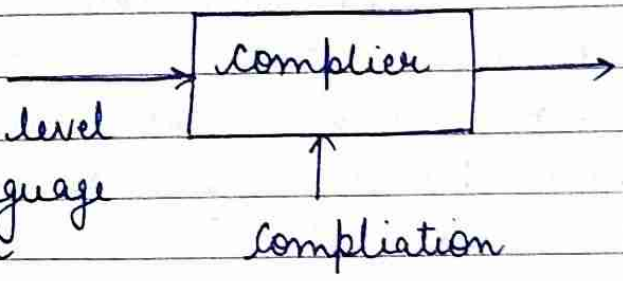
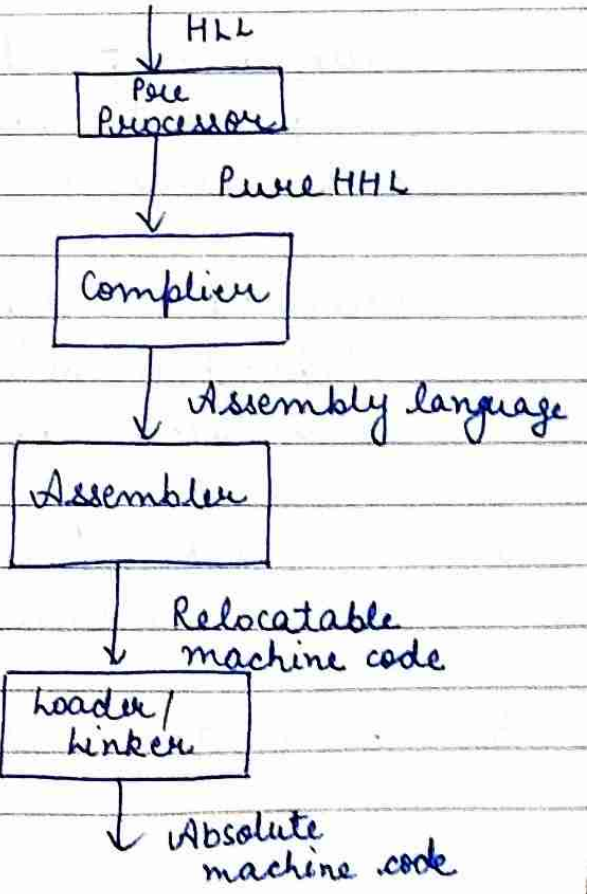
Date
Page

Complier design:-

A complier is a special program that translates a program written in high-level programming language into machine code or bytecode i.e. understandable by computer. An important role of the complier to report the errors in the source program that is detected during translation process. The output generated by complier is called machine code / object code / target code / 0's / 1's.

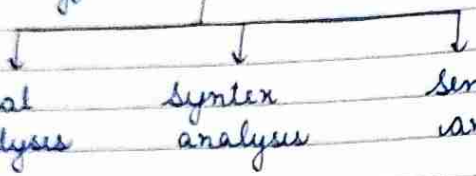


language processing system:-
(using complier.



Analysis of Source program:

The structure of the source program can be analyzed in 3 phases:



1. **Lexical analysis:**
It is the lowest level. It is also called linear analysis or scanning. In this, the source programs are viewed as sequence of characters called tokens. A token is a group of strings that make sense. It can be single character or sequence of characters. Tokens can be classified as:

1. **Identifiers:** These are names chosen to represent variables, functions, classes, data items etc.

2. **Keywords:** They are the reserved words of any programming language. Keywords can't be used as identifiers.

3. **Operators:** They are used to perform operations on operands.

4. **Separators:** These are punctuation marks used to group together a sequence of tokens that have a single meaning.
E.g. → (,), ;, /, {, }.

5. **Literals:** Some language support literals which denote direct values.

6. **Comments:** non-executable part of program which is used to increase readability of a program.

E.g.:
A statement:
Position = Initial + rate * 60 is grouped into following characters:

a) Position → identifier
b) Initial → identifier
c) + → operator
d) = → Assignment symbol
e) rate → identifier
f) * → multiplication sign (operator)
g) 60 is a number

[Note: Blanks separating characters of these tokens are eliminated during lexical analysis]

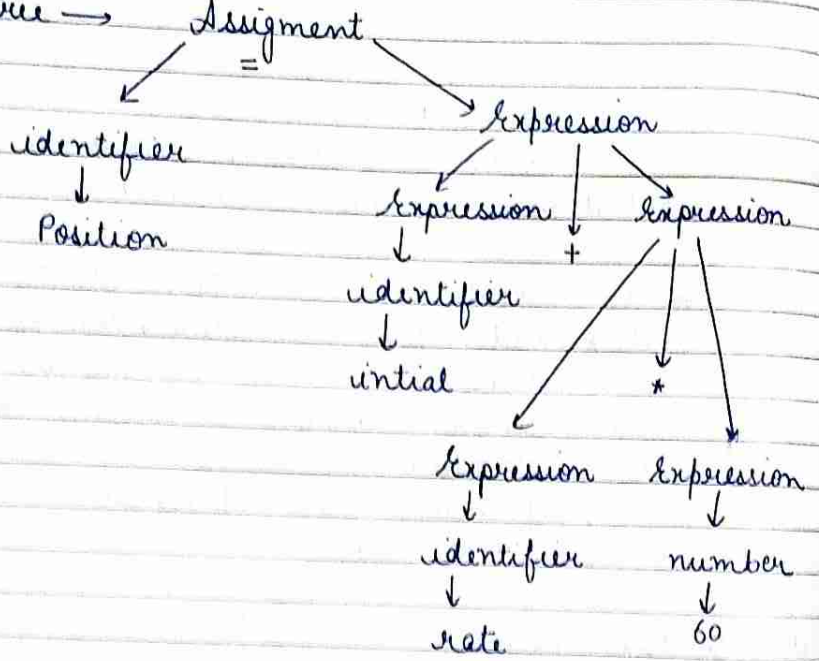
2. **Syntax analysis:**
It describes the way in which program statements are constructed from tokens. This is defined using context free grammar & represented using parse tree. It involves grouping of source program into grammatical phases that are

used by the complier to synthesize output.

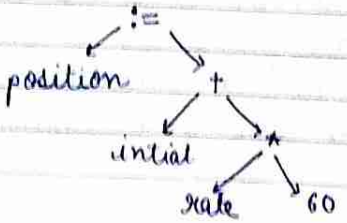
→ A ~~syntax~~ tree is a tree in which internal nodes are the operators & the external nodes are operands. This analysis shows an error if syntax is incorrect.

E.g.: Position = initial + rate * 60

Parse tree →



Syntax tree: Compressed representation of parse tree.

