Unit - 1

Introduction to Compiler

Aniket
Date
Page

Translator

is one kind of program that takes one form of program as input and converts it into another form. The input program is called source program and the output program is called target program.

- The source language can be low-level language or a high-level language like C, C++, FORTRAN.
- The target language can be a low-level language or a machine language.

Source language → [Translator] → Target language

Compiler

is a program which takes one language (source language) as input and translates it into an equivalent another language (target program).

During this process of translation, if some errors are occured, then compiler display them as error messages.

(Input) Source program → [Compiler] → target program (output)

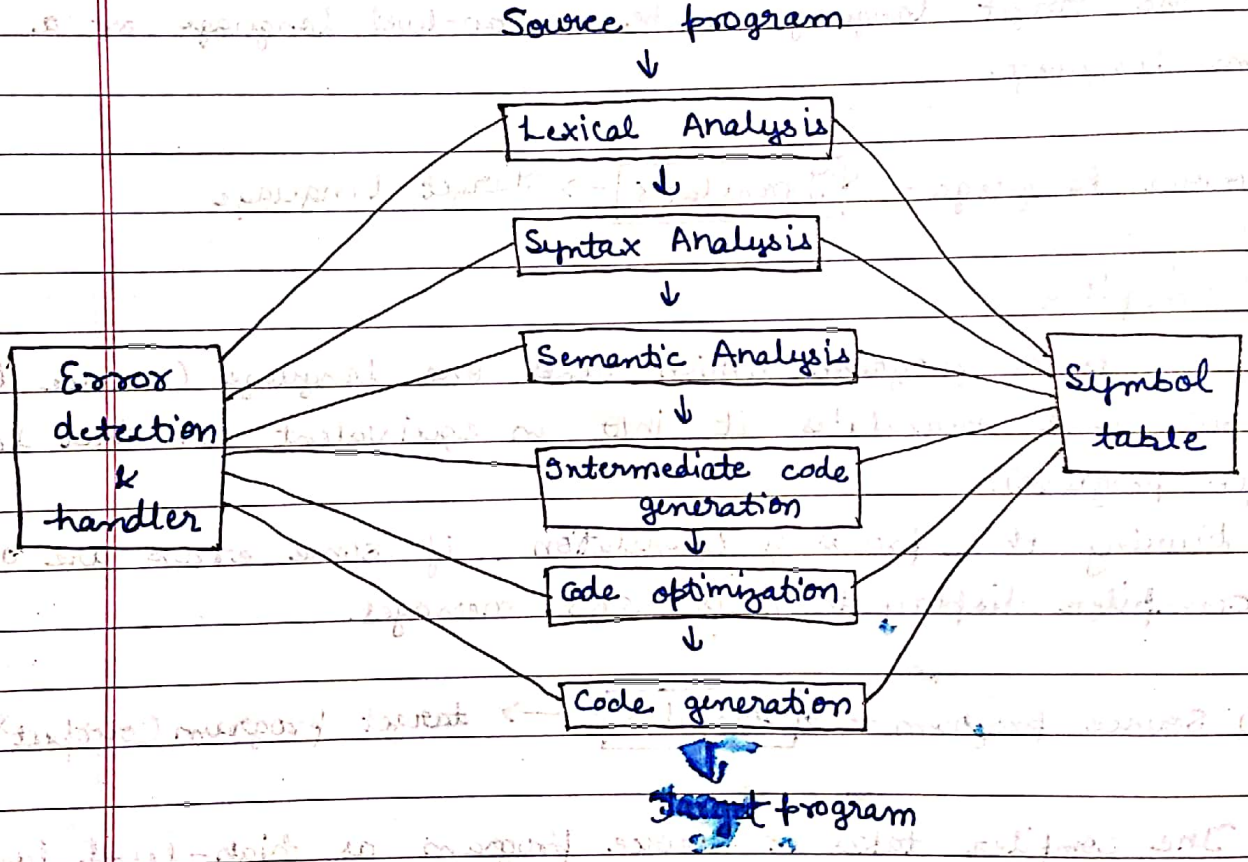
The compiler takes a source program as high-level language and converts it into low level language or machine language like assembly language.

Properties of Compiler:

When a compiler is built, it should posses following properties: 1) The compiler itself must be bug free.
2) It must generate correct machine code.
3) The generated machine code must run fast.

4) The compiler itself must run fast (compilation time must be proportional to program size).
5) The compiler must be portable.
6) It must give good diagnostic & error messages.
7) It must have consistent optimization.

Phases of Compiler



1) Lexical Analysis:

The lexical analysis is also called scanning. It is the first phase of compilation in which the complete source code can be scanned & source program is broken up into group of strings called tokens. A token is a sequence of characters having a collective meaning.

for ex: total = count + rate * 10

The identifier 'total' id1
The 'assignment' Symbol
The identifier 'count' id2
The 'plus' symbol
The identifier 'rate' id3
The 'multiplication' Symbol
The constant number '10'

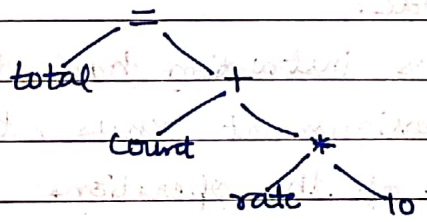
} tokens

The blank characters which are used in the programming statements are eliminated during the lexical analysis phase.

2) Syntax Analysis:

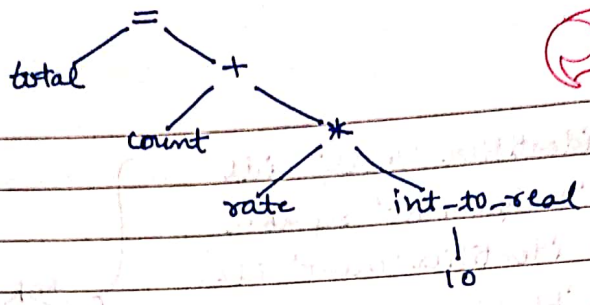
This phase is also called parsing phase. In this the tokens generated by the lexical analyzer are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the token together. The hierarchical structure generated in this phase is called parse tree or syntax tree.

e.g. total - count + rate * 10



3) Semantic Analysis:

Once the syntax is checked in the syntax analyzer phase, the next phase i.e. semantic analysis determines the meaning of the source string e.g. meaning of source string means matching parenthesis in the expression or matching of if-else statements or performing arithmetic operation of the expression that are type compatible or checking the scope of the operation.



4) Intermediate Code generation:

The intermediate code generation is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as three address code, quadruple, triples and postfix form. Here we will consider an intermediate code in three address code form, this is like an assembly language. The three address code consists of instruction each of which has atmost three operands.

t_1 = int_to_real(10)

t_2 = rate * t_1

t_3 = count + t_2

total = t_3

There are certain properties which should be followed by the three address code.

1) Each three address instruction has atmost one operator in addition to the assignment. Thus, the compiler has to divide the order of the operations in the three address code.

2) The compiler must generate a temporarily name to hold the value computed by each instruction. Some three-address instruction may have fewer than three operands. for ex - 1st and last instruction in above example.

5) Code optimization:

This phase attempts to improve the intermediate code this is necessary to have a faster executing code or less

consumption of memory. Thus, by optimizing the code, the overall running time of the target program can be improved.

```
t1 = rate * int_to_real(10)
total = count * t1
```

6) Code generation:

This phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instruction.

```
MOV   R1, rate
MUL   R1, #10.0
MOV   R2, count
ADD   R1, R2
MOV   total, R1
```

Two reasons for phase of compiler to be grouped are:

1) It helps in producing compiler for different source language.
2) Front end phases of many compilers are generally same.

Two parts of a compilation are:

1) Analysis part:

It breaks up the source program into constituent pieces and creates an intermediate representation of source program.

2) Synthesis part:

It constructs the desire target program from the intermediate representation.

Aniket
Date
Page

Symbol Table management:

1) To support these phases of compiler, a symbol table is maintained. The task of symbol table is to store identifier (variables) use in the program.

2) The symbol table also stores information about attributes of each identifier. The attributes of identifier are usually its time, scope and information about the storage allocated for it.

3) The symbol table also stores information about the sub-routines use in the program.

4) The symbol table allows us to find the record for each identifier quickly and to store & retreive data from that record efficiently.

5) During compilation, the lexical analysis detects the identifier and makes its entry in the symbol table.

Error detection & recovery:

To err is human: As programs are written by human beings, therefore they cannot be free from errors. On compilation, each phase detects error. These error must be reported to the error handler whose task is to handle the error. So, that the compilation can proceed.

Normally, the error are reported in the form of message. When the input characters from the input

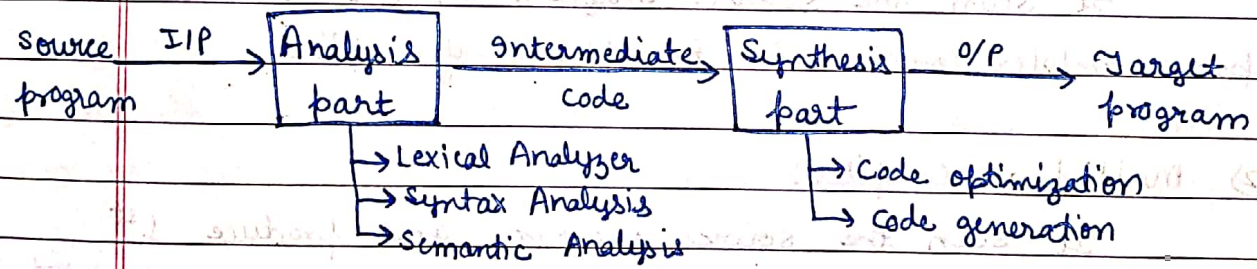
do not form the token, the lexical analyzer detects it as an error.

Large number of errors can be detected in syntax analysis phase are known as syntax errors. During semantics analysis, type mismatch kind of error is usually detected.

Analysis & Synthesis model of compiler:

Analysis part contains all those phases that depends on programming language such as lexical phase, syntax phase and semantic phase.

Synthesis part contains all those phases that depends on machine language such as intermediate code generation, code optimization and code generation.



Parser: is a program which check the syntax. Parser perform two task to construct parse tree.

1) Scan the source program input string from left to right.
2) Select the appropriate production to construct the parse tree.

Parser use two approaches or divided into two categories:

1) Top down parser (from starting to input string) i.e. Expansion
2) Bottom up parser (from input string to starting) i.e. Reduction

Difference between Interpretor and compiler

Interpretor	compiler
1) It directly generate executable code.	1) It may generate executable code or not.
2) It use three phases: lexical, syntax and semantics.	2) It uses all six phases.
3) It read/ scan the source program line by line.	3) It read the whole program.
4) It is slow.	4) It is fast.
5) It takes less space.	5) It takes more space.

Pass: grouping of phases is called pass.

Types of pass:

1) Single pass compiler:

It scan the source program in one time. It means single pass compiler contain all the phases of compiler.

2) Multi pass compiler:

It scan the source program and produce 1st modified form then scans the 1st modified form & produce 2nd modified form and so on. Untill the target code is reached such a compiler is called multipass compiler.

Advantage and disadvantage of multi pass compiler:

1) Single pass compiler is fast because all the compiler code is loaded into memory at once & multi pass compiler is slow because the output of each pass of multi pass compiler is stored on the memory and must be read in each time where next pass start.

2) Single pass compiler takes more space in memory as compare to multi pass compiler.

Compiler Construction Tool:

Writing a compiler is a complicated process & time consuming task. There are some specialised tool for helping in implementation of various phases of compiler. It is called compiler construction tool.

1) Scanner generator: works on lexical analyzer. It takes as an input regular expression, Here, Lex tool is used.

2) Parser generator: Works on Syntax analyzer. It takes as an input context free Grammar. Here, tool is used to implement the parser YACC (yet another compiler compiler).

3) Syntax direction translation engine (SDTE): are used to produce the collection of syntax directed translation scheme which traverse the parse tree and generate intermediate code generation.

4) Data flow engine or analysis: are used to perform data flow analysis needed to perform good code optimization. The data flow analysis involves the gathering of information about how values are transformed formed from one part of program to each part of program.

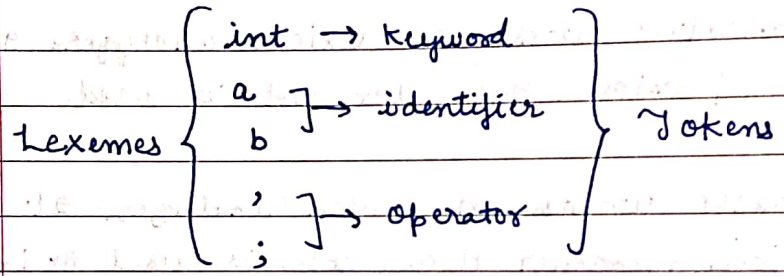
5) Automatic code generator: is used to generate the machine language for the target machine.

Aniket
Date
Page

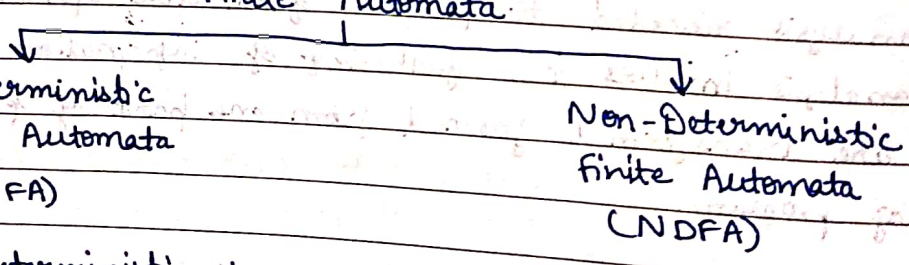
Token: It describes the class or category of input string. for ex - identifier, keywords or constant.

