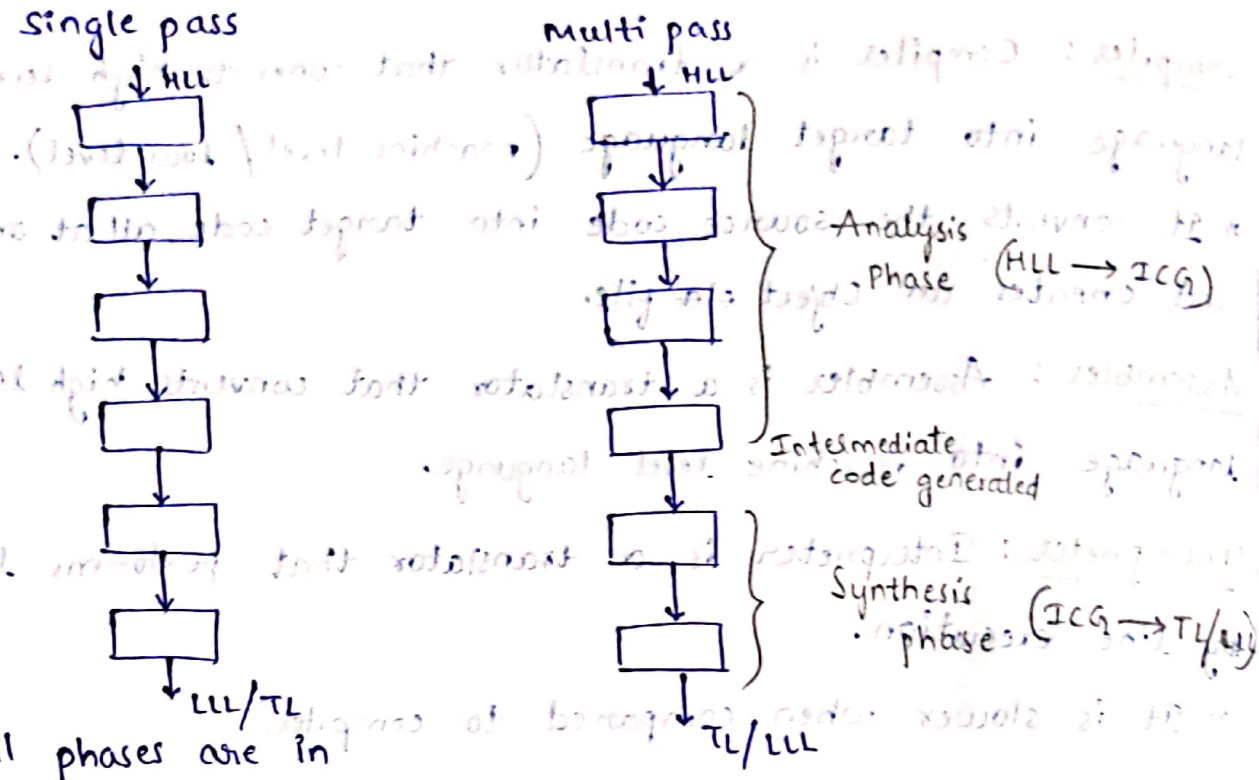
Types of compilers

1) **Single pass compiler:**

It is a type of compiler that processes the source code only once. (All 6 phases are performed at once).

2) **Multi pass compiler:** 4-phases — Pass I, 2-phases — Pass II.

It is a type of compiler that processes the source code multiple times i.e, to convert source code into target/object code.



All phases are in single module.

There are two parts of compilation process (phases).

Two phases → Analysis phase
Two phases → Synthesis phase

Analysis Phase

It breaks up the source code or program into smaller parts ~~or~~ and creates an intermediate representation of the source program.

Synthesis Phase

It takes the intermediate representation of source code as input and creates the target code that is desired.

Properties of the compiler

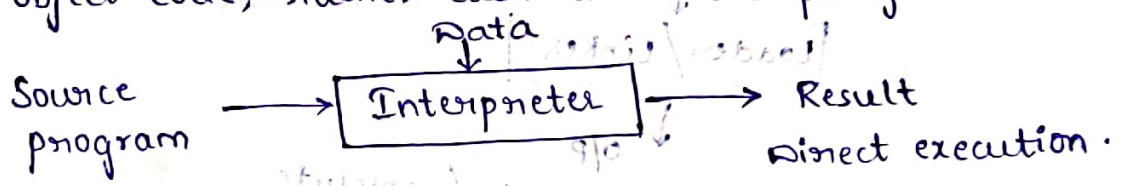
When a compiler is built, it should posses the following properties.

» The compiler itself must be bug-free.

» It must generated correct machine code.

» The generated machine code must ~~done~~ run fast.
» Compilation time must be proportional to program size.
» The compiler must be portable.
» It must give good diagnostics and error messages.
» The generated code must work well with existing debuggers
» It must have consistent optimization.

Interpreter :- An interpreter is a kind of translator which produces the result directly when the source program and data is given to it as input. It does not produce the object code, rather each time the program needs execution

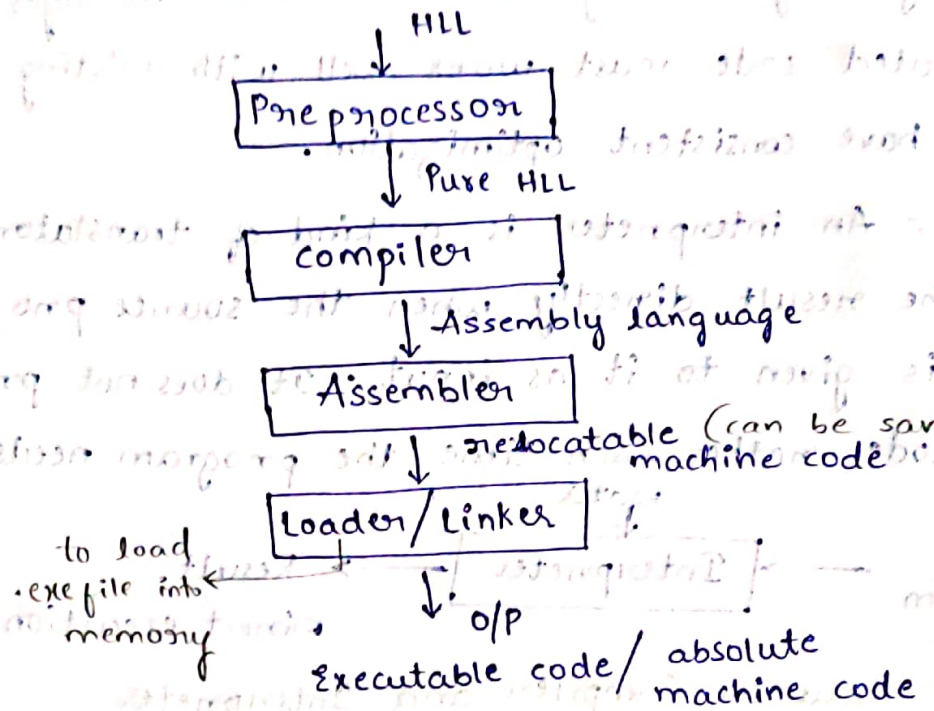


Differences between Compiler and Interpreter

Compiler	Interpreter
» It translates entire program as input.	» It takes single instruction as input.
» Intermediate object code is generated.	» No intermediate object code is generated.
» Memory requirement is more (since object code is generated).	» Memory requirement is less.
» Program need not be compiled every time.	» Every time, high level program is converted into low level language
» Errors are displayed after the entire program is checked.	» Errors are displayed for every instruction interpreted.
» C, Java, C++ are compiled languages.	» Python is interpreted language.

→ The main goal of any translator is that, it must preserve the meaning of the source code.

Language Processing System



The high-level language programs are fed into a series of tools and operating system components to get the desired code that can be used by the machine. This is called language processing system.

» Preprocessor : In preprocessing, HLL is converted into pure HLL. Preprocessor removes the preprocessing directives (#include<stdio.h>) and adds the respective files ie, file inclusions. It also performs macro expansion, operation conversions (Ex: a-- ⇒ a=a-1)

» Compiler : During compilation, compiler converts pure HLL to assembly language.

» Assembly language : It is a low-level programming language. It is not in binary form.

» Assembler : For every platform, we have different assemblers. Assembler for one platform will not work for another platform.

Assembly code is converted into ~~relocatable~~ executable machine code

» Relocatable machine code: that can be loaded anywhere in the memory.

Assembly code $\xrightarrow{\text{converted}}$ executable machine code.

» Linker / Loader: It converts relocatable machine code into absolute machine code.

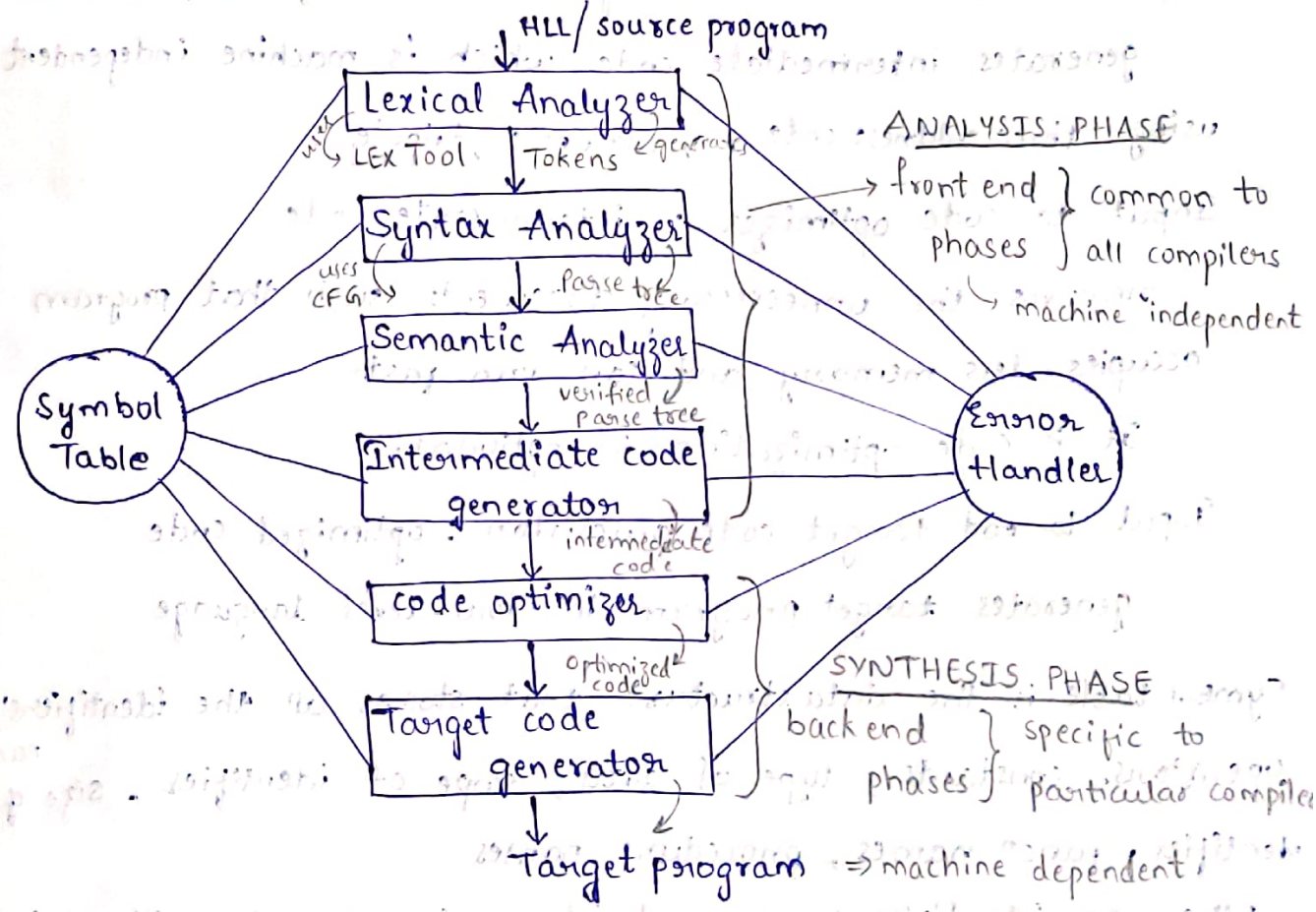
» Linker: It links different object files into a single executable file.

» Loader: It loads the executable file in the memory and executes it.

» Absolute machine code: Code that is assembled to work at one specific address.

30/08/2023

* Phases of the Compiler



① Input to lexical analyzer : HLL source code

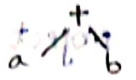
generates stream of tokens using LEX Tool that uses regular exp.

regular expression → that has rules for a valid token.

: removes all spaces, commas, semi colons, tab spaces.

Errors like repeated declarations, using keywords as identifiers are detected.

② Input to syntax analyzer : stream of tokens

generates parse tree 

uses CFG (Context Free Grammar) to check the syntax.

~~eliminates~~ identifies all the syntactical errors (if any).

③ Input to semantic analyzer : parse tree

checks for compatibility errors, parentheses matching, if-else statements, control flow statements

generates verified parse tree

④ Input to Intermediate code generator : verified parse tree

generates intermediate code, which is machine independent.

using three address code, quadra pole, tripole

$$\begin{cases} a = b + c + d \\ t_1 = b + c \\ t_2 = t_1 + d \end{cases}$$

⑤ Input to code optimizer : intermediate code

removes the unnecessary statements such that program occupies less memory and can run fast.

~~It~~ Code optimization is optional.

⑥ Input to ~~cod~~ target code generator : optimized code

generates target program in low-level language.

Symbol table is the data structure that stores all the identifiers, operators, constants, type of data, scope of identifier, size of identifier, name, func^n names, procedure names

When an identifier is encountered again, then it will not be entered into symbol table rather, it points to the already entered identifier.

It is also used to retrieve the identifiers.

It is associated with each phase.

Error handler is associated with ~~eo~~ every phase.

It handles all the errors and ensures smooth compilation of the entire program.

*Show how an input $a = b + c * 60$ get processed in compiler. show the output of each stage of compiler. Also show the contents of symbol table.

$a = b + c * 60$

↓

lexical analyzer

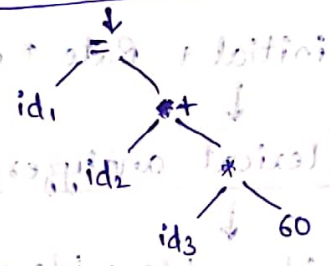
↓

$id_1 = id_2 + id_3 * 60$ (stream of tokens)

↓

Syntax analyzer

↓



(parse tree)

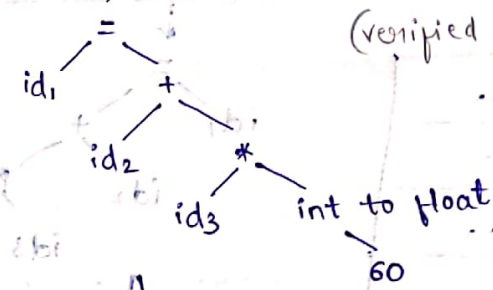
Symbol table

a	...
b	...
c	...
⋮	...

↓

Semantic analyzer

↓



(verified parse tree)

• if a, b, c are floating point

↓

Intermediate code generator

↓

$t_1 = \text{int to float}(60)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

(3-address code)

↓

code optimization

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

target code generator

↓

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

(Target code)

Describe the output for the various phases of the compiler w.r.to following statement

position = initial + Rate * 60

position = initial + Rate * 60

↓

lexical analyzer

↓

$id_1 = id_2 + id_3 * 60$

Stream of tokens

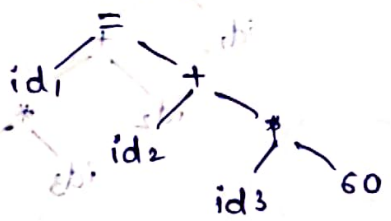
↓

syntax analyzer

↓

Symbol table

position	..
initial	..
Rate	..
⋮	..

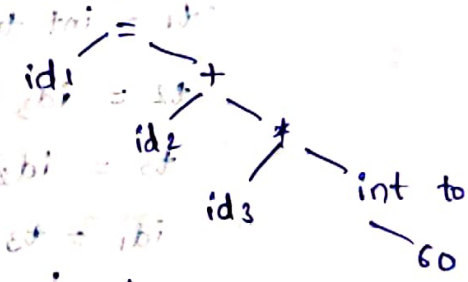


parse tree

↓

semantic analyzer

↓



verified parse tree

↓

intermediate code generator

↓

t_1 = int to float

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

~~t_4~~ $id_1 = t_3$

3-address code

↓

code optimization

↓

$t_1 = id_3 * 60.0$

$id_1 = id_2 + t_1$

↓

target code generator

↓

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

↓

target code

1/9/2023

Show the output generated by each phase for the following example.