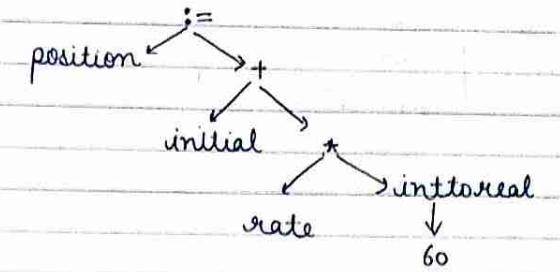


3. Semantic analysis:

This phase checks the source program for semantic errors & gathers type information for code generation phase.
It uses hierarchial structure determined by syntax level to identify operands & operators.

E.g. → Typechecking ⇒ checking for permitted operands.

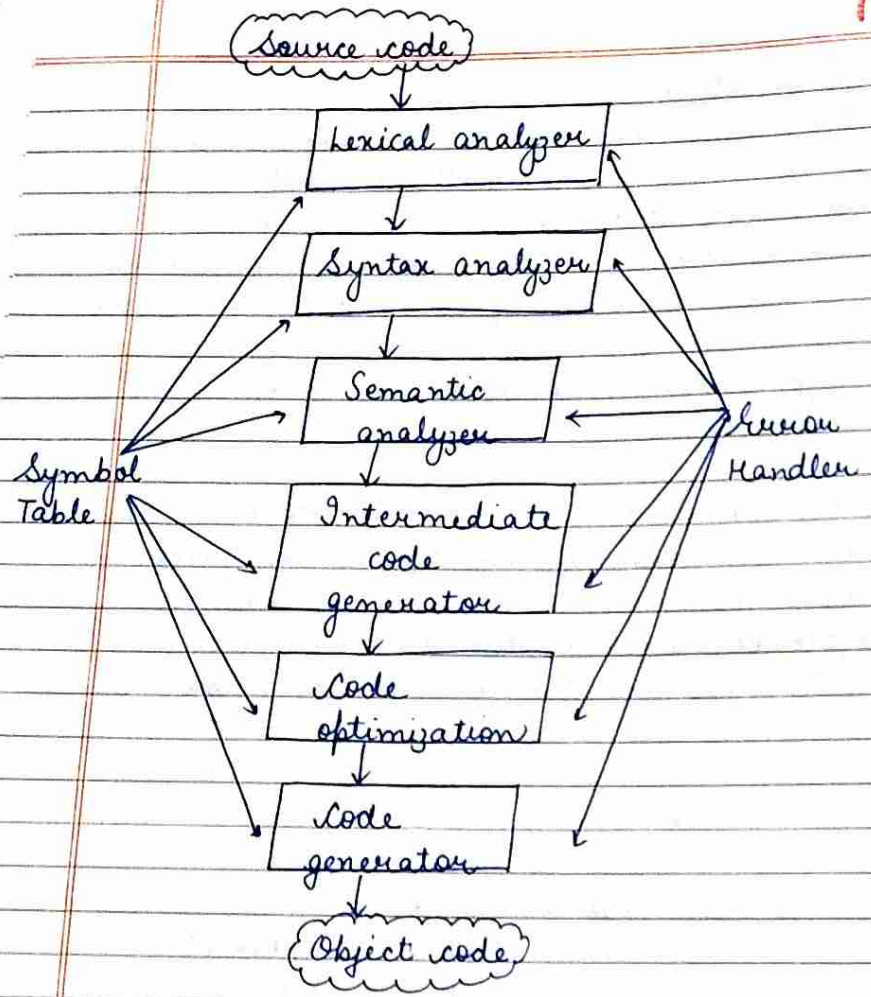
In statement, position = initial + rate * 60, the identifiers position, initial & rate are declared as real numbers, but 60 is a integer.
Therefore 60 is converted to type real.
Thus, syntax tree becomes as shown below:



Phases of Complier:

a) lexical analysis:

→ It is first phase of complier.
→ It reads the stream of characters & group the characters into meaningful sequences



called lexemes.

For each lexeme, the output is a token in form of

<token-name, attribute-value>

which is passes to next phase.

In the token, the first component token-name is an abstract symbol that is used during syntax analysis & the second component attribute-value points to an entry in symbol table for this token.
Information from symbol table is needed for semantic analysis & code generation.

E.g. → position = initial + rate * 60

The character in these statements would be grouped into following lexemes :-

1. position is a lexeme that would be mapped into token <id, 1> where id is an abstract symbol for identifier & 1 points to symbol table entry for 'position'. The symbol table entry holds information about the identifier, such as name & type.

2. The assignment symbol = is a lexeme that is mapped into token <=>. It needs no attribute value, so second component is omitted.

3. Initial is the lexeme that is mapped into <id, 2>, where 2 points to entry for initial.

4. + is a lexeme that is mapped into <+>.

5. Rate is the lexeme that is mapped into <id, 3>

6. 60 is the lexeme that is mapped into <60>.

Blanks separating the lexeme would be eliminated by lexical analyzer.

<id,1> <=> <id,2> <+> <id,3> <60>

Token: Token is a single character or sequence of character treated as single entity.
- → Identifiers
- → Keywords
- → operators
- → Separators
- → literals / constants

Pattern: A set of strings in input for which the same token is produced as output. This set of strings is described by rule called pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in source program that is matched by pattern for the token.

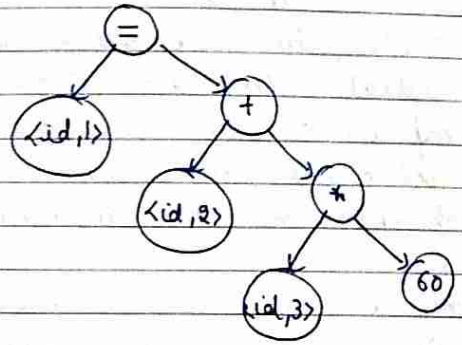
e.g:

Token	Informal description	Sample lexeme
if	characters i, f	if
else	characters e, l, s, e	else
comparisons	<, >, >=, <=, ==, !=	<=, !=
id	letters followed by letters & digits	score, d1, a2
number	any numeric constant	2.14, 3.22
literal	anything but surrou-ned by " " 's	"score"

b.) Syntax analysis:

→ The second phase of complier is parsing.
→ The parser uses the first component of the tokens produced by lexical analyzer to create a tree which is intermediate representation that tells about grammatical structure of token stream.
→ From Parse tree, syntax tree is generated.

Syntax tree:



→ The internal node is labelled with '*' has (id,3) has its left child & 60 as right child.
→ (id,3) represents identifier rate.
→ The node labelled with * makes it explicit that we must multiply <id,3> with 60.
→ The node labelled + indicates that we must add the result of multiplication to value of initial.

