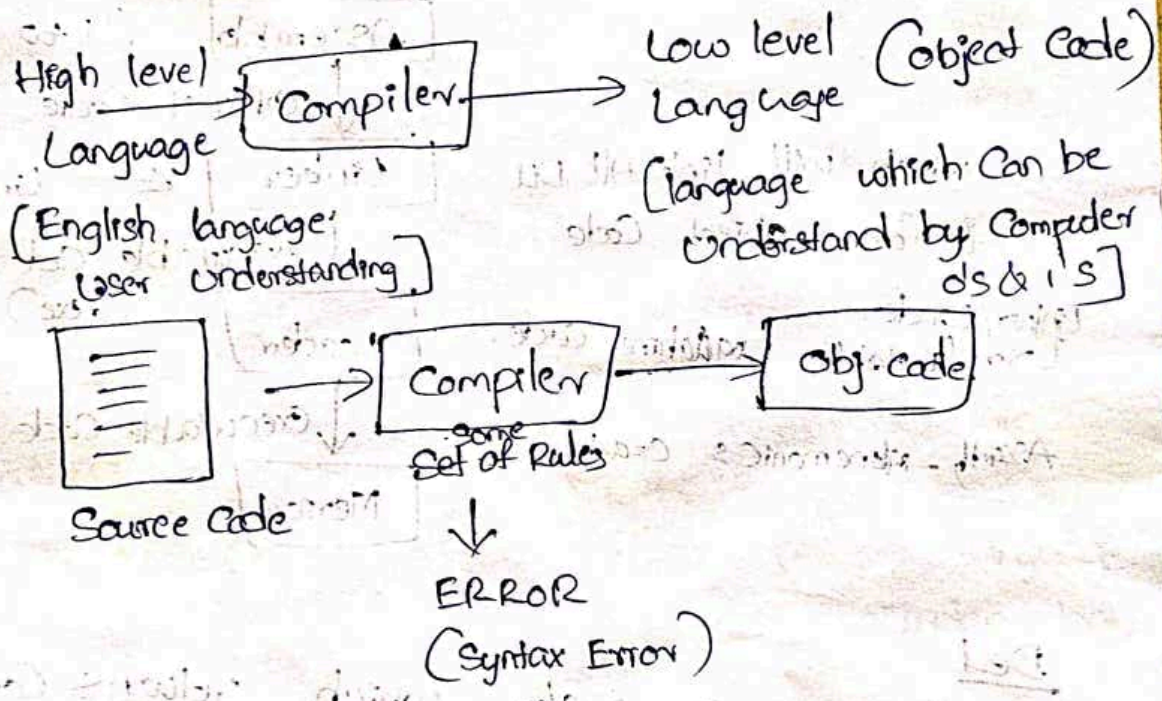
Compiler Design

Introduction to Compiler.

The Compiler is also known as language Translator [It] which will Converts one language to another language.

Compiler → language Translator.



If the instruction is Correct given to the Compiler Some set of rule in the Compiler.

If the instruction is violence the Rules By Compiler The Compiler is sent generate the Errors. these errors is called Syntax Error.

* The Compiler is Total Source Code is Converted to object code.
* Another language also there i.e Interpreter.

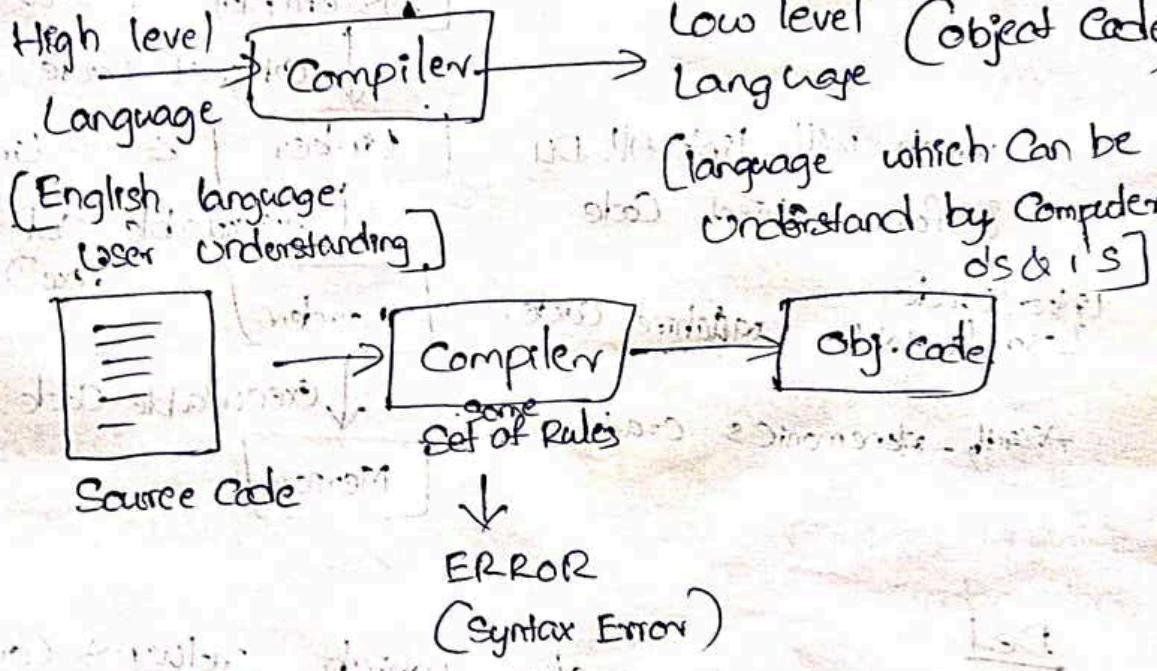
Interpreter is executes the instruction line by line.

Compiler Design

Introduction to Compiler.

The Compiler is also known as language Translator. [It] which will Converts one language to another language.

Compiler → language Translator.



If the instruction is Correct given to the compiler Some set of rule in the compiler.

If the instruction is violence the Rules By Compil The Compiler is sent generate the Errors. these errors is called Syntax Error.

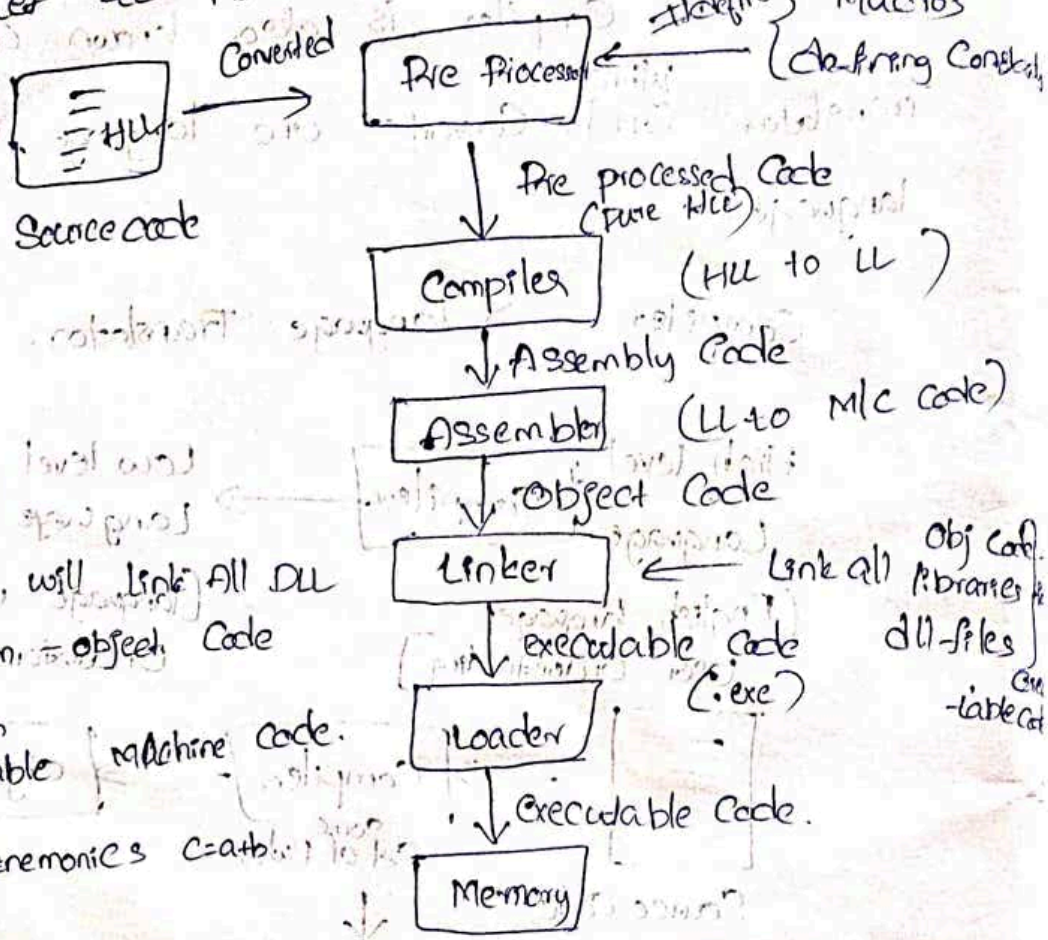
* The Compiler is Total Source Code is Converted to object code.
* Another language also there i.e Interpreter.

Interpreter is execudes the instruction line by line.

Basic Language Processing System

What the exact Compiler will do. by defining a file using Preprocessor Language

Let see how it looks like



Which will link All DLL & form = object Code

Linker/Loader → relocatable

Assembly - mnemonics

Def

A Compiler is a s/w which converts a program written in HLL to Low level language.

Eg① C program {HLL}

↓

② C Compiler translates Prog in to low level lang (Assembly)

↓

③ Assembler translate LL in to MLC code

↓

④ Linker used to Link all parts of Prog together for execution

↓

⑤ The loader loads All them into memory the Prog is executed

A Compiler Converts HLL to LLL

Assembler Converts LLL to MLL

Statements → Mnemonics → binary nos

HLL → Assembly → M/C

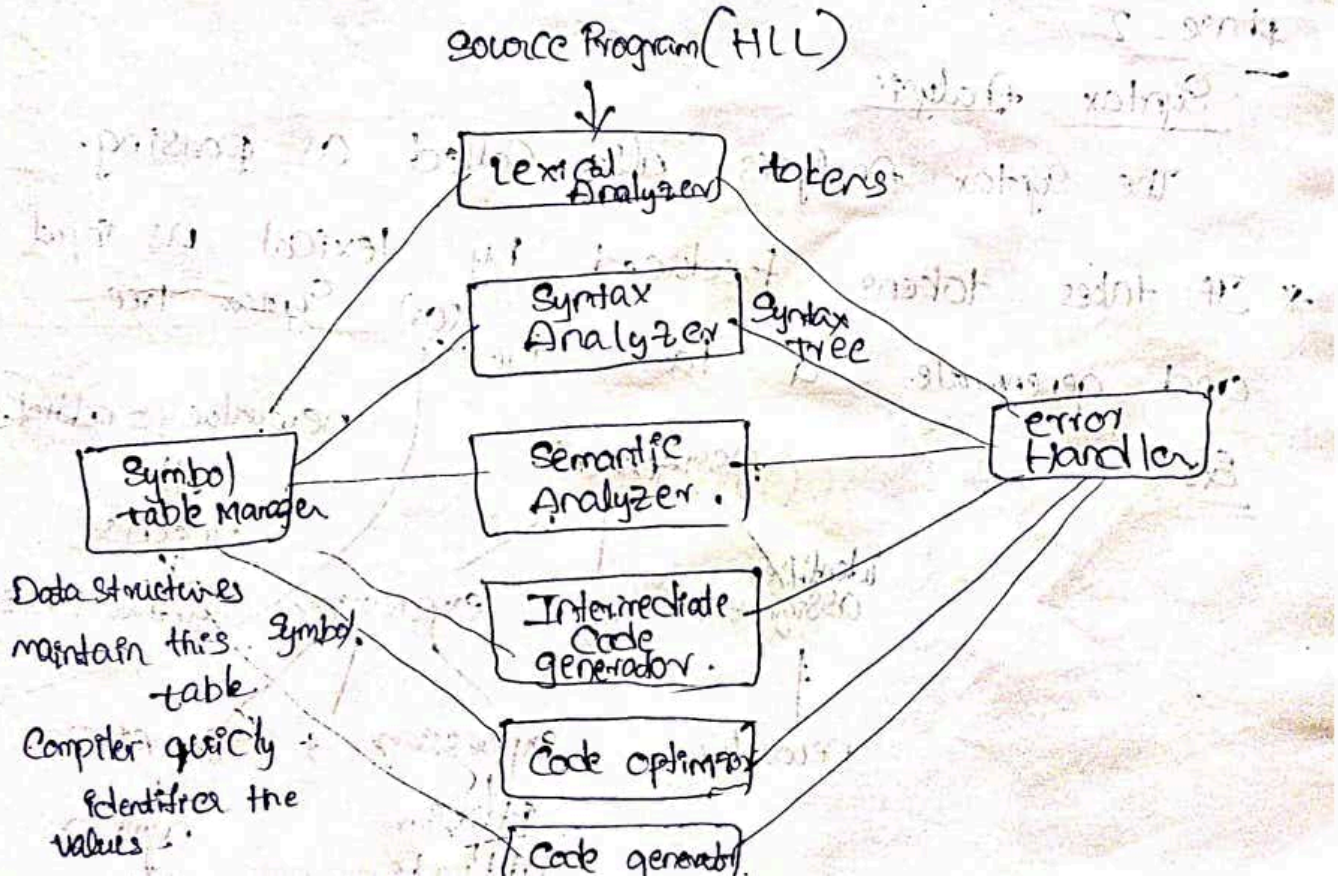
Phases of Compiler

phase is a logically inter-related operations that takes the source program in one representation & produce o/p in another representation.