price = amount + rate * 50

price = amount + rate * 50

↓

lexical analyzer

↓

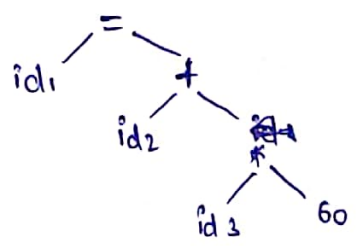
$id_1 = id_2 + id_3 * 50$

stream of tokens

↓

syntax analyzer

↓

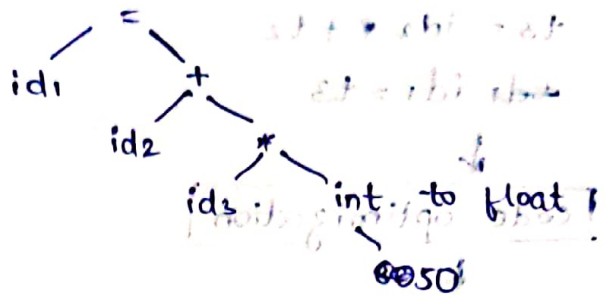


parse tree

↓

Semantic analyzer

↓



↓

intermediate code generator

↓

t1 = int to float

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

↓

code optimization

↓

t1 = id3 * 50.0

id1 = id2 + t1

↓

Target code generator

↓

```
MOVF  id3, R2
MULF  #50.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

Show the output generated by each phase for the following example.

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

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

↓

lexical analyzer

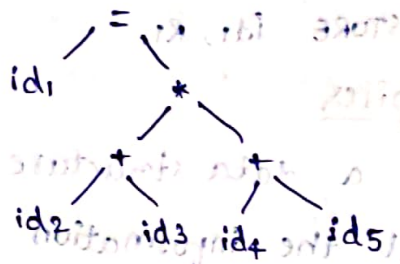
↓

$$id_1 = (id_2 + id_3) * (id_4 + id_5)$$

stream of tokens

↓

Syntax analyzer

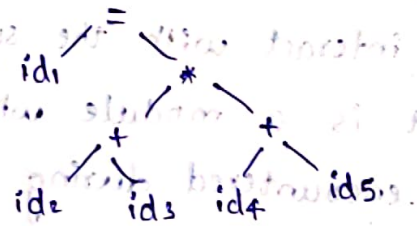


parse tree/ syntax tree

↓

semantic analyzer

↓



verified parse tree

↓

Intermediate code generator

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 + id_5$

$t_3 = t_1 * t_2$

$id_1 = t_3$

3-address code

↓

code optimization

↓

$t_1 = id_2 + id_3$

$t_2 = id_4 + id_5$

$id_1 = t_1 * t_2$

↓

code generator

↓

~~Load id1, id2~~

~~ADD~~

LOAD R1, id2

ADD R1, id3

LOAD R2, id4

ADD R2, id5

MUL R1, R2

STORE id1, R1

Phases of the compiler

Symbol Table : It is a data structure being used by the compiler to store all the information related to identifiers.

Ex: its type, scope, size, location, name etc

» It helps the compiler to function smoothly by finding the identifiers quickly.

» All the phases interact with the symbol table manager.

Error Handler : It is a module which takes care of the events which are encountered during compilation, and it takes care to continue the compilation process even if error is encountered.

Lexical Analysis :

» Reads the program and converts it into tokens using a tool called Lex tool.

» Tokens are identified by regular expressions which are understood by the lexical analyzer.

» Removes whitespaces, comments, tab spaces etc.

» Lexical analyzer is also called as Scanner

Syntax Analyzer / Parser :

» constructs parse tree

» Takes the tokens one by one and uses CFG to construct parse tre

» If it is not possible to construct the parse tree, then input is syntactically incorrect and error messages are displayed.

» By using Context Free Grammar (CFG), the input has to be checked whether it is in the desired format or not.

» Syntax errors can be detected by it, if the input is not according to the grammar given.

» It takes input as tokens and checks their syntactic correctness.

Semantic Analysis:

» Once the syntax is checked in the syntax analysis phase, the next phase is semantic analysis, which determines the meaning of the source string. For ex: meaning of the source string is matching of parantheses in the expression, matching of if-else statements, performing arithmetic operations of the expressions that are type-compatible, checking the scope of the operation.

Intermediate code generation:

» The intermediate code is a kind of code, which is easy to generate and this code can be easily converted to target code.

» This code can be in variety of forms such as 3-address code, quadraples, tripules, post-fix notation.

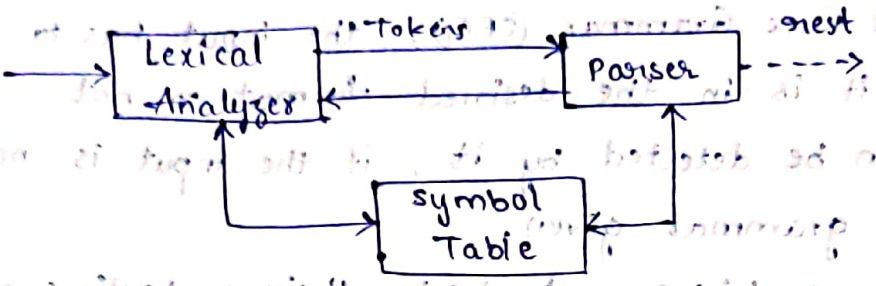
Code ~~optimation~~ Optimization:

» The code optimization phase attempts to improve the intermediate code. This is necessary to have faster executing code or less consumption of memory. Thus, by optimizing the code, the overall running time of the target program can be improved.

Code generation:

» In code generation phase, the target code gets generated. The intermediated code instructions are translated into sequence of machine instructions.

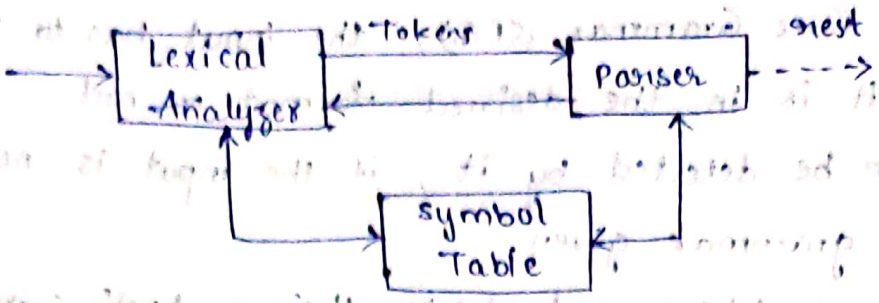
Lexical Analysis Phase



» Lexical Analysis is the first phase of the compiler, also called as Linear Analysis or ~~Scanner~~ Scanning.

» In this phase, the stream of characters making up source code is read from left to right and grouped into tokens that are sequence of characters having collective meaning.

Lexical Analysis Phase



» Lexical Analysis is the first phase of the compiler, also called as linear Analysis or Scanner Scanning.

» In this phase, the stream of characters making up source code is read from left to right and grouped into tokens that are sequence of characters having collective meaning.

» Tokens include identifiers, operators, keywords ...

4/9/2023

Functions of Lexical Analyzer

» It produces stream of tokens.

» It eliminates blank spaces and comments.

» It generates the symbol table which stores the information about identifiers, constants encountered in the input.

» It keeps track of the line numbers.

» It reports the errors encountered while generating the tokens.

Tokens :

It describes the class or category of input string.

~~For~~ ex :- identifiers, keywords, constants.

Patterns :

Set of rules that describe the tokens

Lexemes :

Sequence of characters in the source program that are matched with the patterns or the tokens.

ex :- num is a valid token ie, identifier that follows the pattern

Ex: Identify tokens in the expression —

if (a < b)

if – keyword
(→ operator
a → identifier
< → relational operator
b → identifier
) → operator

**

Q) Define lexeme, token and pattern. Identify the lexemes that make up the tokens in the following program set.
~~Identify~~ Indicate the corresponding token and pattern.

```
void swap (int i, int j){
    int t;
    t = i;
    i = j;
    j = t;
}
```

Lexeme	Token	Lexeme	Token
void	keyword	t	identifier
swap	identify	=	operator
(	operator	i	identifier
int	keyword	i	identifier
~~&~~	~~operator~~	=	operator
i	identifier	j	identifier
, int	separation operator (keyword)	j	identifier
j	identifier	=	operator
)	operator	t	identifier
{	operator	}	operator
int	keyword		
t	identifier		
~~&~~	~~operator~~		
~~*~~	~~operator~~		

Patterns

(i) Identifiers

» Identifiers are a collection of letters

» Identifier is a collection of alphanumeric characters)

» The first character of an identifier must be a letter.

(ii) Operator

» Operator can be arithmetic, logical, relational operators.

» The parantheses are considered as operators

» comma is treated as separation operator

» assignment is denoted by operator

(iii) keyword

» keywords are special words to which some meaning is associated with.

» int, void are keywords for denoting data types.

Q)
```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

~~Token~~	~~lexeme~~ token
int	keyword
max	identifier
(	operator
a	identifier
b	identifier
{	operator
if	keyword
>	relational operator
return	keyword
else	keyword
}	operator

lexeme ~~token~~	token ~~lexeme~~
int	- keyword
max	- identifier
(	- operator
int	- keyword
a	- identifier
,	- operator
int	- keyword
b	- identifier
)	- operator
{	- operator
if	- keyword
(	- operator

lexeme ~~Token~~	token
a	- identifier
>	- operator
b	- identifier
)	- operator
return	- keyword
a	- identifier
else	- keyword
return	- keyword
b	- identifier
}	- operator

operator

lexeme	token
Patterns	

(i) Identifiers

» Identifiers are a collection of letters

» Identifier is a collection of alphanumeric characters

» The first character of an identifier must be a letter

(ii) Operator

» Operator can be arithmetic, logical, relational

» Parantheses are considered as operators

» Comma is treated as separation operator.

» Assignment is denoted by operator,

» greater than symbol (>) is an operator

(iii) keyword

» keywords are special words to which some meaning is associated with.

» int, void are keywords for denoting data types.

» return is a keyword

» if, else are keywords for checking whether a condition is true.

Differences between Phases and Pass.

Phase	Pass
1) The processing of compilation is carried out in various ways. These steps are preferred as Phases. The phases are: Lexical phase, Syntax phase, Semantic phase, Intermediate code generation, code optimization, Target code generation	1) Various phases are logically grouped together to form a Pass. The process of compilation can be carried out in single pass or in multiple passes.

2) The task of compilation is carried out in analysis and synthesis phase.

3) The phases namely -
lexical analysis
Syntax analysis
Semantic analysis
Intermediate code generation
are machine independent.
code optimization
Target code generation
are machine dependent.

2) The task of compilation is carried out in single or multiple passes.

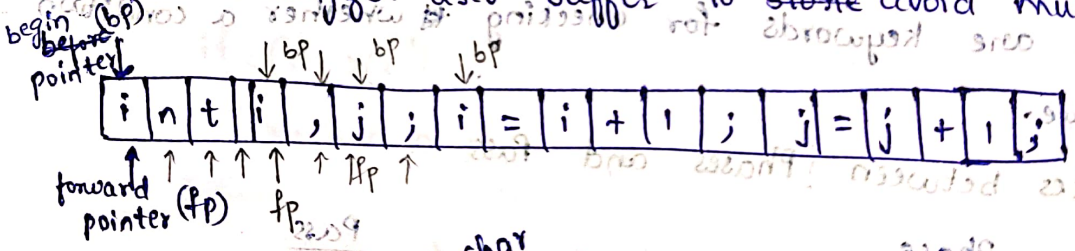
3) The execution of program in passes, The compilation model can be viewed as front end and back end

05/9/2023

** Input Buffering

```
int i, j;
i = i+1 ;
j = j+1 ;
```

lexical analyzer uses buffer to ~~store~~ avoid multiple system calls.



when it reads each char then, fp is incremented until space is identified.

when it encounters blank space, then a token is identified.

Here, int is identified which is a keyword.

Then, begin pointer is moved to i

both fp and bp are at i

Then, fp is incremented.

next char is ,

so, bp is also shifted to ,

When ; is encountered it indicates end of the statement so, both bp and fp are moved to it.

» Problem of using buffer is size.

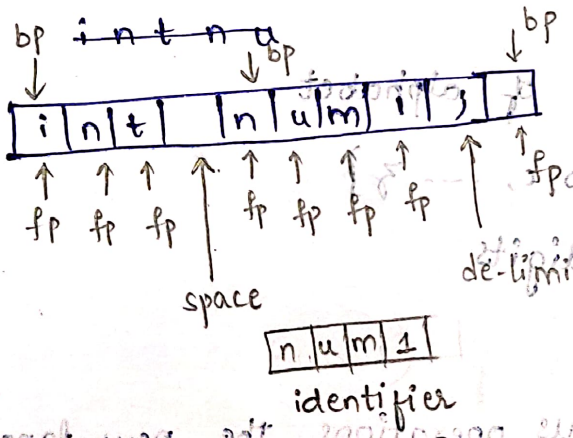
categories of buffer :-

(i) one buffer

(ii) two buffer

(i) In single buffer, data is overwritten when buffer size is insufficient. In this method, data may be lost.

(ii) To overcome the problems of single buffer, we use two buffer. When the end point of buffer (eof) is reached, the remaining data is stored in second buffer. And, when second ~~dat~~ buffer is reached to its end, it again uses first buffer. Thereby, the data of first buffer is lost.



Specifications of tokens (UNIT – I) 1st Half

» To specify tokens, regular expressions are used.

When a pattern is matched by some regular expression then, token can be recognized.

Language and String

» Language : Finite set of non-empty strings

» String : Collection of finite number of alphabet or letters.

ε indicates null string.

» The length of the string is denoted by $|s|$

» The empty string is denoted by ε

» The empty set of strings is denoted by $\emptyset$

Operations on Language

1) Union of two languages ~~L1 and L2~~ L1 and L2 :-

$L1 \cup L2 = \{$ set of strings in L1 and set of strings in L2 $\}$

2) Concatenation of two languages L1 and L2 :-

$L1 \cdot L2 = \{$ set of strings in L1 followed by set of strings in L2 $\}$

3) Kleen closure of language $L \Rightarrow L^*$

L^* denotes 0 or more concatenations of L

~~Include~~ This may contain ε

4) Positive closure of L $\Rightarrow L^+$

L^+ denotes 1 or more concatenations of L

Example

Let L represent the set of alphabet.

$L = \{ A, B, C, \ldots, Z, a, b, c, \ldots z \}$

D be the set of digits

$D = \{ 0, 1, 2, 3, \ldots, 9 \}$

Then, by performing various operations the new language is generated as follows :-

1) $L \cup D$ is a set of alphabet and digits

2) $L \cdot D$ (or) LD is a set of strings consisting of alphabet followed by digits.

L^5 is a set of strings having length of 5 each

3) L^* is a ~~strino~~ set of strings having all the strings including ε.

4) L^+ is a set of strings except ε.

Suppose, $L = \{a\}$ then,

$$L^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$$

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

suppose, $L = \{a, b\}$ then

$$L^* = \{\varepsilon, a, b, ab, ba, aa, bbb, \ldots\}$$

$$L^+ = \{a, b, ab, ba, aaa, bbb, \ldots\}$$

3) L^* is a string set of strings having all the strings including ε.

4) L^+ is a set of strings except ε.

Suppose, $L = \{a\}$ then,

$L^* = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

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

suppose, $L = \{a, b\}$ then

$L^* = \{\varepsilon, a, b, ab, ba, aa, bbb, \ldots\}$

$L^+ = \{a, b, ab, ba, aaa, bbb, \ldots\}$

11/09/2023

Regular exp. has set of rules.

Regular Set

The finite set which denotes a regular language and the set which can be described by regular expressions is called Regular set.

Ex: A set of identifiers.

Because, it can be represented using regular expressions.

Regular Expressions

Regular expressions are mathematical symbolisms which describe the set of strings of specific language.

~~Rules for writi~~

Following are the rules that describe the definition of regular expressions over the input set denoted by Σ.

1. ε is a regular expression that denotes the set containing empty string

2. If R_1 and R_2 are regular expressions, then

$R = R_1 + R_2$ $(R = R_1 | R_2)$ is also a regular expression which represents Union operation.

3. If R_1 and R_2 are regular expressions, then

$R = R_1 R_2$ is also a regular expression which represents concatenation operation.

4. If R_1 is a regular expression, then

$R = R_1^*$ is also a regular expression which denotes Kleen closure.

* The language denoted by regular expressions is called Regular set / Regular Language.

Q) Write a regular expression for the language accepting all the combinations of a's including null over $\Sigma = \{a\}$

$L = \{\varepsilon, a, aa, aaa, \ldots\}$

$L = a^*$

Q) Accept all the combinations of a except the null string

$L = \{a, aa, aaa, \ldots\}$

$L = a^+$

Q) Write a regular expression for the language containing the strings of length 2 over $\Sigma = \{0,1\}$

$L = \{00, 01, 10, 11\}$

$RE = (0+1)(0+1)$

OR $(0+1)$ for length 1

Q) Write a regular expression for a language combining strings which ends with abb over $\Sigma = \{a,b\}$.

$L = \{abb, aabb, babb, ababb, baabb, \ldots\}$

$RE = (a+b)^* abb$

$\because (a+b)^* = \{\varepsilon, ab, aa, bb, ba, \ldots\}$

Q) construct a regular expression for the language containing all the strings having any number of a's and b's except ε. null string

$RE = (a+b)^*$ $(\because L = \{a, b, aa, bb, ab, \ldots\}$

Q) construct regular expression for the language accepting all the strings ending with 00 over the set $\Sigma = \{0,1\}$

$RE = (0+1)^* 00$

$L = \{000, 100, 0100, 1000, 01100, 10100, \ldots, 00\}$

Q) write a regular exp for the language accepting the strings which are starting with 1 and ending with 0 over the set $\Sigma = \{0,1\}$

$RE = 1(0+1)^* 0$

$L = \{10, 1010, 1110, 100100, 100100, \ldots\}$

Q) Write RE to denote the language L over Σ^* where $\Sigma = \{a, b, c\}$ in which every string will be such that any number of a's followed by any number of b's followed by any number of c's.

$RE = a^* b^* c^*$

$L = \{a, b, c, ab, bc, aabc, abcc, \ldots\}$

Q) write a RE to denote L over Σ^* where, $\Sigma = \{a, b\}$ such that the third character from right end of the string is always a.

$RE = (a+b)^* a (a+b)(a+b)$

Q) construct RE for the language which consists of exactly 2 b's over the set $\Sigma = \{a, b\}$

$$RE = a^* b a^* b a^*$$

Q) Describe the language for the following REs.

(1) $(0+1)^* \, 0 \, (0+1)^*$

The language L containing all the strings made up of 0's and 1's which contains atleast one zero.

(2) $0^* 1 0^* 1 0^* 1 0^*$

The language L containing all the strings made up of 0's and 1's which contains exactly 3 ones.

12/09/2023

Finite Automata

» Finite Automata is an abstract computing device.

» It is a mathematical model of a system with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet Σ

A string is said to be Accepted String, if all the characters are accepted by the finite automata, based on the specific patterns.

To accept a string, the final state has to be reached.

num1 ↑↑↑↑ ✓ nu#m1 ↑↑x ✗

Finite Automata Representation

1) Graphical representation (Transition diagram)

2) Tabular form (Transition Table)

3) Mathematical form (Transition function (or) Mapping function)

Formal definition of Finite Automata

A finite Automata is a 5-tuple notation. They are

$M = (Q, \Sigma, \delta, q_0, F)$ where,

Q is a finite set of states

Σ is a finite set of input symbols or alphabet

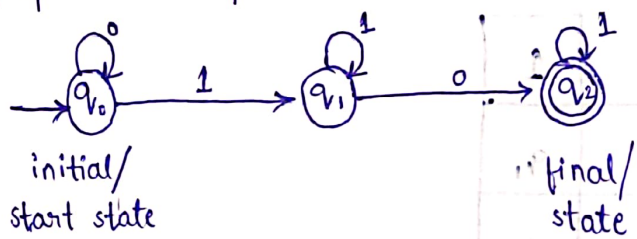
$\delta : Q \times \Sigma \rightarrow Q$ is a transition function.

$q_0 \in Q$ is the start state, also called as Initial state

F is the set of accepted states, also called as final state.

Transition Diagram

It is a directed graph associated with the vertices of a graph corresponds to the state of finite automata.



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

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

q_0 = initial state

$F = q_2$ (final state)

Each state accepts one symbol at a time.

Here, q_0 takes the symbol 0 and stays at q_0 itself

Transition Table

It is the tabular representation of the transition function that takes two arguments (a state and symbol) and returns a value (the next state).

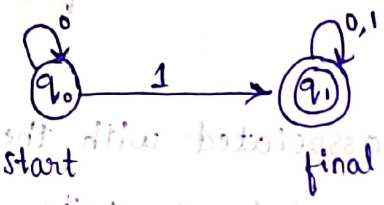
Rows corresponds to states.

Columns correspond to input symbols

Entries correspond to next states

The start state is marked with an arrow. →

The accepted state is marked with *



	symbols / states	0	1
→	q_0	q_0	q_1
*	q_1	q_1	q_1

Transition function / Mapping function

» It is denoted by δ.

» Two parameters are passed to this transition function.

(i) current state

(ii) Input symbol.

» The transition function returns a state, which can be called as next state.

δ (current state, input symbol) = next state.

Ex:-

$\delta(q_0, 0) = q_0$

$\delta(q_0, 1) = q_1$

$\delta(q_1, 0) = q_1$

$\delta(q_1, 1) = q_1$

Types of Finite Automata

- Finite Automata
  - Finite Automata without output
    - Deterministic finite automata (DFA)
    - Non-deterministic finite automata NDFA (or) NFA
    - NFA with ε - moves
  - Finite Automata with output
    - Moore machine
    - Melay machine

13/9/2023

Deterministic Finite Automata (DFA)

1) DFA, if the machine can read an input string one symbol at a time.

2) In DFA, there is only one path for specific input from the current state to ~~rei~~ next state.

$$q_0 \xrightarrow{a} q_1$$

3) DFA does not accept the null move
ie, DFA cannot change state without any input symbol.

4) DFA can contain multiple final states. It is used in lexical analysis in compiler.

Formal Definition of DFA

DFA is a collection of 5 tuple notation,

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : finite set of states

Σ : finite set of input symbols

δ : transition function

q_0 : start state

F : final ~~set~~ state

Acceptance of a Language

Language acceptance is defined by, if a string w is accepted by machine M ie, if it is reaching the final state F by taking the string w.

Not accepted if not reaching the final state.

Ex :-

⇒ $L = \{\} \Rightarrow \emptyset$

If language contains nothing ⇒ q_0 (no string)

It is represented by single state.

⇒ $L = \{\varepsilon\}$ ⇒ q_0 (no move)

initial & final state.

0 len string.

we don't have I/P so, it will not move to next state.

⇒ If we are able to express reg. exp. for ε

It means that our DFA can accept it.

⇒ DFA to accept 'a' ⇒

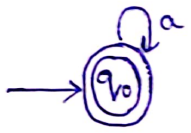
⇒ DFA to accept zero or more no. of a's

$L = \{\varepsilon, a, aa, aaa, \ldots\}$

regular exp. = a^*

No. of states depends on min length string

NOTE : No. of states = min str length + 1

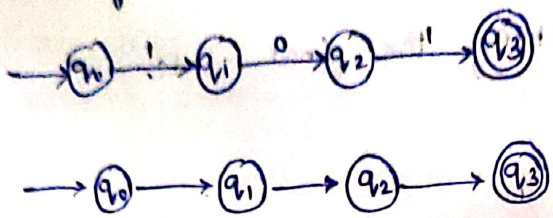


Q) Design DFA which accepts 101 string over

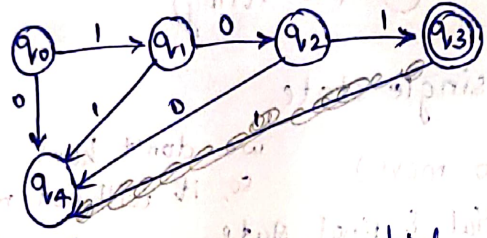
$\Sigma = \{0,1\}$

$L = \{101\}$, length of the string = 3

no. of states = 3+1 = 4



2nd I/P : 0101

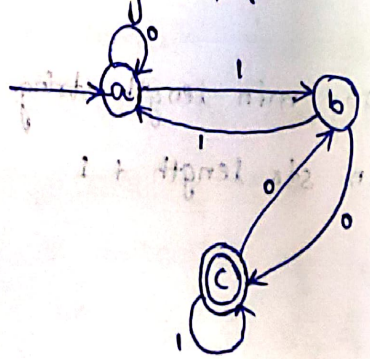


dead / trap state : which will never reach final state.