Patterns: is a set of rules that describes the token.

Lexeme: is a sequence of characters in the source program that are matched with the pattern of the token. for ex - int i, switch, num etc.



Automata: is an abstract model of digital computer and automata has a mechanism to read from input tape. Any language is recognized by some language. Hence, automata are basically language acceptor or language recognizer.



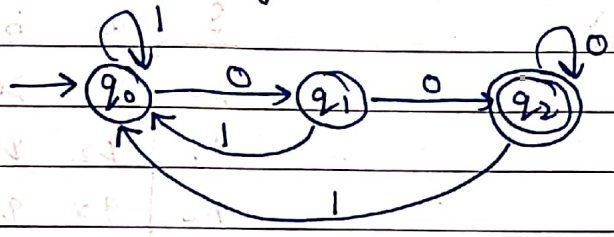
DFA: is deterministic in nature in the sense that each move in the automata is uniquely determined on current input & current state on the other hand.

Aniket
Date
Page

NDFA: is non-deterministic in nature in the sense that each move in the automata is not uniquely determined on current i/p input & current state on the other hand.

Q. Design a finite automata that accepts set of strings such that every string ends with 00. over alphabet $\Sigma = \{0, 1\}$.

Sol.

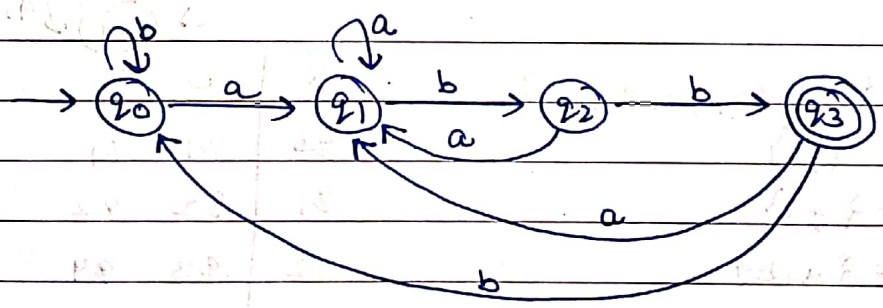


$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
$q_0 = \{q_0\}$
$F = \{q_2\}$

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_0

Q. Design a finite automata which ends with abb.

Sol.



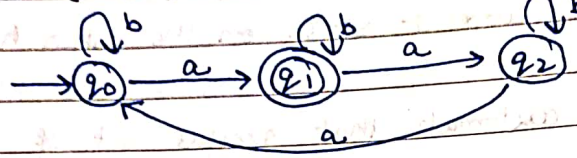
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$
$q_0 = \{q_0\}$
$F = \{q_3\}$

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_0

Aniket
Date
Page

Q. Design a finite automata that accepts the language,
$L = \{ w / n_a \bmod 3 = 1 \;/\; w \in (a,b)^* \}$

Sol.

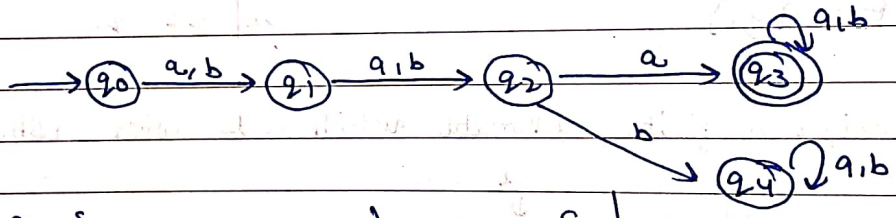


$Q = \{ q_0, q_1, q_2 \}$
$q_0 = \{ q_0 \}$
$F = \{ q_1 \}$
$\Sigma = \{ a, b \}$

δ	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_0	q_2

Q. Design a finite automata in which 3rd symbol from the left hand is always a.

Sol.

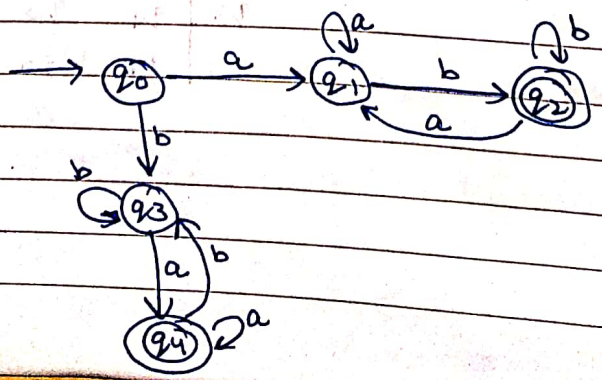


$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$
$q_0 = \{ q_0 \}$
$F = \{ q_3 \}$
$\Sigma = \{ a, b \}$

δ	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_3	q_4
q_3	q_3	q_3
q_4	q_4	q_4

Q. Design a finite automata in which left most symbol is differ from right most symbol.

Sol.



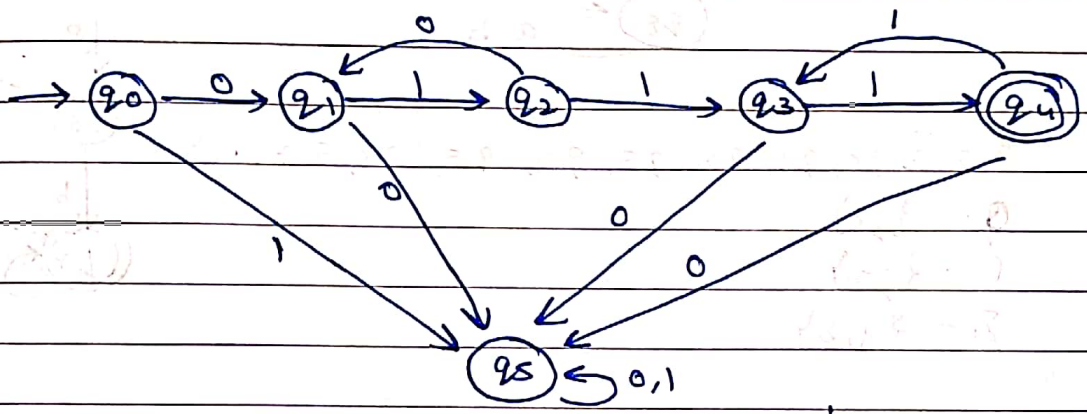
Aniket
Date
Page

$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$q_0 = \{q_0\}$
$F = \{q_2, q_4\}$
$\Sigma = \{a, b\}$

δ	a	b
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_4	q_3
q_4	q_4	q_3

Q. Design a finite automata in which $L = \{(01)^i 1^{2j} \mid (i,j) \geq 1\}$.

Sol.

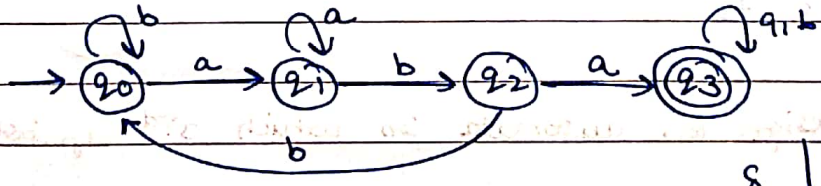


$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
$q_0 = \{q_0\}$
$F = \{q_4\}$
$\Sigma = \{0, 1\}$

δ	0	1
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_1	q_3
q_3	q_5	q_4
q_4	q_5	q_3
q_5	q_5	q_5

Q. Design a finite automata in which $L = \{w \mid$ substring aba $w \in (a,b)^*\}$.

Sol.



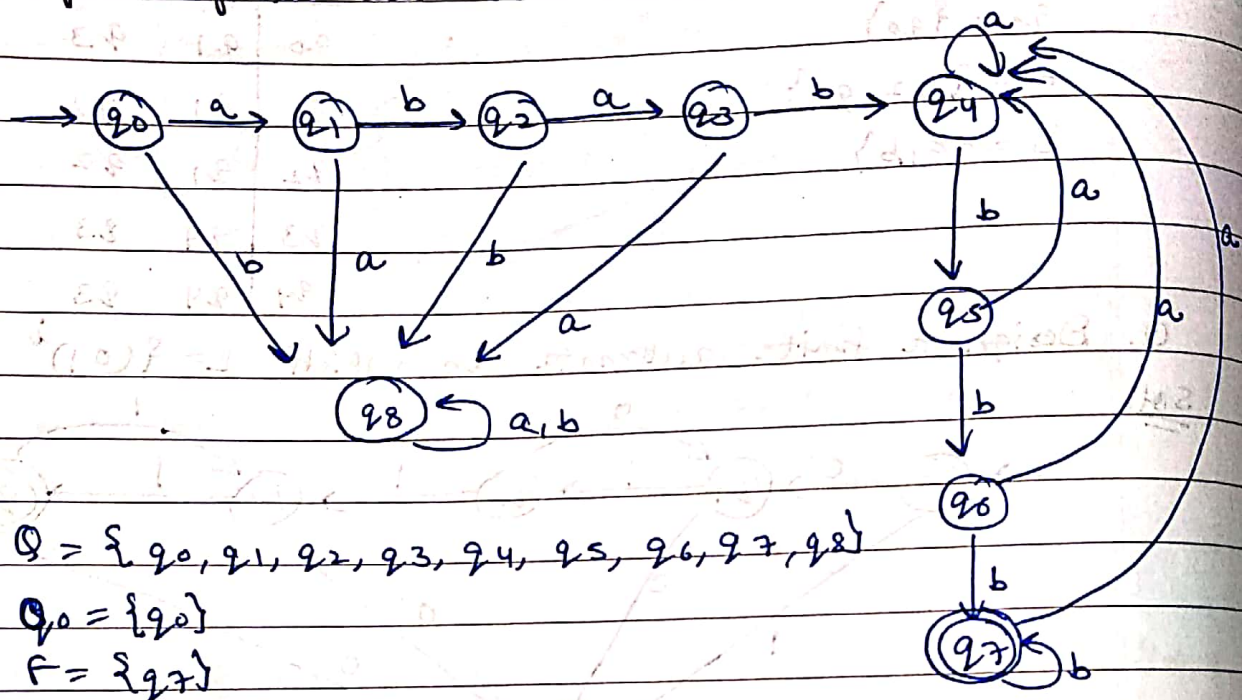
$Q = \{q_0, q_1, q_2, q_3\}$
$F = \{q_3\}$
$q_0 = \{q_0\}$
$\Sigma = \{a, b\}$

δ	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_3	q_0
q_3	q_3	q_3

Aniket
Date
Page

Q. Design a finite automata in which $L = \{(ab)^2 w b^3\}$.

Sol.



$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$

$Q_0 = \{q_0\}$

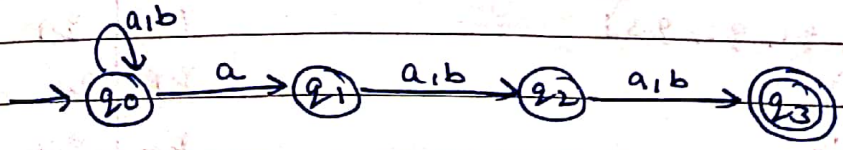
$F = \{q_7\}$

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

δ	a	b
→ q_0	q_1	q_8
q_1	q_8	q_2
q_2	q_3	q_8
q_3	q_8	q_4
q_4	q_4	q_5
q_5	q_4	q_6
q_6	q_4	q_7
* q_7	q_4	q_7
q_8	q_8	q_8

Q. Design an automata in which 3rd symbol from right is a.

Sol



Aniket
Date
Page

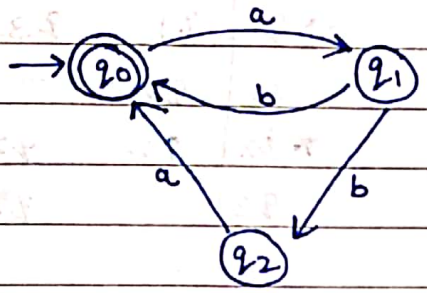
$Q = \{q_0, q_1, q_2, q_3\}$
$q_0 = \{q_0\}$
$F = \{q_3\}$
$\Sigma = \{a, b\}$

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	q_0
q_1	q_2	q_2
q_2	q_3	q_3
$*q_3$	–	–

Q. Design an automata in which $L = \{(ab \cup aba)^*\}$

Sol.