→ The root of tree, labelled with = indicates we must store the result of this addition for identifier position.

c) Semantic analysis :

The semantic analysis uses the synter tree & the information in symbol table to check the source program for semantic consistency with the langubge defination.

→ It also gathers type of information & saves it either in synter tree or the symbol table, for subsequent use during intermediate code generation.

→ Most important part of semantic analysis is type checking where complier checks the each operator has matching operands.

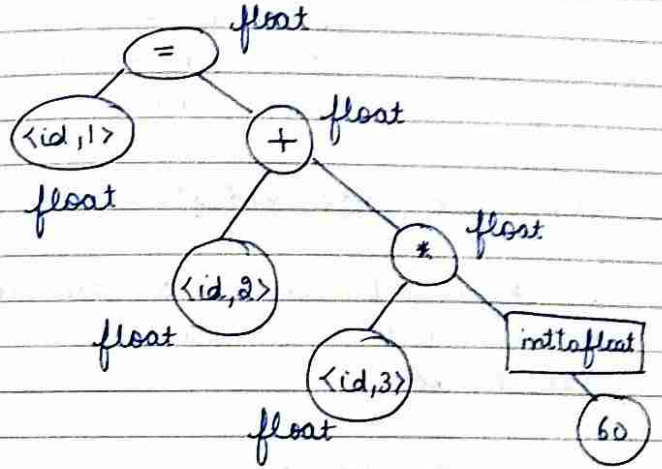
E.g. → Many programming languages requires an array index to be integer, the complier must report an error if floating point number is used to index an array.

Some sort of type conversion is done by complier.

E.g. → If we add, 1 floating point & 1 integer, the complier may automatically convert integer to float.

E.g: Suppose, the identifiers position, initial, rate are floating point numbers & the lexeme 60 by itself forms an integer. The semantic analyzer discovers that the operator * is applied to floating point & integer.

So, the output of semantic analyzer adds an extra node for operator inttofloat which explicitly converts integer to floating point number.



d) Intermediate code generation :

In the process of translating a source program, a complier may construct a one or more intermediate representations, which can have a variety of forms.

Syntax trees are form of intermediate representations which are commonly used during syntax & semantics analysis.

After that, the complier generate an explicit low-level or machine like representation which is like a program for abstract machine. The generated code is machine independent.

E.g.→ In our example, we use 3-address code which consists of sequence of instructions with 3 operands per instruction.

E.g. → t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Properties of intermediate representation:

1. It should be simple & easy to produce.
2. It should be easy to translated into target code.

e) Code optimization:

The machine-independent code optimization phase attempts to improve the intermediate code so that the better target code will be generated.
It is done to make the execution faster, shorter code or the target code consumes less power. Thus, it increases the efficiency.
E.g.:

The optimized code of above example is:

t1 = id3 * 60.0
id1 = id2 + t1

The 4 line of code is converted to 2 line code.

f. Code generator:

It is the last phase who takes intermediate representation of source program as its input & maps it into the target language.

The target code can be

→ If target language is machine code, registers or memory locations are used for each of variables used by source program. So, it is converted into sequences of machine instructions to perform task.

→ If the target language is assembly language, it generates assembly code as a target output.

E.g. Code generated:

```
LDF R2, id3
MULF R2, #60.0
LDF R1, id2
```

```
ADDF  R1, R2
STF   id1, R1.
```

The F in each instruction refers to floating point number.

Explanation:-

1. Load the contents of address id3 into register R2, then multiply it with floating point constant number 60.0. '#' indicates that it is intermediate constant.
2. The 3rd instruction moves id2 contents into register R1 & 4th adds the contents of R1 & R2. The resulted value is stored in register R1.
3. Finally, the value in register R1 is stored in address of id1.

So, the code correctly implements statement, position = initial + rate * 60.

Symbol table :-

Symbol table is the data structure containing a record for each variable name, with fields for attributes of the name.

They attributes may provide information about storage allocated for a name, its type, its scope (where in program its value maybe used) & in case of functions such things as number & type of arguments, the method of passing arguments (call by value, call by reference) & type returned.

The data structure stores records in such a way it is easy to find the record very quickly & retrieve the data for the record. The symbol table is used by every phase of complier to get the information.

Error detection & reporting :-

→ Each phase can encounter error.
→ However, after detecting an error, a phase must somehow deal with that error, so that compliation can proceed, allowing further errors in source program to be detected.

Grouping of phases :-

The process of compilation is split up into following phases :-

1. Analysis phases : It mainly performs 4 actions :-

a) Lexical analysis.

b.) Syntax Analysis.
c.) Semantic analysis.
d.) Intermediate code generation.

The analysis part breaks up the source program into constituent pieces & impose a grammatical structure on them. Then this structure is used to generate intermediate representation.

If analysis part detects that source program is either syntatically ill formed or semantically unsound, then it must provide informative message, so that user can take corrective action.

It also collects info about source program & stores it in data structure called symbol table, which is passed along with intermediate representation to the synthesis part.

Symbol table + Intermediate code → Synthesis phase.

Synthesis phase:-

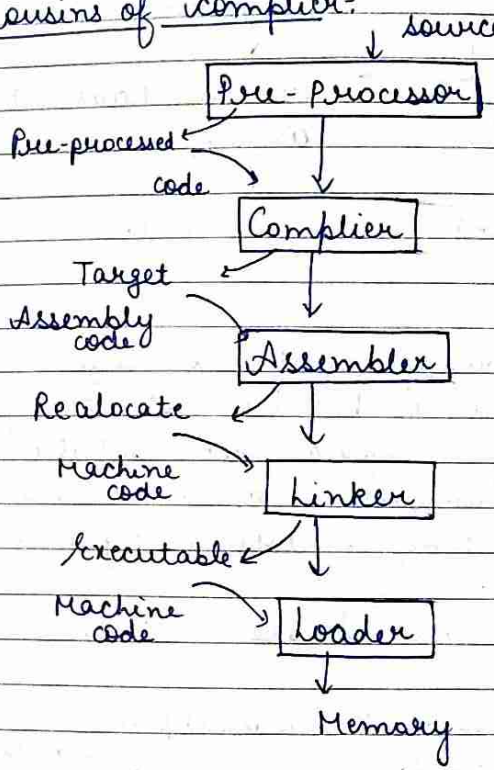
It mainly performs 2 actions:-

a.) Code optimization.
b.) Code generation.

The synthesis part constructs the desired target program from intermediate represent-ation & info. in symbol table. It also performs optimization so that execution takes less time, lesser space & thus provides efficiency.

The analysis part is called front part/end of complier & synthesis part is called back end.