Compiler divided into two major phases

① Analysis phase :- (It is M/c independent/language dependent)

② Synthesis phase :- (It is M/c dependent/language Independent)



phase - 1

lexical Analysis

It Reads the stream of characters making up the source program & group the characters into meaningful sequences is called lexeme.

→ lexical Analyzer represents these lexems in the form of tokens.

<token-name, attribute-value>

Ex newvalue := oldvalue + 12

Tokens : newvalu identifier

= Assignment Operator

old val identifier.

+ Add operator.

12 Number

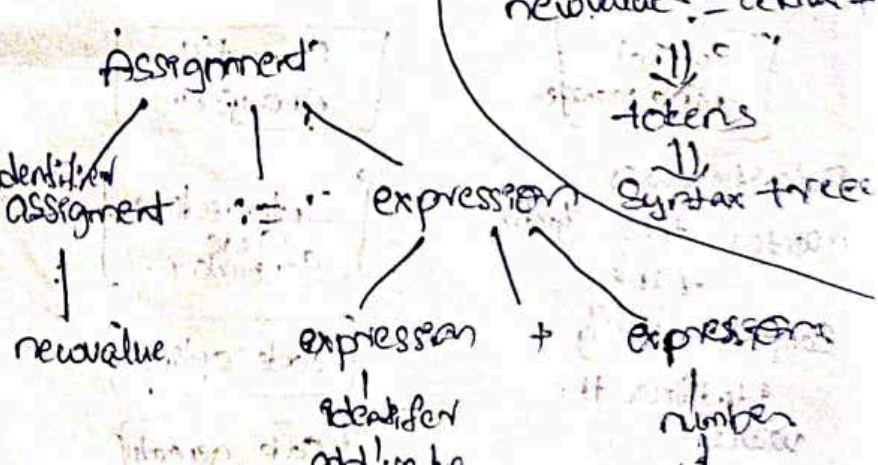
phase - 2

Syntax Analysis

The Syntax Analysis also called as parsing.

* It takes tokens produced by lexical as input and generevade a Parse tree (or) Syntax tree.

Ex:



phase 3:

Semantic Analysis

It checks whether the Parse tree Constructed follows the rules of languages or not.

Eg: assignment of values is b/w Compatable datatypes.

→ also keep track of identifiers their types & expressions

Ex:

```
sum = a + b
int   a;
double sum;
char   b;
```

data types mismatch

```
Semantic record
a: int
sum: double
b: char
```

* It is Syntactically correct but Semantically incorrect.

phase 4

Intermediate Code generation.

It is the representation of final m/c language

Code is produced.

⇒ This phase bridges the Analysis & synthesis phases of translation.

new val := old val + fact * 1

↓

id1 := id2 + id3 * 1

↓ ↘

Temp1 := int real(1); temp2 = id3 * Temp1

Temp 3 := id2 + temp2

id1 := temp3

phase 5 - Code optimization.

It is used to optimize the code.

[It removes unnecessary files. Manage the statements inorder]

The output runs faster & takes less space.

```
Temp 1 = id3 * 1
id1 = id2 + Temp 1
```

newval := oldval + 1

fact * 1 taken as temp1

phase 6 - Code generation.

It translate the intermediate code into sequ

of relocatable m/c code.

Eg: id1 := id2 + id3 * 1

```
MOV  R1, Id3
MUL  R1 * 1
MOV  R2, Id2
ADD  R1, R2
MOV  Id1, R1
```

} Assembly langue. (LLL)

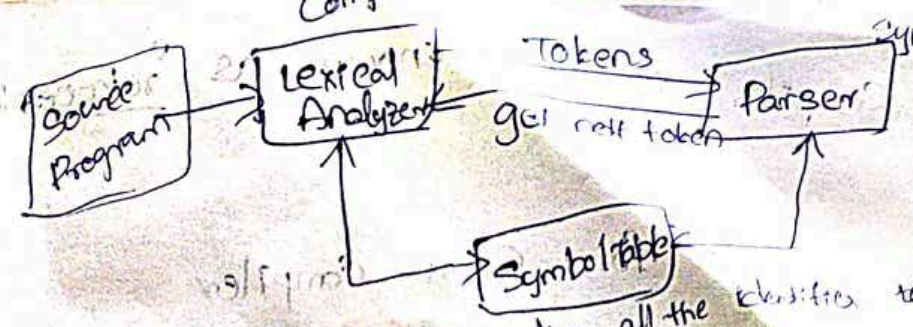
The Compiler converts HLL into LLL

Lexical Analyzer (1st phase):

LA is the first phase of Compiler also called a

linear Analysis (or) Scanning.

In this phase the the stream of characters making up the source program is read from left-to- and grouped into tokens that are sequences of character having collective meaning



* Upon receiving a 'get next token' command from Parser, the lexical analyzer reads the source program until it can identified the next token.
* LA may also perform certain secondary tasks as user interface. One such task is striping out from source program the commands & white spaces in the form of blank, tab and line characters.
* Another is correlating error messages from the compiler with the source code.

Cross Compiler

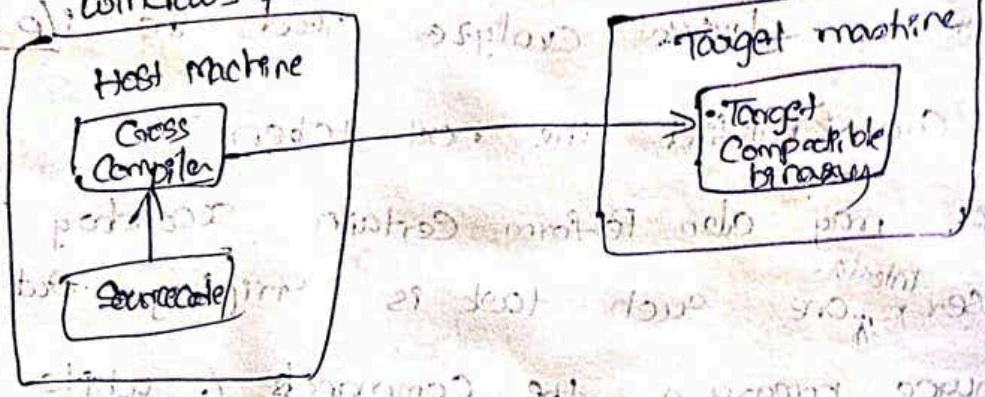
A Cross Compiler is a type of Compiler, that generates machine code targeted to run on a system different the one generating it.

* For example - A Compiler that runs on windows platform also generates code that runs on Linux platform is a cross compiler.
* The process of creating executable code for different machines is also called "retargeting".

The Cross Compiler is also known as "retargetable Compiler"

GNU GCC is example for Cross Compiler

Cross Compiler operation.



With the help of Cross Compiler we can execute our target program in Target machine

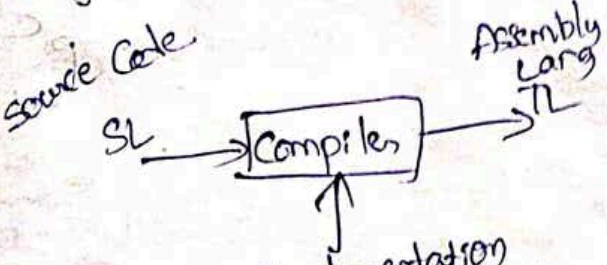
Bootstrapping

* Bootstrapping is widely used in the Compilation development
* It is used to produce a self hosting table compiler
* It is a type of Compiler that can compile its own source code.

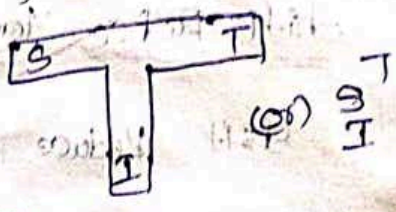
A Compiler can be characterized by three langgs

1) Source language (S)
2) Target language (T)
3) Implementation language (I)

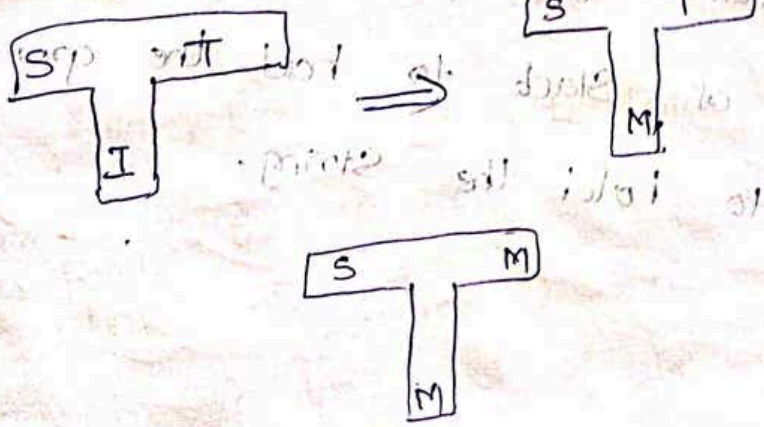
Boot Strapping is the process by which simple langug is used to translate more complicated program, which in turn may handle an even more complicated Program so on.

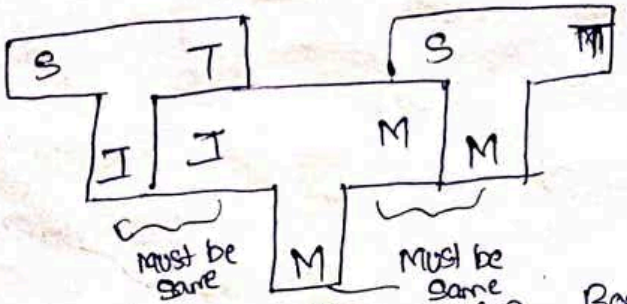


In boot slapping this Compiler diagram is represented as 'T' diagram.



Eg



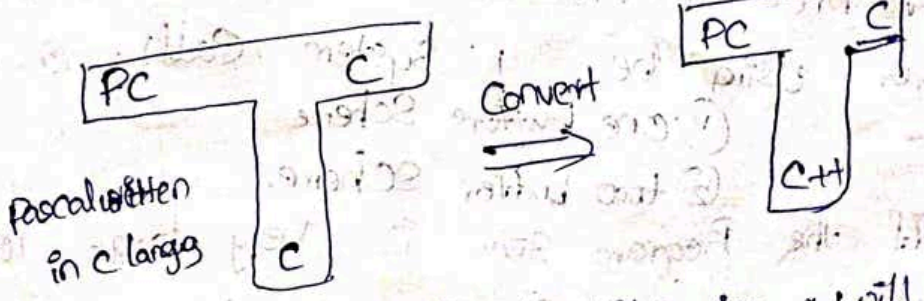


How the Compilation works by using Bootstraping.

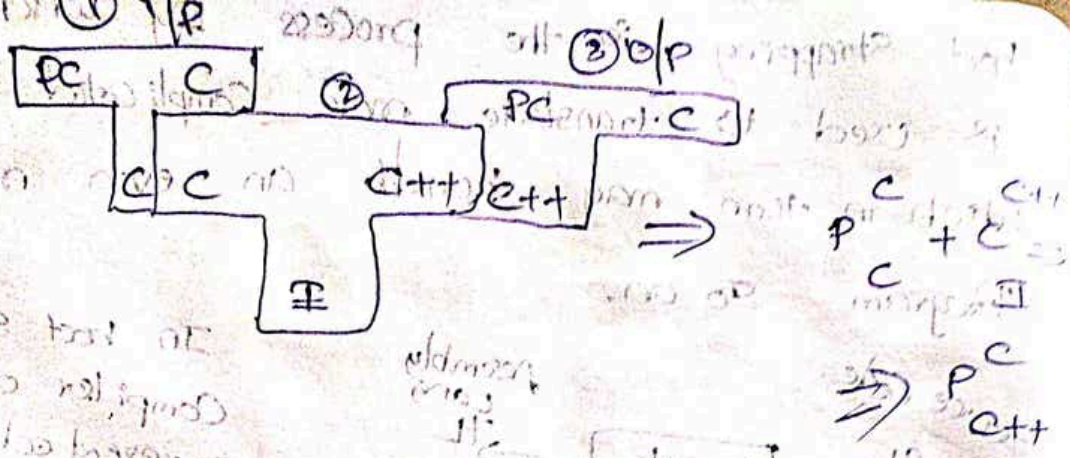
Eg: Pascal translator - C lang

PC - i/p Create a Pascal translator in C++

C - o/p



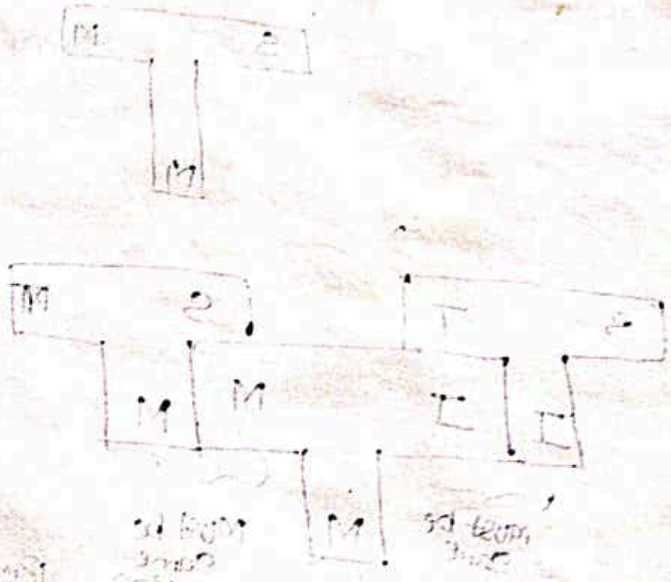
With the help of Boot Stranping we will Convert



Shift Reduce Parsing

Shift Reduce Parsing is a process of reducing a string to the start symbol of a grammer.