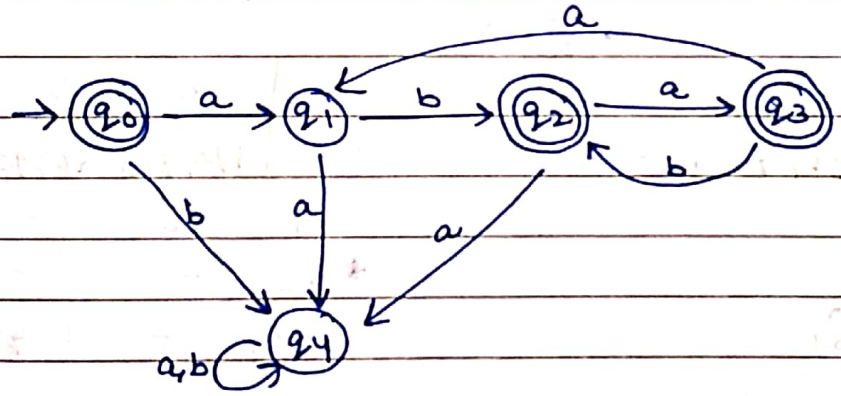
NFA:



δ	a	b
$\rightarrow^* q_0$	q_1	–
q_1	–	$\{q_0, q_2\}$
q_2	q_0	–

$Q = \{q_0, q_1, q_2\}$
$q_0 = \{q_0\}$
$F = \{q_0\}$
$\Sigma = \{a, b\}$

DFA:



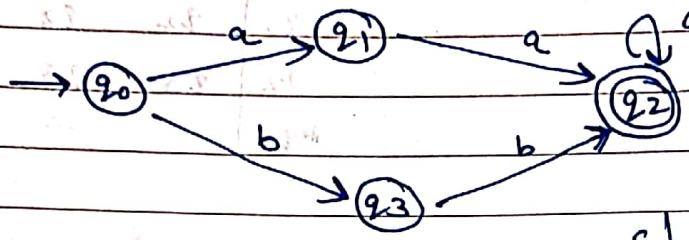
	a	b
$\rightarrow^* q_0$	q_1	q_4
q_1	q_4	q_2
$*q_2$	q_3	q_4
$*q_3$	q_1	q_2
q_4	q_4	q_4

$Q = \{q_0, q_1, q_2, q_3, q_4\}$
$q_0 = \{q_0\}$
$F = \{q_0, q_2, q_3\}$
$\Sigma = \{a, b\}$

Aniket
Date
Page

Q. Design an NFA in which string starts with aa or bb.

Sol.



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

$q_0 = \{q_0\}$

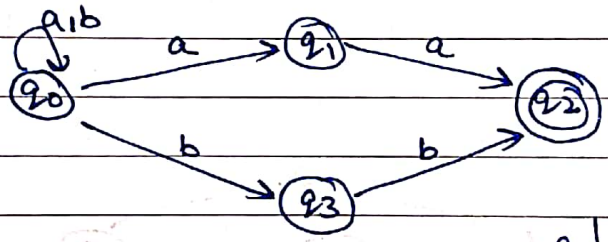
$F = \{q_2\}$

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

δ	a	b
$\rightarrow q_0$	q_1	q_3
q_1	q_2	–
$*q_2$	q_2	q_2
q_3	–	q_2

Q. Design an NFA in which string ends with aa or bb.

Sol.



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

$q_0 = \{q_0\}$

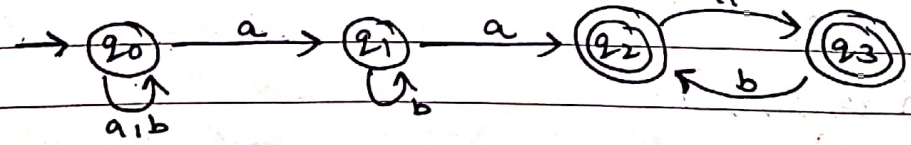
$F = \{q_2\}$

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

δ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
q_1	q_2	–
$*q_2$	–	–
q_3	–	q_2

NFA to DFA conversion:

Q:



Sol: NFA table

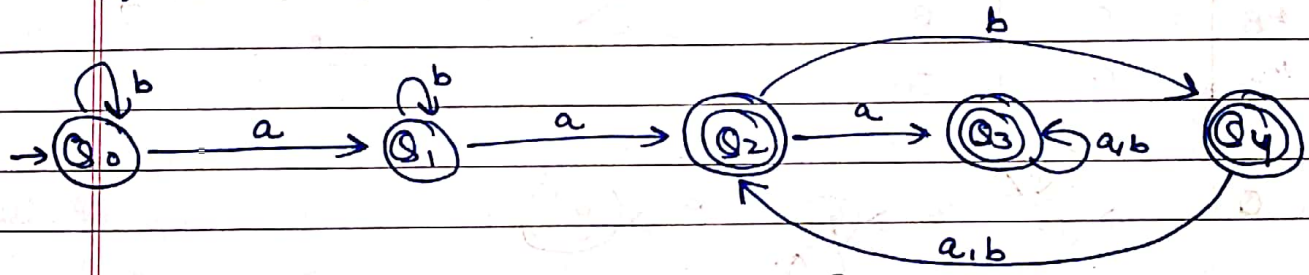
	a	b
q_0	$\{q_0, q_1\}$	q_0
q_1	q_2	q_1
q_2	q_3	q_3
q_3	–	q_2

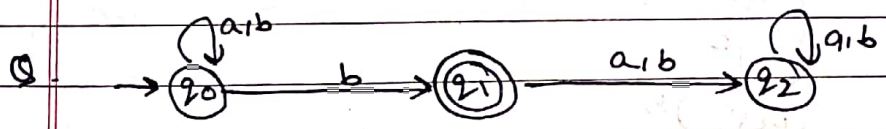
DFA conversion table:

δ	a	b
→ q_0	$\{q_0, q_1\}$	q_0
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$
* $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_3\}$
* $\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$
* $\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$

$q_0 = Q_0$ $\quad \{q_0, q_1\} = Q_1$ $\quad \{q_0, q_1, q_2\} = Q_2$

$\{q_0, q_1, q_2, q_3\} = Q_3$ $\quad \{q_0, q_1, q_3\} = Q_4$





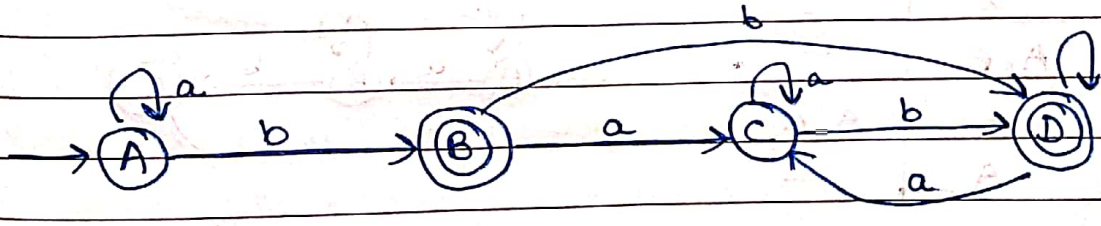
Sol.

NFA

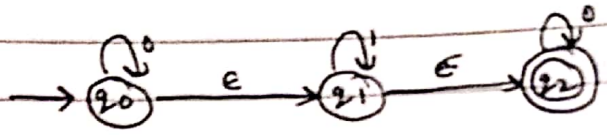
	a	b
→ q_0	q_0	$\{q_0, q_1\}$
* q_1	q_2	q_2
q_2	q_2	q_2

DFA

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

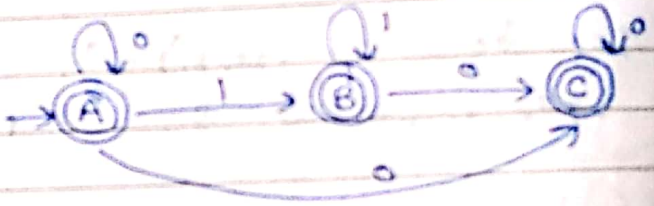


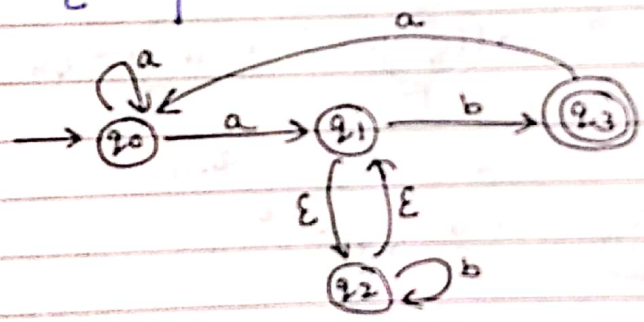
NFA with ε-move ~~to~~ NFA:

Q. 

Sol. ε-closure of $q_0 = \{q_0, q_1, q_2\} = A$
ε-closure of $q_1 = \{q_1, q_2\} = B$
ε-closure of $q_2 = \{q_2\} = C$

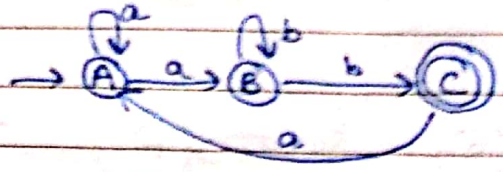
	ε-closure of	
	0	1
→*A	A, C	B
*B	C	B
*C	C	–



Q. 

Sol. ε-closure of $q_0 = \{q_0\}$
ε-closure of $q_1 = \{q_1, q_2\}$
ε-closure of $q_2 = \{q_1, q_2\}$
ε-closure of $q_3 = \{q_3\}$

	ε-closure of	
	a	b
→ A	A, B	–
B	–	B, C
* C	A	–



Regular Expression

are mathematical symbolism which describe the set of strings of specific language. It provide convenient and useful notation for representing tokens.

There are some rules that describe definition of the regular expression over the input set denoted by Σ.

Step 1: E is a regular expression that denotes the set containing empty strings.

Rule 2: If R_1 and R_2 are regular expression then $R_3 = R_1 + R_2$ is also regular expression which represent union operation.

Rule 3: If R_1 and R_2 are regular expression then $R = R_1 R_2$ is also a regular expression which represents concatenation operation.

Rule 4: If R_1 is a regular expression then $R = R_1^*$ is also a regular expression which represents Kleen closure.

A language denoted by RE is said to be a regular set or regular language.

Q. Write the RE, where all string ends with 01 over $\Sigma = \{0,1\}$.

Sol. $(0+1)^* 01$

Q. Write the RE in which string starts and ends with b.

Sol. $b\,(a+b)^*\,b$

Q. Write the RE for the language $L = \{a^n b^m / n \geq 4, m \leq 3\}$.

Sol. $RE = a^4 a^* (\epsilon + b + b^2 + b^3)$

Q. Write the RE for the language $L = \{ab^n \; w / n \geq 3 \;\; w \in (a,b)^+\}$

Sol. $ab^3 b^* (a+b)^+$

Aniket
Date
Page

Q. Design a RE where all strings do not end with double letter. $\Sigma = \{a, b\}$.

Sol. $(a+b)^*\ (ab + ba)$

Q. Write a RE in which left most symbol is differ from right most symbol.

Sol. $a\,(a+b)^*\,b + b\,(a+b)^*\,a$

Q. Write a RE in which $L = \{ w \in (a|b)^* \mid n_a(w) \bmod 3 = 0 \}$.

Sol. $[b^*\,a\,b^*\,a\,b^*\,a\,b^*]^*$

Q. Write a RE in which $L = \{ a^{2n}\, b^{2m+1} \mid n, m \geq 0 \}$.

Sol. $(aa)^*\,(bb)^*\,b$

Q. Write a RE in which second symbol is 0 and 4th symbol is 1

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

Q. Write a RE in which string has exactly four 1's.

Sol. $0^*\,1\,0^*\,1\,0^*\,1\,0^*\,1\,0^*$

Q. Write a RE in which $L = \{ (01)^i\, 1^{2j} \mid i, j \geq 0 \}$.

Sol. $(01)^*\,(11)^*$

Q. Write the RE in which $L = \{ a^n\, b^m$, where $n+m$ is even$\}$

Sol. Case 1: when both are even

$R_1 = (aa)^*\,(bb)^*$

Case 2: when both are odd

$R_2 = a\,(aa)^*\,b\,(bb)^*$

$R = R_1 + R_2 = (aa)^*\,(bb)^* + a\,(aa)^*\,b\,(bb)^*$

Aniket
Date
Page

Q. Write the RE in which $L = \{a^n b^m$, where $n+m$ is odd$\}$.

Sol. Case 1: when a is odd & b is even.

$R_1 = a(aa)^* (bb)^*$

Case 2: when a is even & b is odd

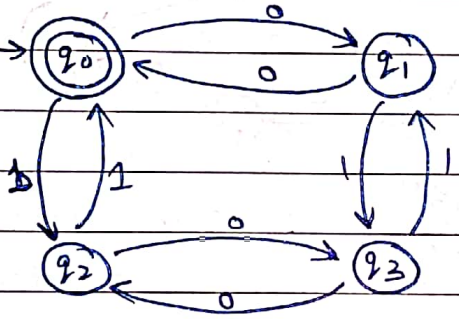
$R_2 = (aa)^* b (bb)^*$

$R = R_1 + R_2$

$= a(aa)^* (bb)^* + (aa)^* b (bb)^*$

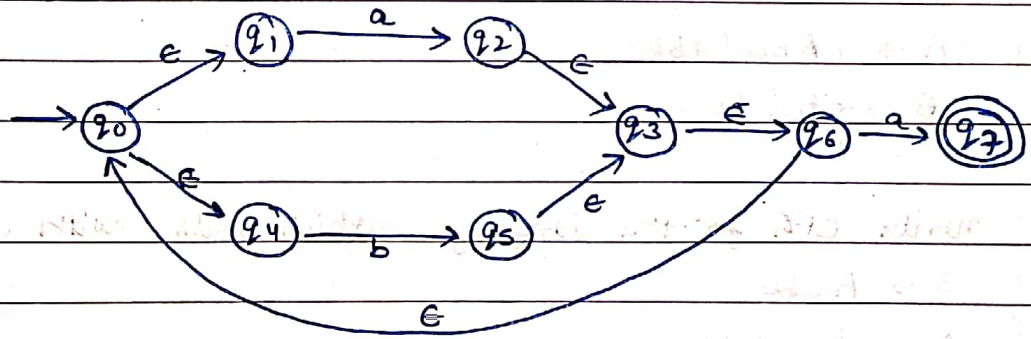
Q. Draw a DFA for the language in which no. of a's and no. of b's are even.