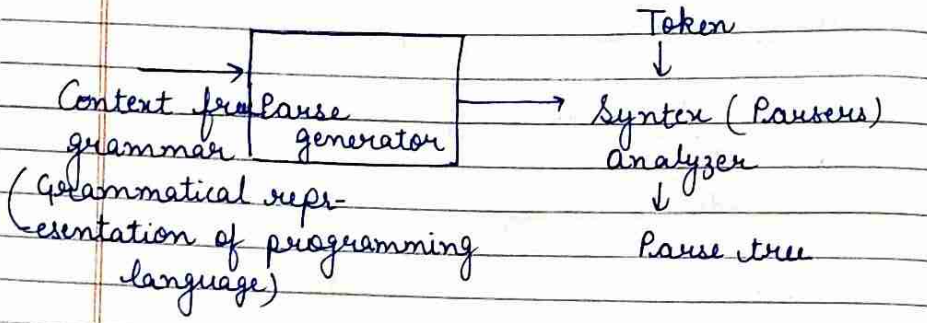
Cousins of complier:-



Complier construction tools :

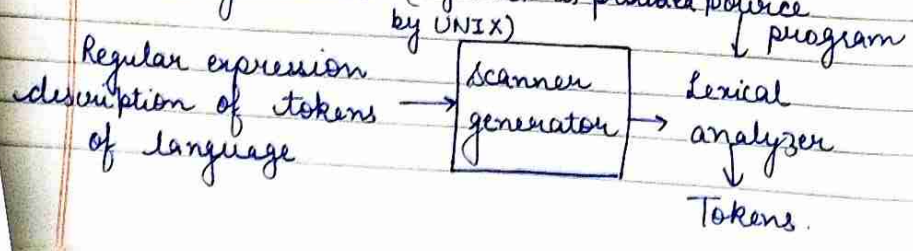
Complier writers use software development tools & more specialized tools for implementing various phases of complier.
Some commanly used construction tools are :

1. Parser generator :



They normally produce syntax analyzers, from input that is based on context free grammer or grammatical representation of program. It is useful as syntax analysis phase is highly complex which requires lot of time & effort.
It is one of easiest phase to implement.

2. Scanner generators : (E.g. → LEX is provided by UNIX)



It generates lexical analyzer from the input consists of specifications of regular expression which is description of token of a language.
It generates finite automata (DFA & NDFA) to recognize regular expression.

3. Syntax directed Translation engine :

Input : Parse tree
Output : Intermediate code.

It generates intermediate code with three address format from input consists of a parse tree.
These engines have routines (functions) to traverse (visit) each node in parse tree & produces intermediate code for that node.
The basic concept is that one or more translations are related or associated with each node of parse tree & each translation is represented because of translation at its neighbor nodes in the tree.

4. Automatic code generators :

Input : Intermediate Code.
Output : Machine language.
It generates the machine language for a target machine. Each operation of intermediate

code is translated using collection of rules & then is taken as input by the code generator
The rules must include sufficient detail that we can handle the different possible access methods for data.
In this process, template matching is used.

5. Data flow analysis engine:-
It is used in code optimization. It is a keypart of the code optimization that gathers the information i.e. the values that flow from one part of a program to another.

Native Complier & Cross complier:-

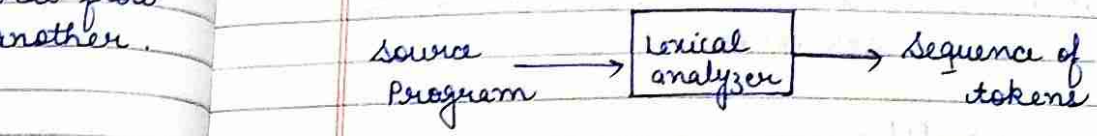
Native	Cross
1. Native complier are compliers that generate code for same platform on which it runs.	1. Cross complier is a complier that generates executable code for a platform other than one on which complier is running.
2. It is dependent on machine or OS.	2. It is independent of machine.
3. It is used to build program for same system / OS it is installed.	3. It is used to build programs for other system like AVM / ARM.
4. It can generate executable file like .exe.	4. It generate raw code .hex.

e.g. → Turbo C or GCC is native complier

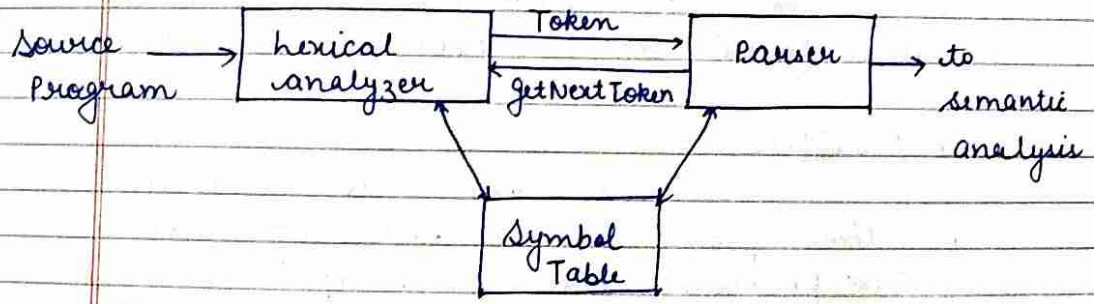
e.g. → Keil.
[CC is that complier running on windows produce object code for MAC os etc.]

Role of lexical analyzer:-

1. a) The main task of lexical analyzer is to read the input characters of source program, group them into lexemes & produce as output a sequence of tokens.



b) When a lexeme is discovered, it needs the lexeme into the symbol table. The entries in the symbol table are needed by the other phases of complier



The interaction is implemented by having the parser call the lexical analyzer. GetNextToken command causes the lexical analyzer to read from its input until it can identify

Lexical error:- A character sequence that can't be scanned into any valid token is a lexical error.

the next lexeme & produce a token, which it returns to parser.

2. It removes all the comments & whitespaces (blank, newline, tab or other characters that are used to separate tokens)

3. It also generates error messages like exceeding length, unmatched string, illegal character. For instance, lexical analyzer may keep the track of number of newline characters seen, so it can associate a line number with each error message.

Attributes for token:-

A token is associated with different piece of information i.e. each token has different properties or attributes.

For e.g. → Information about identifier is its lexeme, type, location where it is first found etc is kept in symbol table.

Input buffering:-

The lexical analyzer scans at input at a time from left to right. It makes use of two pointers:- begin ptr (bptr) & forward pointer (fptr) to keep track of the input scanned.

Initially, both the pointers point to the first character of the input string.

int a = 1;

i	n	t		a	=	1	;

fptr
bptr

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered it indicates end of lexeme.

In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified. The fp ptr moves ahead, when it encounters blank space, then both bp & fp are set at next token. The input character is thus read from secondary storage, but reading this way from secondary storage is costly. Hence, buffering technique is used. A block of data is first read into a buffer, & then by lexical analyzer. There are 2 methods used in this content.