It uses of stack to hold the grammer and an input to hold the string.



Buffer

A block of characters to be read into the Buffer using the 1 system calls.

① one buffers scheme

② two buffer scheme

If the Program size is very large 1000 system calls are need so this is

Input Buffering

Lexical Analysis has to access secondary memory each time to identify tokens. It is time consuming and costly. So the input strings are stored into a buffer and then scanned by Lexical Analysis.

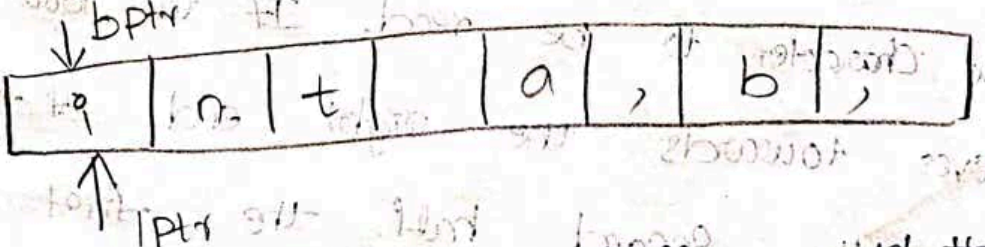
Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens.

Begin Pointer (bPtr) - It Points to the beginning of the string to be read.

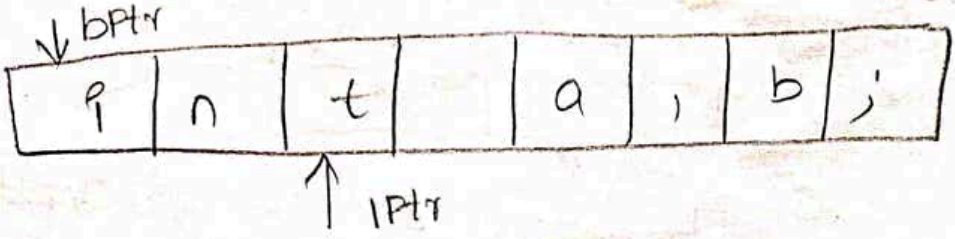
Look ahead pointer (lPtr) - It moves ahead to search for the end of the token.

Example - For Statement: int a, b;

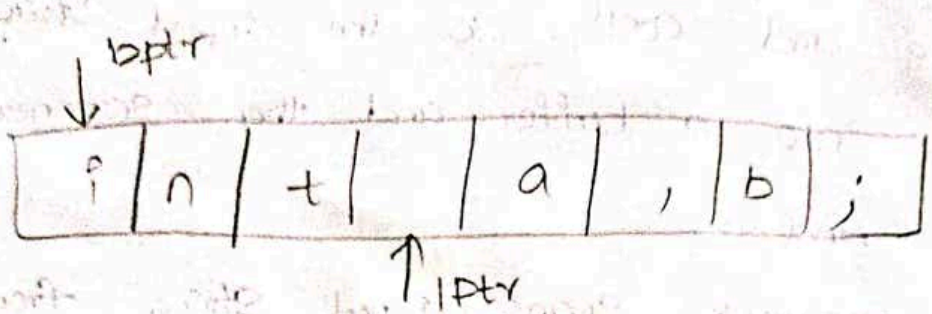
* Both pointers start at the beginning of the string, which is stored in the buffer.



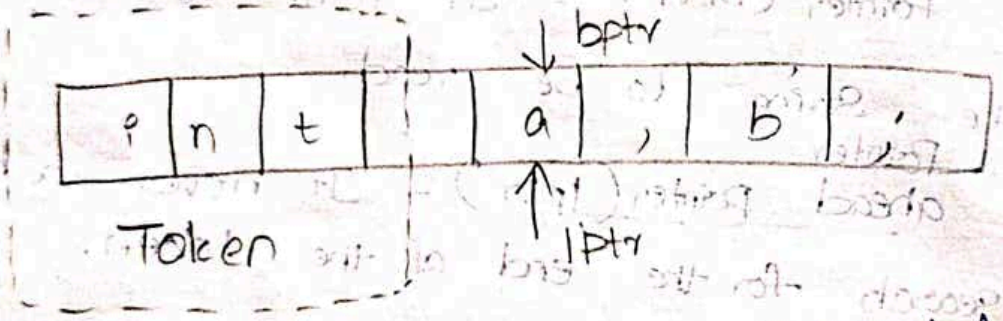
* Look Ahead Pointer scans buffer until the token is found.



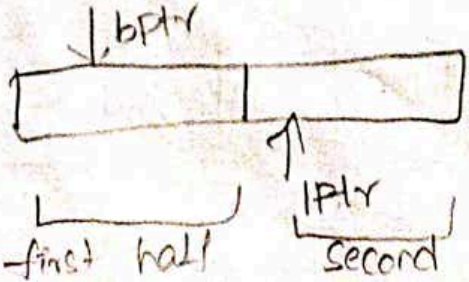
* The character ("blank space") beyond the token ("int") have to be examined before the token ("int") will be determined.



* After processing token ("int") both pointers will set to the next token ('a'), & this process will be repeated for the whole program.

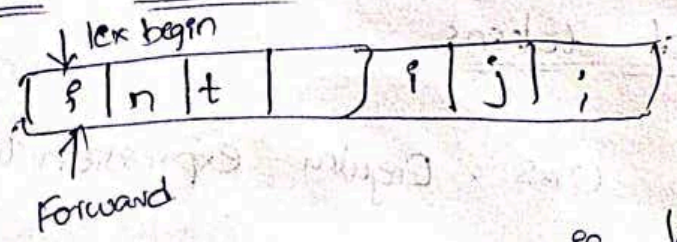


A buffer can be divided into two halves. If the look Ahead pointer moves towards halfway in first Half, the second half is filled with new character to be read. If the look Ahead pointer moves towards the right end of the second half. the first half will be filled with new characters and it goes



Input Buffering

① One buffer Scheme



i/p character is too

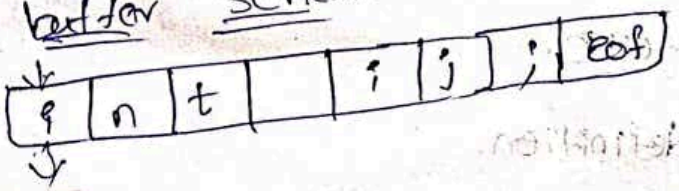
If the i/p string size is bager then in order to store large data is imposible. i/p string 500 char

The buffer has to be overwritten.

Over come this Problem we can move two buffer.

scheme.

② two buffer scheme



eof end of the file character.

After the 1st buffer ~~filled~~ completed filled or not we can identify by using eof.

Specification of tokens

In order to use, Regular expression to specify tokens RE generates Regular language, language has set of Rules.

1. Alphabets
2. About string, languages.
3. Operations on language
4. Regular expression
5. Regular definition.

Alphabet : A set of symbols & we denote Alphabet with the help of Σ

$\Sigma = \{0, 1\}$ It is a binary alphabet.

$\Sigma = \{a, b, \ldots\}$ It is a set of lowercase letters

It is defined as

String : A finite set of symbols generated from Σ

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

a, b, ab, ba, aab, ------ we derived in number of strings

1. length of string
2. empty string
3. Prefix of a string
4. Proper Prefix
5. Suffix of a String
6. Proper Suffix of a String
7. Substring
8. Proper substring
9. Concatination of two String.

Length of String: The total No. of characters present in the string

$S = 1100$

$|S| = 4$

Empty String: If the length of String is zero It is empty

$\epsilon = 0$

Prefix of a String: It is any No. of leading symbols in the string.

Let, $S = abc$

Prefix - ϵ, abc, a, ab

$\epsilon . abc$

$abc . \epsilon$

$\epsilon . abc . \epsilon$

Proper Prefix String:- expert ϵ, abc are remaing all

a, ab

Sufix : It is defined any no. of trailing symbols in the string.

Let $S = abc$

Suffix - ϵ, abc, c, bc

Proper Suffix : except ϵ, abc. remaing all

c, bc

Substring : It is obtained by deleting Prefix & Suffix from the String.

Let $S =$ banana.

Substring :- ϵ, banana., nan, anan, nang

Proper Substring :- nan, anan, nang

Concatination of two Strings:- Combining two String.

Let $x = abc$

$y = de$

$xy = abcde$

$yx = deabc$

Languages

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

$L = \{a, b, ab, ba, abb, \dots\}$

Union, Concatination, Clean closure, Positive closure

① Union of two languages : $L \cup M = \{s \mid s$ is in L or s is in $M\}$

$L = \{0, 1\}$ $\quad M = \{00, 11\}$

$L \cup M = \{0, 1, 00, 11\}$

② Intersection of two languages.

$LM = \{xy \mid x$ is in L and y is in $M\}$

$L = \{0, 1\}$ $\quad M = \{00, 11\}$

$LM = \{000, 011, 100, 111\}$

③ Clean Closure :- It is denoted by * symbol

set of string which include empty string

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

$L^0 \cup L^1 \cup L^2 \dots$

$\Sigma = \{a\}$

$\Sigma^* = \epsilon, a, aa$

④ Positive Closure : It is denoted by (L^+)

A set of strings except Null strings.

$$L^* = \bigcup_{i=1}^{\infty} L^i \quad L^1 \cup L^2 \dots$$

$\Sigma^+ = a, aa$

Regular Expressions

A RE over Σ (Sigma) can be defined as

(Pie)
1. ϕ is a RE for empty set.
2. ϵ is a RE for Noll string
3. If a is a symbol in Σ then a is RE
4. If R & S are two RE then
   - (a) Union of two RE is a RE
   - (b) Concatination of two RE is a RE
   - (c) Clean closure of any RE is a RE

Theese there are known as Properties of Comporent of RE

Algebraic identity of RE

Rules

1. $R+S = S+R$
2. $(R+S)+\epsilon = R+(S+T)$
3. $(RS)T = R(ST)$
4. $(R+S)T = RT+ST = (S+R) = ST+RT$
5. $\epsilon \cdot R = R \cdot \epsilon = R$
6. $R^* = (\epsilon + R)^*$
7. $R^{**} = R^*$

Meta Chaoaders are Patterns of RE

Meta character	Description.
x	It matches with character x
.	It matches with any character except new line.
R^*	zero or more occurences of R
$R+$	one or more occurrences of R.

R? 0 or 1 occurrence of R

R^ It matches with beginning of the line.

R$ It matches with end of the line.

R1/R2

[abc] a, b, c

Regular Definition

It is a name which is given to the RE and we can use that name line.

d1 → R1 [Produces]

d2 → R2

d3 → R3

di → symbol Not a part of Alphabet

ri → Regular expression.

Regular expression for identifiers.

Pascal language start with any identifier or k

letter → a|b|c| ... |z|A|B|C| ... |Z

digit → 0|1|2| ... |9

id → letter (letter (digit)*

id → [a ... z A ... Z] [a ... z A-Z 0 ... 9]*

Eg: RE for digits. (integer no, floating point no)

digits → 0|1| ... |9

digits → digit (digit)*

number → digits (. digits)? (E[+-]? digits)

123 → integers

123.45 → float

12.345E23

12.345e-23

Examples of Regular Expression

1) write RE for the language accepting all strings containing any numbers of a's and b's.

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

This will give the set as $L = \{\varepsilon, a, aa, b, bb, ab, ba, aba \ldots\}$

any combination of a & b

the $(a+b)^*$ shows any combination which a & b.

even null string also

② write RE for the language accepting all the string which are starting with 1 & ending with 0 over $\Sigma = \{0, 1\}$

sol The first symbol is 1 $L = 10, 100, 1000$

last symbol is 0

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

3. write RE for the lang starting with a but not having consecutive b's

sol $L = \{a, aba, aab, aaa, abab, \ldots\}$

$RE = \{a, ab\}^*$ b

4. write the RE for lang over $\Sigma = \{0\}$ having even length of the string.

$L = \{\in, 00, 0000, 000000, \ldots\}$,

$RE = (0.0)^*$

Finite Automata (FA)

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.

FA representations

* Graphical (Transition Diagrams or Transition Table)
* Tabular (Transition table)
* Mathematical (Transition function or Mapping function)

Formal Definition of Finite Automata

A finite automata is a 5-tuples; they are

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

Q - states

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

Q: is a finite set of; called the states.

sigma Σ: is a finite set called the alphabets.