Q) Let DFA be $Q = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $q_0 = \{a\}$, $F = \{c\}$ and construct a DFA based on transition table

	0	1
a	a	b
b	c	a
c	b	c

Transition diagram :-



Accepted string: $L = \{10, 010, 1000, \ldots\}$

Q) Design DFA $\Sigma = \{a, b\}$

(i) str starting with a

(ii) str ending with a

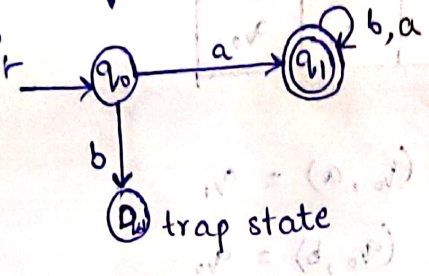
15/09/2023

(i) $L = \{a, ab, abb, aba, ababaa, \ldots\}$

min length str = a => len = 1

no. of states = 1+1 = 2

Transition diagram :-



It is not required to take another state after q_1

~~(ii) L = {a, aa}~~

~~(ii) L = {a, aa, ab, abab~~

Transition table :-

	a	b
→ q_0	q_1	q_d
* q_1	q_1	q_1

Transition function :-

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_d$

$\delta(q_1, a) = q_1$

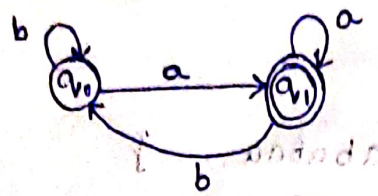
$\delta(q_1, b) = q_1$

(ii) $L = \{a, aa, aba, ba, baa, baba, ababa, \ldots\}$

min length str = a => len = 1

no. of states = 1+1 = 2

Transition diagram :



Transition table :

	a	b
→ q_0	q_1	q_0
* q_1	q_1	q_0

Transition function :

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_0$

$\delta(q_1, a) = q_1$

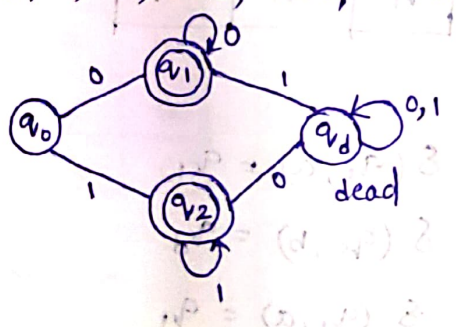
$\delta(q_1, b) = q_0$

Multiple final States

Q) Draw a DFA for the lang accepting strings with 0's and 1's only over the input alphabet, $\Sigma = \{0,1\}$

$L = \{0, 00, 1, 11, 000, 1111, 00000, \ldots\}$

Transition diagram



Transition Table

	0	1
→ q_0	q_1	q_2
* q_1	q_1	q_d
* q_2	q_d	q_2

Transition function :-

$\delta(q_0, 0) = q_1$

$\delta(q_0, 1) = q_2$

$\delta(q_1, 0) = q_1$

$\delta(q_1, 1) = q_d$

$\delta(q_2, 0) = q_d$

$\delta(q_2, 1) = q_2$

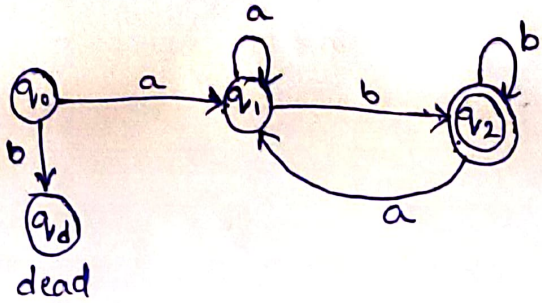
Q) Construct a DFA which accept a language of all the strings starting with 'a' and ending with 'b'.

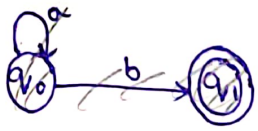
$L = \{ ab, aab, abb, abab, abaabb, \ldots \}$

min length str = 2

no. of states = 2+1 = 3

Transition diagram :-





Transition Table :-

	a	b
→ q_0	q_1	q_d
→ q_1	q_1	q_2
* q_2	q_1	q_2

Transition function :-

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_d$

$\delta(q_1, a) = q_1$

$\delta(q_1, b) = q_2$

$\delta(q_2, a) = q_2$

$\delta(q_2, b) = q_2$

20/9/2023

Non- Deterministic Finite Automata (NFA)

» NFA stands for Non- Deterministic Finite Automata.

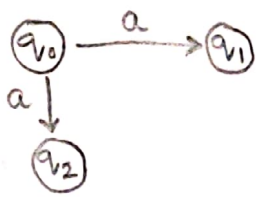
» It is easy to construct an NFA, when compared to DFA for a given regular expression.

» The finite automata are called NFA, when there exists many paths for specific input from the current state to the next state.

» Each NFA can be translated into DFA but, every NFA is non- DFA.

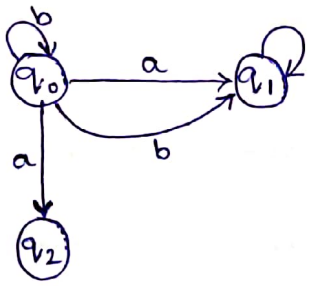
» NFA is defined in the same way as DFA but, with the following two exceptions :-

(i) It contains multiple next paths



(ii) It contains ε- transitions

Ex:-



» NFA also have 5 tuples which are same as DFA, but with different transition function

$$\delta : Q \times \Sigma \longrightarrow 2^Q$$

↳ finite set of states

In DFA,

$$\delta : Q \times \Sigma \rightarrow Q$$

» NFA is defined as a 5 tuple notation

$$M = (Q, \Sigma, \delta, q_0, F) \text{ , where}$$

Q : finite set of states

Σ : finite set of input symbols

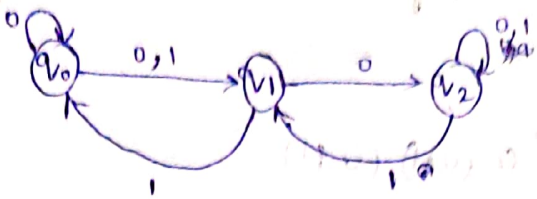
δ : transition function

q_0 : start state

F : end state

Q) Construct the transition table for the given NFA.

$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0,1\}$, $q_0 = \{q_0\}$, $F = \{q_2\}$



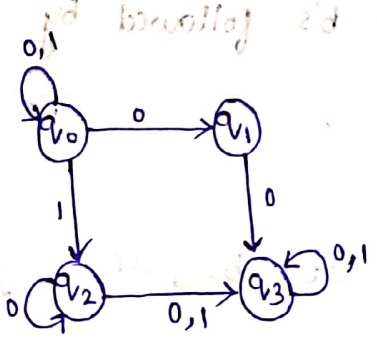
Transition Table :-

	0	1
→ q_0	$\{q_0, q_1\}$	q_1
q_1	q_2	q_0
* q_2	q_2	$\{q_1, q_2\}$

Q) Design NFA transition diagram for the transition table given.

	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_3\}$	
q_2	$\{q_2, q_3\}$	$\{q_3\}$
* q_3	$\{q_3\}$	$\{q_3\}$

Transition diagram :-



Transition function :-

$\delta(q_0, 0) = \{q_0, q_1\}$

$\delta(q_0, 1) = \{q_0, q_2\}$

$\delta(q_1, 0) = \{q_3\}$

$\delta(q_1, 1) = \emptyset$

$\delta(q_2, 0) = \{q_2, q_3\}$

$\delta(q_2, 1) = \{q_3\}$

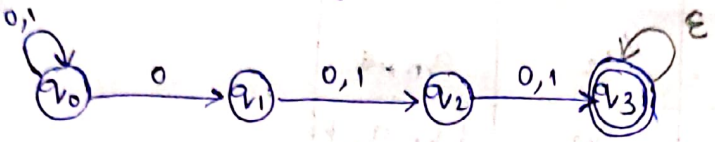
$\delta(q_3, 0) = \{q_3\}$

$\delta(q_3, 1) = \{q_3\}$

Q) Design NFA with $\Sigma = \{0,1\}$ accepting all strings in which the third symbol from the right end is always 0.

RE :- $\underbrace{(0+1)^*}_{\text{self loop}} 0 (0+1)(0+1)$

~~bc~~ & min length str = 4



NFA-ε

It is also a 5-tuple notation

$M = (Q, \Sigma, \delta, q_0, F)$

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

$\Sigma' = \Sigma \cup \{\varepsilon\}$

(NFA-ε) (NFA)

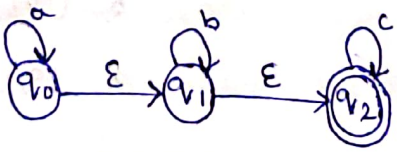
Q) Construct NFA with ε which accepts all strings over the alphabet $\Sigma = \{a, b, c\}$ where, string contains multiple a's followed by multiple b's followed by multiple c's.

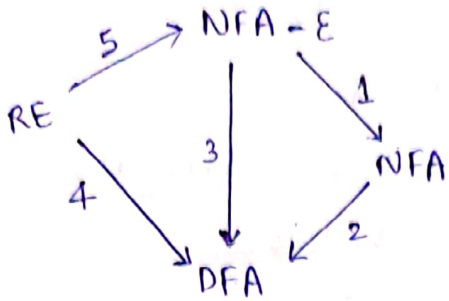
RE = $a^* b^* c^*$

~~L = { a, ab, ac~~ $L = \{a, b, c, ab, bc, ac, \ldots\}$

DFA :- NFA

NFA-ε :-





21/09/2023

① NFA-ε to NFA

2 steps

→ (i) ε - closure (state) = self state + ε-move to other state

→ (ii) $\delta'(q, a)$ = ε- closure $(\delta(\hat{\delta}(q, \varepsilon), a))$

where, $\hat{\delta}(q, \varepsilon) \Rightarrow$ ε- closure (q)

Ex:

NFA - ε

(i) Find ε- closure of states (q_0, q_1, q_2)

ε- closure (q_0) = (self state ~~of~~ for q_0) + (ε-move to other state) → $q_0 \to q_1$ $q_1 \to q_2$

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

ε- closure (q_1) = (self state for q_1) + (ε- move to other state)

$= \{q_1, q_2\}$

ε- closure (q_2) = (self state for q_2) + (ε-move to other state)

$= \{q_2\}$

(ii) $\delta'(q_0, 0)$ = ε- closure $(\delta(\hat{\delta}(q_0, \varepsilon), 0))$

= ε- closure $(\delta(q_0, q_1, q_2), 0)$

= ε- closure $(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

= ε- closure $(q_0 \cup \emptyset \cup \emptyset)$

= ε- closure (q_0)

$\delta'(q_0, 0) = \{q_0, q_1, q_2\}$

$\delta'(q_0, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_0, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), 1)$

$= \varepsilon\text{-closure}(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

$= \varepsilon\text{-closure}(\emptyset \cup q_1 \cup q_2)$

$= \varepsilon\text{-closure}(q_1 \cup q_2)$

$= \varepsilon\text{-closure}(q_1) \cup \varepsilon\text{-closure}(q_2)$

$= \{q_1, q_2\} \cup \{q_2\}$

$\Rightarrow \delta'(q_0, 1) = \{q_1, q_2\}$

$\delta'(q_1, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_1, \varepsilon), 0))$

$= \varepsilon\text{-closure}(\delta(q_1, q_2), 0)$

$= \varepsilon\text{-closure}(\delta(q_1, 0) \cup \delta(q_2, 0))$

$= \varepsilon\text{-closure}(\emptyset \cup \emptyset)$

$= \varepsilon\text{-closure}(\emptyset)$

$\Rightarrow \delta'(q_1, 0) = \emptyset$

$\delta'(q_1, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_1, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_1, q_2), 1)$

$= \varepsilon\text{-closure}(\delta(q_1, 1) \cup \delta(q_2, 1))$

$= \varepsilon\text{-closure}(q_1, q_2)$

$\delta'(q_1, 1) = \{q_1, q_2\}$

$\delta'(q_2, 0) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_2, \varepsilon), 0))$

$= \varepsilon\text{-closure}(\delta(q_2), 0)$

$= \varepsilon\text{-closure}(\delta(q_2, 0))$

$= \varepsilon\text{-closure}(\emptyset)$

$\Rightarrow \delta'(q_2, 0) = \emptyset$

$\delta'(q_2, 1) = \varepsilon\text{-closure}(\delta(\hat{\delta}(q_2, \varepsilon), 1))$

$= \varepsilon\text{-closure}(\delta(q_2), 1)$

$= \varepsilon\text{-closure}(\delta(q_2, 1))$

$= \varepsilon\text{-closure}(q_2)$

$\delta'(q_2, 1) = \{q_2\}$

Transition Table

	0	1
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
q_1	$\emptyset$	$\{q_1, q_2\}$
q_2	$\emptyset$	$\{q_2\}$

Equivalent NFA diagram

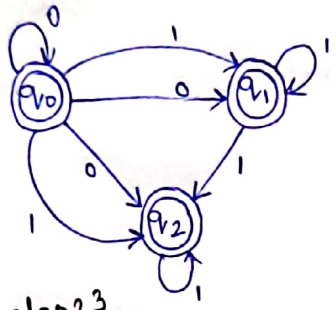
Initial state is always q_0

In NFA-ε, q_2 is final state, so check for

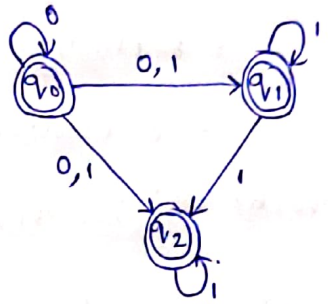
ε-closure (q_0), it contains q_2 so, q_0 is final state

ε-closure (q_1), it contains q_2 so, q_1 is also final state

ε-closure (q_2), it contains q_2 so, q_2 is also final state

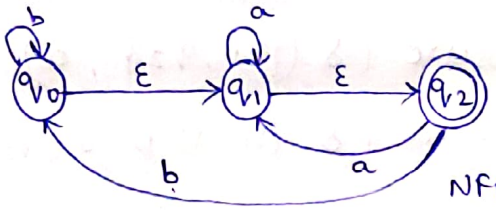


(or)



22/09/2023

Ex (2)



NFA – ε

(i) to find ε-closure (state) here; q_0, q_1, q_2 are states.

= self state + ε-moves to other state

ε-closure (q_0) = self state for q_0 + ε-moves to other state

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

ε-closure (q_1) $= \{q_1, q_2\}$

ε-closure (q_2) $= \{q_2\}$

$$\delta'(q_0, a) = \varepsilon\text{-closure}(\delta(\delta(q_0, \varepsilon), a))$$

$$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0), a))$$

$$= \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}, a))$$

$$= \varepsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2) \cup \delta(q_2, a))$$

$$= \varepsilon\text{-closure}(\emptyset \cup q_1 \cup q_0)$$

$$= \varepsilon\text{-closure}(q_1)$$

$$\Rightarrow \delta'(q_0, a) = \{q_1, q_2\}$$

$$\delta'(q_0, b) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0), b))$$

$$= \varepsilon\text{-closure}(\delta(\{q_0, q_1, q_2\}, b))$$

$$= \varepsilon\text{-closure}(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$$

$$= \varepsilon\text{-closure}(q_0 \cup \emptyset \cup q_0)$$

$$= \varepsilon\text{-closure}(q_0)$$

$$\Rightarrow \delta'(q_0, b) = \{q_0, q_1, q_2\}$$

$$\delta'(q_1, a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_1), a))$$

$$= \varepsilon\text{-closure}(\delta(\{q_1, q_2\}, a))$$

$$= \varepsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$$

$$= \varepsilon\text{-closure}(q_1 \cup q_1)$$

$$= \varepsilon\text{-closure}(q_1)$$

$$\Rightarrow \delta'(q_1, a) = \{q_1, q_2\}$$

$$\delta'(q_1, b) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_1), b))$$

$$= \varepsilon\text{-closure}(\delta(\{q_1, q_2\}, b))$$

$$= \varepsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$$

$$= \varepsilon\text{-closure}(\emptyset \cup q_0)$$

$$= \varepsilon\text{-closure}(q_0)$$

$$\Rightarrow \delta'(q_1, b) = \{q_0, q_1, q_2\}$$

$\delta'(q_2, a) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_2), a))$

$= \varepsilon\text{-closure}(\delta\{q_2\}, a)$

$= \varepsilon\text{-closure}(\delta(q_2, a)) \cancel{\cup \delta(q_2, a)}$

$\cancel{= \varepsilon\text{-closure}(q_1) \cup q_1)}$

$= \varepsilon\text{-closure}(q_1)$

$\Rightarrow \delta'(q_2, a) = \{q_1, q_2\}$

$\delta'(q_2, b) = \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_2), b))$

$= \varepsilon\text{-closure}(\delta(q_2, b))$

$= \varepsilon\text{-closure}(q_0)$

$\Rightarrow \delta'(q_2, b) = \{q_0, q_1, q_2\}$

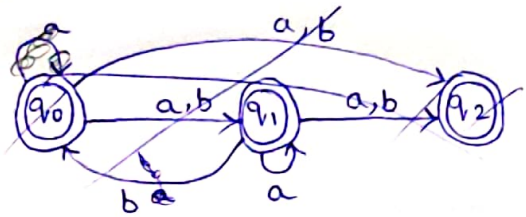
Transition Table

	a	b
q_0	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_2	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

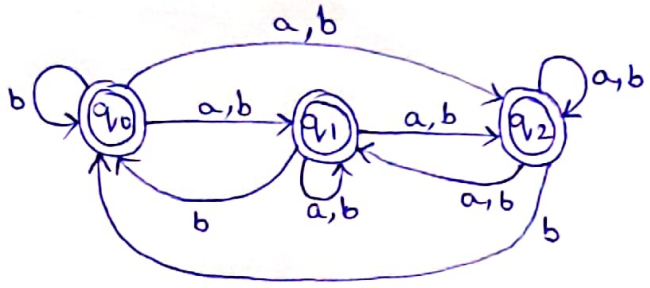
Equivalent NFA diagram

In NFA-ε, q_2 is the final state.

If q_2 is present in ~~each~~ any of the ε-closure (q_0), ε-closure (q_1), ε-closure (q_2) then, it becomes the final state.



Here, q_2 is present in ε-closure (q_0), ε-closure (q_1) and ε-closure (q_2).



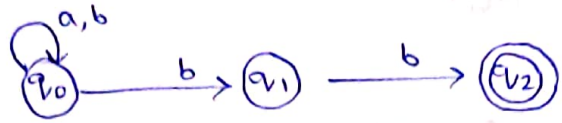
25/9/2023

② NFA to DFA

DFA transition function :- $\delta : Q \times \Sigma \rightarrow Q$

NFA transition function :- $\delta : Q \times \Sigma \rightarrow 2^Q$

Ex (1) :-



Step 1 :- construct NFA transition table.

	a	b
q_0	q_0	q_0, q_1
q_1	ϕ	q_2
q_2	ϕ	ϕ

Step 2 :- construct DFA transition table

	a	b
q_0	q_0	$\{q_0, q_1\}$
q_0, q_1	q_0	$\{q_0, q_1, q_2\}$
q_0, q_1, q_2	q_0	$\{q_0, q_1, q_2\}$

(i) 1st row in DFA TT will be 1st row in NFA TT

(ii) Next row in DFA TT will be last col of its previous row.