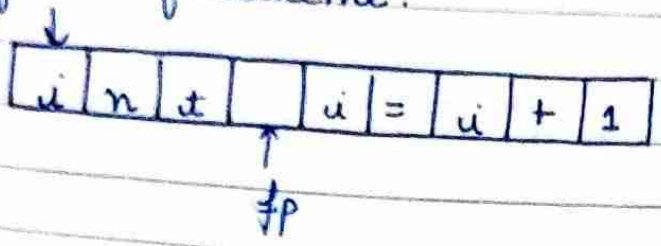
One Buffer Scheme

Two Buffer Scheme.

1. One buffer scheme:-

In this scheme, only one buffer is used to store the input string but in this, the problem is that if lexeme is very long, then it crosses the buffer boundary,

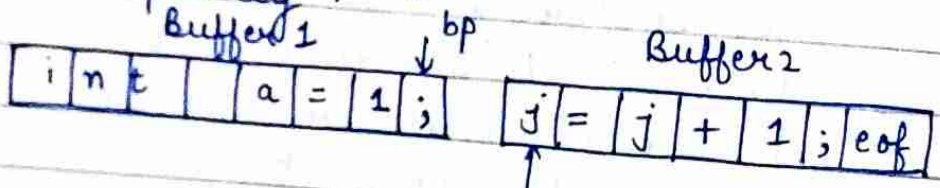
to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



2. Two buffer scheme:

To overcome the problem of one buffer scheme, in this two buffers are used to store the input string. The first & second buffer are scanned alternately. When end of current buffer is reached then the other buffer is filled.

→ The only problem with this method if length of lexeme is longer than length of buffer, then scanning input can't be scanned completely.



→ Initially both are pointing to first characters of first buffer. fp moves foward in search of lexeme. As soon as, blank character is recognized, string b/w fp & bp is identified as token.

→ The "end of" character is placed at end of each buffer to identify the first & second buffer.

→ So, both the buffers can be filled alternatively until the end of input program & stream of token is identified.

Specification of tokens :

To specify a token, we use Regular expressions which generate Regular languages.

Alphabet : An alphabet is a finite non-empty

set of symbols. Symbols can be letters, numbers or any special character.

e.g. :- $\Sigma = \{0, 1\}$ → binary alphabet.
$\Sigma = \{a, b, c \dots x\}$ → lower case letters alphabet.

Strings :- Strings are finite set of symbols generated from alphabet.
e.g. :- $\Sigma = \{0, 1\}$

String → 0, 01, 100, 11, 1 .. etc.
The length of string is total number of characters in the string.

Language :- Set of strings generated from the alphabet is called language. It is collection of string.
e.g :- $\Sigma = \{a, b\}$
$L1$ = Set of all strings of aleast length two.
$L1 = \{aa, bb, aaa, aaaabb \dots\}$
So, $L1$ is infinite language.

So, languages can be finite or infinite.

Regular Expression :-

1.) Regular expression represents patterns of string of characters.
2.) R.E also have a special character called meta-characters. e.g. → *, + etc.

Rules for forming Regular expression:

1. Let 'a' be a character in string S. The regular expression is 'a'. It matches with character 'a' by writing $L(a) = \{a\}$
2. Any terminal symbol (element in Σ), Λ & ϕ are regular expressions.
3. The Union of two regular expression is regular expression.
4. The iteration or closure of regular expression is R^*, which is also a regular expression.
   $R^* = \{\Lambda, a, aa, aaa \ldots\}$
   $R^+ = \{a, aa, aaa \ldots\}$.
5. If the alphabet of language is null i.e. $\Sigma = \phi$ then $L(\phi) = \{\}$

Operations on regular expression:

1. Choice among alternates: (Union)
   Let r and s be two regular expression. Then, union of these two regular expression represented by r/s is also a regular expression.
   $L(r/s) = L(r) \cup L(s)$

   Let us suppose, $L(r) = \{a, b\}$
   $L(s) = \{c, d\}$
   $\therefore L(r/s) = \{a, b, c, d\}$

2. Concatentation: concatentation of 2 regular

expressions is also a r.e. represented by
e.g.

Two regular expressions r & s,
concatentation is written as rs

$L(rs) = L(r)L(s)$ [Character in $L(r)$ followed by char. in $L(s)$.]

$L(r) = \{a\}$
$L(s) = \{b\}$
$L(rs) = \{ab\}$

c.) Repetition: Repetitive operation of a regular expression is called Kleene.
Closure: It is represented by r^*.

e.g. → $L(r) = \{a\}$, $L(s) = (b)$

1.) r^* → It represents the language containing the zero or more occurences of symbol from $L(r)$.

$L(r^*) = \{\varepsilon, a, aa, aaa, aaaa, --\}$

2.) $(rs)^*$

$L((rs)^*) = \{\varepsilon, ab, abab, ababab \ldots\}$

3.) $(r/ss^*)^*$

$L((r/ss)^*) = \{\varepsilon, a, bb, aaa, aaabbbb \ldots\}$

LEX :-

1. Lex is a computer program that generates lexical analyzer. It is commonly used with YACC parse generator.
2. It transforms an input into sequence of tokens.
3. It reads the input stream & produces source code as output through implementing the lexical analyzer in C program.

Structure of a LEX File :-

A lex program is separated into three sections by %% delimiters. The formal of LEX source is as follows :-

```
{ definitions }
  %%
{ rules }
  %%
{ user subroutines }
```

→ The definations section defines macros & import header files written in C. It is also possible to write any C code here including declerations of constant, variables & regular definitions.

→ Rules define the statement of the form p1 {action 1} p2 {action 2} ... pn {action n}, where pi discribes the regular expressions & action i describes actions what action the lexical analyzer should takes when pattern pi matches a lexeme.

→ User subroutines are auxiliary procedure needed by the actions. The subroutine can be loaded with lexical analyzer & compiled separately. It contains C statements & functions which are called by rules in rule section.