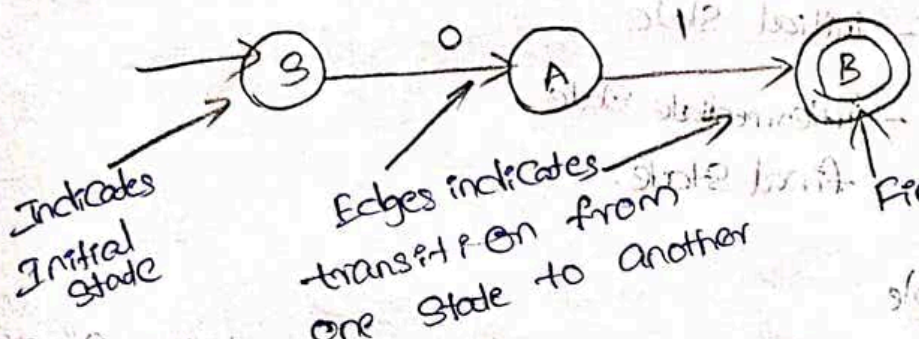
Delta transition function δ: $Q \times \Sigma \rightarrow Q$ is the transition function.

$q_0 \in Q$ is the stard state also called initial

$F \subseteq Q$ is the set of accept states, also called final state.

Transition Diagrams (Transition graph)

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



TD indicates the actions that take place when a LA is called by the parser to get the next token.

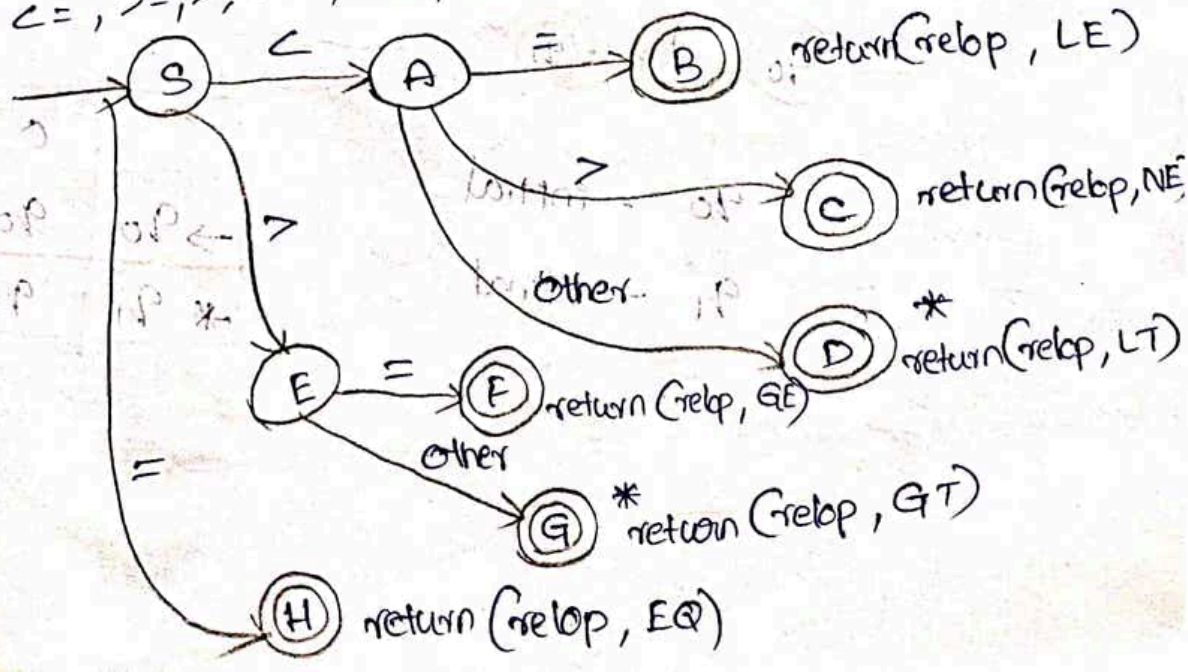
In this example will understand how to convert from RE patterns to TD's. TD's have the collection of circles called STATES.

(S, A and B are different states.

start state = {S}; indicated by → at the start.

Final state = {B}; indicated by ◎

Let us write TD that recognizes the lexemes matching. the tokens <=, >=, >, <, =, and <>



Eg

$\{0, 1\}$ are inputs

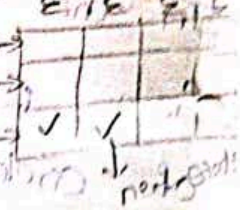
q_0 – initial state

q_1 – intermediate state

q_2 – final state.

Transition Table

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



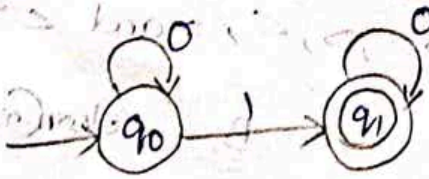
→ Rows corresponds to states

→ Column corresponds to input symbols

→ Entries corresponds to next states.

→ The start state is marked with an arrow (→)

→ The accept state are marked with star (*)



q_0 – initial

q_1 – final

	0	1
→ q_0	q_0	q_1
* q_1	q_1	q_1

Transition function

The mapping functions or transition functions denoted by δ. Two parameters are passed to this transition function

(i) Current State.
(ii) Input symbol

Transition function returns a state which can be called as next state.

$$\delta(\text{Current\_state}, \text{Current\_input\_symbol}) = \text{next state}$$

$$Q \times \Sigma \rightarrow Q$$

Eg:

$$\delta(q_0, a) = q_1$$

$$\delta(q_0, 1) = q_1$$

Converting RE to FA

To convert the RE to FA we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below

Step1 - Design a transition diagram for given regular expression, using NFA with ε moves.

Step2 - Convert this NFA with ε to NFA without ε

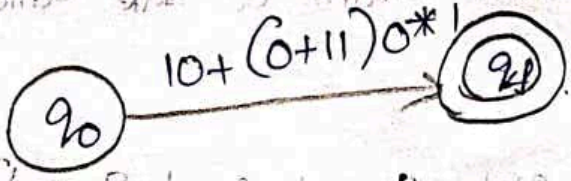
Step3 - Convert the obtained NFA to equivalent DFA

Design a FA from given regular expression

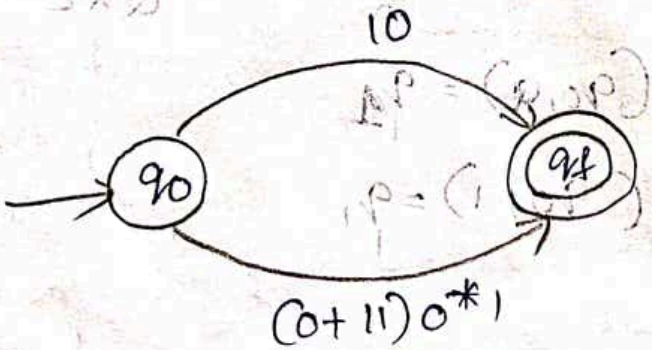
$10+(0+11)0^*1$

Solution: First we will construct the transition diagram for a given regular expression.

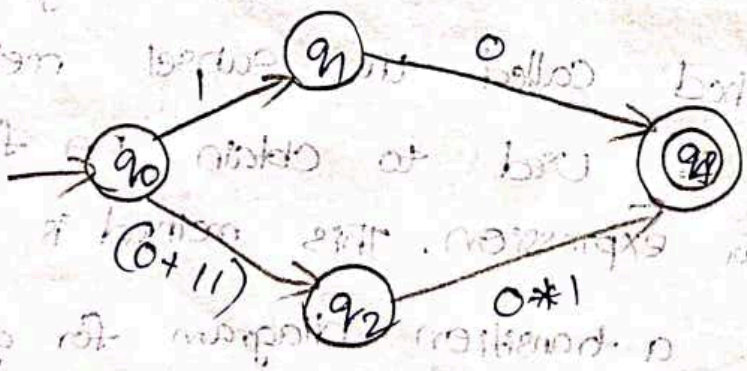
Step1



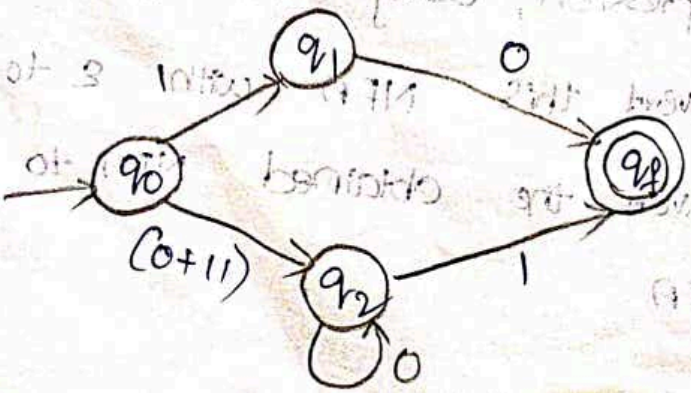
Step2



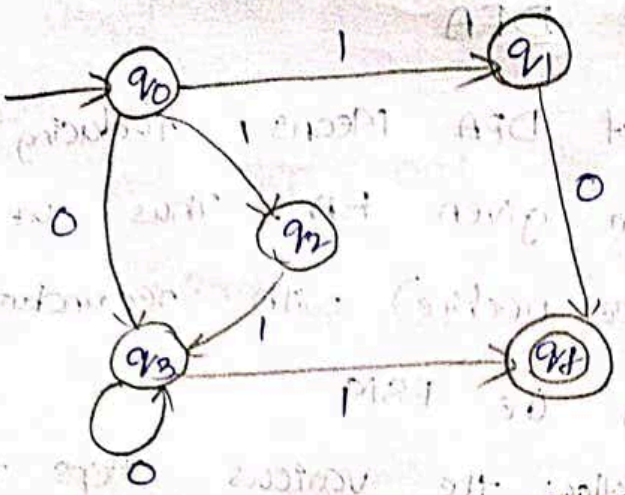
Step3



Step4



Steps



Now we have got NFA without ε. Now we will convert it into required DFA for that we will first write a transition table for this NFA

State	0	1
$\Rightarrow q_0$	q_3	$\{q_1, q_2\}$
q_1	q_f	ϕ
q_2	ϕ	q_3
q_3	q_3	q_f
$*q_f$	ϕ	ϕ

The equivalent DFA will be

State	0	1
$\rightarrow [q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_1]$	$[q_f]$	ϕ
$[q_2]$	ϕ	$[q_3]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_f]$
$*[q_f]$	ϕ	ϕ

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus we get the FSM (finite state machine) with redundant states after minimizing the FSM

we have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA

Step 2: Draw the transition table for all pairs of states.

Step 3: Now split the transition table into two Tables T1 and T2. T1 contains all final states and T2 contains non-final states.

Step 4: Find similar rows from T1 such that.

1. $\delta(q, a) = p$
2. $\delta(r, a) = p$

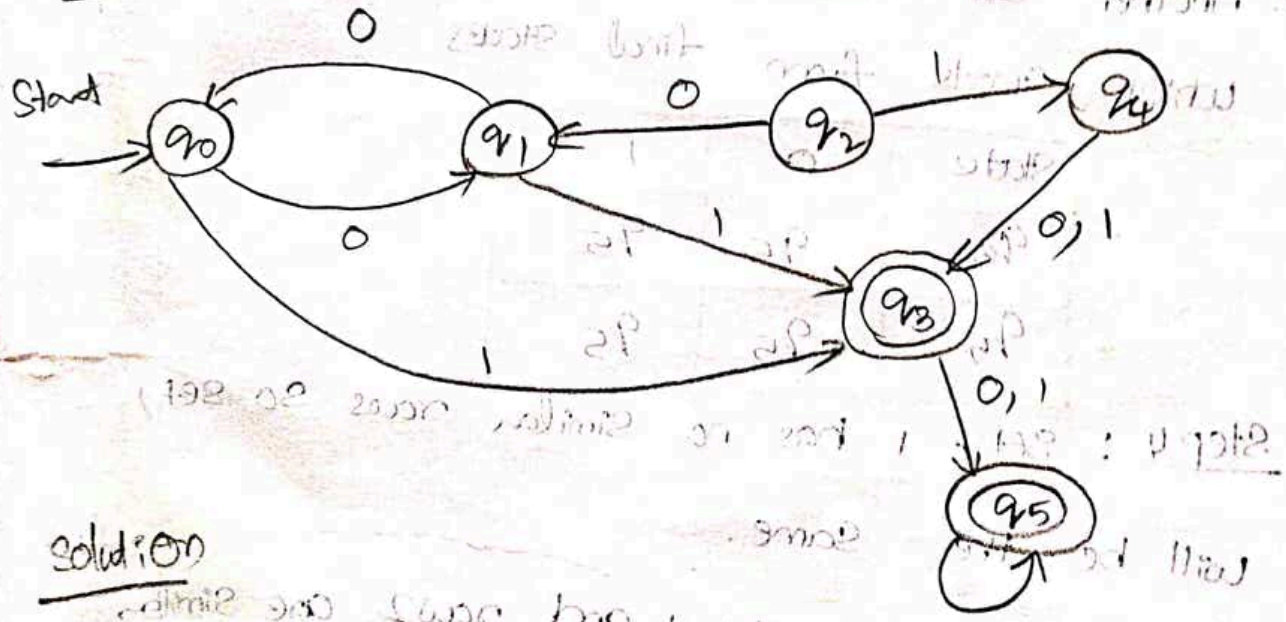
That means find the two states which have the same value of a and b and remove one of them

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also

Step 7: Now Combine the reduced T1 and T2 tables. The Combined transition table is the transition table of minimized DFA.