$\delta((q_0, q_1), a) \Rightarrow \delta(q_0, a) \cup \delta(q_1, a)$

$\Rightarrow q_0 \cup \phi$

$\Rightarrow q_0$

$\delta((q_0, q_1); b) \Rightarrow \delta(q_0, b) \cup \delta(q_1, b)$

$\Rightarrow (q_0, q_1) \cup (q_2)$

$\Rightarrow q_0, q_1, q_2$

$\delta((q_0, q_1, q_2), a) \Rightarrow \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$

$\Rightarrow q_0 \cup \phi \cup \phi$

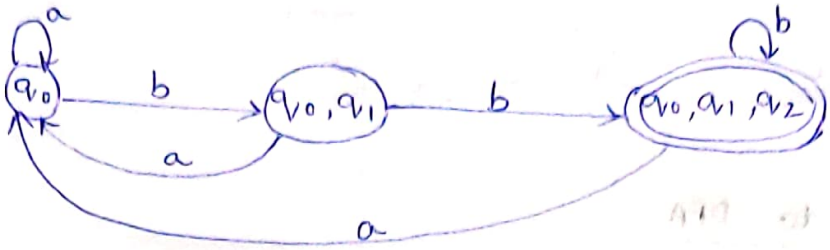
$\Rightarrow q_0$.

$\delta((q_0, q_1, q_2), b) \Rightarrow \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)$

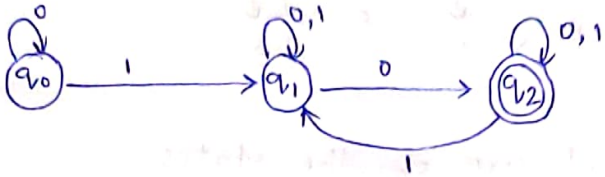
$\Rightarrow (q_0, q_1) \cup q_2 \cup \phi$

$\Rightarrow q_0, q_1, q_2$

step-3 :- construct transition diagram for DFA



Ex (2) :-



step 1 :- construct NFA transition table

	0	1
q_0	q_0	q_1
q_1	q_1, q_2	q_1
q_2	q_2	q_1, q_2

step 2 :- construct DFA transition table

	0	1
q_0	q_0	q_1
q_1	q_1, q_2	q_1
q_1, q_2	q_1, q_2	q_1, q_2

$\delta(q_1, 0) = q_1, q_2$

$\delta(q_1, 1) = q_1$

$\delta((q_1, q_2), 0) = \delta(q_1, 0) \cup \delta(q_2, 0)$

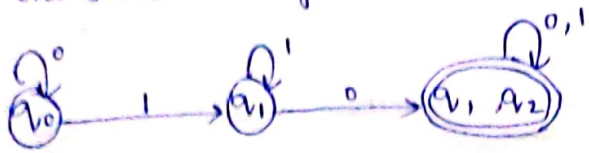
$= (q_1, q_2) \cup q_2$

$= q_1, q_2$

$\delta((q_1, q_2), 1) = \delta(q_1, 1) \cup \delta(q_2, 1)$

$= q_1, q_2$

step3 :- DFA transition diagram.



③ NFA-ε to DFA

Ex (1):

Step1 :- Find ε-closure of the states

ε-closure (state) = self state + ε-move to other state

ε-closure (q_0) = $\{q_0, q_1, q_2\}$

ε-closure (q_1) = $\{q_1, q_2\}$

ε-closure (q_2) = $\{q_2\}$

Step2 :- construct DFA transition table

(i) 1st state in DFA is ε-closure of q_0 (initial state)

		0	1
*	q_0, q_1, q_2	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
*	q_1, q_2	ϕ	$\{q_1, q_2\}$

$\delta((q_0, q_1, q_2), 0)$ = ε-closure $\delta((q_0, q_1, q_2), 0)$

$\Rightarrow$ ε-closure $(\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0))$

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

$\delta((q_0, q_1, q_2), 1)$ = ε-closure $(\delta(q_0, q_1, q_2), 1)$

= ε-closure $(\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1))$

= ε-closure $(\phi \cup q_1 \cup q_2)$

= ε-closure $(q_1 \cup q_2)$

= ε-closure (q_1) ∪ ε-closure (q_2)

= $\{q_1, q_2\} \cup \{q_2\}$

$\Rightarrow \delta((q_0, q_1, q_2), 1)$ = $\{q_1, q_2\}$

$\delta((q_1, q_2), 0) = \varepsilon\text{-closure}(\delta(q_1, q_2), 0)$

$= \varepsilon\text{-closure}(\delta(q_1, 0) \cup \delta(q_2, 0))$

$= \varepsilon\text{-closure}(\phi \cup \phi)$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta((q_1, q_2), 1) = \varepsilon\text{-closure}(\delta(q_1, q_2), 1)$

$= \varepsilon\text{-closure}(\delta(q_1, 1) \cup \delta(q_2, 1))$

$= \varepsilon\text{-closure}(q_1 \cup q_2)$

$= \varepsilon\text{-closure}(q_1) \cup \varepsilon\text{-closure}(q_2)$

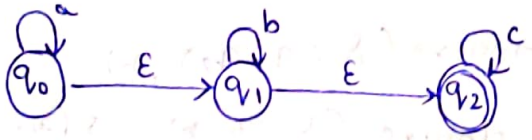
$= \{q_1, q_2\} \cup \{q_2\}$

$= \{q_1, q_2\}$

<u>step-3</u> :- Draw transition diagram for DFA

(or)

<u>Ex (2)</u> :-



<u>step 1</u> : ε-closure of states

ε-closure $(q_0) = \{q_0, q_1, q_2\}$

ε-closure $(q_1) = \{q_1, q_2\}$

ε-closure $(q_2) = \{q_2\}$

<u>step 2</u> : DFA transition table

	a	b	c
q_0, q_1, q_2			
q_1, q_2			
q_2			

$\delta((q_0, q_1, q_2), a) = \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), a)$

$= \varepsilon\text{-closure}(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$

$= \varepsilon\text{-closure}(q_0)$

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

$\delta((q_0, q_1, q_2), b) = \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), b)$

$= \varepsilon\text{-closure}(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(\phi \cup q_1 \cup \phi)$

$= \varepsilon\text{-closure}(q_1)$

$= \{q_1, q_2\}$

$\delta((q_0, q_1, q_2), c) = \varepsilon\text{-closure}(\delta(q_0, q_1, q_2), c)$

$= \varepsilon\text{-closure}(\delta(q_0, c) \cup (q_1, c) \cup (q_2, c))$

$= \varepsilon\text{-closure}(q_2)$

$= \{q_2\}$

$\delta((q_1, q_2), a) = \varepsilon\text{-closure}(\delta(q_1, q_2), a)$

$= \varepsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a)) \cancel{\cup \delta q_2}$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta((q_1, q_2), b) = \varepsilon\text{-closure}(\delta(q_1, q_2), b)$

$= \varepsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \varepsilon\text{-closure}(q_1)$

$= \{q_1, q_2\}$

$\delta((q_1, q_2), c) = \varepsilon\text{-closure}(\delta(q_1, q_2), c)$

$= \varepsilon\text{-closure}(\delta(q_1, c) \cup \delta(q_2, c))$

$= \varepsilon\text{-closure}(q_2)$

$= \{q_2\}$

$\delta(q_2, a) = \varepsilon\text{-closure}(\delta(q_2, a))$

$= \varepsilon\text{-closure}(\phi)$

$= \phi$

$\delta(q_2, b) = \varepsilon\text{-closure}(\delta(q_2, b))$

$= \varepsilon\text{-closure}(\emptyset)$

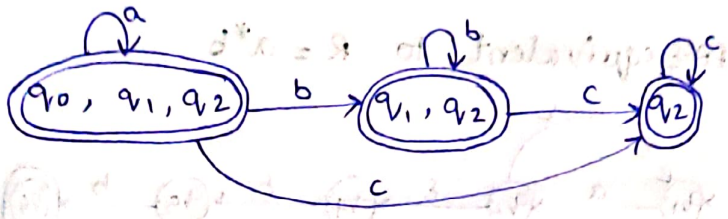
$= \emptyset$

$\delta(q_2, c) = \varepsilon\text{-closure}(\delta(q_2, c))$

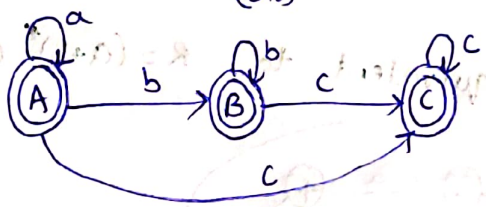
$= \varepsilon\text{-closure}(q_2)$

$= q_2$

step 3 : DFA transition diagram



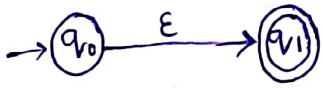
(or)



27/09/2023

(5) Regular Expression to NFA-ε

(1) RE = ε (null string)

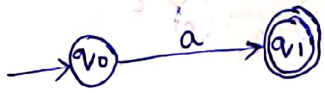


states can be changed without any input.

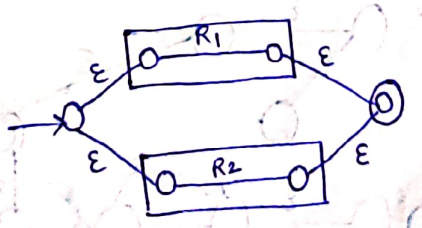
(2) RE = ∅ (empty set of strings)

no transition

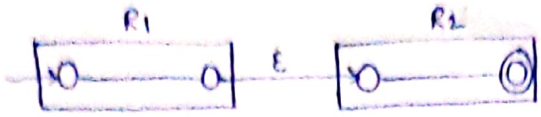
(3) RE = a



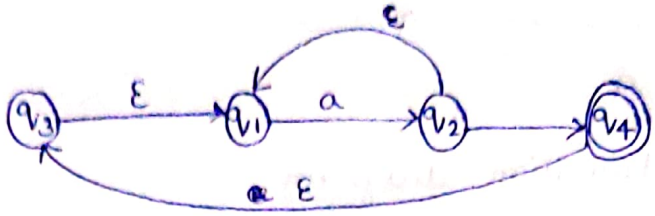
(4) union : RE = $R_1 + R_2$ (or) $R_1 | R_2$



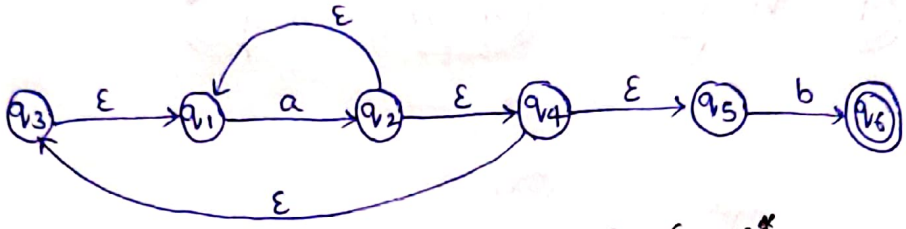
(5) concatenation : $RE = R_1 \cdot R_2$ (or) $R_1 R_2$



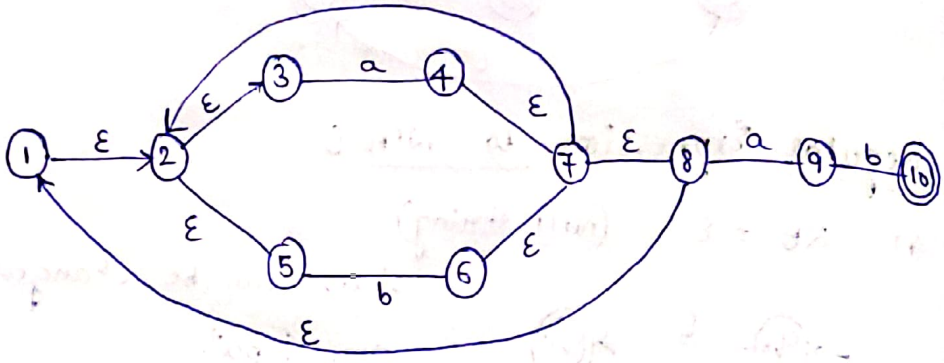
(6) closure : $RE = a^*$



Q) construct NFA-ε equivalent to $R = a^*b$

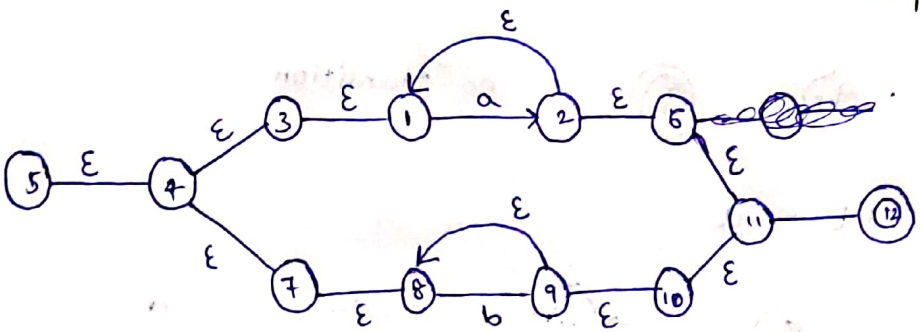


Q) construct NFA-ε equivalent to $R = (a+b)^* ab$

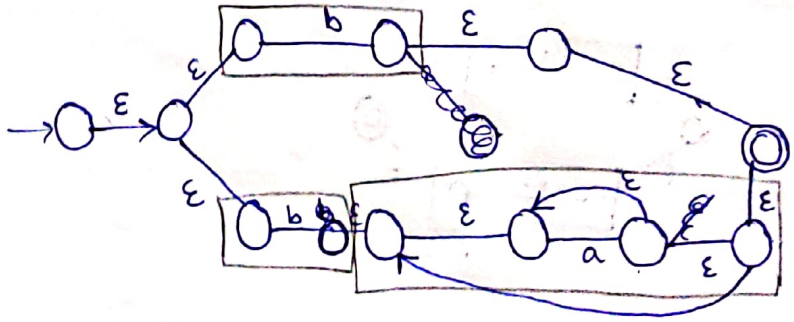


Q) Construct NFA-ε equivalent to $R = a^* + b^*$ (or)

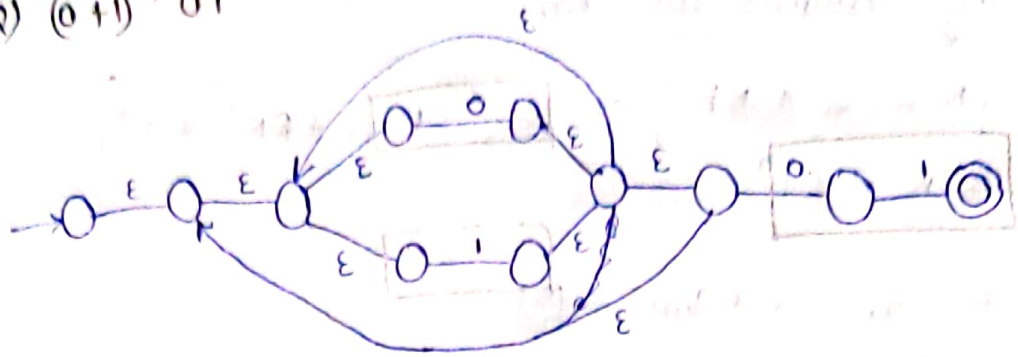
$R = a^* | b^*$



Q) $RE = b + ba^*$



Q) $(0+1)^* 01$



Shanta
27/9/23

④ DFA to RE

step (i) For each state q_0, q_1, q_2, identify all the states that comes into the state. and, write in equation format.

step (ii) For initial state, add ε

step (iii) : Calculate all the equations.

step (iv) : Result is the value of the final state..

Q)

(i) write the eqns for each state individually by considering only incoming edges.

$$q_1 = q_1 0 + \varepsilon \quad \text{—(1)}$$

q_1 is initial state

$$q_2 = q_1 1 + q_2 1 \quad \text{—(2)}$$

$$q_3 = q_2 0 + q_3 0 + q_3 1 \quad \text{—(3)}$$

Now, simplify the eqns including final states.

If there are multiple final states then, apply union operation.

By using Arden's Theorem

$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix} \text{—④} \quad ; \quad [\varepsilon + RR^* = R^*]$$

eq ① $\Rightarrow$ $q_1 = \varepsilon + q_1 0$ —⑤

on comparing eq's ④, ⑤ ; we get

$R = q_1$, $Q = \varepsilon$, $RP = q_1 0$

$\therefore$ $q_1 = 0^*$

eq ② $\Rightarrow$ $q_2 = q_1 1 + q_2 1$

$\Rightarrow$ $q_2 = 0^* 1 + q_2 1$ —⑥

on comparing eq's ④, ⑥ ; we get

$q_2 = 0^* 1 1^*$

$\therefore$ Resultant Regular Expression is,

$RE = q_1 + q_2$

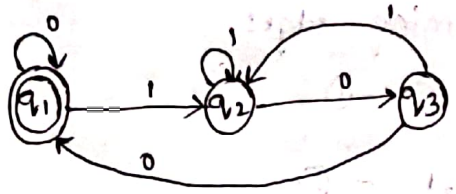
($\because$ q_1 and q_2 are final states, we ~~appl~~ perform union operation on q_1 and q_2)

$\Rightarrow$ $RE = 0^* + 0^* 1 1^*$

$= 0^* (\varepsilon + 11^*)$

$= 0^* 1^*$ $\quad (\because \varepsilon + 11^* = 1^*)$

Q)



A) $q_1 = q_1 0 + q_3 0 + \varepsilon$ —①

$q_2 = q_2 1 + q_1 1 + q_3 1$ —②

$q_3 = q_2 0$ —③

The above eq^ns cannot be simplified by using Aden's Theorem.

substitute eq (3) in eq (2)

$\therefore$ eq (2) $\Rightarrow q_2 = q_2 1 + q_1 1 + q_2 01$

$\Rightarrow q_2 = q_1 1 + q_2 (1 + 01)$

$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix}$$

$\Rightarrow q_2 = q_1 1 (1+01)^*$ — (4)

Now, eq (3) $\Rightarrow q_3 = q_1 1 (1+01)^* 0$ — (5)

Then, eq (1) $\Rightarrow q_1 = q_1 0 + q_1 1 (1+01)^* 00 + \varepsilon$

$\Rightarrow q_1 = q_1 (0 + 1 (1+01)^* 00) + \varepsilon$

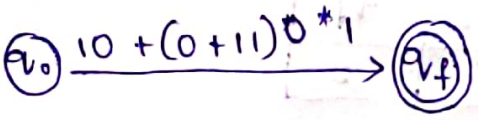
$\Rightarrow q_1 = \varepsilon + q_1 (0 + 1 (1+01)^* 00)$

$$\begin{bmatrix} R = Q + RP \\ R = QP^* \end{bmatrix}$$

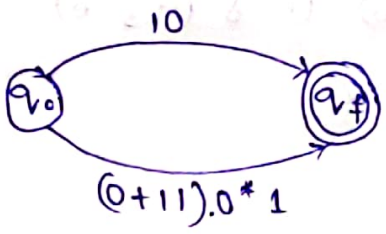
$\Rightarrow q_1 = (0 + 1 (1+01)^* 00)^*$

Q) Design DFA from the given RE: $10 + (0+11) 0^* 1$

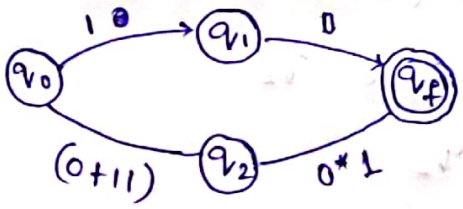
step 1 :-



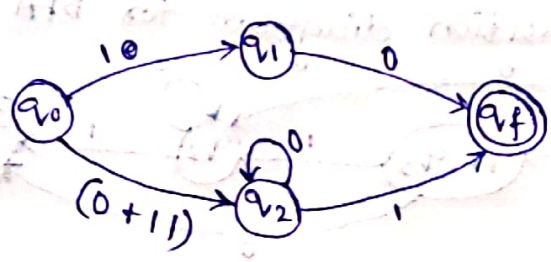
step 2 :-



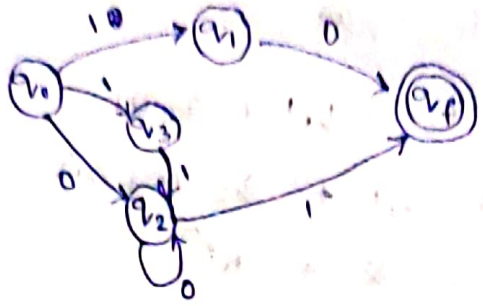
step 3 :-



step 4 :-



step 5 :-



NFA

step 6 :- Transition Table for NFA

	0	1
q_0	q_2, q_3	q_1, q_3
q_1	q_f	$\emptyset$
q_2	q_2	q_f
q_3	$\emptyset$	q_2

step 7 :- Transition Table for DFA

	0	1
q_0	q_2	q_1, q_3
q_1, q_3	q_f	q_2
q_2	q_2	q_f
q_f	$\emptyset$	$\emptyset$

$$\delta((q_1, q_3), 0) = \delta(q_1, 0) \cup \delta(q_3, 0)$$
$$= q_f \cup \emptyset$$
$$= q_f$$

$$\delta((q_1, q_3), 1) = \delta(q_1, 1) \cup \delta(q_3, 1)$$
$$= \emptyset \cup q_2$$
$$= q_2$$

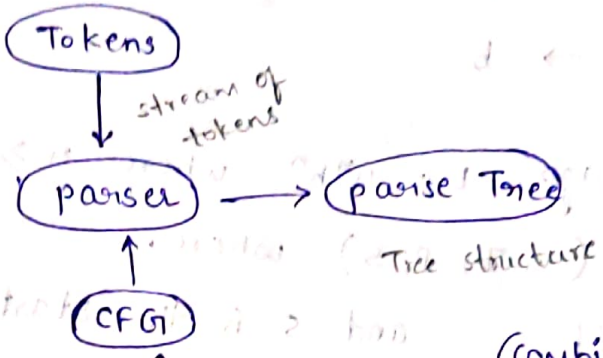
step 8 :- Transition diagram for DFA

03/10/2023

UNIT - 2

Syntax Analysis

A parser or syntax analysis is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntactic errors.



production rules - (combinat^n of terminal & non-terminal symbols)

$S = T + const$

start

non-terminal → other than capitals

terminal → capitals

Every non terminal symbol is ended with ~~non-~~terminal

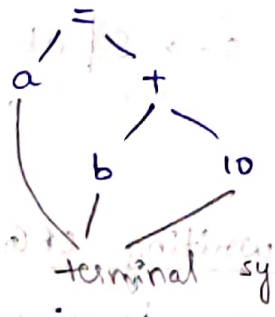
Ex: $a = b + 10$

$S \rightarrow T + const$

$T \rightarrow b$

$const \rightarrow 10$

$S \rightarrow a$



Parse Tree is generated by using terminal & non-terminal symbols.