Example of LEX file :

```
/* Definition section */
%{
  # include <Stdio.h>
%}
%%

/* Rules section */
/* [0-9]+ matches string of one or more
   digits */
```

```
[0-9]+ {
        printf ("Saw an integer: %s\n",
            yytext);
       }
.|\n {
        /* Ignores all other characters */}
%%

/* C code section */

int main (void)
{
   /* Call the lexer, then quit. */
   yylex();
   return 0;
}
```

Given input: abc 123z.!g &* 2gh6

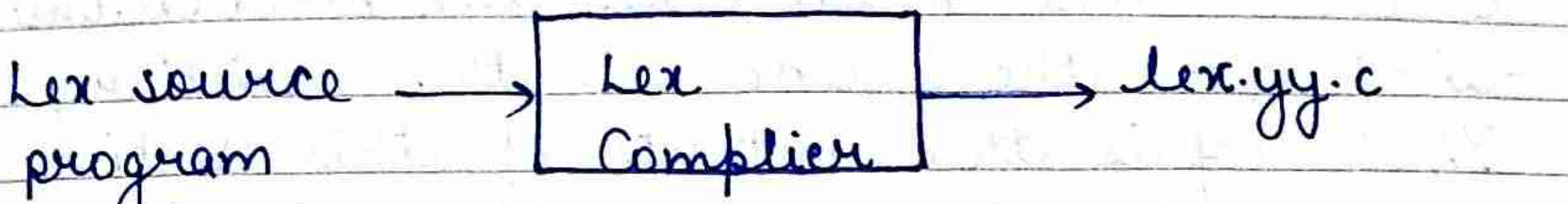
Output: Saw an integer: 123
Saw an integer: 2
Saw an integer: 6

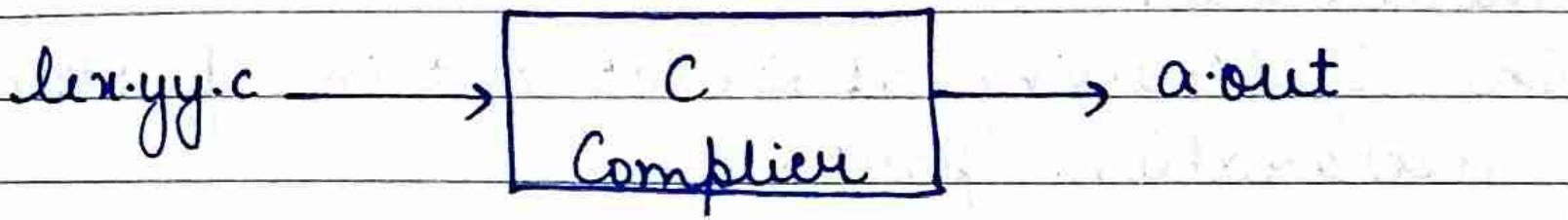
Function of LEX:

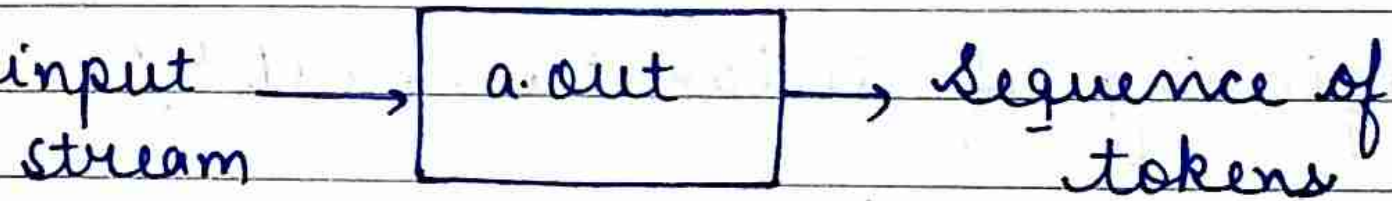
1. Firstly, lexical analyzer creates a program lex.l in lex language. The lex complier runs the lex.l program & produce a C program lex.yy.c.
2. Finally C complier runs the lex.yy.c program

& produces a object program a.out.

3. a.out is lexical analyzer that transforms an input into sequence of token.

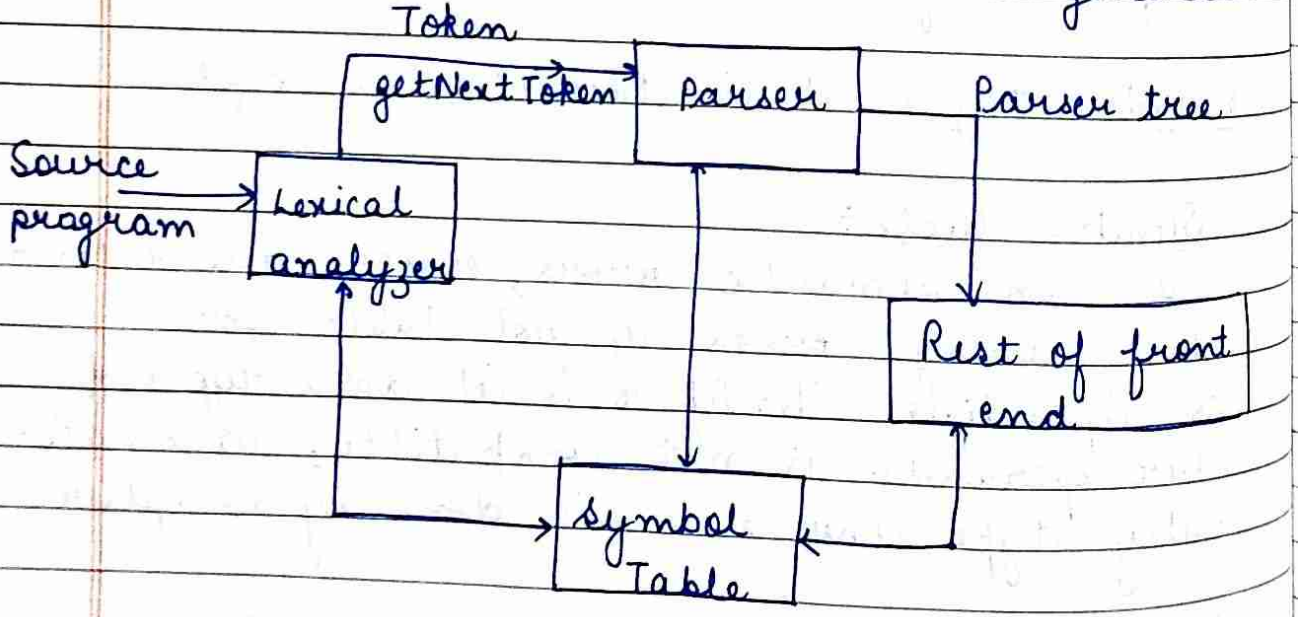






Role of parser:

In syntax analysis phase, a complier verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntatic rules of the language. This is done by parser. The parser obtain string of tokens from lexical analyzer & verifies that the string can be the grammar ~~from~~ for ~~the lexical analyzer~~ source language. It ~~decides~~ detects & reports any syntax error & produce a parse tree from which intermediate code can be generated.