Example



Solution

Step 1: In the given DFA q2 & q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states

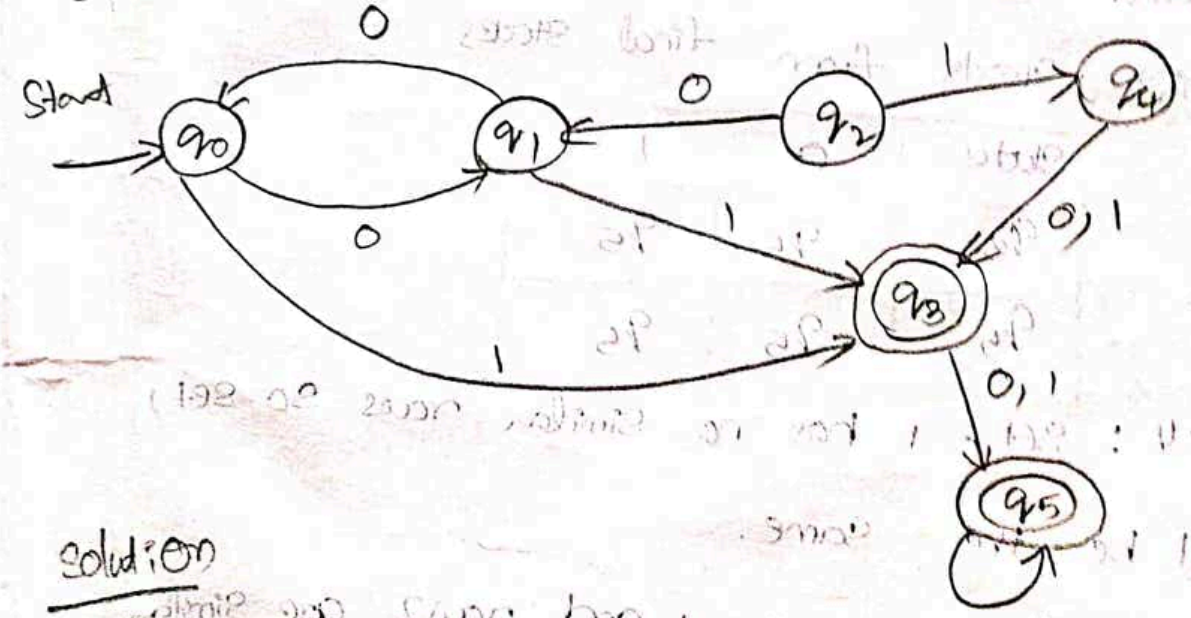
state	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also

Step 7: Now Combine the reduced T1 and T2 tables. The Combined transition table is the transition table of minimized DFA.

Example



Solution

Step 1: In the given DFA q_2 & q_4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states

state	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

Step 3 : Now divide rows of transition table into two sets as :

1. One set contains those rows, which start from non-final states

State	0	1
q_0	q_1	q_3
q_1	q_0	q_3

2. Another set contains above those rows, which starts from final states

State	0	1
q_3	q_5	q_5
q_5	q_5	q_5

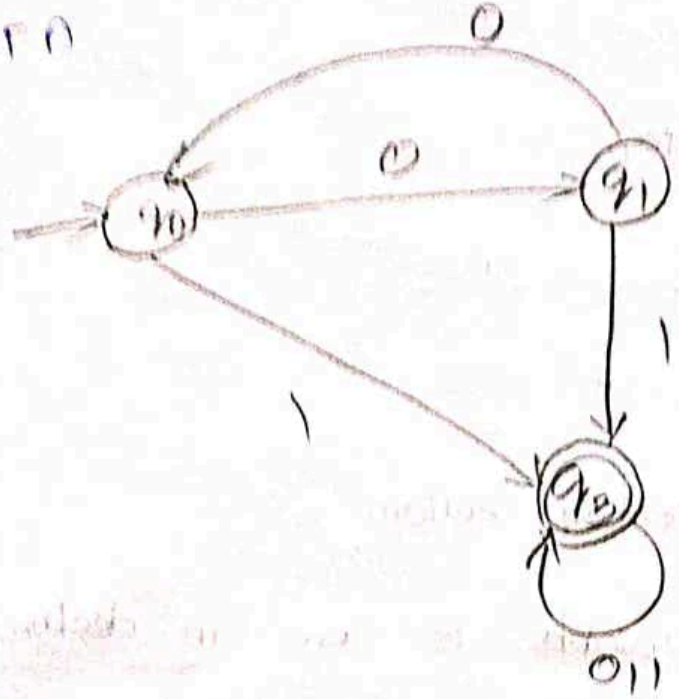
Step 4 : Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2 row 1 and row 2 are similar since q_3 and q_5 transit to the same state on 0 and 1. So skip q_5 and then replace q_5 by q_3 in the rest.

State	0	1
q_3	q_5	q_3

Step 6: Now combine set 1 and set 2 as

State	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
*q_3	q_5	q_3

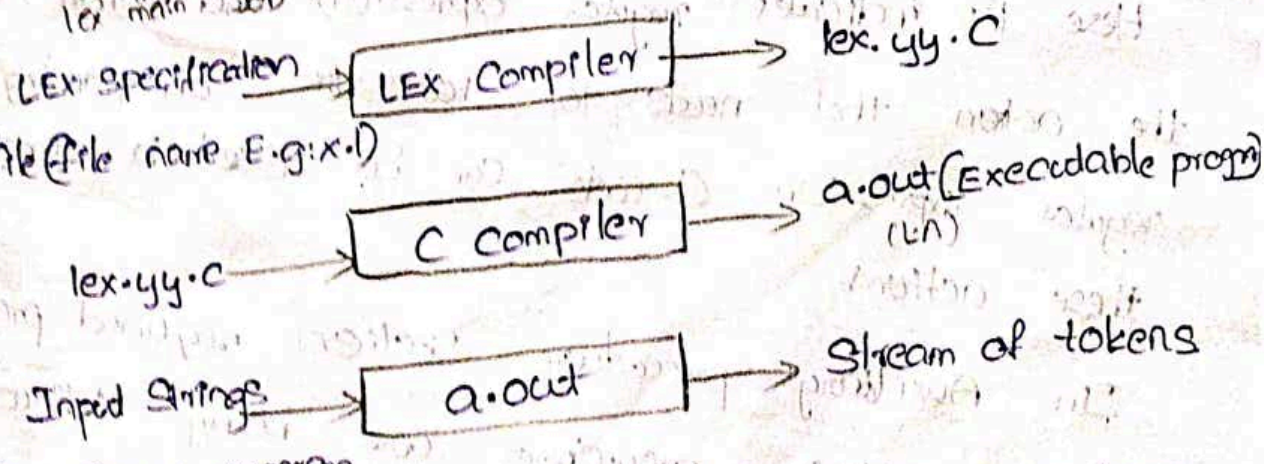


Lex : Lexical Analyzer Generator

LEX is a simple lexical analyzer generator

1. extension can be used to create LEX specification file.

LEX can be generated as shown. In the fig. lex main job is to break up an i/p stream into token



LEX program consists of three sections:
Declaration section, rule section, Procedure section

LEX Source program has

%{
Declaration section
%}
%%
Rule section
%%
Auxiliary procedure section.

* Declaration of variables is done in declaration section. Regular definitions can also be written here
* Rule section consist of regular expressions with associated actions. The translation rules can take the form as:

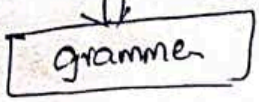
R1 {action 1}
R2 {action 2}
!
Rn {action}

Here R1 predicates / regular expression and action define the action that needs to take for corresponding regular expression. C-code can be used to specify these actions.

In Auxiliary procedure section required procedures are being defined. section requires These procedures may also be required by the actions in the rule section.

Lex program

Prog to count the no. of vowels & consonants in a given grammer.

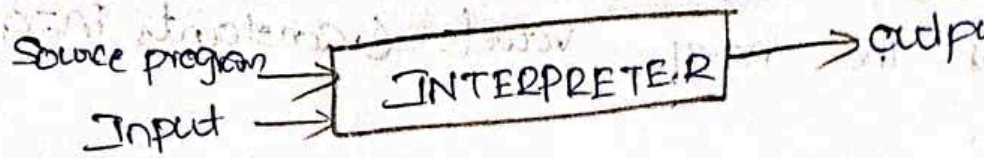



```
%{ #include<stdio.h>
   int vowels = 0;
   int cons = 0;
%}
%%{ aeiou AEIOU } {vowels ++;}
   { a-z  A-Z }  { cons ++;}

int yywrap()
{
   return 1;
}
main()
{
Printf("Enter the string and press ^d \n");
yylex(); -tool
Printf("no. of vowels %d \n"
       no: of cons %d \n", vowels, cons);
}
```

Interpreters

Interpreter is also a language translator like a compiler. Interpreter directly executes the operations in the source program on inputs provides by user rather than producing a target program as a translation. Interpreter is a common type of language processor. It executes the source program statement by statement and therefor it provides better error diagnostics in comparison with Compiler. An Interpreter is shown in fig



Context - free Grammar (CFG)

CFG stands for Context free grammar. It is a formal grammar which is used to generate all possible patterns of strings in a given F.L

Context free grammer G can be defined by four tuples as.

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

→ G is grammer, which consist of a set of production rule. It is used to generate the string of a language

→ T is the final set of terminal symbol. It is denoted by lowercase letters.

T - Terminals
(lowercase letters)

→ V is the final set of non-terminal symbol

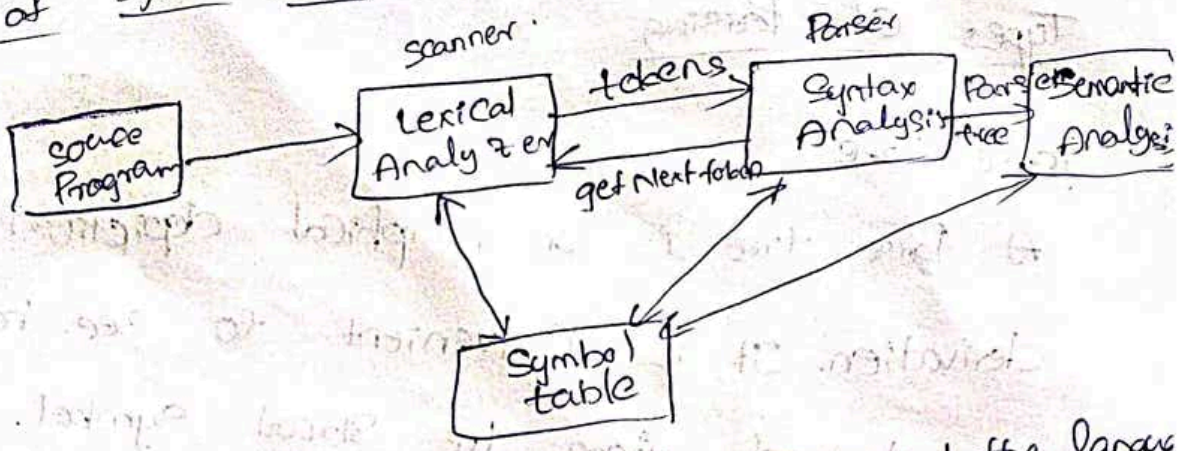
It is denoted by ~~capital letters~~

V - Non-Terminals

→ P is a set of Production rules

→ S is a start symbols.

Role of Syntax Analysis

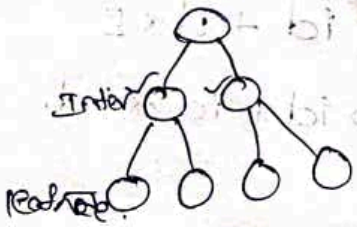


If the tokens are following grammer of the langu

Root node → Start Symbol

Internal node → Non Terminals

leaf node → terminals.



Ex: Consider the following grammer

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

$$id+id*id.$$

left most derivation.

$E \to E+E$ (∵ $E \to E+E$)

$\to id+E$ (∵ $E \to id$)

$\to id+E*E$ (∵ $E \to E*E$)

$\to id+id*E$ (∵ $E \to id$)

Types of Parsing

Parse Tree

A Parse tree is a graphical depiction of a derivation. It is convenient to see how string are derived from the start symbol. The start symbol of the derivation becomes the root of the Parse tree. Let us see this by an example from the last topic.