Context Free Grammar (CFG)

The Context Free Grammar is a collection of the following :-

lexical analyzer & syntax analyzer work together

$G = (V, T, P, S)$

where, V : set of non-terminal symbols

T : set of terminal symbols (other than capitals)

S : start symbol

P : set of production rules

Ex: (S, V, T, A, B)

The Production rules are given in the following form

Non Terminal $\longrightarrow (V \cup T)^*$

Ex: $E \rightarrow E + E \mid E * E$ (or) $\cong$ $E \rightarrow E + E$ (non Ter) (Terminal)

$E \rightarrow E * E$

$E \rightarrow T \mid T * F$

$T \rightarrow a$

$F \rightarrow b$

Q) The language $L = a^n b^n$ where $n \geq 1$

Let $G = (V, T, P, S)$ where,

$V = \{S\}$ and S is the start symbol.

$T = \{a, b\}$ Give the production rules ie, $P = ?$

Ans:- $n \geq 1$ implies that $L = \{ab, aabb, aaabbb, \ldots\}$

$\downarrow$ min length string

$P = \{$

$S \rightarrow aSb$ $\quad \because a \varepsilon b = ab$

$S \rightarrow \varepsilon \mid ab$ for only ab

$S \rightarrow \varepsilon \mid aSb \mid$ for any no. of a's and b's

$\}$

$a \underset{\varepsilon}{S} b$

$\boxed{ab}$

$a \underset{\varepsilon}{a S b} b$

$\boxed{aabb}$

aSb

$a a S b b$

$a a \underset{\varepsilon}{S} b b$

$\boxed{aaabbb}$

Rules for writing CFG

1) A single non-terminal should be at LHS
2) The rule should be always in the form of LHS $\longrightarrow$ RHS where, RHS may be the combination of terminal and non-terminal symbols and LHS is non-terminal symbol
3) The null derivation can be specified as non-terminal $\longrightarrow \varepsilon$
4) One of the non-terminals should be start symbol and conventionally, we should write the write the rules for these non-terminal symbols.

Q) Write a CFG for while statement in C language

Ans:- P = { S → while (condition) stmt

→ non-terminal

condition → id relOp id

relOp → > | >= | < | <= | != | ==

S → while (condition) stmt L

stmt L → stmt L | stmt

while (cond)

{

while (cond)

{

}

}

V = { S, condition, relOp, stmt L }

T = { while, (,), <, >, <=, >=, !=, ==

Q) what is the string generated by CFG A → (A) A

Every non-terminal symbol should be ended with the terminal symbol.

Here, A is not ending with terminal symbol.

As there is no terminal symbol generation by the given CFG, we cannot derive any string. from

A → (A) A

A → ( ) A

A → ( ) | ε

Possible strings :-

take, A → (A) A

A → (ε) ε

A → ( )

A → ((( )) ( ).

A → (( )) ε

A → (( ))

Derivation and Parse Trees

Derivation from S means, generation of string w from S.

For constructing derivation, two rules are important.

(1) Choice of non-terminal from several others

(2) Choice of rule from production rules for corresponding non-terminal.

Derivation can be of two types

1) Left most derivation (LMD) replace non-ter with ter sym from left no

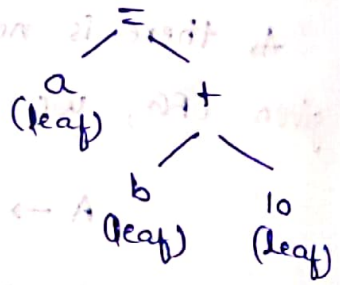
2) Right most derivation (RMD) replace ter with non Ter from right most side

Parsing:

The syntax Analyzer is the second phase in compilation. The syntax analyzer (parser) basically checks for the syntax of language.

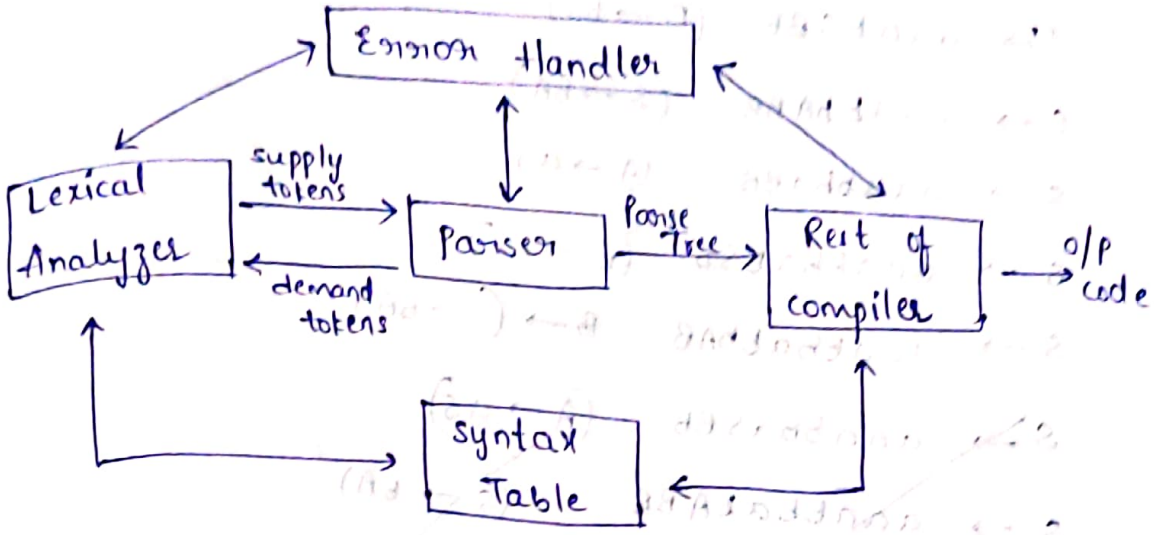
Ex: a = b + 10

This statement is first given to lexical analyzer. It will divide into group of tokens. Syntax analyzer takes the tokens as input and generates a tree like structure called parse tree.



Role of Parser

In the process of compilation, the parser and LA work together. That means, when parser requires stream of tokens, it invokes lexical analyzer. In return, the lexical analyzer supplies tokens to syntax analyzer.



The parser collects sufficient no. of tokens and builds a parse tree. Thus by building the parse tree, parser finds the syntactical error if any. It is also necessary that, the parser should recover from commonly occuring errors, so that remaining task of process the i/p can be continued.

Why Lexical analyzer and Syntax analyzer are separated out?

→ It speeds up the process of compilation

→ The errors in the source input can be identified precisely (correctly).

04/10/2023

Q) Let G be Context Free Grammar for which the production rules are given as below.

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB.$

Derive the string aaabbabbba using above grammar.

Ans:

$w = aaabbabbba$

$S \rightarrow aB$

$\cancel{S \rightarrow aaBB \quad (B \rightarrow aBB)}$

$S \rightarrow aaaBBB \quad (B \rightarrow aBB)$

$\cancel{S \rightarrow aaabBB \quad (B \rightarrow b)}$

$S \to aaabSBB$ $(B \to bS)$

$S \to aaabbABB$ $(S \to bA)$

$S \to aaabbaBB$ $(A \to a)$

$S \to aaabbabSB$ $(B \to bS)$

$S \to aaabbabbAB$ $(S \to bA)$

$S \to aaabbaSBB$ $(A \to aS)$

$S \to aaabbabABB$ $(S \to bA)$

$S \to aaabbabb$ $(A \to bAA)$

LMD

$S \to aB$

$S \to aaBB$ $(B \to aBB)$

$S \to aaaBBB$ $(B \to aBB)$

$S \to aaabBB$ $(B \to b)$

$S \to aaabbB$ $(B \to b)$

$S \to aaabbaBB$ $(B \to aBB)$

$S \to aaabbabB$ $(B \to b)$

$S \to aaabbabbS$ $(B \to bS)$

$S \to aaabbabbbA$ $(S \to bA)$

$S \to aaabbabbba$ $(A \to a)$

RMD

$S \to aB$

$S \to aaBB$ $(B \to aBB)$

~~$S \to aaaBBB$ $(B \to aBB)$~~

$S \to aaBbS$ $(B \to bS)$

$S \to aaBbbA$ $(S \to bA)$

$S \to aaBbba$ $(A \to a)$

$S \to aaaBBbba$ $(B \to aBB)$

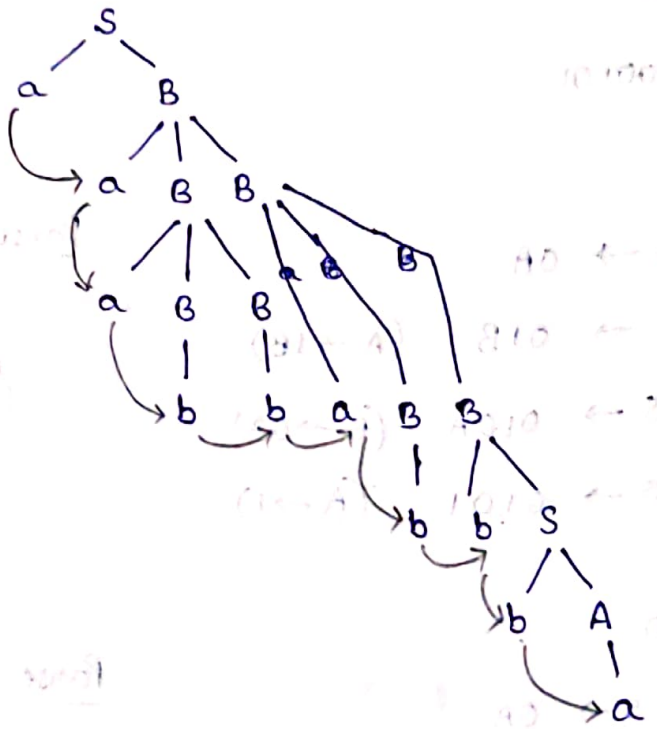
$S \to aaaBbbba$ $(B \to b)$

$S \rightarrow aaabSbbba \quad (B \rightarrow bS)$

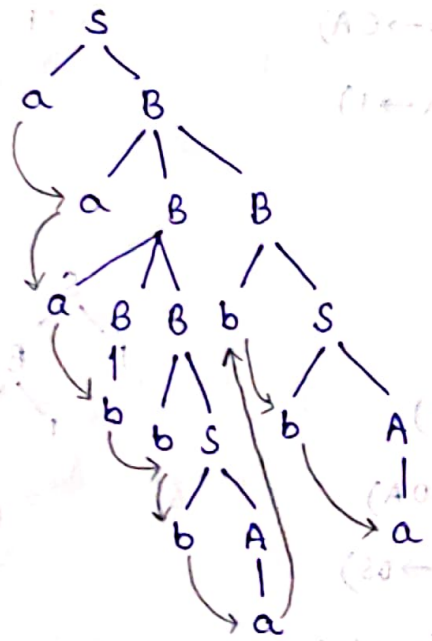
$S \rightarrow aaabbAbbba \quad (S \rightarrow bA)$

$S \rightarrow aaabbabbba \quad (A \rightarrow a)$

Parse Tree for LMD



Parse Tree for RMB



Q) $S \rightarrow 0A \mid 1B \mid 0 \mid 1$

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

$B \rightarrow 0A \mid 1S$

(i) 0101

(ii) 1100101

Ans:

(i) LMD

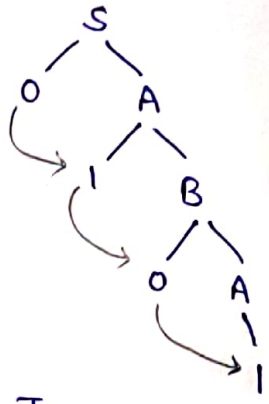
$S \rightarrow 0A$

$S \rightarrow 01B \quad (A \rightarrow 1B)$

$S \rightarrow 010A \quad (B \rightarrow 0A)$

$S \rightarrow 0101 \quad (A \rightarrow 1)$

Parse Tree



RMD

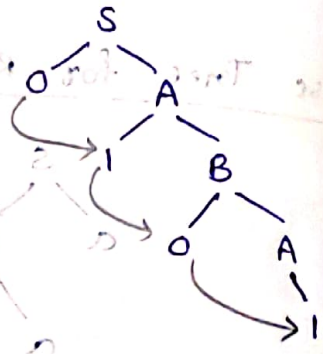
$S \rightarrow 0A$

$S \rightarrow 01B \quad (A \rightarrow 1B)$

$S \rightarrow 010A \quad (B \rightarrow 0A)$

$S \rightarrow 0101 \quad (A \rightarrow 1)$

Parse Tree



(ii) LMD

$S \rightarrow 1B$

$S \rightarrow 11S \quad (B \rightarrow 1S)$

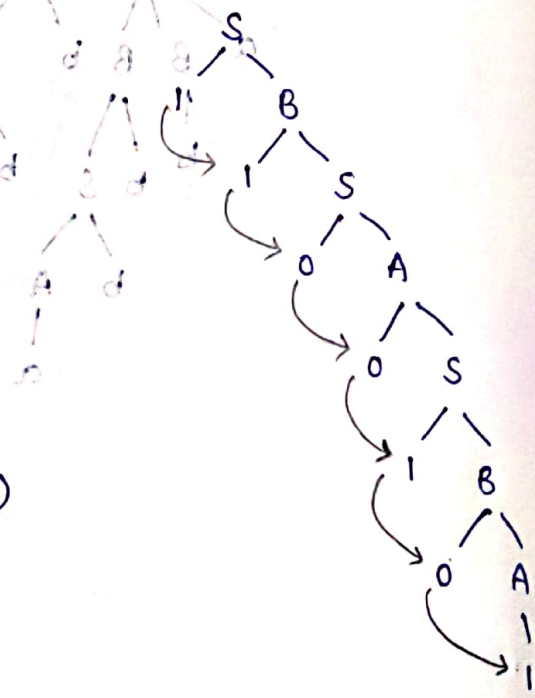
$S \rightarrow 110A \quad (S \rightarrow 0A)$

$S \rightarrow 1100S \quad (A \rightarrow 0S)$

$S \rightarrow 11001B \quad (S \rightarrow 1B)$

$S \rightarrow 110010A \quad (B \rightarrow 0A)$

$S \rightarrow 1100101 \quad (A \rightarrow 1)$



RMD

$S \to 1B$

$S \to 11S \quad (B \to 1S)$

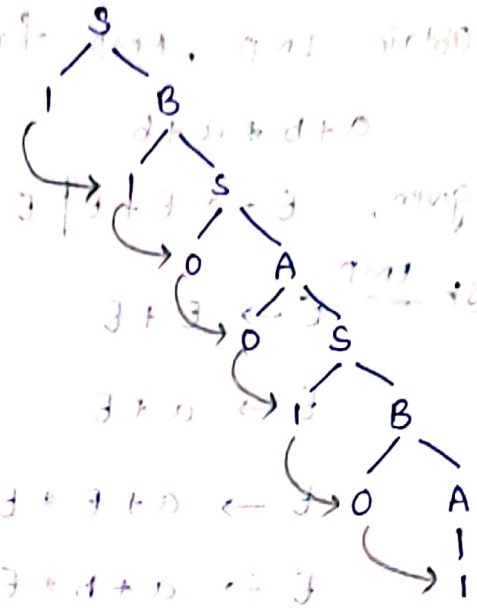
$S \to 110A \quad (S \to 0A)$

$S \to 1100S \quad (A \to 0S)$

$S \to 11001B \quad (S \to 1B)$

$S \to 110010A \quad (B \to 0A)$

$S \to 1100101 \quad (A \to 1)$



HW

Q) Consider the grammar

$S \to (L) \mid a$

$L \to L, S \mid S$

(i) what are the terminals and non-terminals and start symbol

(ii) Find the parse trees for the following sentences.

(1) (a,a)

(2) (a, (a, a))

(3) (a, ((a,a), (a,a)))

(iii) construct LMD and RMD

(iv) what languages thus the grammar generate.

Ans (iv) This grammar generates the strings with well-formedness of parantheses.