Context-free grammar:

The syntax of any programming language is described by context-free grammar. It is represented by tuples (V, S, Σ, P) where V is set of non-terminals, Σ is set of terminals, S is start symbol &

P is set of production in the form,

$$V \rightarrow (V + \Sigma)^*$$

Parse tree :

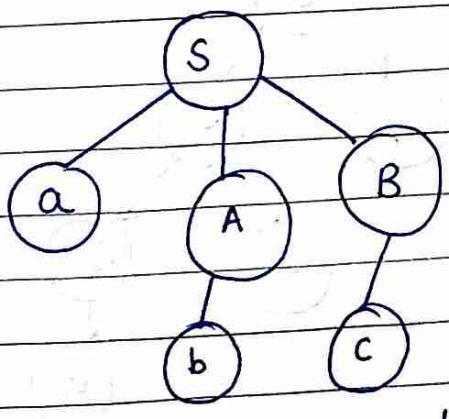
Parse tree or derivation tree is a ordered, rooted tree that represents the syntatic structure of a string according to some context-free grammar.

1. The starting symbol of a grammar must be used as root of the Parse tree
2. Leaves or external nodes represents terminals.
3. Each interior symbol represents an production in the grammar.

Eg: Grammar: $S \rightarrow aAB$

$A \rightarrow b$

$B \rightarrow c$



A parse tree for a CFG $G = (V, \Sigma, P, S)$ is a tree satisfying these conditions:

1. Root has label S, S is start symbol.
2. Each vertex of parse tree has a label which can be variable (V), terminal (Σ) or ε(null).

3. If $A \rightarrow C_1 C_2 \ldots C_n$ is a production, then $C_1, C_2 \ldots C_n$ are children of node labelled with A.

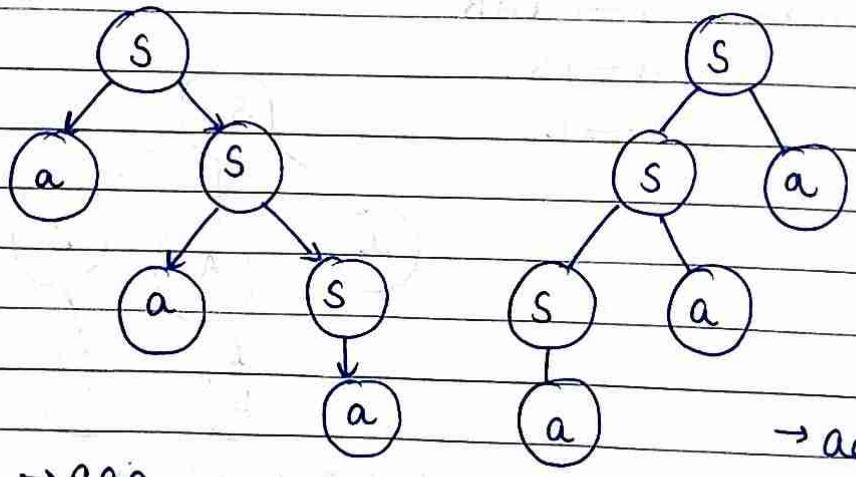
4. Leaf nodes are terminal (Σ) & interior nodes are (V) variable.

Yield: Yield of parse tree is concatenation of labels of the leaves in the left to right ordering.

Ambiguity: A grammar that produces more than one parse tree for some sentence is said to be ambigious.

E.g: $S \rightarrow aS \mid Sa \mid a$

For string → aaa



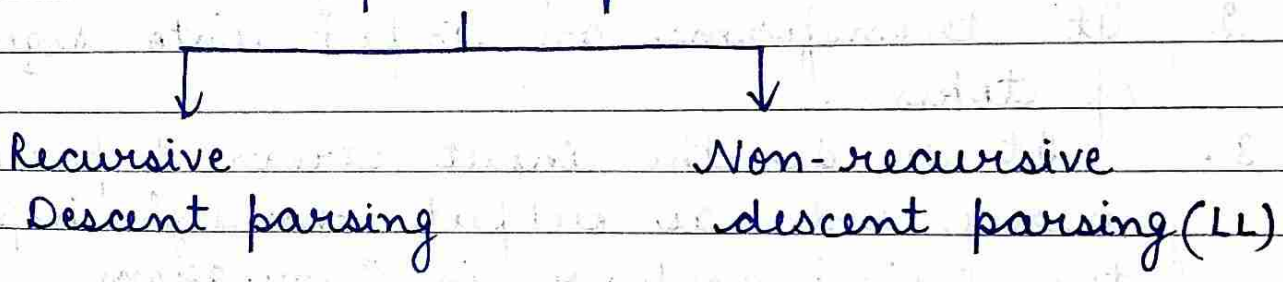
→ aaa → aaa

Parsing: The process of deriving the string from the given grammar is known as parsing (derivation).

Depending on how derivation is done, we have 2 kind of parsers :-

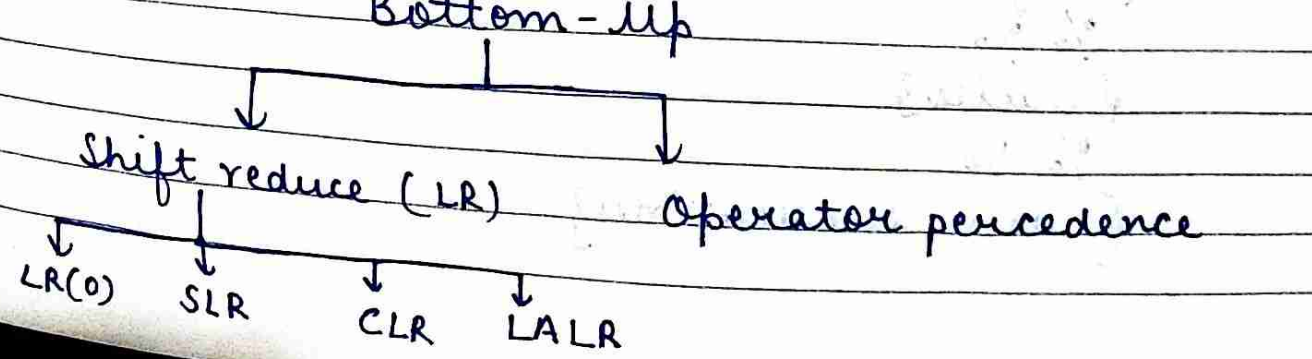
1. Top-Down Parsing :-

Top-down parsing attempts to build the parse-tree from root-to-leaf. The top down parser starts from the start symbol & proceed to the string. It follows left most derivation. In left most derivation, the left most non-terminal in each sentimental is always choosen.