Sol.

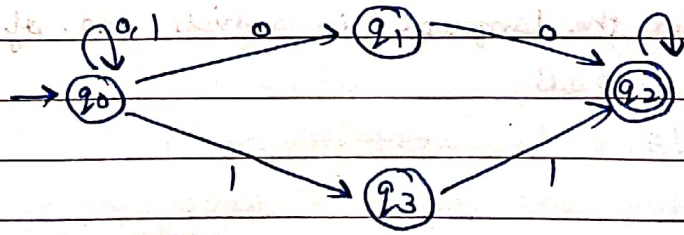


Direct Method for designing Finite automata from regular Exp. :- Thompson's rule:

Q. $(a+b)^* a$



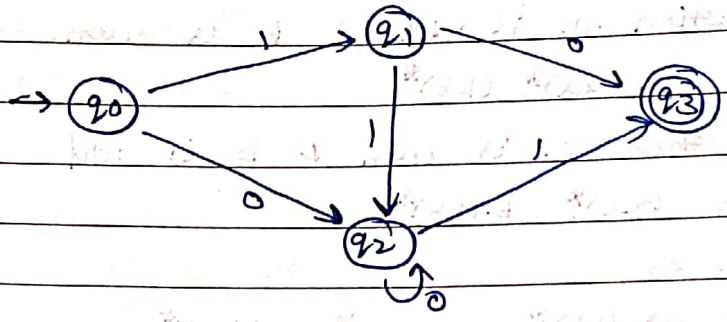
Q. $(0+1)^* (00+11)(0+1)^*$



Aniket
Date
Page

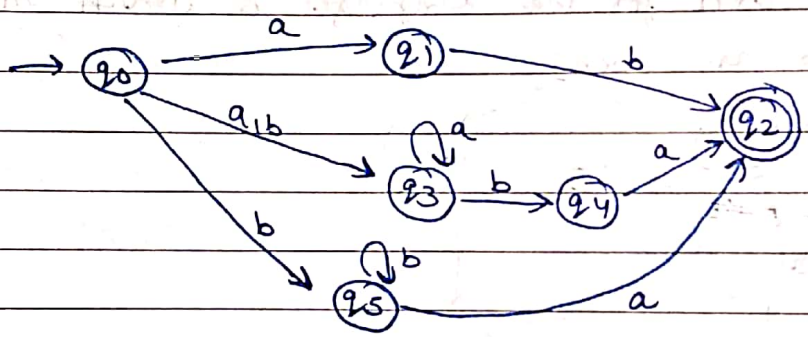
Q. $10 + (0+11)0^*1$

Sol.



Q. $ab + (a+b)a^*ba + b^+a$

Sol.



Context free Grammar

Q. $L = a^n b^n \; ; n \geq 1$ $\quad S \rightarrow aSb \mid ab$

Q. $L = a^n b^{2n} c^m d^m \quad n \geq 1, m > 0$

$S \rightarrow AB$

$A \rightarrow aAbb \mid abb$

$B \rightarrow cBd \mid \epsilon$

Q. Write CFG for the language which ends with abb.

$S \rightarrow Aabb$

$A \rightarrow aA \mid bA \mid \epsilon$

Q. Write CFG for the language in which no. of a is 3.

$S \rightarrow BaBaBaB$

$B \rightarrow bB \mid \epsilon$

Q. Write CFG for balanced set of parenthesis.

$S \rightarrow (S) / SS / \epsilon$

Q. Write CFG for palindrome grammar:

$S \rightarrow aSa / bSb / a / b / \epsilon$

Parse Tree: is an ordered tree in which nodes are labelled by left side of production and children are represented corresponding right side. There are two approaches to construct parse tree:

1) Left most derivation
2) Right most derivation

Q. Draw parse tree and write left most derivation and right most derivation for the string id+id*id by

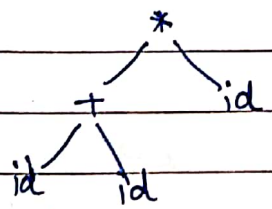
$E \rightarrow E+E / E*E / id$

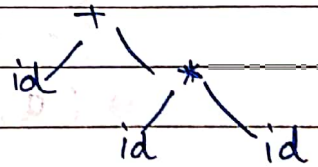
Sol. Left Most Derivation:

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

Right Most Derivation:

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





Q. Consider the following grammar:

$S \rightarrow (L) / a$

$L \rightarrow L, S / S$

i) Write terminal or non-terminal for the given grammar.
ii) Construct the parse tree for the given string:

Aniket
Date
Page

a) (a,a) b) (a,(a,a)) c) (a,((a,a),(a,a)))

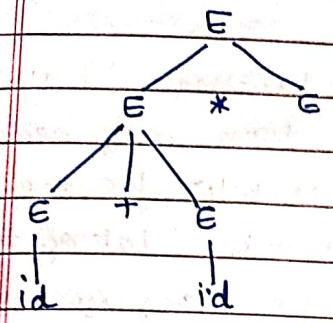
i) Terminal - ( ), a
ii) Non-terminal - L, S

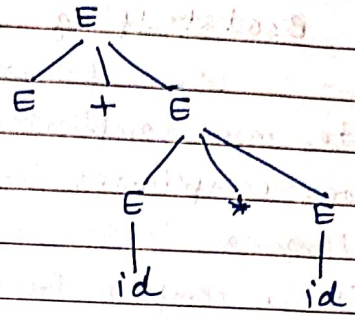
ii) a) S → (L)
(L,S)
(S,S) → (a,a)

b) S → (L,S)
(L,(L))
(S,(L,S))
(a,(S,S))
→ (a,(a,a))

c) S → (L)
(L,S)
(S,(L))
(S,(L,S))
(S,(S,S))
(S,((L),(L)))
(S,((L,S),(L,S)))
(S,((S,S),(S,S)))
→ (a,((a,a),(a,a)))

Ambiguity: If two different parse tree are generated with same approach (LMD & RMD) for same input string then grammar is called ambigious grammar.
for ex - E → E+E / E*E / id
w = E+E*E





Cousins of Compiler:

1) Preprocessor: The output of preprocessor may be given as the input to compiler. Preprocessor allow user to use macros in the program. Macros means some set of instruction which can be used repeatedly in the program.

2) Assembler: Some compiler produce the assembly code as output which gives as a input to the assembler. The assembler is a kind of translator which takes a assembly program as input and produces the machine code as output.

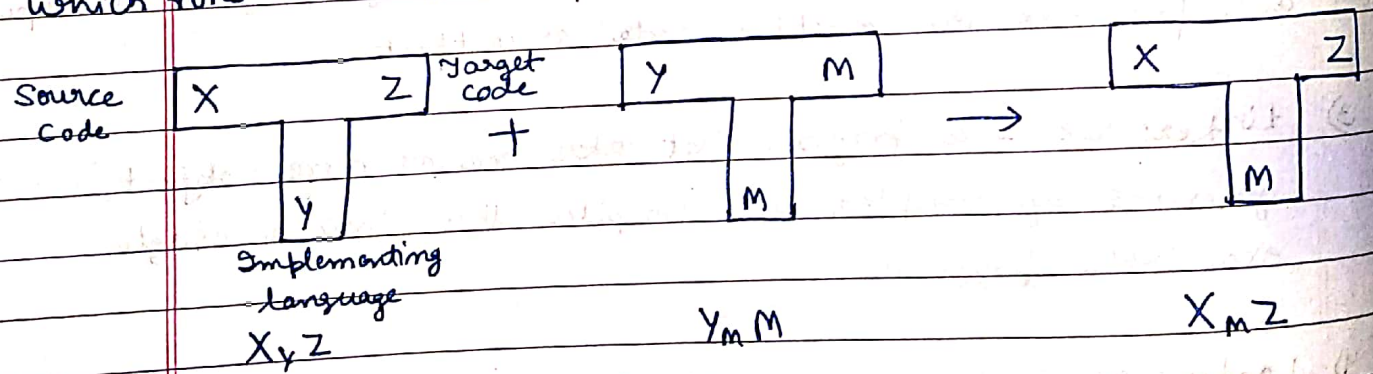
3) Linker: It is a program that takes one or more object generated by compiler and compiles them into a single executable program.

4) Loader: It is part of an OS that is responsible for loading a program.

Bootstrapping

is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated programs. Writing a compiler for any high level language is the complicated process. It takes lot of time to write a compiler from scratch hence simple language is used to generate the target code in some stages.

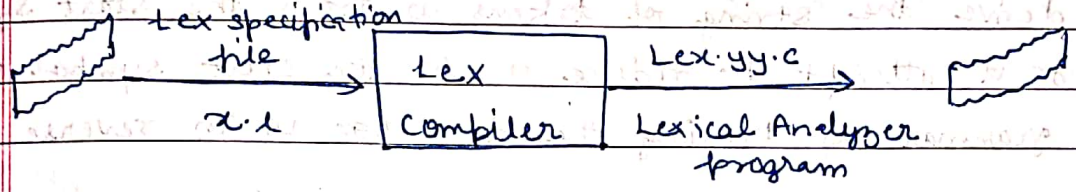
Ex: Suppose we want to write a cross compiler for new language X, the implementation language for this language is Y and target code being generate is in the language Z. It means if we want to create $X_Y Z$, now if existence compiler Y runs on machine M and generate the code for M. Then it is denoted as $Y_M M$. Now if we run the machine $X_Y Z$ by $Y_M M$ then we get a compiler $X_M Z$ that means a compiler for source language X that generate a target code in language Z which runs on machine M.

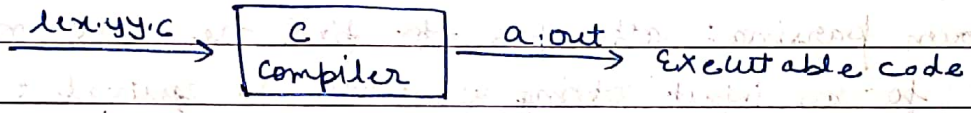


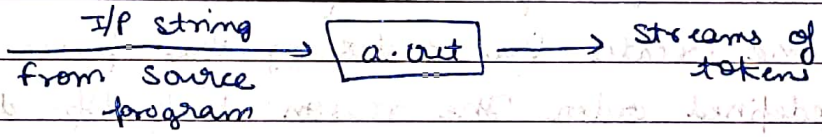
Lex Compiler:

The Lex specification file can be created by using the extension .l. for ex- the specification file can be x.l. this file is given to the lex compiler to produce lex.yy.c. this lex.yy.c is C program which is actually a lexical analyzer program. this Lex specification file store the RE for the token and lex.yy.c file consists of the tabular repres

In the specific file, Lex actions are associated with every RE. These actions are simply the pieces of C code. This C code is then directly carried to the lex.yy.c. finally the compiler compiles this generated lex.yy.c and produces the object program a.out when some input string is given to a.out then sequence of tokens get generated.







Unit - 2

Basic Parsing techniques

Aniket
Date
Page

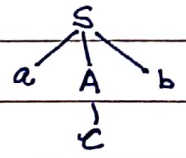
Parser: A syntax analyzer or parser is a program that perform syntax analysis. A parser obtain the string of tokens from the lexical analyzer and verifies whether or not a string is a valid construct of the source language. It means whether or not it can be generated by the grammar for the source language. The parser either attempts to derive the string of tokens w from the start symbol S or it attempt to reduce w to the start symbol of the grammar by tracing the derivation w in reverse.

Top-down parsing: attempts to find the left most derivation for an input string w, which is equivalent to constructing a parse tree for input string w that starts from the root and creates the nodes of the parse tree in the predefined order. The reason that top-down parsing seeks the LMD for an input string w and not RMD is dead to the input string w is scan by the parser from left to right one token at a time and LMD generates the leaf of the parse tree in left to right order which match the input scan order.

Q. $S \rightarrow aAb$

$A \rightarrow cd / c$

$w = acb$



Problems with top-down parsing:

There are certain problems in top-down parsing in order to implemented the parsing. We need to eliminate these problems:

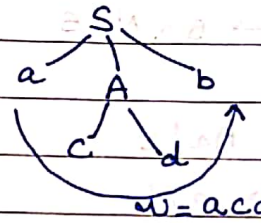
Aniket
Date
Page

1) Backtracking: is a technique in which for the expansion of non-terminal symbol we choose one alternative and if some mis-match occur then we try another alternative if any.

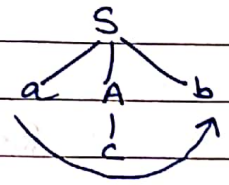
for ex- $S \to aAb$

$A \to cd / c$

$w = acb$



$w = acdb$



$w = acb.$

If for any non-terminal, there are multiple production rules beginning with the same input symbol then to get the correct derivation, we need to try all these alternatives. Secondly in backtracking, we need to need to move some levels upward in order to check the possibilities.

2) Left Recursion: If left recursion is present in the grammar, then it creates a serious problem because of the left recursion, the top-down parser can enter in infinite loop.