We take the left most derivation of a+b*c

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

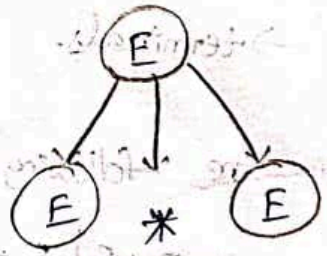
$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id.$

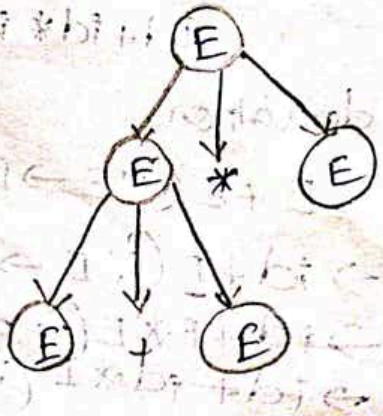
Step 1

$E \rightarrow E * E$



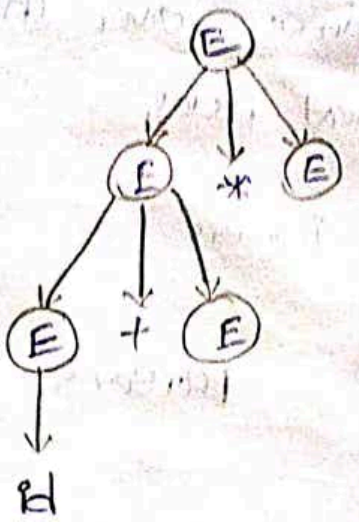
Step 2

$E \rightarrow E + E * E$



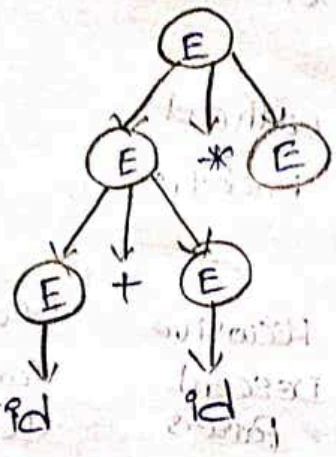
Step 3:

$E \rightarrow id + E * E$



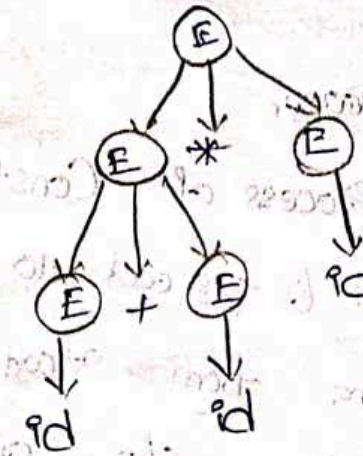
Step 4:

$E \rightarrow id + id * E$



Step 5:

$E \rightarrow id + id * id$



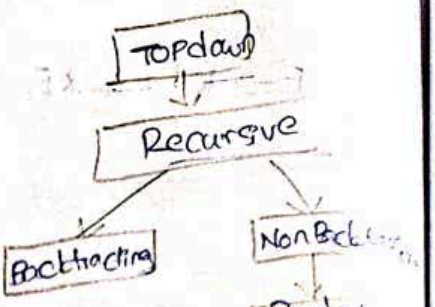
In a Parse tree:

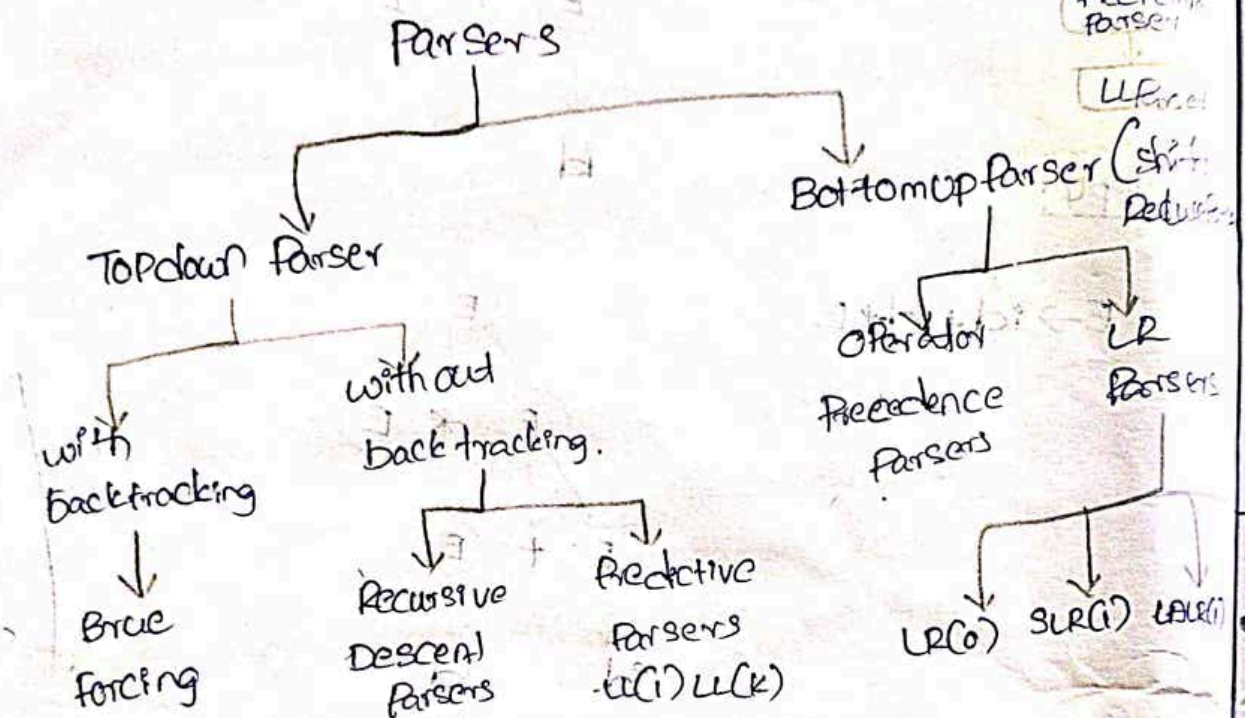
* All leaf nodes are terminals.
* All interior nodes are non-terminals.
* In-order traversal gives original input string

A Parse tree depicts associativity and Precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree

gets Precedence over the operator which is in the parent nodes.

Types of Parsers





TOP down Parser

The Process of Construction of Parser tree starting from root & proceed to children is called TDP i.e starting from start symbol of grammer and reaching the i/p string.

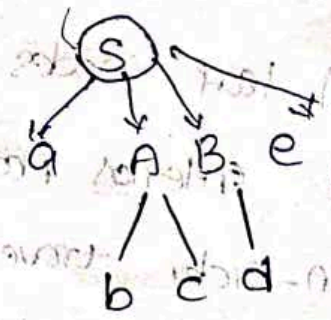
Eg: Start Symbol

$S \rightarrow aABe$

$A \rightarrow bc$

$B \rightarrow d$

$w \rightarrow abcde$



* TDP internally uses left most derivation.

It is free from grammer & Ambiguity.

*TDP Constructed for the grammer if it is free from ambiguity & left recursion.

Avg time complexity is $O(n^2)$

Classification of TDP

* with backtracking – eg: for brute force technique
* without backtracking – eg: Predective Parse
  - LL(1)
  - recursive descent Parsing

Recursive descent Parsing

CLR(1)

Parse is Constructed from top & i/p is read from left to right.

⇒ The i/p is recursively Parsed for preparing a Parse tree. with or without back-tracking

Back tracking

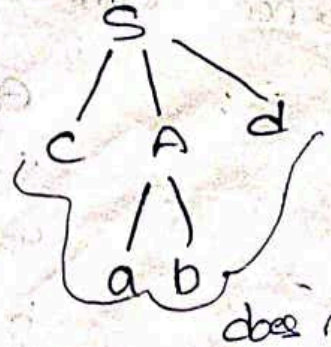
The Parse tree is started from root node and i/p string is matched against the production rules for replacing them.

Eg: Let grammer G be.

$S \rightarrow CAd$

$A \rightarrow ab|d$

String $\boxed{w = cdd}$



w = cdd

does not match to given string

Succeed to do BT

Limitations of Backtracking

* If the given grammer has more no. of Alternatives then the cost of backtracking is high.

Shift reduce Parsing

* Shift Reduce parsing is a process of reducing a string to the start symbol of a grammer
* It uses of a stack to hold the grammer and an input to hold the string.

> A string $\xrightarrow[\text{reduce to}]{}$ the starting symbol

* Shift reduce Parsing perform two actions
  Shift (1) & Reduce (2)
* At shift action, the covered symbol in the i/p string is pushed to a stack
* At each reduction, the symbol will replaced by the non-terminals.
* The symbol is the right side of the production & non terminal is the left side of the production

Ex:

Grammer

$S \to S+S$

$S \to S-S$

$S \to (S)$

$S \to a$

Input String

$a_1 - (a_2 + a_3)$

Stack Content	Input String	Actions
$	$a_1 - (a_2 + a_3)$ $	shift a_1
$a_1	- (a_2 + a_3) $	reduce by S→a
$ S	- (a2 + a3) $	shift -
$ S -	(a2 + a3) $	shift (
$ S - (	a2 + a3) $	shift a_2
$ S - (a2	+ a3) $	Reduce by S→a
$ S - (S	+ a3) $	shift +
$ S - (S +	a3) $	shift a3
$ S - (S + a3	) $	reduce by S→a
$ S - (S + S	) $	shift)
$ S - (S + S)	$	reduce by S→S+S
$ S - (S)	$	reduce by S→(S)
$ S - S	$	reduce by S→S-S
$ S	$	Accept

Eg: Consider the grammer.

E → 2E2
E → 3E3
E → 4

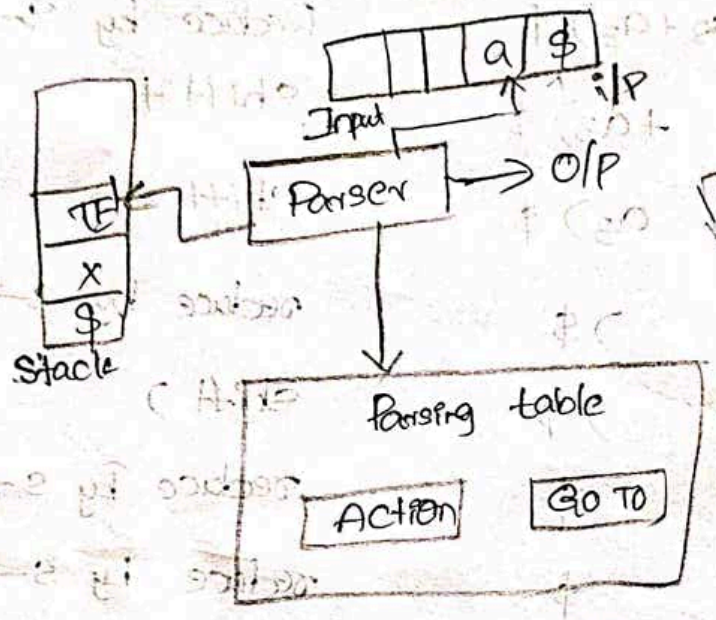
I/P string 32423

Stack Content	Input String	Action
$	32423$	shift 3
$3	2423$	shift 2
$32	423$	shift 4
$324	23$	reduce E→4
$32E	23$	shift 2
$32E2	3$	Reduce E→2E2
$3	3$	Shift 3

$3E3	$	reduce E→3E3
$E	$	Accepted.

Predictive Parser / LL(1) Parser

It is a type of Recursive Descent with no backtracking



LL(k) grammar contains some Constructs

(LL(1) & LL(k) is some

*Each Production has almost one Product to choose

Predictive Parser is a recursive descent Parser, which has the capability to predict which Production is to be used to replace the input string. The Predictive Parser does not suffer from backtracking.

To accomplish its tasks, the Predictive Parser uses a look-ahead pointer, which points to the next input symbols. To make the Parser backtracking free, the Predictive Parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

Predictive Parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

TOP-Bottom Parser

↓ Remove Left Recursion
Left Factored Grammar

Recursive Descent

↓ Remove Back-tracking

Predictive Parser

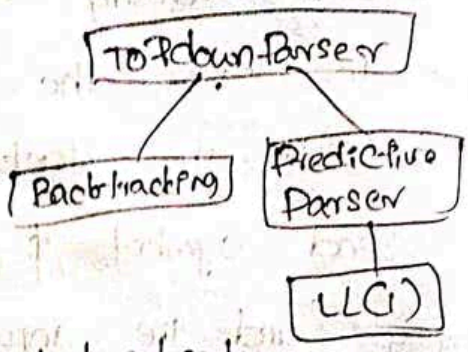
↓ Use Table
Remove Recursion.

Non Recursive Predictive Parser.

In recursive descend parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string making the parsing procedure to fail.

Construction of

LL Parser



It Accepts LL grammer and

It is denoted as LL(k) → No. of look a heads
left most derivation
input from left to Right.

Generally k=1 ∴ LL(k) = LL(1)

* A grammer G is LL(1). If there are two distinct Productions.

$$A \rightarrow \alpha / \beta$$

① For non-terminal α & β derive string begining with a.

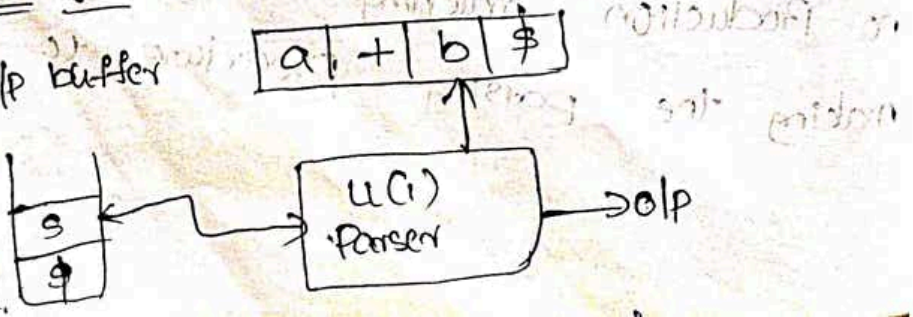
② At most one of α, β can derive Empty string.