(i) Terminal symbols :-

$T = \{ (,), , , a \}$

Non-terminal Symbols :-

$V = \{ L, S \}$

start symbol :- S

$S \to (L) \mid a$

Q) Obtain LMD, RMD for the string

$a+b*a+b$

given, $E \rightarrow E+E \mid E-E \mid E*E \mid E \mid a \mid b$

Ans: LMD

$E \rightarrow E+E$

$E \rightarrow a+E \quad (E \rightarrow a)$

$E \rightarrow a+E*E \quad (E \rightarrow E*E)$

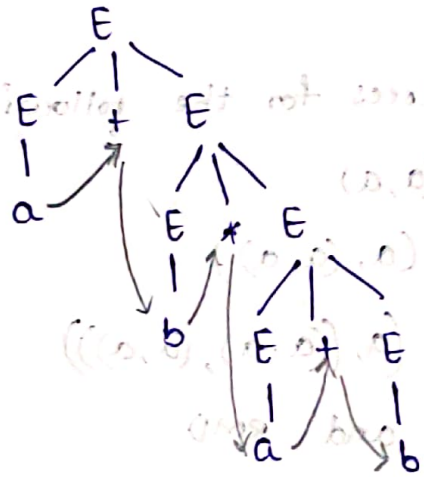
$E \rightarrow a+b*E \quad (E \rightarrow b)$

$E \rightarrow a+b*E+E \quad (E \rightarrow E+E)$

$E \rightarrow a+b*a+E \quad (E \rightarrow a)$

$E \rightarrow a+b*a+b \quad (E \rightarrow b)$

Parse Tree



RMD

$E \rightarrow E+E$

$E \rightarrow E+b \quad (E \rightarrow b)$

$E \rightarrow E*E+b \quad (E \rightarrow E*E)$

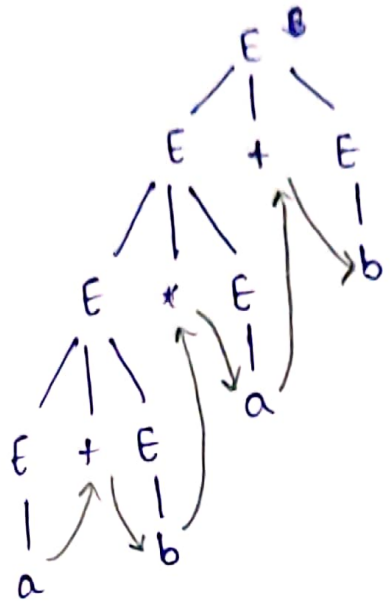
$E \rightarrow E*a+b \quad (E \rightarrow a)$

$E \rightarrow E+E*a+b \quad (E \rightarrow E+E)$

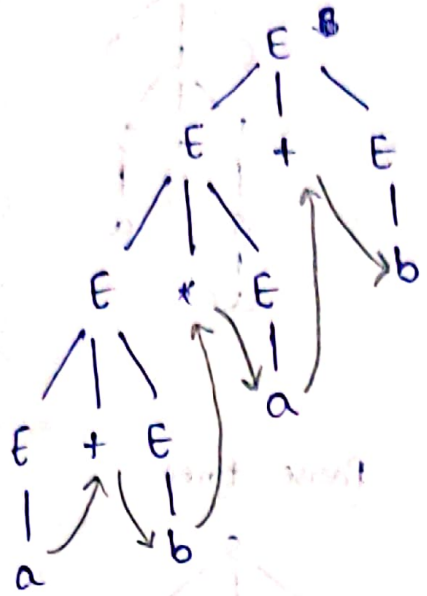
$E \rightarrow E+b*a+b \quad (E \rightarrow b)$

$E \rightarrow a+b*a+b \quad (E \rightarrow a)$

Parse Tree



Parse Tree



HW

Sol :-

(ii), (iii)

i) (a, a)

LMD :-

$S \to (L) \cancel{|a}$

$S \to (L, S)$ $(L \to L, S)$

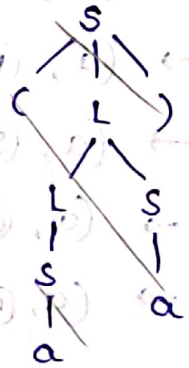
$S \to (S, S)$ $(L \to S)$

$S \to (a, S)$ $(S \to a)$

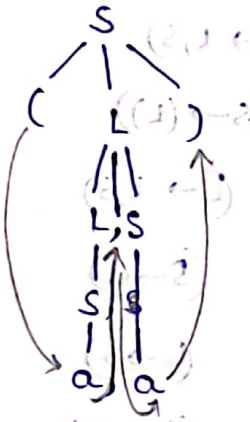
$S \to (a, a)$

$S \to (L) \mid a$

$L \to L, S \mid S$



Parse tree :-



RMD:-

$S \to (L)$

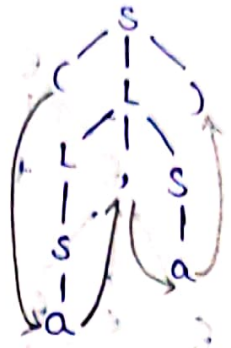
$S \to (L, S)$ $\quad (L \to L, S)$

$S \to (L, a)$ $\quad (S \to a)$

$S \to (S, a)$ $\quad (L \to S)$

$S \to (a, a)$ $\quad (S \to a)$

Parse tree



2) $(a, (a, a))$

LMD:-

$S \to (L)$

$S \to (L, S)$ $\quad (L \to L, S)$

$S \to (S, S)$ $\quad (L \to S)$

$S \to (a, S)$ $\quad (S \to a)$

$S \to (a, (L))$ $\quad (S \to (L))$

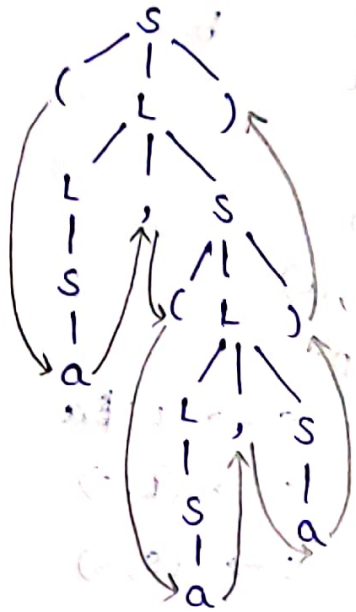
$S \to (a, (L, S))$ $\quad (L \to L, S)$

$S \to (a, (S, S))$ $\quad (L \to S)$

$S \to (a, (a, S))$ $\quad (S \to a)$

$S \to (a, (a, a))$ $\quad (S \to a)$

Parse tree



RMD:-

$S \to (L)$

$S \to (L, S)$ $\quad (L \to L, S)$

$S \to (L, (L))$ $\quad (S \to (L))$

$S \to (L, (L, S))$ $\quad (L \to L, S)$

$S \to (L, (L, a))$ $\quad (S \to a)$

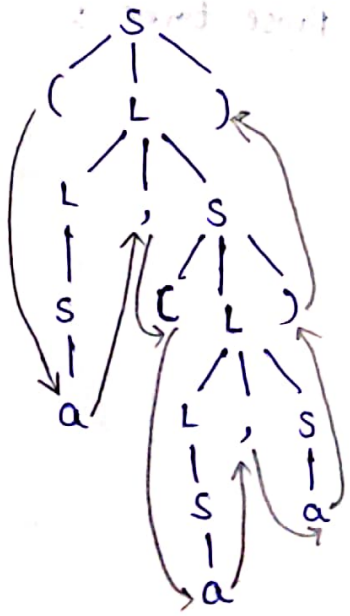
$S \to (L, (S, a))$ $\quad (L \to S)$

$S \to (L, (a, a))$ $\quad (S \to a)$

$S \to (S, (a, a))$ $\quad (L \to S)$

$S \to (a, (a, a))$ $\quad (S \to a)$

Parse tree



3) (a, ((a,a), (a,a)))

LMD:

$S \to (L)$

$S \to (L,S)$ $(L \to L,S)$

$S \to (S,S)$ $(L \to S)$

$S \to (a, S)$ $(S \to a)$

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

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

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

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

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

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

$S \to (a, ((a,S),S))$ $(S \to a)$

$S \to (a, ((a,a),S))$ $(S \to a)$

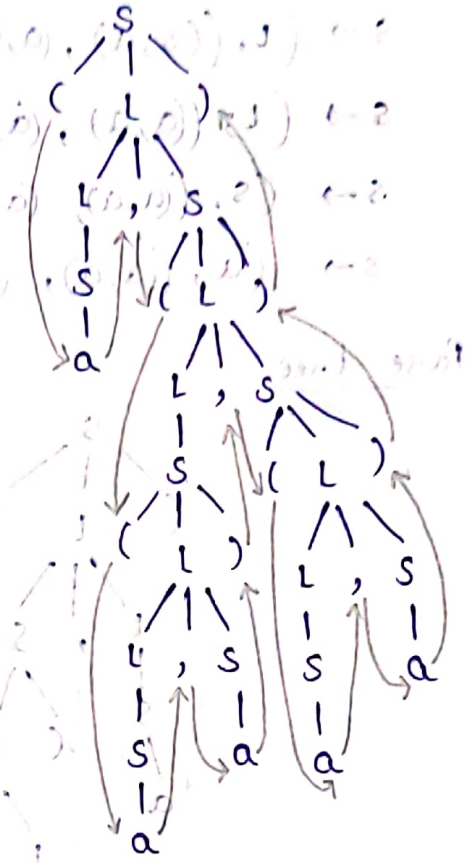
$S \to (a, ((a,a),(L)))$ $(S \to (L))$

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

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

$S \to (a, ((a,a),(a,S)))$ $(S \to a)$

$S \to (a, ((a,a),(a,a)))$ $(S \to a)$



RMD:

$S \to (L)$

$S \to (L,S)$ $(L \to L,S)$

$S \to (L,(L))$ $(S \to (L))$

$S \to (L,(L,S))$ $(L \to L,S)$

$S \to (L,(L,(L)))$ $(S \to (L))$

$S \to (L,(L,(L,S)))$ $(L \to L,S)$

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

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

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

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

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

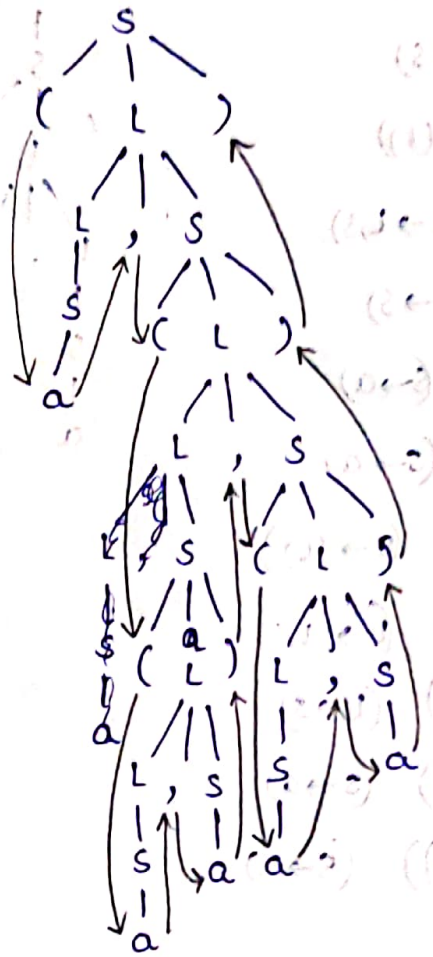
$S \to (L, ((S,a), (a,a)))$ $(L \to S)$

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

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

$S \to (a, ((a,a), (a,a)))$ $(S \to a)$

Parse tree



05/10/2023

Ambiguous Grammar

A grammar G is said to be ambiguous if it generates more than one parse tree for sentences of language L(G)

We need to remove the ambiguity so that there would be a single parse tree.

To eliminate the ambiguity, we use some techniques

Q) $E \rightarrow E+E \mid E*E \mid (E) \mid id$

(i) $w = id + id + id$

LMD

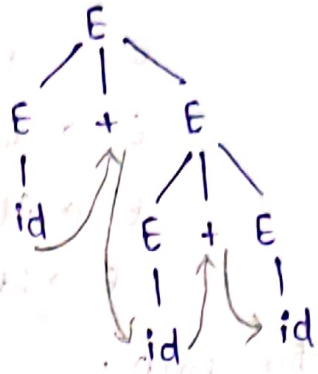
$E \rightarrow E + E$

$E \rightarrow id + E \quad (E \rightarrow id)$

$E \rightarrow id + E + E \quad (E \rightarrow E+E)$

$E \rightarrow id + id + E \quad (E \rightarrow id)$

$E \rightarrow id + id + id \quad (E \rightarrow id)$



RMD

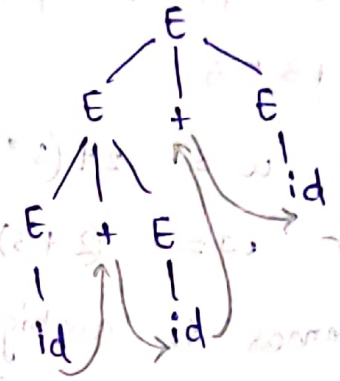
$E \rightarrow E+E$

$E \rightarrow E + id$

$E \rightarrow E + E + id$

$E \rightarrow E + id + id$

$E \rightarrow id + id + id$



(ii) $w = id + id * id$

LMD

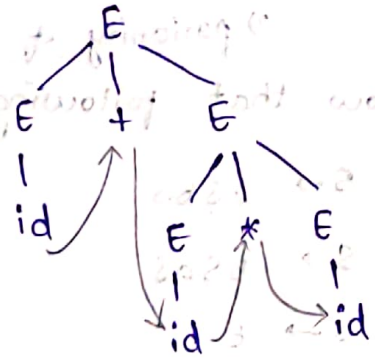
$E \rightarrow E + E$

$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse Tree 1

RMD

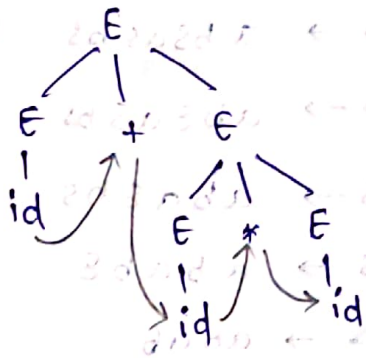
$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$



LMD:

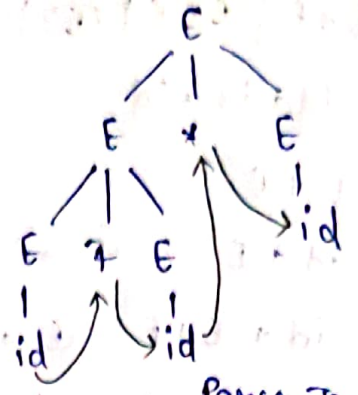
$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse Tree 2

for LMD;

there are two possible parse trees.

If $\omega = 2 + 3 * 5$

Parse Tree 1 :- $\omega = 2 + (3 * 5)$ ✓

Parse Tree 2 :- $\omega = (2 + 3) * 5$ ✗

∴ Given grammar is ambiguous.

So, To eliminate this ambiguity :-

consider - 1) associativity

2) priority of operators

Q) Show that following grammar is ambiguous.

$S \rightarrow aSbS$ $\omega = abab$

$S \rightarrow bSaS$

$S \rightarrow \varepsilon$

$S \rightarrow aSbS$

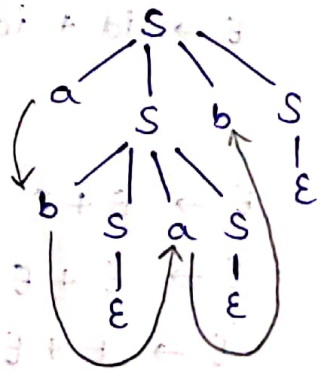
$S \rightarrow abSaSbS$ $(S \rightarrow bSaS)$

$S \rightarrow ab\varepsilon aSbS$ $(S \rightarrow \varepsilon)$

$S \rightarrow aba\varepsilon bS$ $(S \rightarrow \varepsilon)$

$S \rightarrow aba b\varepsilon$ $(S \rightarrow \varepsilon)$

$S \rightarrow abab$



Parse Tree 1

$S \to aSbS$

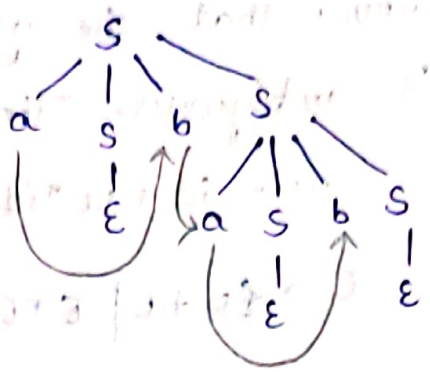
$S \to a\varepsilon bS$

$S \to abaSbS$

$S \to aba\varepsilon bS$

$S \to ababe$

$S \to abab$



Parse Tree 2

Thus, the given grammar is ambiguous.
Hence proved

Removal of Ambiguity

The ambiguity in the given grammar occurs due to use of single non-terminal more in multiple definitions or production rules.

> If single non-terminal is being used repeatedly, then the ambiguity arises.

By introducing multiple non-terminals, the ambiguity can be removed.

For removing ambiguity of grammar, defining arithmetic expressions, we apply two rules.

1) If the grammar has left associative operators (+, -, *, /) then, apply Left Recursion.

Ex:- $E \to E + T$ (Left recursion grammar)

2) If the grammar has right associative operator (↑) (exponential operator) then, apply Right Recursion.

Ex:- $E \to T + E$ (Right recursive production rule)

Q) Show that the grammar $E \to E+E \mid E*E \mid (E) \mid id$ is ambiguous. Eliminate the ambiguity.

$w = id + id * id \Rightarrow$ left recursive $(\because +, *)$

$E \to E+E \mid E*E \mid (E) \mid id$

$E \to E+E$

$\Rightarrow E \to E+T \mid T$ $\quad (\because E \to E+E$, $E \to E+T \mid T$, $E \to T)$

~~$E \to E*E$~~

~~$T \to T*F$~~

$\Rightarrow T \to T*F \mid F$ $\quad (\because E \to T$ & $E \to E*E$, $T \to T*T$, $T \to T*F \mid F \to T \to F)$

$\Rightarrow F \to (E) \mid id$

LMD

$E \to E+T$

$E \to T+T$ $\quad (E \to T)$

$E \to F+T$ $\quad (T \to F)$

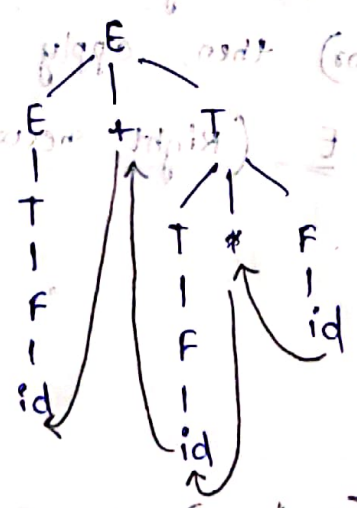
$E \to id+T$ $\quad (F \to id)$

$E \to id+T*F$ $\quad (T \to T*F)$

$E \to id+F*F$ $\quad (T \to F)$

$E \to id+id*F$

$E \to id+id*id$



$2 + (3 * 5)$

$2 + 15 \Rightarrow 17$

RMD

$E \rightarrow E + T$

$E \rightarrow E + T * F$

$E \rightarrow E + T * id$

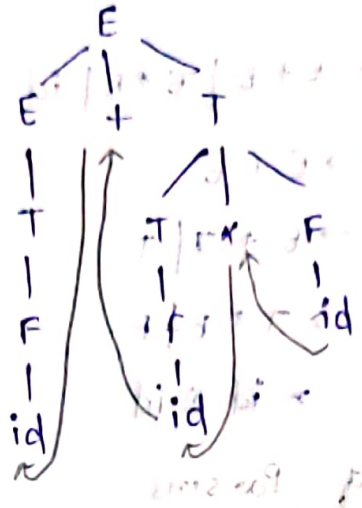
$E \rightarrow E + F * id$

$E \rightarrow E + id * id$

$E \rightarrow T + id * id$

$E \rightarrow F + id * id$

$E \rightarrow id + id * id$



Q) bExp → bExp OR bExp

| bExp AND bExp

| Not bExp

| True

| False

bExp → bExp OR bExp

~~bExp → T OR bEx~~

bExp → bExp OR T | T

T → T OR F | F

F → F AND G | G

G → Not H | H

H → True | False

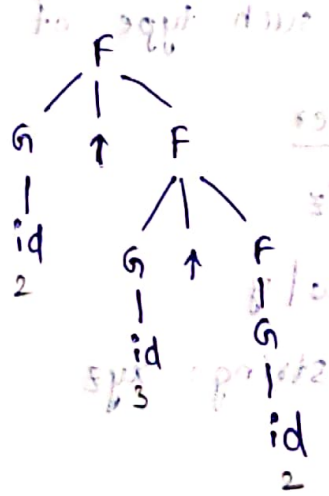
Q) $2 \uparrow 3 \uparrow 2$

$(2 \uparrow (3 \uparrow 2))$

$= 2^{3^2} = 2^9$

$F \rightarrow G \uparrow F \mid G$

$G \rightarrow id$



06/10/2023

$E \rightarrow E+E \mid E*E \mid id$

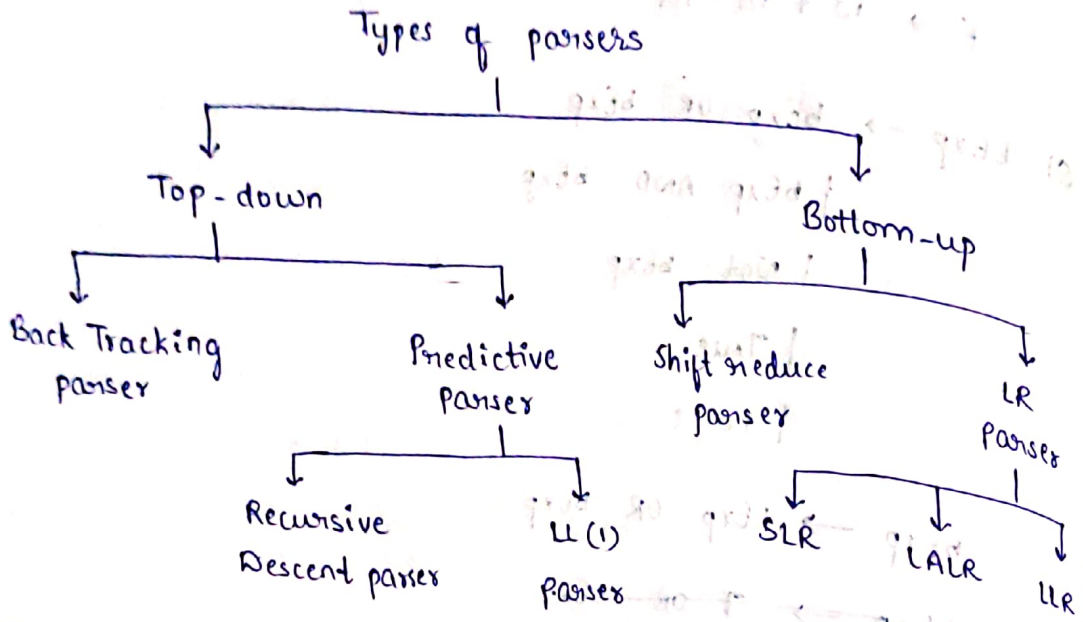
$E \rightarrow E+E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

Types of Parsers



Top-down Parser

When parse tree can be constructed from root & expanded to leaves then, such type of parser is called Top-down Parser.