To eliminate left recursion we need to modify the grammar. Let G is the CFG having a production rule with left recursion.

$A \to A\alpha / \beta$ has a solution

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

{ After removing left recursion }

Q. $E \to E+T/T$, remove left recursion from given grammar.

Sol. $E \to TE'$

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

Aniket
Date
Page

Q. $A \rightarrow ABd / Aa / a$ Remove left recursion.

$B \rightarrow Be / b$

Sol. for $A \rightarrow ABd / a$

$A \rightarrow aA'$

$A' \rightarrow BdA' / \epsilon$

for $A \rightarrow Aa / a$

$A \rightarrow aA'$

$A' \rightarrow aA' / \epsilon$

for $B \rightarrow Be / b$

$B \rightarrow bB'$

$B' \rightarrow eB' / \epsilon$

So,

$A \rightarrow aA'$

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

$B \rightarrow bB'$

$B' \rightarrow eB' / \epsilon$

3) Left factoring: If the grammar is left factored then it becomes suitable for the use. Basically left factoring is used when it is not clear that which of the two alternative is used to expand the non-terminal.

$A \rightarrow \alpha B_1 / \alpha B_2$ is a production then it is not possible for us to take a decision whether to choose 1st rule or 2nd rule. On such a situation, the above grammar can be left factored as follows:

$A \rightarrow \alpha A'$

$A' \rightarrow B_1 / B_2$

Q. $S \rightarrow iEtS / iEtSeS / a$

$E \rightarrow b$

Maintain the left factoring in the above grammar.

Sol. $S \rightarrow iEtSS' / a$

$S' \rightarrow eS / \epsilon$

$E \rightarrow b$

Aniket
Date
Page

Q. $A \rightarrow aAB / aA / a$

$B \rightarrow bB / b$

Sol. $A \rightarrow aA'$

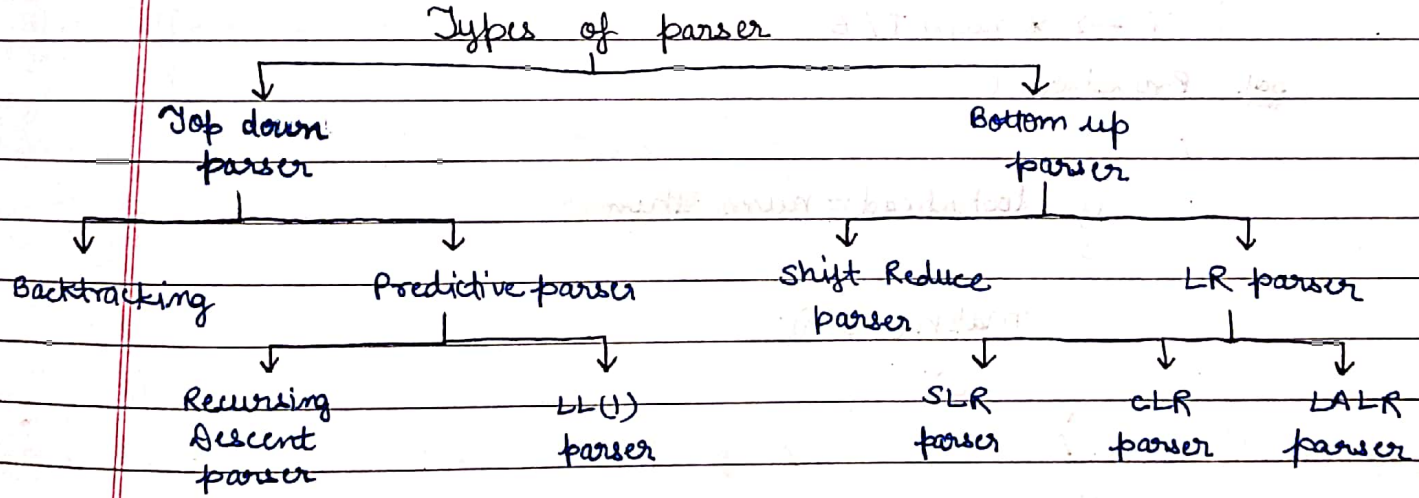
$A' \rightarrow AA'' / \epsilon$

$B \rightarrow bB'$

$B' \rightarrow B / \epsilon$

4) Ambiguity: The ambigious grammar is not desirable in top-down parsing. Hence, we need to remove the ambiguity from the grammar, if it is present.
for ex:- $E \rightarrow E+E / E*E / id$ is an ambigious grammar.
If we design the parse tree for the input string id+id*id

Note: One thing that the grammar is unambigious but it is left recursive and elimination of such recursion is again must.



Recursive Descent parser

A parser that uses collection of recursive procedure for parsing the given input string is called recursive descent parser. In this parser, the CFG is used to build a recursive routines.

Aniket
Date
Page

Basic steps for construction of RDP:

a) The RHS of the rule is directly converted into program code symbol by symbol. If the input symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.

b) If the input symbol is terminal then it is matched with the look ahead from input. The look ahead pointer has to be advance or matching of the input symbol.

c) If the producing rule has many alternatives then all these alternate has to be combined into a single body of procedure.

d) The parser should be activated by procedure corresponding to the start symbol.

Q. Construct a recursive descent parser for the given CFG.

E → num T

T → * num T / ε

Sol.

```
Procedure E
{
    if lookahead = num then
    {
        match(num);
        T();
    }
    else
        error;
    if lookahead = $ then
    {
        declare success;
    }
    else
        error;
}
```

```
Procedure T
{
   if lookahead = '*' then
   {
      match('*');
      if lookahead = num then
      {
         match(num);
         T();
      }
      else
         error;
   }
   else NULL;
}

Procedure match(token t)
{
   if lookahead = t then
      lookahead = next_token;
   else error;
}
```

Consider that the input string 3 * 4. We will pass this string using given procedure. The parser will be activated by calling procedure E. Since the first input character 3 is matching with num, the procedure match num will be invoked and then the lookahead points to the next token and the procedure T is given. A match with * sign is found, hence lookahead = next_token. Now 4 is matching with num, hence the procedure for num is fulfilled with procedure T is invoked and T is replace by null.

As lookahead points to $ and we quit by reported succes

Q. Construct RDP for the given CFG:

$A \rightarrow AB / +aB$

$B \rightarrow eC$

$C \rightarrow *cC / \epsilon$

Sol.

```
Procedure A
{
    if lookahead = '+' then
    {
        match ('+');
        if lookahead = 'a' then
        {
            match ('a');
            B();
        }
        else error;
    }
    else
    {
        A();
        B();
    }
} if lookahead = $ then
  {
    declare success;
  }
  else
      error;
}
```

```
Procedure B
{
    if lookahead = 'c' then
    {
        match ('c');
        C ();
    }
    else error;
}

Procedure C
{
    if lookahead = '*' then
    {
        match ('*');
        if lookahead = 'c' then
        {
            match ('c');
            C ();
        }
        else error;
    }
    else NULL;
}

Procedure match (token t)
{
    if lookahead = t
        lookahead = next_token;
    else error;
}
```

Aniket
Date
Page

FIRST & FOLLOW function:

1) FIRST(): FIRST(α) is a set of terminal symbol that are first symbol appearing at RHS in the derivation of α. There are following rules used to compute the FIRST():

a) If the terminal symbol is a, then FIRST(α) = {a}.

b) If there is a rule α → NULL, then FIRST(α) = {ε}.

c) for the rule A → X1/X2/X3/---/Xk then

FIRST(A) = FIRST(X1) ∪ FIRST(X2) ∪ ..... ∪ FIRST(Xk)

2) FOLLOW(): FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A. The rules for computing FOLLOW() are as follows:

a) for the start symbol S, place $ in FOLLOW(S).

b) If there is a production A → αBβ

i) FIRST(β) does not contain NULL then

FOLLOW(B) = FIRST(β)

ii) FIRST(β) contain NULL then

FOLLOW(B) = [FIRST(β) − ε] ∪ FOLLOW(A)

c) If there is a production A → αB then

FOLLOW(B) = FOLLOW(A).

Q. Find FIRST() and FOLLOW() of the grammar.

E → TE'

E' → +TE'/ε

T → FT'

T' → *FT'/ε

F → (E)/id

Sol. FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

FOLLOW(E){), $ }
FOLLOW(E'){), $ }
FOLLOW(T){ +, $,) }
FOLLOW(T'){ +, $,) }
FOLLOW(F){ *, +, $,) }

Q. Find the FIRST() and FOLLOW() of the grammar:
S → AaAb / BbBa
A → ϵ
B → ϵ

Sol. FIRST(S) = { a, b }
FIRST(A) = { a }
FIRST(B) = { b }
FOLLOW(S) = { $ }
FOLLOW(A) = { a, b }
FOLLOW(B) = { a, b }

Q. S → aAB / bA / ϵ
A → aAb / ϵ
B → bB / ϵ

Sol. FIRST(S) = { a, b, ϵ }
FIRST(A) = { a, ϵ }
FIRST(B) = { b, ϵ }
FOLLOW(S) = { $ }
FOLLOW(A) = { b, $ }
FOLLOW(B) = { $ }

Q. For the following grammar, find the set for leading & trailing:
S → iCtSA / a
A → eS / ϵ
C → b

Sol. FIRST(S) = { i, a }
FIRST(A) = { e, ϵ }
FIRST(C) = { b }
FOLLOW(S) = { $, e }
FOLLOW(A) = { $, e }
FOLLOW(C) = { t }

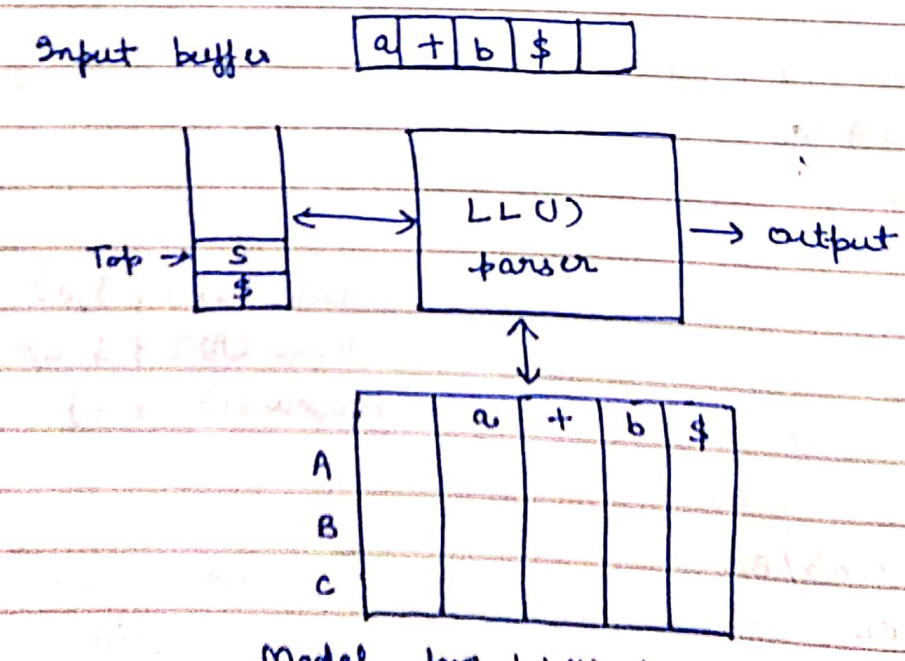
Q. S → ACB / CbB / Ba
A → da / BC
B → g / ϵ
C → h / ϵ

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

Q.

S' → S#	FIRST(S') = { a, b }	FOLLOW(S') = { $ }
S → ABC	FIRST(S) = { a, b }	FOLLOW(S) = { # }
A → a / bbD	FIRST(A) = { a, b }	FOLLOW(A) = { a, b, # }
B → a / ϵ	FIRST(B) = { a, ϵ }	FOLLOW(B) = { b, # }
C → b / ϵ	FIRST(C) = { b, ϵ }	FOLLOW(C) = { # }
D → c / ϵ	FIRST(D) = { c, ϵ }	FOLLOW(D) = {

Predictive LL(1) parser: This top-down parsing algorithm is of non-recursive type. In this type of parsing, a table is built for LL(1), the first L means the input is scanned from left to right. The second L means it uses left most derivation for input string and the number 1 in the input symbol means it uses only 1 input symbol to predict the parsing process.



Model for LL(1) parser

The data structure used by LL(1) parser

1) Input buffer
2) Stack
3) Parsing table

The LL(1) parser uses input buffer to store the input tokens. The stack is used to hold the left sentential form. The symbol in right hand side of rule are pushed into the stack in reverse order. It means from right to left. The table can be represented by M[A,a] where A is non-terminal & a is current input symbol.

Algorithm for Predictive Parser table:

The construction of predictive parser table is an important activity in predictive parsing method. This algorithm requires FIRST & FOLLOW function.

Input: The CFG G

Output: Predictive parsing table (M)

Algo: for the rule $A \to \alpha$ of grammar G

1) for each a in FIRST(α) create entry $M[A,a] = A \to \alpha$ where a is the terminal symbol.
2) for null in FIRST(α) create entry $M[A,b] = A \to \alpha$ where b symbols from FOLLOW(A).
3) If null in FIRST(α) and $ is in FOLLOW(A) then create entry in the table $M[A,\$] = A \to \alpha$.
4) All the remaining entry in the table M are marked as syntax errors.

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

Check whether the given grammar is LL(1) or not.