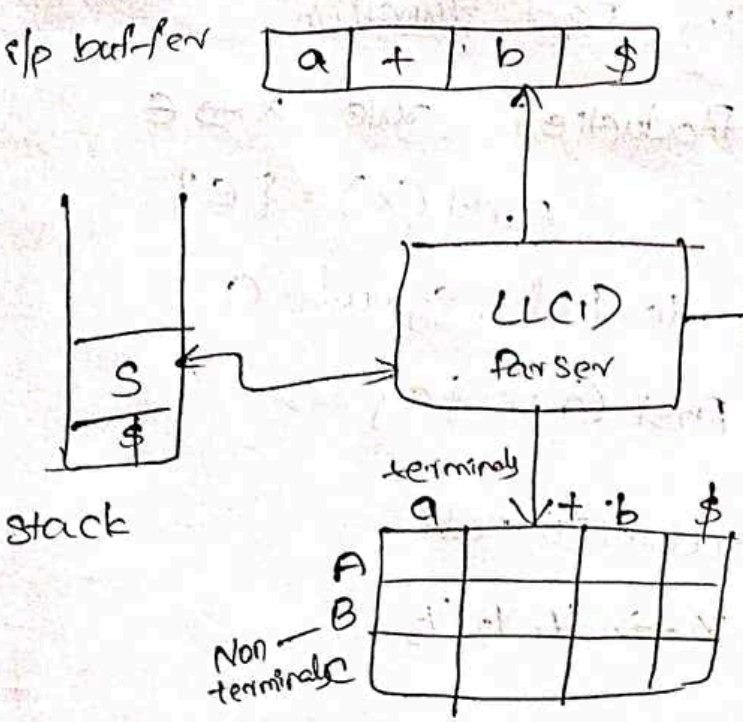
③ If $\beta \rightarrow \epsilon$ then 'a' doesn't derive any str begining with a terminal in Follow(A)

Data structures used by LL(1)

- i/p buffer
- stack
- Parsing table.

Structure of LL(1)





Construction of Predictive LL(1)

① First ( ) / Leading ( )
Follow ( ) / Trailing ( )

② Predictive Parsing table by using first & follow functions

③ Stack implementation.

④ Parse the i/p string with the help of the table.

⑤ Parse the i/p string

Finding FIRST And FOLLOW

First and follow sets are needed so that the Parser can properly apply the needed Production rule at the correct

FIRST Function

* First (α) is a set of terminal symbol that Designs in strings derived from α

Eg: $A \rightarrow abc / def / ghi$

then First (A) = {a, d, g}

Rules for Calculating First

Rule 1 - for a Production rule $X \to \epsilon$

$First(X) = \{\epsilon\}$

Rule 2 - for any terminal symbol a

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

Rule 3 - for a Production rule

$X \to Y_1 Y_2 Y_3$

Calculating First(X)

If $\epsilon \notin First(Y_1)$, then $First(X) = First(Y_1)$

If $\epsilon \in First(Y_1)$, then $First(X) = \{First(Y_1) - \epsilon\} \cup First(Y_2 Y_3)$

Calculating $First(Y_2 Y_3)$

If $\epsilon \notin First(Y_2)$, then $First(Y_2 Y_3) = First(Y_2)$

If $\epsilon \in First(Y_2)$, then $First(Y_2 Y_3) = \{First(Y_2) - \epsilon\} \cup First(Y_3)$

Similarly we can expand the rule for any Production $X \to Y_1 Y_2 Y_3 \ldots$

Follow Functions

Follow(X) is a set of terminal symbol that appear immediately to the right of X

$S \to aAc$ $First(S) = a$

$A \to bd$ $First(A) = b$

$Follow(A) = c$

Rules for Calculating Follow function.

Rule 1 : For the start symbol S, place $ in follow(S)

Rule 2: For any Production rule $A \Rightarrow \alpha B$.

Follow (B) = Follow (A)

Rule 3: For any Production rule $A \rightarrow \alpha B \beta$

If $\epsilon \notin$ First (B), then follow (B) = First (B)

If $\epsilon \in$ First (B), then follow (B) = { first (B) − ϵ } ∪ Follow (A)

Note: * ϵ may appear in the first function of a Non-terminal.

* ϵ will never appear in the follow function of a non-terminal.

* It is recommended to Eliminate left Recursion from grammar if present before calculating first & follow functions.

* we will calculate the follow function of a non-terminal by looking where it is present on RHS of a production rule.

Operator Precedence Parsing

Any grammer G is called an operator precedence grammer if it meets the following two conditions

1. There exist no production rule which contains ε (epsilon) on its right hand side.
2. There exist no production rule which contains two non-terminals adjacent to each other on its right hand side.

$E \to E+E$
$E \to E*E$

* A parser that reads and understand an operator precedence grammer is called as Operator Precedence parser.

Example: which is not an operator precedence grammer.

$E \to EAE \mid (E) \mid -E \mid id$. ✗ This rule

$A \to + \mid - \mid \times \mid / \mid \wedge$

Example: which is an operator precedence grammer.

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

In this terminal is the (+, -, *, /)

Operator Precedence can only established between the terminals of the grammer. It ignores the non terminals

Parsing Action

* Both end of the given input string, add the $ symbol.
* Now scan the i/p string from left right until the > is encounted
* scan towards left over all the equal precedance until the first left most < is encounted
* Every between left most < and right most > is handle.
* $ on $ means parsing is successful.

There are three Operator Precedance relations:

a > b Terminal a has higher Precedence than b
a < b Terminal a has lower Precedence than b
a = b Terminal a & b have same precedence.

Precedance table.

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	x
)	>	>	x	>	x	>
id	>	>	x	>	x	>
$	<	<	<	x	<	x

Rules
id, a, b, c → high
$ - low
+ > +
* > *
id ≠ id
$ A $

Precedence.

Example 1: Consider the following grammer and Construct the Operator Precedence parser.

E → E+E / id.

A → + / x

Now parse the following String id + id x id

Step 1: Check Operator Precedence grammer or not

Step 2: Construct the Relation table.

Step 3: You need to Parse the String.

Step 4: Generate the Parse tree.

Step 1: Convert the given grammer to operator Precedence grammer.

E → E+E | EXE | id.

Step 2: Construct the Operator precedence table.

terminal symbols are.

{ $, id, +, x, $ }

table.

	id	+	x	$
id	−	>	>	>
+	<	>	<	>
x	<	>	>	>
$	<	<	<	A

id → high
$ → low

+ > +
* > *

$ A $
↓
Accepted String

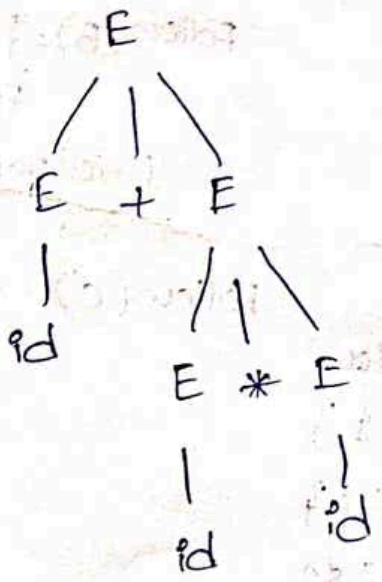
Step 3: Parse the given String id+id*id.

E→E+E
E→E*E
E→id

Stack	Relation	Input	Action
$	<	id+id*id$	shift id
$id	>	+id*id$	Replace E→id.
$E +	<	+id*id$	shift +
$E+	<	id*id$	shift id
$E+id	>	*id$	Reduce E→id.
$E+E	<	*id$	shift *
$E+E*	<	id$	shift id
$E+E*id	>	$	Reduce E→id
$E+E*E	>	$	Reduce E→E*E
$E+E	>	$	Reduce E→E+E
$E	A	$	Acceptance. string.

↑ bottom up approach to construct Parse tree

Step 4: Generate Parse tree. (follow bottomup approach to construct Parsetree)



Calculate the first and follow

Eg:
$S \to A$
$A \to aB / Ad$
$B \to b$
$C \to g$

The given grammer is a left recursion.

So, first remove left recursion from the grammer

After eliminating left recursion.

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

$S \to A$
$A \to aBA'$
$A' \to dA' / \epsilon$
$B \to b$
$C \to g$

First function.

First(S) = First(A) = {a}
First(A) = {a}
First(A') = {d, ∈}
First(B) = {b}
First(C) = {g}

Follow function

Follow(S) = {$}
Follow(A) = Follow(S) = {$}
Follow(A') = Follow(A) = {$}
Follow(B) = First(A') − ∈ ∪ follow(A)
= {d, $}
Follow(C) = NA

Production	First	follow
$S \to A$	{a}	{$}
$A \to aBA'$	{a}	{$}
$A' \to dA' / \epsilon$	{d, ∈}	{$}
$B \to b$	{b}	{d, $}
$C \to g$	{g}	NA

Step2: Construct Parse table using -first & -follow -functions

	a	d	b	g	$
S	S→A				
A	A→aBA'				
A'		A'→dA'			A'→ϵ
B			B→b		
C				C→g	

S→A
A→aBA'
A'→dA'/ϵ
B→b
C→g

Step 3 Implementation by using Parsing table.

Stack	i/p	Production
S $	abd$	S→A
A $	abd$	A→aBA'
aBA' $	abd$	POP a
BA' $	bd$	B→b
bA' $	bd$	POP b
A' d	d$	A'→dA'
dA' $	d$	POP d
A' $	$	A→ϵ
$	$	Accept ∴ The i/p is Properly parsed

Top down parser

Step 4: Generate Parse tree using Stack implementation using TOP down approach.

String w: a b d e

Eg: 2 Construction of Predective Parser LL(1)

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

① first follow
2 Parsing table
3 stack Implementation
4 Parse tree

Step 1 first function

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

$First(T') = \{*, \epsilon\}$

$First(T) = \{(, id\}$

$First(E') = \{+, \epsilon\}$

$First(E) = \{(, id\}$

Follow function.

$Follow(E) = \{\$,)\}$

$Follow(E') = \{\$,)\}$

$Follow(T) = \{First(E') - \epsilon\} \cup Follow(E)$

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

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

Step 2

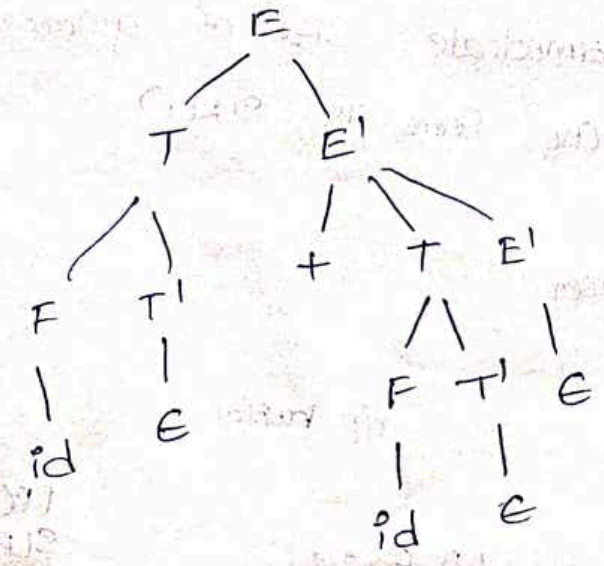
Parsing Table.

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

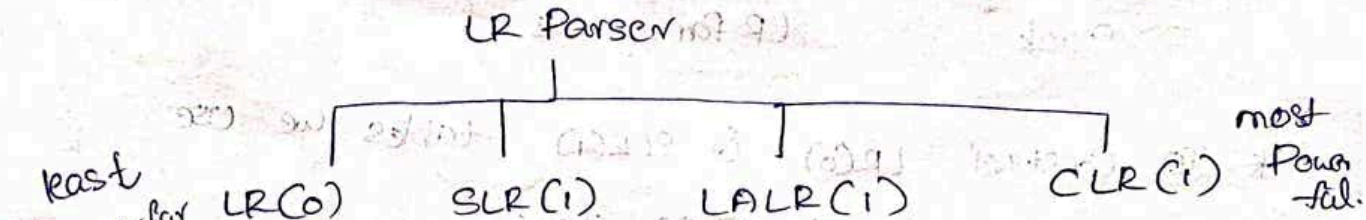
Step 3 Stack implementation.

Stack	I/P	Action
E	id+id$	$E \to TE'$
TE' $	id+id$	$T \to FT'$
$FT'E'$ $	id+id$	$F \to id$
id $T'E'$ $	id+id$	POP id
$T'E'$ $	+id$	$T' \to \epsilon$
E' $	+id$	$E' \to +TE'$
$+TE'$ $	+id$	POP +
TE' $	id$	$T \to FT'$
$FT'E'$ $	id$	$F \to id$
id $T'E'$ $	id$	POP id
$T'E'$ $	$	$T' \to \epsilon$
E' $	$	$E' \to \epsilon$
$	$	Accepted

Step 4 Parse tree (following Top down approach):



LR Parsers



LR Parser : Non recursive shift reduce bottom up parser.