Bottom-up Parser

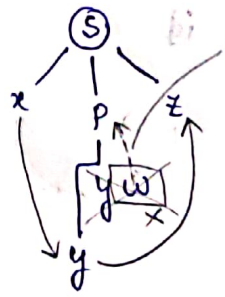
When the parse tree can be constructed from leaves to root then, such type of parser is called Bottom-up Parser.

Top-down Parser

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

string: xyz



Back Tracking causes burden to the parser

Problems of Top-down Parser

- Back tracking
- Left Recursion
- left Factoring
- Ambiguity

Left Recursion

case (i) :- $A \rightarrow A\alpha \mid \beta$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

Ex: $E \rightarrow E+T \mid a$

$A \rightarrow A\alpha \quad \beta$

$$E \rightarrow aE'$$

$$E' \rightarrow TE' \mid \varepsilon$$

Q) $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

First, identify if it is left recursion.

Take, $E \rightarrow E+T \mid T$

$(A \rightarrow A\alpha \mid \beta$ then, $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon)$

$$E \rightarrow TE'$$

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

Thus, left recursion is eliminated.

Now, $T \rightarrow T*F \mid F$

$(A \rightarrow A\alpha \mid \beta$ then, $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon)$

$$T \rightarrow FT'$$

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

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

∴ After removing left recursion,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

Q) $S \rightarrow S0S1S \mid 01$

$\left(A \rightarrow A\alpha \mid \beta \quad \text{then,} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array} \right)$

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' \mid \varepsilon$

Q) $S \rightarrow (L) \mid x$

$L \rightarrow L,S \mid S$

terminal

Take, $L \rightarrow L,S \mid S$

$\left(A \rightarrow A\alpha \mid \beta \quad \text{then,} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array} \right)$

$L \rightarrow SL'$

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

After removing left recursion,

$S \rightarrow (L) \mid x$

$L \rightarrow SL'$

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

case (ii) : $A \to A\alpha_1 \mid A\alpha_2 \ldots \mid \beta_1 \mid \beta_2$

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

$A' \to \alpha_1 A' \mid \alpha_2 A' \ldots \mid \varepsilon$

Q) $\underset{A}{expr} \to \underset{A'}{expr} + \underset{\alpha_1}{expr} \mid \underset{A}{expr} * \underset{\alpha_2}{expr} \mid \underset{\beta}{id}$

$expr \to id\, expr' \mid expr'$

~~$expr' \to +expr' \mid *expr' \mid \varepsilon$~~

$expr' \to +expr\ expr' \mid *expr\ expr' \mid \varepsilon$

10/10/2023

Consider the following grammar

$A \to ABb \mid Aa \mid a$

$B \to Be \mid d$

remove the left recursion.

(i) $\underset{A}{A} \to \underset{A}{A}\underset{\alpha}{Bb} \mid \underset{\beta}{a}$ then $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$A \to aA'$

$A' \to BbA' \mid \varepsilon$

(ii) $\underset{A}{A} \to \underset{A}{A}\underset{\alpha}{a} \mid \underset{\beta}{a}$ then, $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$A \to aA'$

$A' \to aA' \mid \varepsilon$

(iii) $\underset{A}{B} \to \underset{A}{B}\underset{\alpha}{c} \mid \underset{\beta}{d}$ then, $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$B \to dB'$

$B' \to eB' \mid \varepsilon$

∴ The production rules after removing left recursion are :-

$A \to aA'$

$A' \to aA' \mid BbA' \mid \varepsilon$

$B \to dB'$

$B' \to eB' \mid \varepsilon$

Left Factoring

- (It) is used when it is not clear that which of the two alternatives is used to expand the non-termin

If $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ is a production,

then it is not possible to take a decision whether to choose Rule 1 or Rule 2. In such situations, the above grammar can be left factored as :-

$A \rightarrow \alpha A'$ when $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ is given.

$A' \rightarrow \beta_1 \mid \beta_2 \ldots$

Ex :-

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

take, $S \rightarrow \underbrace{iEtS}_{\alpha\ \beta_1} \mid \underbrace{iEtSeS}_{\alpha\ \beta_2} \mid a$ then, $A \rightarrow \alpha A'$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

$A' \rightarrow \beta_1 \mid \beta_2$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

$E \rightarrow b$

Q) Do left factoring in the following grammar

$A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b$

Ans :- (1) $A \rightarrow \underbrace{aAB}_{\alpha\ \beta_1} \mid \underbrace{aA}_{\alpha\ \beta_2} \mid \underbrace{a}_{\alpha\ \beta_3}$ then, $A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \varepsilon$

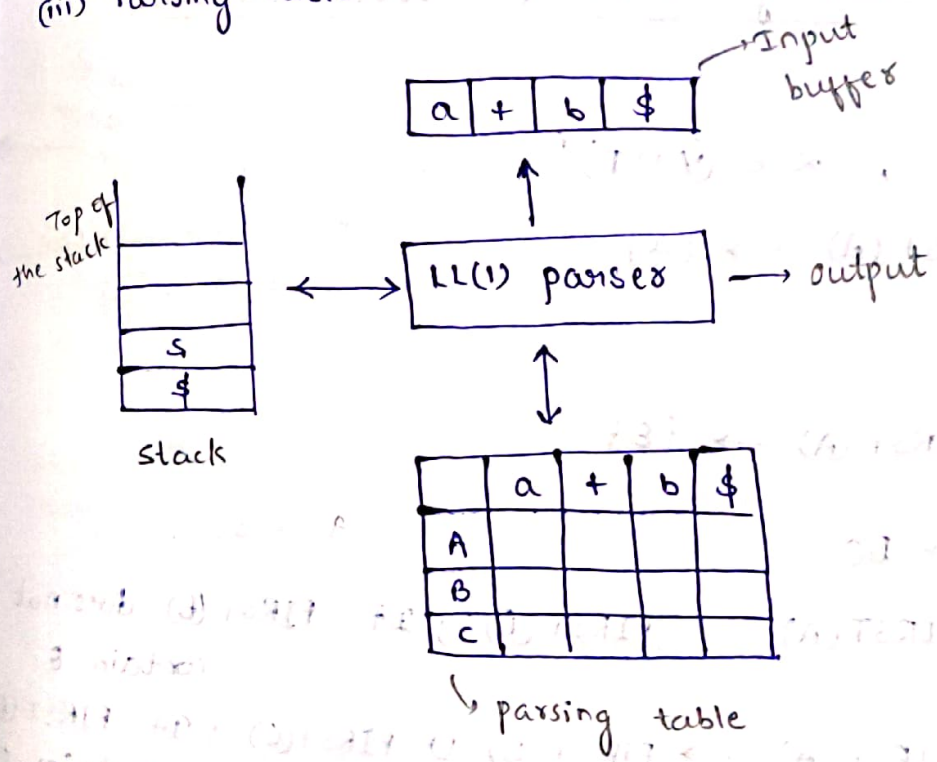
(ii) $B \rightarrow bB \mid b$ then, $A \rightarrow \alpha A'$
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ $A' \rightarrow \beta_1 \mid \beta_2$
$B \rightarrow bB'$
$B' \rightarrow B \mid \varepsilon$

Predictive LL(1) parser

LL(1)
- left to right
- left most derivative
- look ahead symbol that uses single char

» For LL(1) parser,
the first L means that, the input is scanned from left to right.
the second L means that, it uses left most derivation for input string.
the number 1 in the input string means that, it uses only 1 input symbol (look ahead symbol) to predict the parsing process.

» The data structures used by LL(1) parser are
(i) Input buffer
(ii) Stack
(iii) Parsing table



» LL(1) parser uses input buffer to store the input tokens & indicates that, it is the end of the string.

» Stack is used to hold the left sentential form. The symbols in the RHS of the rule are pushed into the stack in reverse order.

» Parsing table is a 2-D array. It has rows for the non-terminal symbols and columns for terminal symbols.

Construction of Predictive LL(1) parser

For the construction of this parser, we need to follow the steps given below:

1) Computation of FIRST & FOLLOW functions
2) Construct the predictive parsing table using the FIRST & FOLLOW functions.
3) Parse the input string with the help of predictive parsing table

Rules for constructing FIRST & FOLLOW functions

FIRST :- [it is found for symbols that are on the LHS of production rule]

1) If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$, Ex: $A \rightarrow \underbrace{a}_{\text{terminal}}\underbrace{BaC}_{\text{combination of terminals \& non-terminals}}$

then, FIRST (A) $\rightarrow \{a\}$

FIRST (A) $\rightarrow$ FIRST (aBac)

$\Rightarrow$ FIRST (A) $\rightarrow \{a\}$

2) If $A \rightarrow \varepsilon$

then, FIRST (A) $\rightarrow \{\varepsilon\}$

3) If $A \rightarrow BC$

$A \rightarrow \underbrace{BC}_{\text{all are non-terminal symbols}}$

then, FIRST (A) $\rightarrow$ FIRST (B) ; If FIRST (B) does not contain ε

4) If FIRST (A) $\rightarrow$ FIRST (B) $\cup$ FIRST (C) ; If FIRST (B) contains ε

FOLLOW :- [It is found for symbols that are on the RHS of production rule]

1) If 'S' is a starting symbol

then, FOLLOW (S) = { $ }

2) If $A \rightarrow \alpha B \beta$

then, FOLLOW (B) = FIRST (β) ; if FIRST (β) doesn't contain ε

3) If $A \rightarrow \alpha B$

Here, B should be the last symbol.

then, FOLLOW (B) $\rightarrow$ FOLLOW (A)

4) If $A \rightarrow \alpha B \beta$ where $\beta \rightarrow \varepsilon$ then, B becomes the last symbol.

then, FOLLOW (B) $\rightarrow$ FOLLOW (A)

FOLLOW :- [it is found for symbols that are on the RHS of Product^n rule]

1) If 'S' is a starting symbol
then, FOLLOW (S) = { $ }

FOLLOW never contains ε

2) If $A \rightarrow \alpha B \beta$
then, FOLLOW (B) = FIRST (β) ; if FIRST (β) doesn't contain ε

3) If $A \rightarrow \alpha B$ → Here, B should be the last symbol
then, FOLLOW (B) → FOLLOW (A)

4) If $A \rightarrow \alpha B \beta$ where $\beta \rightarrow \varepsilon$ then, B becomes the last symbol.
then, FOLLOW (B) → FOLLOW (A)

11/10/2023
Consider the following grammar. Remove left recursion and left factoring and construct LL(1) parsing table.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

After removing left recursion :-
given; $E \rightarrow E + T \mid T$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

1) FIRST & FOLLOW functions

FIRST (E) → FIRST (TE')

FIRST (T) → FIRST (FT')

FIRST (F) → FIRST ((E)) ∪ FIRST(id) = { (, id }
a

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

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε) (∵ E' → +TE' | ε)

= { +, ε }

Now,

FIRST (T) = ~~FIRST~~ { (, id }

FIRST (T') → FIRST (*FT') ∪ FIRST (ε)

= { *, ε }

FIRST (F) = { (, id }

FOLLOW (E) → { $,) } (∵ E is the starting symbol)

FOLLOW (E) = FIRST ( )) E →

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

FOLLOW (T) = FIRST (E') = { +, ε } ✗ E → TE'

α β β, ↳ ε

But, FOLLOW should not contain ε. E' → +TE' | ε

FOLLOW (T) = ~~FOLLOW~~ FOLLOW(E) = { +, $,) }

FOLLOW (T') = FOLLOW (T)

= { +, $,) }

FOLLOW (F) = FIRST (T') = { *, ε } ✗

FOLLOW (F) = FOLLOW (T) = { +, $,) }

2) Constructing the parsing table

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

To construct the parsing table, following rules are used.

1) $M[A, a] = A \to \alpha$, if a is in FIRST (α)

2) $M[A, b] = A \to \alpha$, if ε is in FIRST (α)
b is in FOLLOW (A)

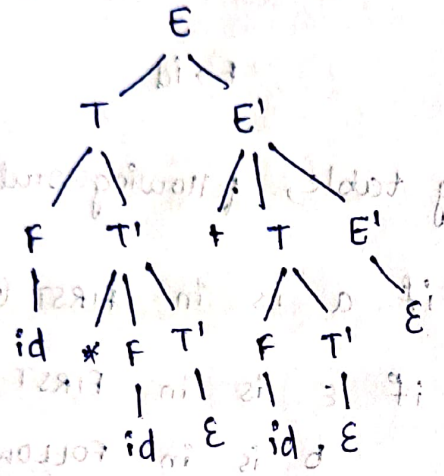
3) Parsing the input string.

ω = id * id + id

Stack	input	output
$E	id * id + id $	
$ E'T	id * id + id $	$E \to TE'$
$ E'T'F	id * id + id $	$T \to FT'$
$ E'T'id	id * id + id $	$F \to id$
$E'T'	* id + id $	
$E'T'F*	* id + id $	$T' \to *FT'$
$ E'T'F	id + id $	
$ E'T'id	id + id $	~~$T \to FT'$~~ $F \to id$
$ E'T'	+ id $	
$ E'	+ id $	$T' \to \varepsilon$
$ E'T+	+ id $	$E' \to +TE'$

check ∵ $E \to id$ in Parsing table

$ E'T	id $	
$ E'T'F	id$	T → FT'
$ E'T'id	id $	F → id
$ E'T'	$	
$ E'	$	T' → ε
$	$	E' → ε



Q) Construct the predictive parsing table for given grammar

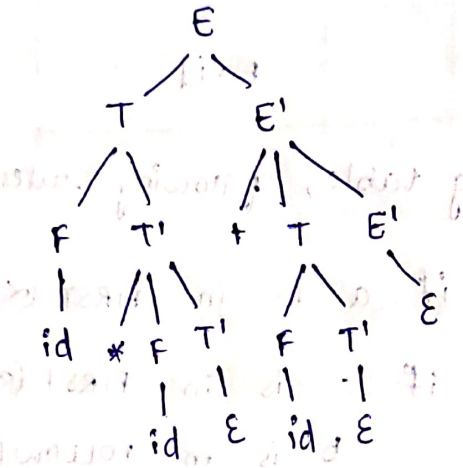
E → E+T | T

T → TF | F

F → F* | a | b

Sol:

$ E'T	id $	
$ E'T'F	id $	T → FT'
$ E'T'id	id $	F → id
$ E'T'	$	
$ E'	$	T' → ε
$	$	E' → ε



Q) Construct the predictive parsing table for given grammar

$E \rightarrow E+T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F* \mid a \mid b$

Sol:-

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

$E \rightarrow TE'$

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

$T \rightarrow \underset{A}{T} \underset{\alpha}{F} \mid \underset{\beta}{F}$

$T \rightarrow FT'$

$T' \rightarrow FT' \mid \varepsilon$

~~$F \rightarrow F* \mid a \mid b$~~

~~$F \rightarrow$~~

$F \rightarrow \underset{A}{F} \underset{\alpha}{*} \mid \underset{\beta}{a}$

$F \rightarrow aF'$

$F' \rightarrow *F' \mid \varepsilon$

$F \rightarrow \underset{A}{F} \underset{\alpha}{*} \mid \underset{\beta}{b}$

$F \rightarrow bF'$

$F' \rightarrow *F' \mid \varepsilon$

After removing left recursion :-

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

$T' \to FT' \mid \varepsilon$

$F' \to aF' \mid bF'$

$F' \to *F' \mid \varepsilon$

$E \to TE'$

$E \to TE'$

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

$T \to FT'$

$T' \to FT' \mid \varepsilon$

$F \to aF' \mid bF'$

$F' \to *F' \mid \varepsilon$

1) FIRST & FOLLOW functions

FIRST (E) = FIRST (TE')

FIRST (T) = FIRST (FT')

FIRST (F) = FIRST (aF') ∪ FIRST (bF')

α α

= {a, b}

⇒ FIRST (E) = {a, b}

⇒ FIRST (T) = {a, b}

⇒ FIRST (F) = {a, b}

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε)

⇒ FIRST (E') = {+, ε}

FIRST (T') = FIRST (FT') ∪ FIRST (ε)

⇒ FIRST (T') = {a, b, ε} (∵ FIRST (F) = {a, b})

FIRST (F') = FIRST (*F') ∪ FIRST (ε)

⇒ FIRST (F') = {*, ε}

⇒ FOLLOW (E) = {$}

FOLLOW (T) = FIRST (E')

= {+, ε} ⊗

= FOLLOW (E)

$E \to \underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{}$

$E \to \underset{\alpha}{T}\underset{B}{\varepsilon}\underset{\beta}{}$

⇒ FOLLOW (T) = {+, $}

FOLLOW (F) = FIRST (T')

$T \to FT'$ $\quad$ $T \to F\varepsilon$

$\alpha B \beta$ $\quad$ $\alpha B \beta$

$\Rightarrow$ FOLLOW (F) = $\{a, b, \varepsilon\} = \{a, b, +, \$\}$

FOLLOW (E') = FOLLOW (E)

$E \to TE'$

αB

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

$T \to FT'$

FOLLOW (T') = FOLLOW (T)

αB

$\Rightarrow$ FOLLOW (T') = $\{+, \$\}$.

$F \to aF'$

FOLLOW (F') = FOLLOW (F)

αB

$\Rightarrow$ FOLLOW (F') = $\{a, b, +, \$\}$

2) Constructing parsing table.

	+	*	a	b	$
E			$E \to TE'$	$E \to TE'$	
E'	$E' \to +TE'$				$E' \to \varepsilon$
T			$T \to FT'$	$T \to FT'$	
T'	$T' \to \varepsilon$		$T' \to FT'$	$T' \to FT'$	$T' \to \varepsilon$
F			$F \to aF'$	$F \to bF'$	
F'	$F' \to \varepsilon$	$F' \to *F'$	$F' \to \varepsilon$	$F' \to \varepsilon$	$F' \to \varepsilon$

12/10/2023

Consider the grammar

$S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

$E \to b$

Check whether it is LL1 grammar.

Step 1 :- FIRST & FOLLOW functions.

FIRST (S) = $\{i, a\}$

FIRST (S') = $\{e, \varepsilon\}$

FIRST (E) = $\{b\}$

FOLLOW (S) = { $, e, ε }

if S → S' FIRST (S') = { e, ε }

FOLLOW (S) = { $, e }

replace S' by ε

S → iEtS
α β
FOLLOW (S)

S' → eS
α β
FOLLOW (S')

FOLLOW (S') = FOLLOW (S) = { $, e }

FOLLOW (E) = FIRST (tSS') = { t }

step 2 :-

	i	t	a	e	b	$
S	S → iEtSS'		S → a			
S'				S' → eS S' → ε		S' → ε
E					E → b	

Here, one of the cells of parsing table contains more than one production rule.