Aniket
Date
Page

Sol. FIRST(E) = { (, id}
FIRST(E') = { +, ϵ}
FIRST(T) = { (, id}
FIRST(T') = { *, ϵ}
FIRST(F) = { (, id}

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

	+	*	(	)	id	$
E	Error	Error	E→TE'	Error	E→TE'	Error
E'	E'→+TE'	Error	Error	E'→ϵ	Error	E'→ϵ
T	Error	Error	T→FT'	Error	T→FT'	Error
T'	T'→ϵ	T'→*FT'	Error	T'→ϵ	Error	T'→ϵ
F	Error	Error	F→(E)	Error	F→id	Error

So, there is no double entry. So, it is LL(1) parser.

Q. S → AaAb / BbBa
A → ϵ
B → ϵ

Sol. FIRST(S) = { a, b}
FIRST(A) = { ϵ}
FIRST(B) = { ϵ}

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

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

So, it is LL(1) parser.

Q. S → iCtSA / a
A → eS / ϵ
C → b

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

FOLLOW(S) = { e, $}
FOLLOW(A) = { e, $}
FOLLOW(C) = { t}

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

No, it is not LL(1) parser because M[A,e] has double entry A→eS, A→ϵ, so, it is not LL(1) parser.

Q. find FIRST and FOLLOW function of the given CFG:

S→aBDh — FIRST(S) = {a} — FOLLOW(S) = {$}

B→cC — FIRST(B) = {c} — FOLLOW(B) = {g, f, h, $}

C→bC/ϵ — FIRST(C) = {b, ϵ} — FOLLOW(C) = {g, f, h, $}

D→EF — FIRST(D) = {g, f, ϵ} — FOLLOW(D) = {h}

E→g/ϵ — FIRST(E) = {g, ϵ} — FOLLOW(E) = {f, h}

F→f/ϵ — FIRST(F) = {f, ϵ} — FOLLOW(F) = {h}

Q. check the given grammar is LL(1) or not:

S→Aa/b

A→Ac/Sd/ϵ

Sol. first, we have to remove left recursion

S→Aa/b — FIRST(S) = {a, b, c} — FOLLOW(S) = {d, $}

A→SdA'/A' — FIRST(A) = {a, b, c, ϵ} — FOLLOW(A) = {a}

A'→cA'/ϵ — FIRST(A') = {c, ϵ} — FOLLOW(A') = {a}

	a	b	c	d	$
S	S→Aa	S→b	S→Aa	Error	Error
A	A→SdA' A→ϵ	A→SdA'	A→A'	Error	Error
A'	A'→ϵ	Error	A'→cA'	Error	Error

No, it is not LL(1) parser because M[A,a] has double entry i.e. A→SdA' & A'→ϵ. so it is not LL(1) parser.

Aniket
Date
Page

Q. Check the given grammar in LL(1) or not.

$A \to aAB / aA / a$

$B \to bB / b$

Sol: Remove left factors:

$A \to aA'$	$A \to a\,A'$
$A' \to AB / A / \epsilon$	$A' \to AA'' / \epsilon$
$B \to bB'$	$A'' \to B / \epsilon$
$B' \to B / \epsilon$	$B \to bB'$
	$B' \to B / \epsilon$

FIRST(A) = {a}
FIRST(A') = {a, ϵ}
FIRST(A'') = {b, ϵ}
FIRST(B) = {b}
FIRST(B') = {b, ϵ}

FOLLOW(A) = {b, $}
FOLLOW(A') = {b, $}
FOLLOW(A'') = {b, $}
FOLLOW(B) = {b, $}
FOLLOW(B') = {b, $}

	a	b	$
A	$A \to aA'$	Error	Error
A'	$A' \to AA''$	$A' \to \epsilon$	$A' \to \epsilon$
A''	Error	$A'' \to B/\epsilon$	$A'' \to \epsilon$
B	Error	$B \to bB'$	Error
B'	Error	$B' \to B/\epsilon$	$B' \to \epsilon$

Here M[A'', b] & M[B', b] has double entry. So, it is not LL(1) parser.

Bottom-up parsing:

In this method, the input string is taken first and we try to reduce this string with the help of grammars and try to obtain the start symbol. The process of parsing holds

success as soon as we reach to a start symbol. The parse tree constructed from bottom to up, it means from leaf to root. In this process, the input symbols are placed at the leaf node after successfully parsing. The leaf nodes together are reduced further to internal nodes. These internal nodes are further reduced and finally a root node is obtained.

In this process, basically parser tries to identify the right hand side of production rule and replaced it corresponding left hand side. This activity is called reduction. Thus, the parse-tree in bottom-up parsing is designed from leaf to root and in the parsing, the main task is to find the production that can be used for the production.

Operator Precedence Parser:

A grammar is said to be operator precedance, if it holds following property:

1) No production on the right hand side is null.
2) There should not be any production rule containing two adjacent non-terminal at the right hand side.

Q. Consider the grammar for arithmetic expression:

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \wedge$

This is not operator precedance grammar. As in the rules $E \rightarrow EAE$, it contain two adjacent non-terminal. Hence, first we convert it into equivalent operator precedance grammar by removing A.

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E \mid (E) \mid -E \mid id.$

Aniket
Date
Page

In operator precedance parsing, we will first define precedance relation $< \; > \; =$ between terminal. The meaning of these relation is

1) $p < q$: It means p gives more precedance to q.
2) $p = q$: It means p has same precedance as q.
3) $p > q$: It means p takes precedance over q.

	id	+	*	$
id		.>	.>	.>
+	<.	.>	.>	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Now consider the string id + id * id, we will insert $ symbol at start and end of the input string. We will also insert precedance operator by refering the precedance relation table:

$ <. id .> + <. id .> * <. id .> $

Advantages: 1) This type of parsing is simple to implement.

Disadvantage: 1) The operator like '-' has two different precedance (unary, binary). Hence, it is hard to handle.
2) This type of parsing is applicable to only small class of grammar.

Application: The operator precedance parsing is done in a language having operators. Hence, in SNOBOL operation parsing is done

Shift Reduce Parser:

attempts to construct the parse tree from leaves to root. Thus, it works on the principle of bottom up parser. A shift reduce parser requires following data structure. The input buffer storing the input string. A stack for storing and access the left hand sides and right hand side of the rules.

The parser performs following basic operation:

a) shift:

Moving off the symbols from input buffer on to the stack. This action is called shift.

b) Reduce:

If the handle appears on the top of the stack then the reduction of it by appropriate rule is done that means R.H.S of the rule is popped and left hand side is pushed in. This action is called reduce action.

c) ACCEPT:

If the stack contain the start symbol only and input buffer is empty at the sametime then that action is called accept. When accept state is obtained in the process of parsing then it means a successful parsing is done.

d) Error:

A situation in which parser cannot either reduce or shift. The symbol cannot even perform the accept action is called an error.

Aniket
Date
Page

Q. $E \rightarrow E+E / E*E / id$. Perform shift reduce parsing of input string $id_1 + id_2 * id_3$.

Sol.

Stack	Input Buffer	Parsing action
$	$id_1+id_2*id_3\$$	shift
$\$id_1$	$+id_2*id_3\$$	Reduce $E \rightarrow id$
$\$E$	$+id_2*id_3\$$	shift
$\$E+$	$id_2*id_3\$$	Shift
$\$E+id_2$	$*id_3\$$	Reduce $E \rightarrow id$
$\$E+E$	$*id_3\$$	Reduce $E \rightarrow E+E$
$\$E$	$*id_3\$$	Shift
$\$E*$	$id_3\$$	Shift
$\$E*id_3$	$\$$	Reduce $E \rightarrow id$
$\$E*E$	$\$$	Reduce $E \rightarrow E*E$
$\$E$	$\$$	Accept

Q. $S \rightarrow TL;$
$T \rightarrow int / float$
$L \rightarrow L,id / id$

Parse the input string
int id, id;

Stack	Input buffer	Parsing Action
$	int id, id; $	Shift
$ int	id, id; $	Reduce $T \rightarrow int$
$ T	id, id; $	Shift
$ T id	,id; $	Shift
$ T L	, id; $	Reduce $L \rightarrow id$
$ T L,	id; $	Shift
$ T L, id	; $	shift
$ T L	; $	Reduce $L \rightarrow L,id$
$ TL;	$	shift
$ S	$	Accept

Aniket
Date
Page

Q. $S \rightarrow (L)/a$

$L \rightarrow L,S / S$

Perform shift reduce parsing of input string (a, (a,a))

Sol.

Stack	Input buffer	Parsing action
$	(a, (a,a)) $	Shift
$(	a, (a,a)) $	Shift
$(a	, (a,a)) $	Reduce S→a
$(S	, (a,a)) $	Reduce L→S
$(L	, (a,a)) $	shift
$(L,	(a,a)) $	shift
$(L,(	a,a)) $	shift
$(L,(a	,a)) $	Reduce S→a
$(L,(S	,a)) $	Reduce L→S
$(L,(L	,a)) $	shift
$(L,(L)	)) $	shift
$(L,(L)a	)) $	Reduce S→a
$(L,(L,S	)) $	Reduce L→L,S
$(L,(L	) $	shift
$(L,(L)	) $	Reduce L→(L)
$(L,S)	$	Shift
$(L)	$	Reduce S→(L)
$S	$	Accept

Q. $S \rightarrow abSa / aaAb$

$A \rightarrow baAb / b$

Check whether the given grammar is LL(1) or not.

Sol. Grammar is contain left factor.

So, $S \rightarrow aS' / b$

$S' \rightarrow bSa / aAb$

$A \rightarrow bA'$

$A' \rightarrow aAb / \epsilon$

Aniket
Date
Page

FIRST (S) = { a, b }
FIRST (S') = { a, b }
FIRST (A) = { b }
FIRST (A') = { a, ∈ }

FOLLOW(S) = { $, a }
FOLLOW(S') = { $, a }
FOLLOW(A) = { b }
FOLLOW(A') = { b }

	a	b	$
S	S→aS'	S→b	S.∈
S'	S'→aAb	S'→bSa	S.∈
A	S.∈	A→bA'	S.∈
A'	A'→aAb	A'→∈	S.∈

So, yes, it is LL(1) parser.

LR parser

This is the most efficient method of bottom up parsing which can be used to parse the large class of CFG. This method is also called LR (k) parser. Here, left stands for left to right scanning & R stands for right most derivation in reverse and k is the number of input symbol.

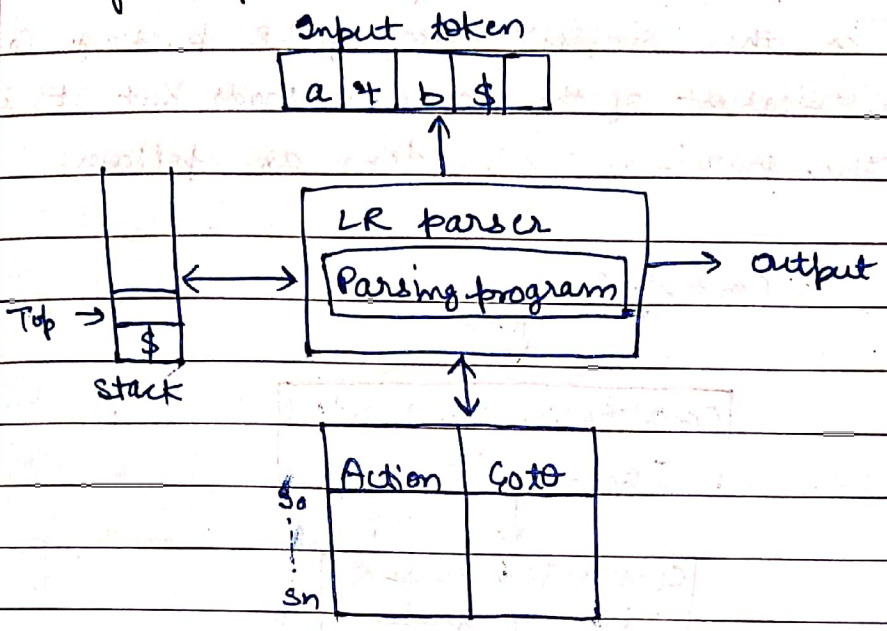
when k is omitted it assume to be one.

Properties: 1) LR parser can be constructed to recognize most of the programming languages for which CFG can be written.
2) The class of grammar that can be parsed by LR parser is the superset of class of grammar that can be parsed using predictive parsers.
3) LR parser works using non-backtracking shift reduce technique yet it is efficient.

Note: LR parser detects synthetical errors very efficiently

Aniket
Date
Page

Structure of LR parser:



It consist of input buffer for string the input string. A stack for storing the grammar symbols output & the parsing table consists of two parts namely action & goto. There is one parsing program which is actually a driving program and read the input symbol one at a time from the input buffer.