It is also known as LR(k)
- L look a head symbol
- L construction of right most derivation in reverse.

Left to right scanning i/p stream.

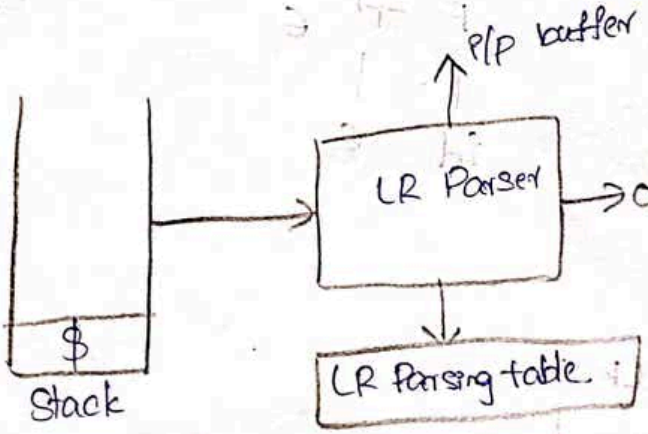
1. SLR(1): Simple LR Parser
   * It is works on smallest class of grammer.
   * It is having few no. of states.
   * Simple & fast Construction.
2. LR(1) - LR Parser
   * It works on Complete set of LR(1) grammer
   * It generates large no. of states.
   * Slow Construction

③ LALR(1)

Look a head LR Parser

* Works on intermediate size of grammer
* No. of States are same as SLR(1)

Structure of LR Parser



LR(0)
SLR(1)
LALR(1)

* To Construct LR(0) & SLR(1) tables we use Canonical Collection of LR(0) items.
* To Construct LALR(1) & CLR(1) tables we use Canon Collection of LR(1) Items.

LR(0) Parsing:

Various Steps involved in Parsing.

* For the given I/P String write Context free gr
* Check ambiguity of the grammer
* Add augmente production in the given grammer
* Create Canonical Collection of LR(0) Items
* Draw a data flow diagram (DFA) Data follow ideo

Prob: Construct LR(0) Parsing table.

S → AA
A → aA/b

$S \to S$
$S \to AA$
$A \to aA$
$A \to b$

Augment Product
Add another Pro to the gram

to identify
to stop

Step 3: Add Augment Production

$S' \to S$

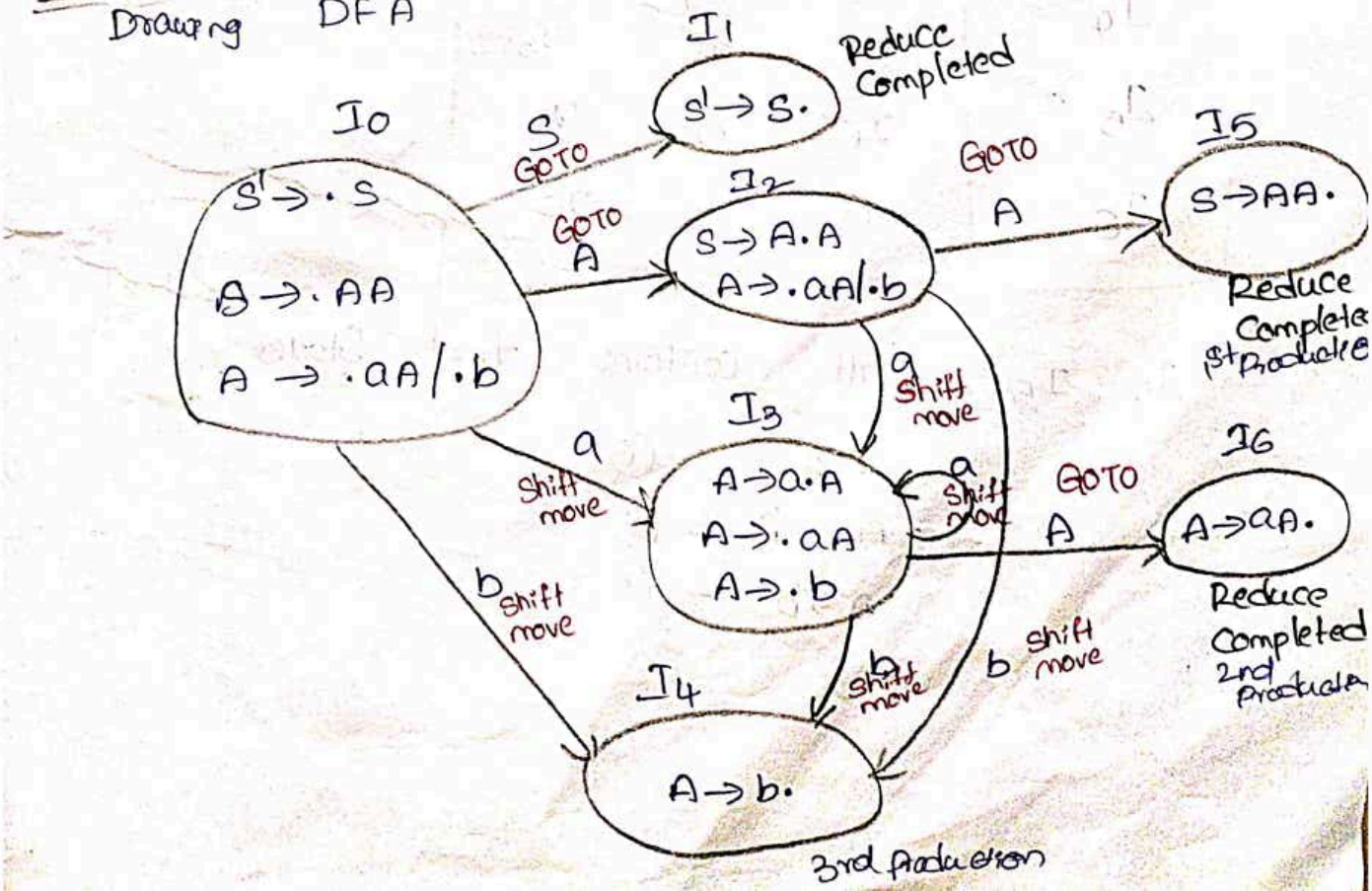
$S \to AA$
$A \to aA / b$ } given grammer.

Step 4: Create canonical collection of LR(0) items.

* An LR(0) item is a production G with dot at some position on the right hand side of Produce
* LR(0) items is useful to indicate that how much of the i/p has been scanned up to a given point in the process of parsing.
* In LR(0) we place the reduce node in entire ra

$S' \to .S$
$S \to .AA$
$A \to .aA / .b$

Step: 5:
Drawing DFA



Step 5 LR(0) table

* If a state is going to some other state on a terminal it is correspond to a shift move
* If a state is going to some other state on a variable it is correspond to goto move
* If a state contains the final item in the production rule then write the reduce node corresponding

States	Action (terminals)			GOTO (variables)	
	a	b	$	A	S
I_0	S_3	S_4		2	1
I_1			accept		
I_2	S_3	S_4		5	
I_3	S_3	S_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

I_4 I_5 I_6 all contains final states.

$S \to AA$ — (1)

$A \to aA$ — (2)

$A \to b$ — (3)

conversion of

SLR(1) Parsing

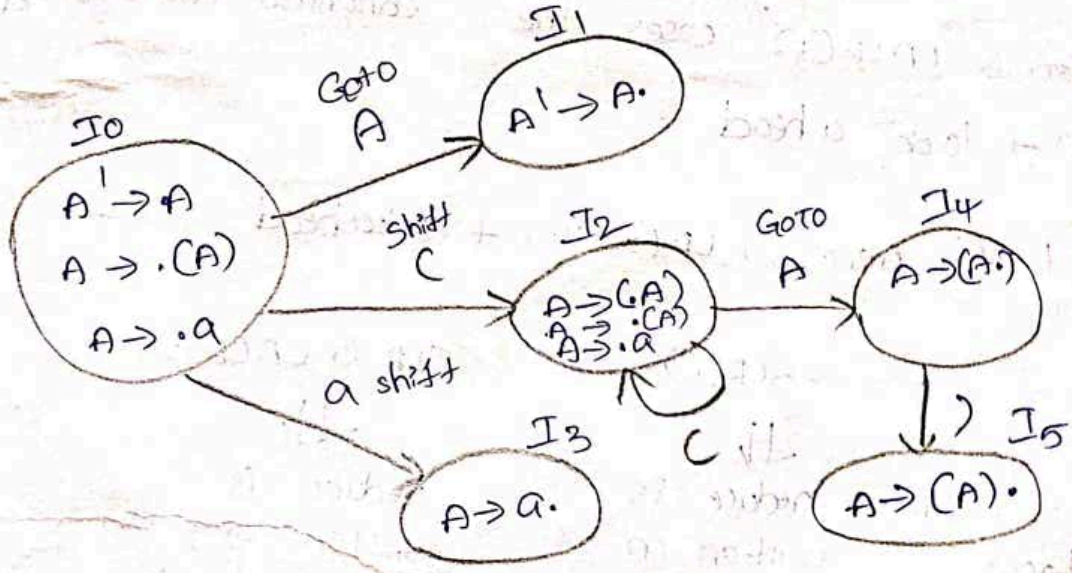
* Smallest class of grammer
* It requires few no. of states.
* Simple & fast to construct
* In SLR we place the reduce move only in the follow of left hand side not to entire row.

$A \to (A) / a$

Augement grammer $A' \to .A$

$A \to .(A)$

$A \to .a$



Parsing table

State	Action.				GOTO
	a	(	)	$	A
I_0	S_3	S_2			1
I_1				Accept.	
I_2	S_2	S_2			
I_3			r_2	r_2	
I_4			S_5		
I_5			r_1	r_1	

$A \to (A)$ - (1)

$A \to .a$ - (2)

LALR (1)

CLR(1) } both uses the Look a head value.
LALR(1)

LALR(1) uses the Canonical collection of LR(1) ite

LR(1) item → LR(0) items + look a head value.

Eg:

$E \to BB$
$B \to CB/d$ $\Rightarrow$

$E' \to \cdot E, \$$
$E \to \cdot BB, \$$
$B \to \cdot CB/\cdot d, c/d$ (first(B) = c/d)

LR(1) items.

I_3+I_6 – Productions are same so Combine them

I_4+I_7 – Production are same and look a head d

I_8+I_9 – " " " "

$I_3 + I_6 \Rightarrow I_{36}$
$I_4 + I_7 \Rightarrow I_{47}$
$I_8 + I_9 \Rightarrow I_{89}$.

	Action			GOTO	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{36}	S_{36}	S_{47}		89	
I_{47}			r_3		
I_{89}	r_2	r_2			
I_{89}			r_2		

Merge

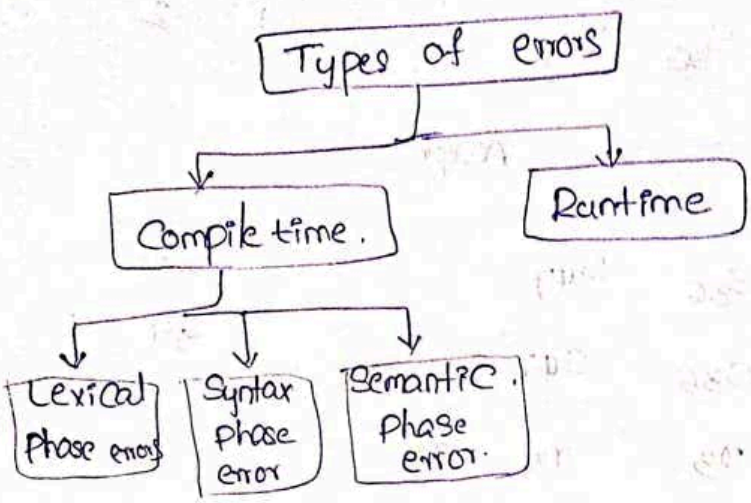
LALR(1)

State	Action			Goto	
	Cel	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accpt		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

LALR(1) having Limited States.

H. Reduce 10 to 7 States.

Error recovery



Lexical Error

* During Lexical phase, lexical error can be detected.
* Lexical error is a sequence of characters that do not match the pattern of any token.
* Lexical phase error is found during the execution of the program.

Lexical phase

* Spelling error.
* Exceeding length of identifier or numeric constants
* Appearence of illegal characters.

To remove the characters that should be present.

To represent replace a character with an incorrect char

Transposition of two characters.

Eg:

```
void main()
{
int x = 10, y = 20;
char a;
a = &x;
x = 4xab;
}
```

this code 4xab is neither a num nor an identifier. So, this code shows lexical

Syntax error

* Syntax error is appears during Syntax anlysis phase.
* It is found during the execution of a program
* Some syntax errors can be
  * Error in structure.
  * Missing operators.
  * Unbalanced paranthesis.

Eg Using "=" when "==" is needed.

```
If (number = 200)X (number == 200)
    Count << "no: is equal to 200";
    Count << "no: is not equal to 200";
```

In this example syntax warning will come,

In this code, if expression used the equal sign which is actualy an assignment operator not the rebational operator which test for equality.

Eg:2 float x = 1.2 // semicolon is missing

Eg 3 Errors in Expressions.