Hence, given grammar is not an LL(1) grammar

Q) Find FIRST & FOLLOW functions

S → ABCDE

A → a | ε

B → b | ε

C → c

D → d | ε

E → e | ε

② Q) S → ACB | CbB | Ba

A → da | BC

B → g | ε

C → h | ε

③ Q) Construct the predictive parsing table for the grammar

S → (L) | a

L → L, S | S

① FIRST (S) = FIRST (ABCDE) = ~~FIRST (A) ∪ FIRST (B) ∪ FIRST (C) ∪ FIRST (D) ∪ FIRST (E)~~

~~= FIRST (S)~~

~~= {a, b, c, d, e, ε}~~

FIRST (A) = {a, ε}

FIRST (B) = {b, ε}

FIRST (C) = {c}

FIRST (D) = {d, ε}

FIRST (E) = {e, ε}

FIRST (S) = FIRST (ABCDE)

= FIRST (A) ∪ FIRST (BCDE)

= {a, ε} ∪ {b, ε} ∪ FIRST (CDE)

= {a, b, c, ε}

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (BCDE) = FIRST (B) ∪ FIRST (CDE) = {b, ε}

= {b, FIRST (CDE) = FIRST (C)

= {c} ;

= {b, c}

FOLLOW (B) = FIRST (CDE)

= FIRST (C) = {c}

FOLLOW (C) = FIRST (DE)

= FIRST (D) ∪ FIRST (E)

= {d, FIRST (BCE)} ∪ {e, FIRST (D)}

= {d, FIRST (B) ∪ FIRST (CE)} ∪ {e, d, FIRST (E)}

= {d, {b, FIRST (DE)} ∪ {c}} ∪ {e, d, e, $}

= {d, {b, FIRST (D) ∪ FIRST (E)} ∪ {c}} ∪ {e, d, $}

= {d, b, d,

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (BCDE)
= {b, c}

$\{b, \varepsilon\}$ = FIRST (B) ∪ FIRST (C)

S → ABCDE
S → ACDE
S → ABCE

FIRST (CDE) {c}

FOLLOW (B) = FIRST (CDE)
= {c}

FOLLOW (C) = FIRST (DE)
= {d, e, $}

S → ABCD

S → ABCDE

FOLLOW (D) = FIRST (E)
= {e, $}

S → ABCDE

FOLLOW (E) = {$}

② FIRST (S) = FIRST (ACB) ∪ FIRST (CbB) ∪ FIRST (Ba)

FIRST(A) ∪ F

= {d, g, h, ε} ∪ {h, ε} ∪ {g, ε}
= {d, g, h, ε}

ABCDE
ACDE

FIRST (A) = FIRST (da) ∪ FIRST (BC)
= {d} ∪ {g, h, ε}
= {d, g, h, ε}

A → da
A → BC

FIRST(B) ∪ FIRST(C)
{g, ε} ∪ {h, ε}

FIRST (B) = {g, ε}

FIRST (C) = {h, ε}

FOLLOW (S) = {$}

S → ACB

S → ACB
ε ε

FOLLOW (A) = FIRST (CB) = {h, ε} ∪ FIRST(B) ∪ FOLLOW(S)

{g, ε}

= {h, g, $}

FOLLOW (B) = FOLLOW (S) ~~= {$}~~
S → ACB

FIRST (a) = {h, g, $, a}
S → Ba

~~FOLLOW (C)~~ = ~~FIRST (B)~~ ~~= {g, ε}~~

FOLLOW (C) = FIRST (B) ∪ FIRST(bB) ∪ FOLLOW (A)
S → ACB S → CbB A → BC

= {g, b, h, $}

(3) $S \to (L) \mid a$

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

Removing left recursion :-

$S \to (L) \mid a$

$L \to SL'$

$L' \to ,SL' \mid \varepsilon$

FIRST & FOLLOW functions

FIRST (S) = FIRST((L)) ∪ FIRST (a) = { (, a }

FIRST (L) = FIRST (SL') = { (, a }

FIRST (L') = FIRST (,SL') ∪ FIRST(ε) = { , , ε }

FOLLOW (S) = { $, , ,) }

FOLLOW (L) = FIRST ()) = {) }

FOLLOW (L') = FOLLOW(L) = {) }

$L \to \underset{\alpha}{S}L' \mid \underset{\beta}{\varepsilon}$

$L \to \underset{\alpha}{S} \mid \underset{\beta}{\varepsilon}$

$S \to (\underset{\alpha}{L}) \mid \underset{\beta}{a}$

$L' \to \underset{\alpha}{,S}L' \mid \underset{\beta}{\varepsilon}$

Passing table.

	(	)	,	a	$
S	S→ (L)			S→a	
L	L→SL'			L→SL'	
L'		L'→ε	L'→,SL'		

LR(0) Parser

$S \to AA$ — 1

$A \to aA$ — 2

$A \to b$ — 3

Include new produc^n rule

⇒ Augmented grammar is the result

step 1 :- Include new production rule

Augmented grammar :-

$S' \to S$

$S \to AA$

$A \to aA$

$A \to b$

step 2 :- Closure operation Introduce · at the RHS ~~at~~ at left most.

I_0

$S' \to \cdot S$

$S \to \cdot AA$

$A \to \cdot aA$

$A \to \cdot b$

closure of the production rules

step 3 :- ~~Find~~ Apply goto operation

(move · towards right)

(no I/p symbol is scanned) $S \to \cdot AA$

(1 I/p symbol is scanned) $S \to A \cdot A$

(all I/p symbols are scanned) $S \to AA \cdot$

$S' \to \cdot S \xrightarrow{S}$ I_1: $S' \to S \cdot$

$S \to \cdot AA \xrightarrow{A}$ I_2:

$S \to A \cdot A$

$A \to \cdot aA$

$A \to \cdot b$

(Check symbols that are after · & include its produc^n rules)

$A \to \cdot aA \xrightarrow{a}$ I_3:

$A \to a \cdot A$

$A \to \cdot aA$

$A \to \cdot b$

$A \to \cdot b \xrightarrow{b}$ I_4: $A \to b \cdot$

$S' \to S.$ —— (no I/P symbol)

$S \to A.A \xrightarrow{A}$ $(S \to AA.)$ I_5

$A \to .aA \xrightarrow{a}$ $A \to a.A$ already exists.

$A \to .b \xrightarrow{b}$ $A \to b.$ already exists

$A \to a.A \xrightarrow{A}$ $(A \to aA.)$ I_6

$A \to .aA \longrightarrow$ $A \to a.A$

$A \to .b \longrightarrow$ $A \to b.$

$A \to b.$ —— (no I/P symbol)

<u>step 4</u>: construct parsing table.

FOLLOW (S) = {$}

FOLLOW (A) = FIRST (A) ∪ FOLLOW(S) = {a, b, $}

$A \to aA \mid b$

~~S → aA~~

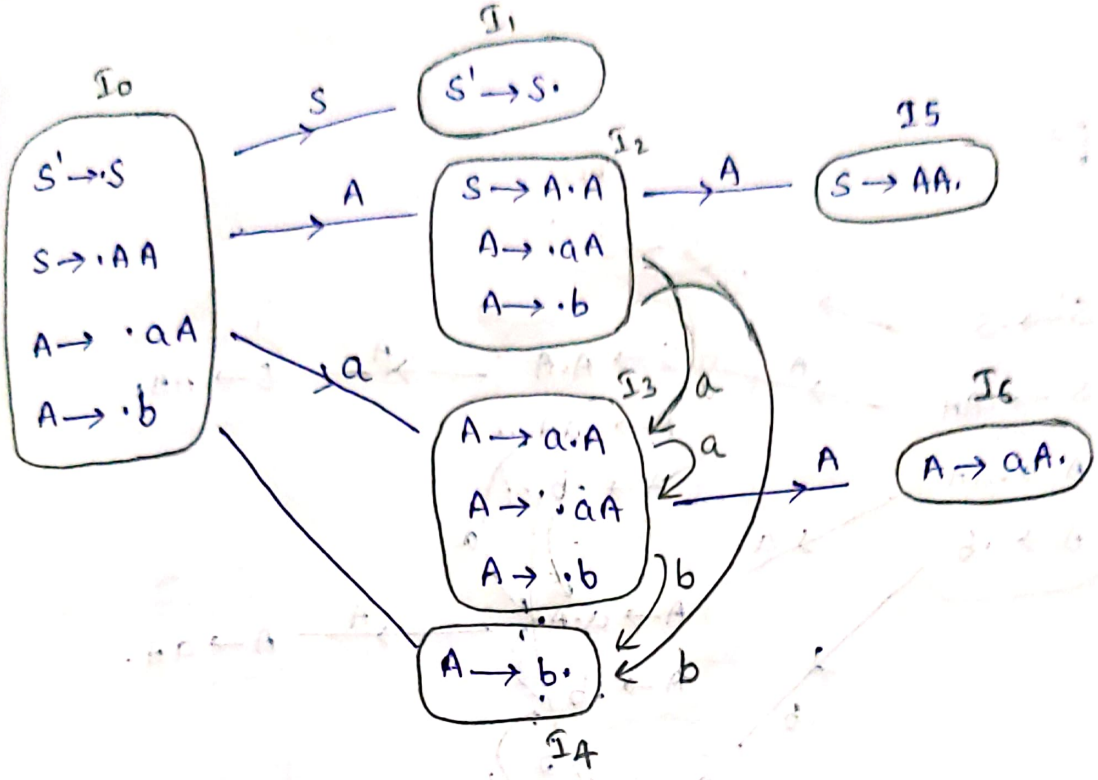
~~S → AA~~

~~α B~~

S → A A
α B

S → A A
α B β

	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		



SLR (1) parser

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

step 1

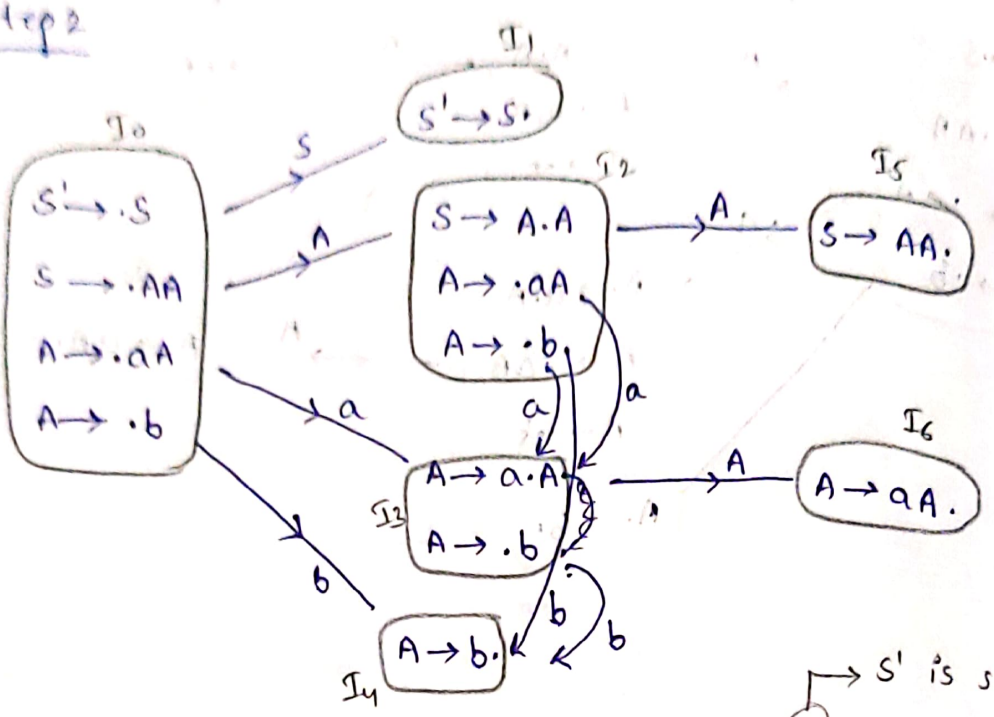
$S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

Step 2



I1 : S' → S.

→ S' is starting symbol so write accept

Step 3

	Action			goto	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

I6 ⇒ A → aA. — (2)

↳ A is not starting symbol

So, take FOLLOW(A) = {a, b, $}

In these places, write r2

18/10/2023

UNIT - 3

Syntax Directed Definition (SDD)

SDD = CFG + Semantic Rules

all with

attribute

$E \rightarrow E + T \Rightarrow E.val = E.val + T.val$

$E \rightarrow T \Rightarrow E.val = T.val$ → semantic rule

An SDD is a Context Free Grammar together with semantic rule.

Attributes are associated with grammar symbols and semantic rules are associated with production rules.

Attributes may be numbers, strings, references, datatypes etc.

Ex: Production rule — semantic rules.

$E \rightarrow E + T$ — $E.val = E.val + T.val$

Types of Attributes (2)

1) Synthesized attribute : If a parent node takes value from its children, then it is called Synthesized Attribute.

Ex: $A \rightarrow BCD \Rightarrow A.S = B.S$, $A.S = C.S$, $A.S = D.S$

(A: parent node; BCD: child nodes)

2) Inherited Attribute : If a node takes a value from its parent or sibling nodes then, it is called Inherited Attribute

Ex: $A \rightarrow BCD \quad C.i = A.i$, $C.i = B.i$, $C.i = D.i$

(A: parent; BCD: siblings; C: child)

** Types of SDDs (2)

1) S-attributed SDD

2) L-attributed SDD

1) S- Attributed SDD

» An SDD that uses only synthesized attributes is called as an S-attributed SDD.

Ex: $A \rightarrow BCD$

$A.S = B.S$

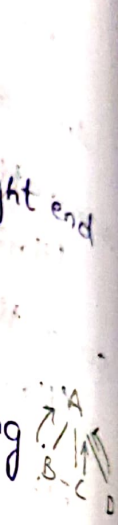
$A.S = C.S$

$A.S = D.S$

» Semantic actions are always placed at the right end of the productions.

Ex: $A \rightarrow BCD \{ \ldots \}$

» Attributes are evaluated with Bottom-up Parsing



2) L- Attributed SDD

» It uses both synthesized and inherited attributes. Inherit attributes can inherit values from either parent or left sibling only

Ex: $A \rightarrow BCD$

$C.i = A.i$

$C.i = B.i$

$D.i = C.i$

$C.i = D.i$ (X)

» Semantic actions are placed anywhere on RHS.

Ex: $A \rightarrow \{ \ldots \} BCD$

$A \rightarrow BCD \{ \ldots \}$

$A \rightarrow B \{ \ldots \} CD$

» Attributes are evaluated by traversing parse string depth-first, left to right

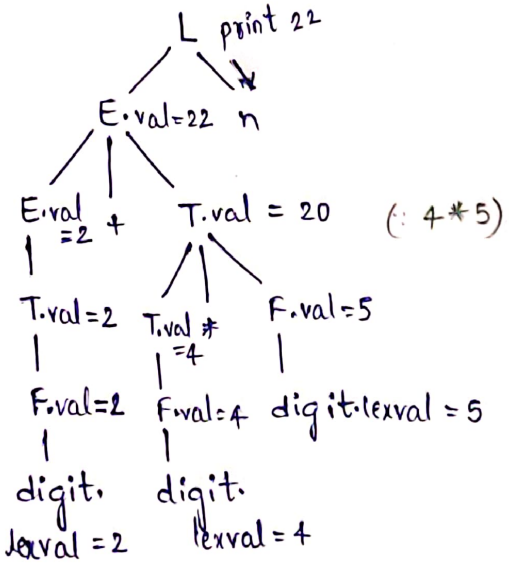
Examples

SDT
Syntax
Directed
Translation

(i) S- attributed SDD

Production rules for Desktop Calc.	Semantic rules
$L \to En$ (n → new line)	print (E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$F \to (E)$	$F.val = (E.val)$
$F \to digit$	$F.val = digit.lexval$

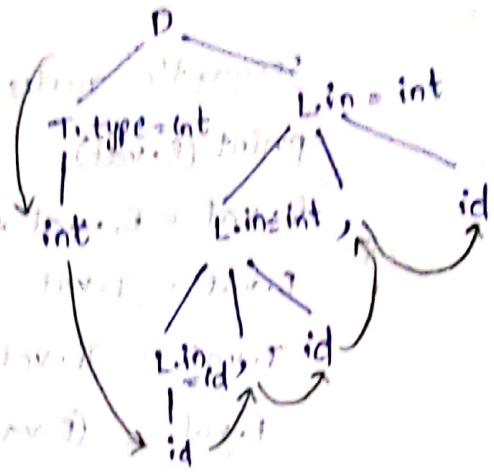
Evaluate the expression 2+4*5n



(ii) L- attributed SDD

Production	Semantic rules
$D \to TL$	$L.in = T.Type$
$T \to int$	$T.Type = integer$
$T \to real$	$T.Type = real$
$L \to L_1, id$	$L_1.in = L.in$ addtype (id.entry , L.in)
$L \to id$	addtype (id.entry , L.in)

Evaluate int id1, id2, id3



20/10/2023

Q) LL(1) Parser

ω = id + id * id

(1) $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Sol:

step 1 : FIRST & FOLLOW functions

FIRST (E) = FIRST (TE')

FIRST (T) = FIRST (FT')

FIRST (F) = FIRST ((E)) ∪ FIRST (id)

= {(, id}

Now,

FIRST (E') = FIRST (+TE') ∪ FIRST (ε)

= {+, ε}

FIRST (T) = FIRST (FT')

= {(, id}

FIRST (T') = FIRST (*FT') ∪ FIRST (ε)

= {*, ε}

~~FIRST (F)~~ ⇒ ~~FIR~~

FOLLOW (E) = {$,) }

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

FOLLOW (T) = FIRST (E')

= {+, ε} ⊗

= {+, $,) }

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

FOLLOW (F) = FIRST (T') = {*, ε} ⊗

= {*, $, +,) }

F → (E)
α B β

E → TE'
α B β
α B β

E → TE'

E → TE
α B β

T → FT'
α B
α B β

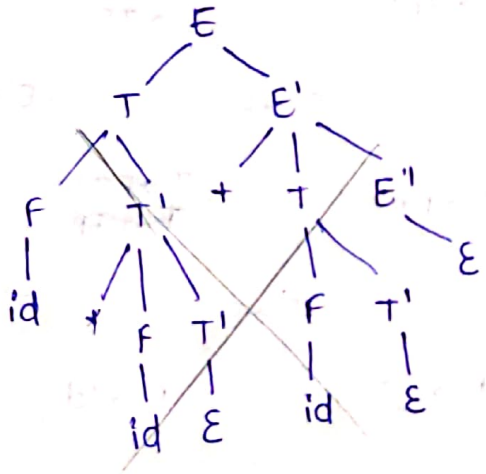
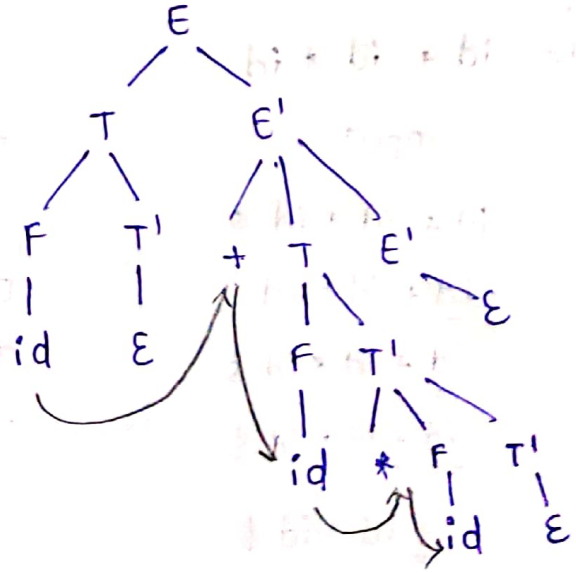
T → FE
α B β

step 2 :- parsing table.

	+	*	(	)	id	$
E			E → TE'		E → TE'	
E'	E' → +TE'			E' → ε		E' → ε
T			T → FT'		T → FT' ~~T → FF'~~	
T'	T' → ε ~~T → ε~~	T' → *FT' ~~T → *FF'~~		T' → ε		T' → ε
F			F → (E)		F → id	

step 3 :- Parsing the input string.

w = id + id * id

Stack	input	output
$E	id + id * id $	
$ E'T	id + id * id $	E → TE'
$E'T'F	id + id * id $	T → FT'
$E'T'id	id + id * id $	F → id
$ E'T'	+ id * id $	
$E'	+ id * id $	T' → ε
$ E'T+	+ id * id $	E' → +TE'

$E'T$	id * id $	$T \to FT'$
$E'T'F$	id * id $	~~F → id~~
$E'T'id$	id * id $	$F \to id$
$E'T'$	* id $	$T' \to *FT'$
$E'T'F*$	* id $	
$E'T'F$	id $	$F \to id$
$E'T'id$	id $	
$E'T'$	$	$T' \to \varepsilon$
E'	$	$E' \to \varepsilon$
$	$