The driving program works on the following principle:

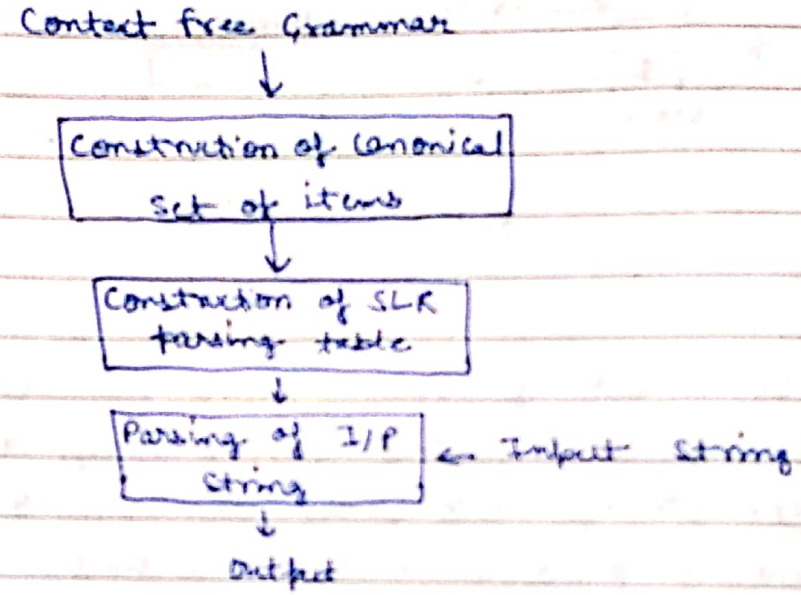
1) It initialize the stack with start symbol and invokes scanner (Lexical analyzer) to get next token.

2) It determine S_j the state currently on the top of the stack & a_i is the current input symbol.

3) It construct the parsing table for the action $[S_j, a_i]$ which can have one of the four values:
   a) S_j is shift state
   b) r_j reduce by rule j
   c) Accept : successfull parsing is done
   d) error : indicate syntactical error

(1) Simple LR parser or LR(0) parser:

It is the simplest form of LR parsing called SLR parser. It is the weakest of the three methods but it is easiest to implement. The parsing can be done as follows:



A grammar for which SLR parser can be constructed is called SLR grammar.

Construction of SLR parsing table:

In the structure of SLR parser, there are two parts of SLR parsing table that are action & goto. By considering basic parsing action such as shift, reduce, accept and error, we will fill up the action table. The goto table can be filled up using goto function.

Input: An augumented grammar
Output: SLR parsing table
Algorithm: 1) Initially construct set of items $C = [I_0, I_1, \ldots, I_n]$ where C is the collection of set of LR(0) items for the input grammar.

2) The parsing action are based on each item I_i, the actions are given as below:

a) If $A \to \alpha . a\beta$ is in I_i and goto $(I_i, a) = I_j$ then set action $[i,a]$ as shift of j. but 'a' must be terminal symbol.

b) If there is a rule $A \to \alpha$ is in I_i then set action $[i,a]$ to reduce $A \to \alpha$ for all symbols 'a' where a $\in$ FOLLOW(A).

c) If $S' \to S$ is in I_i then the entry in the action table $[i, \$]$ = ACCEPT

3) The goto part of SLR table can be filled as S. The goto transition of state I is considered for non-terminal only if goto $[I_i, A] = I_j$ then goto $[I_i, A] = j$

4) All the entries are not defined by rule 2 are considered to be error.

Q. Construction of SLR parsing table for the following set of grammar:

$E \to E+T$ —— (1)

$E \to T$ —— (2)

$T \to T*F$ —— (3)

$T \to F$ —— (4)

$F \to (E)$ —— (5)

$F \to id$ —— (6)

Sol. Augumented grammar:

I_0: $E' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Aniket
Date
Page

goto (I_0, E)
I_1: $E' \to E.$
$E \to E. + T$

goto (I_0, T)
I_2: $E \to T.$
$T \to T. * F$

goto (I_0, F)
I_3: $T \to F.$

goto (I_0, ()
I_4: $F \to (.E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_0, id)
I_5: $F \to id.$

goto (I_1, +)
I_6: $E \to E + .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

goto (I_2, *)
I_7: $T \to T * .F$
$F \to .(E)$
$F \to .id$

goto (I_4, E)
I_8: $F \to (E.)$
$E \to E. + T$

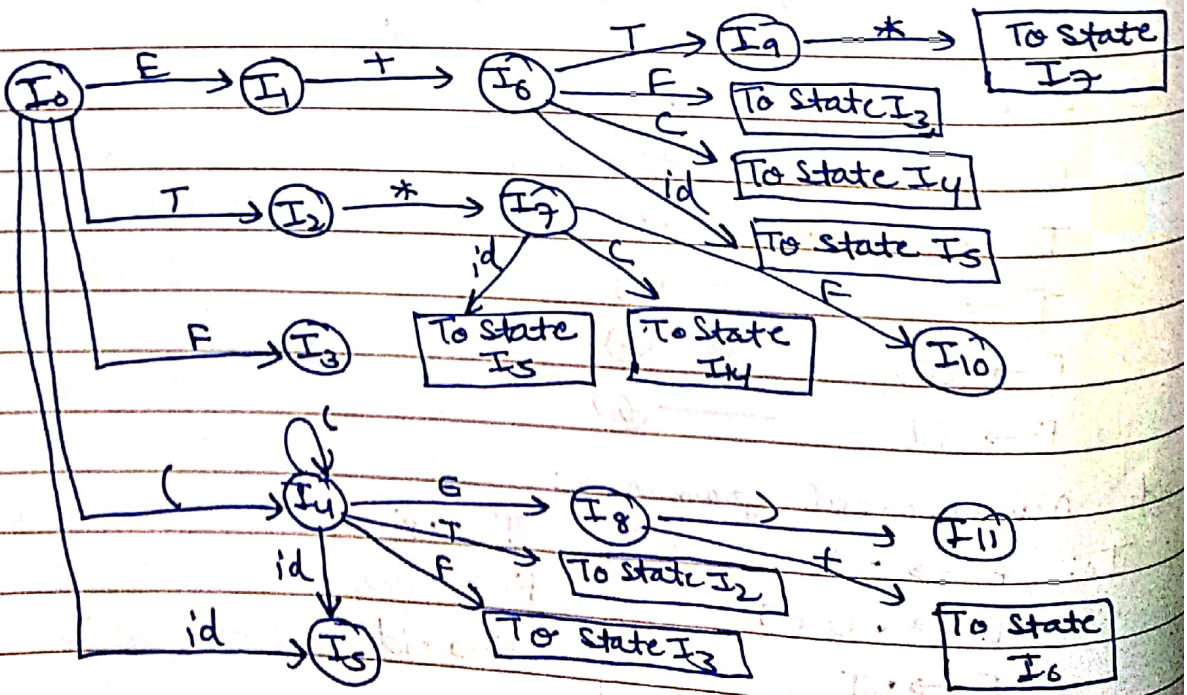
goto (I_6, T)
I_9: $E \to E + T.$
$T \to T. * F$

goto (I_7, F)
I_{10}: $T \to T * F.$

goto (I_8,))
I_{11}: $F \to (E)$

FOLLOW
$E = \{+, \$,)\}$
$F = \{+, *, \$,)\}$
$T = \{+, *, \$,)\}$

DFA:



State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	S_5	Error	Error	Error	Error	Error	1	2	3
1	Error	S_6	Error	Error	Error	Accept	Error	Error	Error
2	Error	r_2	S_7	Error	r_2	Error	Error	Error	Error
3	Error	r_1	r_4	Error	r_4	Error	Error	Error	Error
4	S_5	Error	Error	S_4	Error	Error	8	2	3
5	Error	r_5	r_5	Error	r_5	r_5	Error	Error	Error
6	S_5	Error	Error	S_4	Error	Error	Error	9	3
7	Error	Error	Error	S_4	Error	Error	Error	Error	10
8	Error	S_6	Error	Error	S_{11}	Error	Error	Error	Error
9	Error	r_1	S_7	Error	r_1	Error	Error	Error	Error
10	Error	r_3	r_3	Error	r_3	Error	Error	Error	Error
11	Error	r_5	r_5	Error	r_5	Error	Error	Error	Error

Q. $E \to E+T/T$ | $E \to E+T$ | $T \to F$ | $F \to b$

$T \to TF/F$ | $E \to T$ | $F \to F*$

$F \to F*/a/b$ | $T \to TF$ | $F \to a$

Construct the SLR parsing table.

Sol. Augumented grammar: $E \to E+T$

I_0: $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .TF$

$T \to .F$

$F \to .F*$

$F \to .a$

$F \to .b$

goto (I_0, E)

I_1: $E' \to E.$ ✓

goto (I_0, T)

I_2: $E \to T.$ ✓

$T \to T.F$

$F \to .F*$

$F \to .a$

$F \to .b$

goto (I_0, F)

I_3: $T \to F.$ ✓

$F \to F.*$

goto (I_0, a)

I_4: $F \to a.$ ✓

goto (I_0, b)

I_5: $F \to b.$ ✓

goto $(I_1, +)$

I_6: $E \to E+.T$

$T \to .TF$

$T \to .F$

$F \to .F*$

$F \to .a$

$F \to .b$

Aniket
Date
Page

goto (I_2, F)

I_7: T → TF. ✓

F → F.*

goto (I_3, *)

I_8: F → F*. ✓

goto (I_6, T)

I_9: E → E+T.

T → T.F

F → .F*

F → .a

F → .b

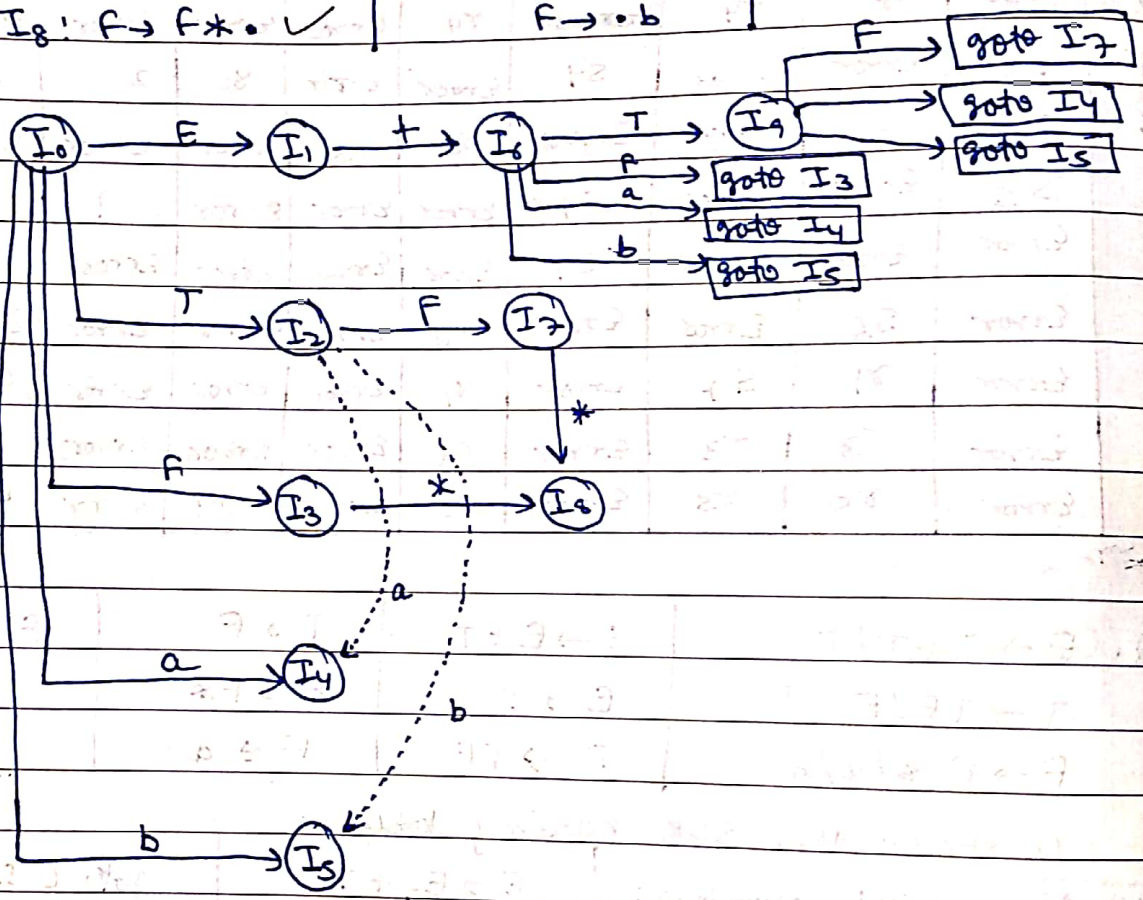
FOLLOW

E = {$, +}

F = {$, *, a, b, +}

T = {$, +, a, b}

DFA:



State	Action					Goto		
	+	*	a	b	$	E	T	F
0			S_4	S_5		1	2	3
1	S_6				Accept			
2	r_2		S_4	S_5	r_2			7
3	r_4	S_8	r_4	r_4	r_4			
4	r_6	r_6	r_6	r_6	r_6			
5	r_7	r_7	r_7	r_7	r_7			
6			S_4	S_5			9	3
7	r_3	S_8	r_3	r_3	r_3			
8	r_5	r_5	r_5	r_5	r_5			
9	r_1		S_4	S_5	r_1			7

CLR or LL(1) parsing

Q. Construct the CLR parsing table of the following grammar:

$S \to CC$ — (1)

$C \to aC$ — (2)

$C \to d$ — (3)

Sol. Augmented grammar:

I_0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

goto (I_0, S)

I_1: $S' \to S., \$$ ✓

goto (I_0, C)

I_2: $S \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

goto (I_0, a)

I_3: $C \to a.C, a/d$
$C \to .aC, a/d$
$C \to .d, a/d$

goto (I_0, d)

I_4: $C \to d., a/d$ ✓

goto (I_2, C)

I_5: $S \to CC., \$$ ✓

goto (I_2, a)

I_6: $C \to a.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

goto (I_2, d)

I_7: $C \to d., \$$ ✓

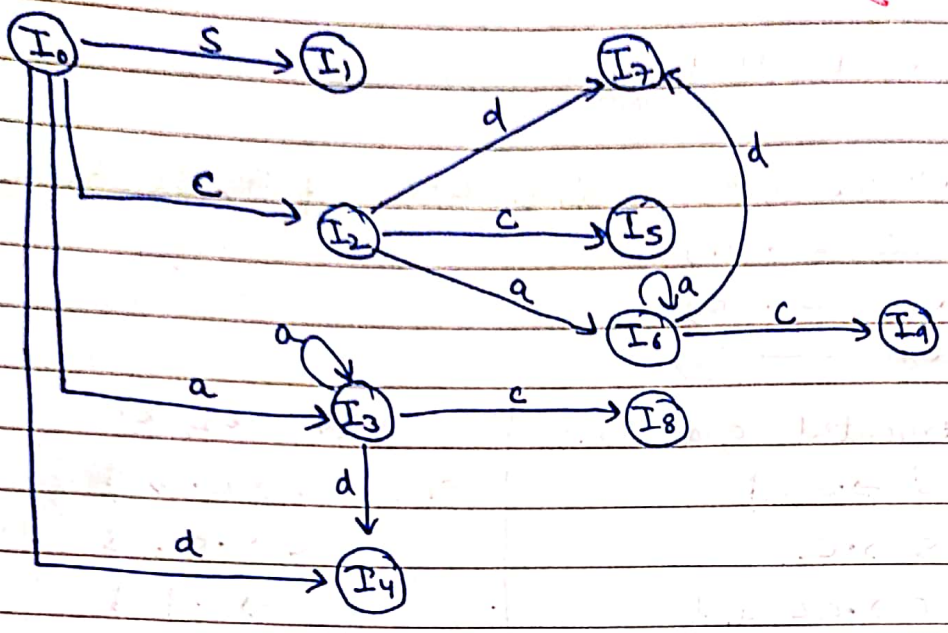
goto (I_3, C)

I_8: $C \to aC., a/d$ ✓

goto (I_6, C)

I_9: $C \to aC., \$$ ✓

	a	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		



LALR parser

Q. Construct the LALR parsing table of the following grammar

$S \to CC$

$C \to aC$

$C \to d$

Sol. Augmented grammar:

I_0: $S \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

goto (I_0, S)
I_1: $S' \to S\cdot, \$$

goto (I_0, C)
I_2: $S \to C\cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

goto (I_0, a)
I_3: $C \to a\cdot C, a/d$
$C \to \cdot aC, a/d$
$C \to \cdot d, a/d$

goto (I_0, a)
I_4: $C \to d\cdot, a/d$

goto (I_2, C)
I_5: $S \to CC\cdot, \$$

goto (I_2, a)
I_6: $C \to a\cdot C, \$$
$C \to \cdot aC, \$$
$C \to \cdot d, \$$

goto (I_2, d)
I_7: $C \to d\cdot, \$$

goto (I_3, C)
I_8: $C \to aC\cdot, a/d$

goto (I_6, C)
$I_9 \to C \to aC\cdot, \$$

Aniket
Date
Page

After reducing the same stages

I_{36}: $C \to a \cdot C, a/d/\$$

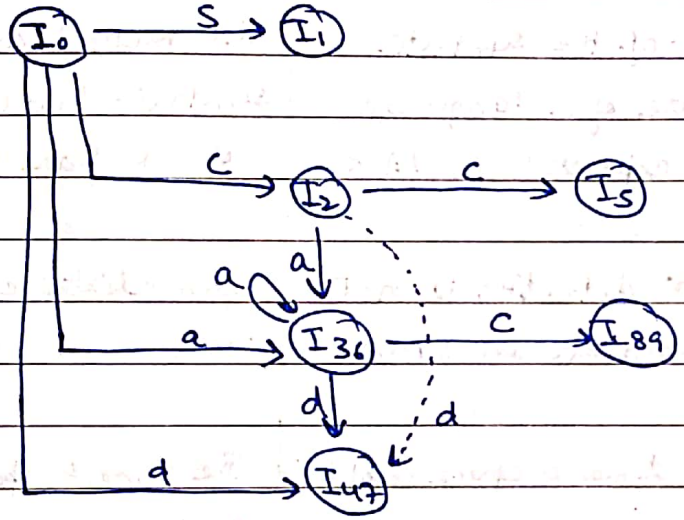
$C \to \cdot aC, a/d/\$$

$C \to \cdot d, a/d/\$$

I_{47}: $C \to d \cdot, a/d/\$$

I_{89}: $C \to aC \cdot, a/d/\$$

DFA:



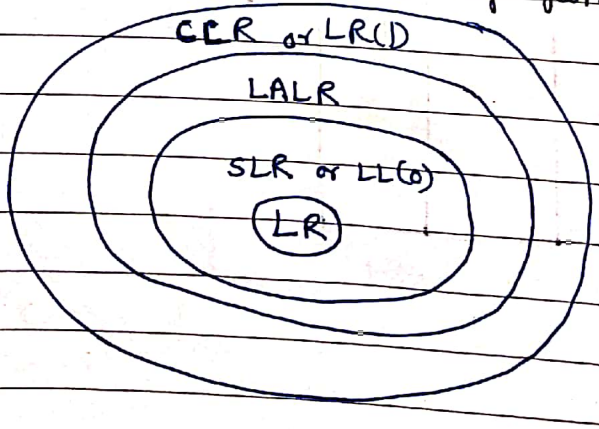
State	Action			goto	
	a	d	$	S	C
0	S_{36}	S_{47}		1	2
1			ACCEPT		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Aniket
Date
Page

Comparison of LR parser

SLR	LALR	CLR
1) SLR is smallest in size.	LALR & SLR have the same size.	CLR parser is largest in size.
2) It is an easiest method based on FOLLOW().	This method is applicable to wider class than SLR	This method is most powerful than SLR & LALR
3) This method exposes less syntactic features than all of LR parser.	Most of the syntactic feature of a language are expressed in LALR.	This method expresses less syntatic features than all of LR parser.
4) Error detection is not immediate in SLR	Error detection is not immediate in LALR.	Immediate error detection is done by CLR parser.
5) It requires less time & space complexity.	The time & space complexity is more in LALR but efficient method exist for constructing LALR parser directly.	The time & space complexity is more for CLR parser.

Graphical representation of LR parsing family:



Aniket
Date
Page

Handle: of a string is the sub-string that match the right side of the production & whose reduction to non-terminal on left side of production represents one step for right most derivation.

In other words, a handle of a right sentential form from Y is a production $A \rightarrow B$ & position of Y where string B may be obtained. This B further can be reduced by A.

Ex: $E \rightarrow E+E$ Here, $E+E$ is handle

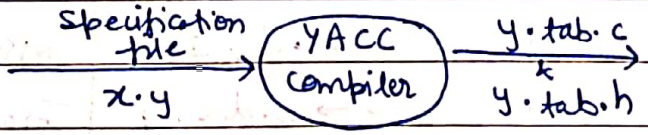
$E \rightarrow E+E*E$ Here, $E*E$ is handle.

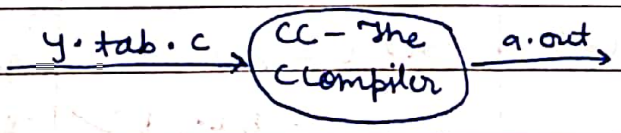
An automatic parser generator:

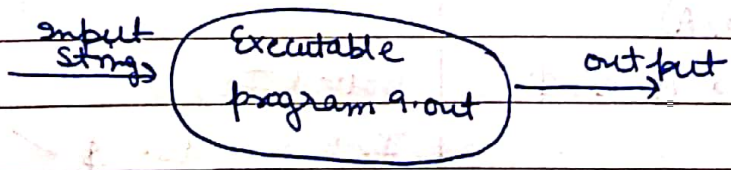
There is a need for the automation of the process in order to achieve the efficiency in parsing the input. Certain automation tools for parser generation are available.

YACC is one such automation tool for generating the parsing program. YACC stands for yet automatic compiler compiler which is basically the utility available from unix. Basically YACC is LALR parser generator.

The YACC can report conflicts or ambiguity in the form of error messages. One such tool LEX for lexical analyzer. LEX & YACC works together to analyze the program syntactically.







First we write a YACC specification file naming x.y. This file is given to the YACC compiler by UNIX command then it will generate a parser program naming y.tab.c this is basically parser program in C generated automatically. We can also give the command -d option. y.tab.h will store all the token.

Specifically, the generated y.tab.c program will be computed by C compiler and generates the executable a.out file. Then we can test our YACC program with the help of some valid & invalid string.

Q. Construct the LALR parser

$S \to Aa$ — (1)

$S \to bAc$ — (2)

$S \to dc$ — (3)

$S \to bda$ — (4)

$A \to d$ — (5)

Sol: Augmented grammar

I_0: $S' \to .S, \$$
$S \to .Aa, \$$
$S \to .dc, \$$
$S \to .bAc, \$$
$S \to .bda, \$$
$A \to .d, d$

goto (I_0, S)
I_1: $S' \to S., \$$ ✓

goto (I_0, A)
I_2: $S \to A.a, \$$

goto (I_0, d)
I_3: $S \to d.c, \$$
$A \to d., d$ ✓

goto (I_0, b)
I_4: $S \to b.Ac, \$$
$A \to .d, \$$
$S \to b.da, \$$

goto (I_2, a)
I_5: $S \to Aa., \$$ ✓

goto (I_3, c)
I_6: $S \to dc., \$$ ✓

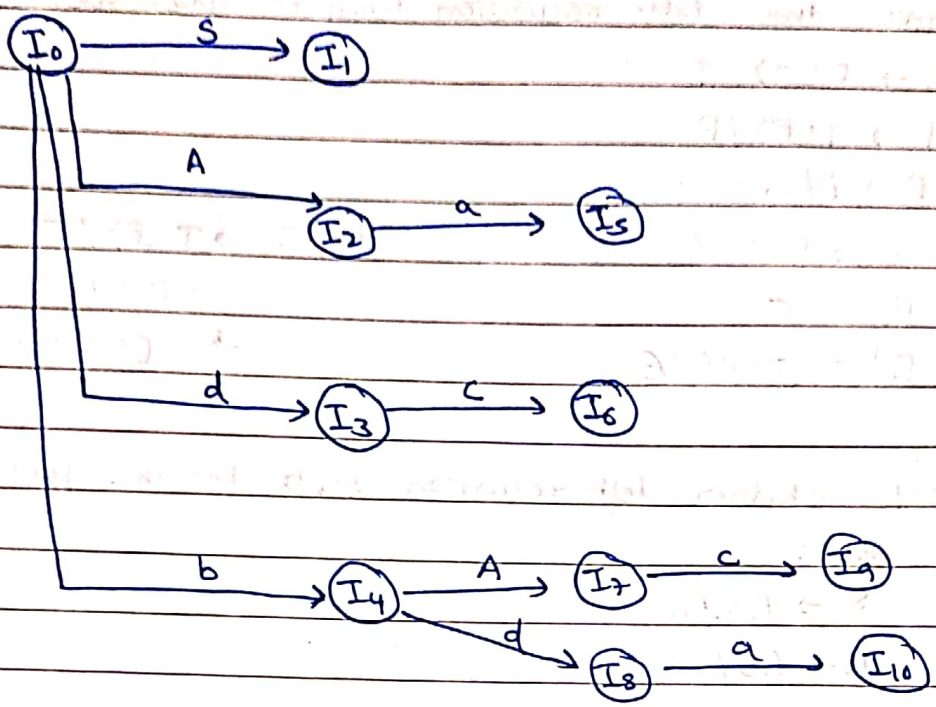
goto (I_4, A)
I_7: $S \to bA.c, \$$

goto (I_7, c)
I_9: $S \to bAc., \$$ ✓

goto (I_8, a)
I_{10}: $S \to bda., \$$ ✓

goto (I_4, d)
I_8: $S \to bd.a, \$$
$A \to d., \$$ ✓

DFA:



State	Action					goto	
	a	b	c	d	$	S	A
0		S_4		S_3		1	2
1					Accept		
2	S_5						
3			S_6	r_5			
4				S_8			7
5					r_1		
6					r_3		
7			S_9				
8	S_{10}				r_5		
9					r_2		
10					r_4		

Aniket
Date
Page

Q. Remove the left recursion from the grammar:

$E \rightarrow E(T) / T$

$T \rightarrow T(F) / F$

$F \rightarrow id$ ✓

Sol. for $E \rightarrow E(T) / T$

$E \rightarrow TE'$

$E' \rightarrow (T)E' / \epsilon$

for $T \rightarrow T(F) / F$

$T \rightarrow FT'$

$T' \rightarrow (F)T' / \epsilon$

Q. Check whether left recursion exists for the following grammar:

$S \rightarrow Aa / b$

$A \rightarrow Ac / Sd / e$

Sol. for $A \rightarrow Ac / Sd / e$

$A \rightarrow eA'$

$A \rightarrow sdA'$

$A' \rightarrow cA' / \epsilon$

~~$S \rightarrow Aa / b$~~

so, $A \rightarrow eA' / sdA'$

$A' \rightarrow cA' / \epsilon$