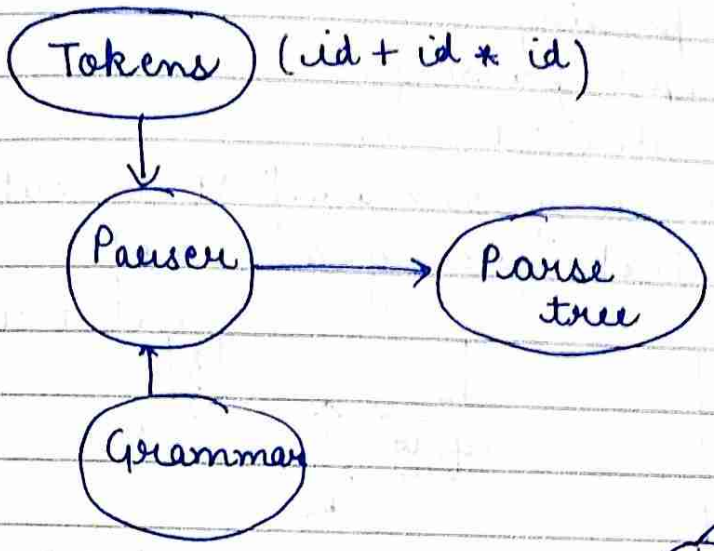


2. Bottom-Up parsing :-

It attempts to build the parse-tree from leaf-to-root. It starts from string & proceed to the start symbol. It follows right-most derivation. In right-most derivation, the right most non-terminal in each sentimental is always choosen.



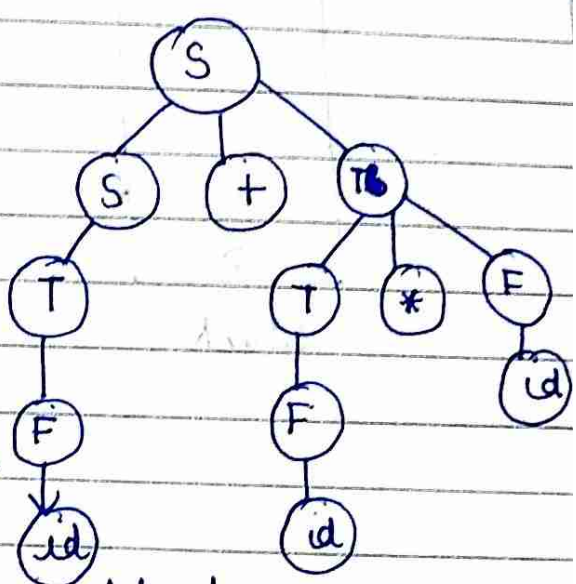
Syntax analysis :



$S \rightarrow S+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$



[Top to down & left to right.]

If the yield of the parse tree isn't matched with the input, it means there is some error.

Top-Down
- → Root to leaf
- → Uses left-most derivation.
- → e.g. → Recursive descent parser.

Bottom-Up
- → leaf to root.
- → Uses right-most derivation.
- → e.g. → Shift reduce, LR parser.

First & Follow :

1. First (A) : It is collection of terminal symbols which are first letters of strings derived from A.

a.) If x is terminal, then
first (x) = {x}.

b.) If x is non-terminal & $x \rightarrow y_1 y_2 \dots y_K$ is the production for $K \geq 1$, the first (x) will be

b1.) If Y is terminal then,

Firist (x) = Firist (y_1, y_2, y_3) = { y_1 }

b2.) If y_1 is non-terminal then
If y_1 doesn't derive to an empty string i.e. if first (y_1) doesn't contain ε then
Firist (x) = Firist (y_1, y_2, y_3) = first (y_1)

b3.) If y_1 contains ε then
First (x) = First (y_1, y_2, y_3) =

First (y) - {ε} ∪ First (y_2, y_3)

Similarly, First (y_2, y_3) = first (y_2) if it doesn't contain ε
If first (y_2) contain ε, then
First (y_2, y_3) = First (y_2) - {ε} ∪ First (y_3)

Similarly, this method will be repeated for further grammar symbols i.e. for y_4, y_5 ... y_k.

c.) If x is non-terminal & $x \to a\alpha$ then First (x) = a.

d.) If $x \to \varepsilon$, then first (x) → ε.

Eg.÷ 1. $S \to ABC \mid ghi \mid jkl$
$A \to a \mid b \mid c$
$B \to b$
$D \to d$

Firist (D) → d
Firist (B) → b
Firist (A) → a, b, c
Firist (S) → a, b, c, g, k

(ii) $S \rightarrow ABC$
$A \rightarrow a \mid b \mid \varepsilon$
$B \rightarrow c \mid d \mid \varepsilon$
$C \rightarrow e \mid f \mid \varepsilon$

First (C) → e, f, ε
First (B) → c, d, ε
First (A) → a, b, ε

First (S) → First (ABC)

First(A) contains ε, So, First (S) → First(A)-{ε} U First (BC)

First (S) → a, b U First (BC)

First (B) contains ε, So, First (BC) = First (B) - {ε} U First (C)

First (BC) = (c, d) U (e, f, ε)
= c, d, e, f, ε

So, First (S) → a, b, c, d, e, f, ε.

(iii) $E \rightarrow TE'$
$E' \rightarrow *TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow \varepsilon \mid +FT'$
$F \rightarrow id \mid (E$

First (F) → id, (.

Furist (T') → ε | +.

Furist (T) = Furist (F) = id, (.

Furist (E') → *, ε.

First (E) → Furist (TE') → Furist (T) = id, (.
[Furist(T) doesn't contain ε.]

iv.) S → (L) / a
L → SL'
L' → , SL' / ε

Furist (S) → (, a
Furist (L) → Furist (S) → (, a
Furist (L') → , , ε.

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

Furist (C) → h, ε
Furist (B) → g, ε

Furist (A) → d, Furist (BC)

Furist (B) contains ε, So, Furist (BC) → Furist (B) − {ε} ∪ Furist (C)
→ g, h, ε.

Furst (A) → d, g, h, ε.

Furst (S) → Furst (ACB), Furst (CbB), Furst (Ba)

Furst (ACB) → d, g, h, ε.
Furst (CbB) → h, b.
Furst (Ba) → g, a.

So, Furst (S) → a, b, d, g, h, ε.

2. Follow (A):

Follow(A) is defined as collection of terminal symbols that occur directly to the right of A.

Rules:

a.) If S is the start symbol, Follow(S) → $
b.) If production is of form,

$A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$

i.) If Furst (β) contains ε, then

Follow(B) = Furst (β) − ε ∪ Follow(A)

∵ when β derives ε, then the terminal after A will follow B.

c.) If production is in form,
$A \rightarrow \alpha B$ then

→ * Follow never contains ε

Date
Page

Follow (B) = { Follow (A) }

e.g.

1) S → AaAb | BbBa
A → ε
B → ε

Follow (S) → $, a, b
Follow (A) → a, b
Follow (B) → b, a.

2) S → ABC
A → DEF
B → ε
C → ε
D → ε
E → ε
F → ε

Follow (A) = First (BC)
First (B) = ε
So, follow (A) = First (C
First (C) = ε
∴, follow (A) = follow (S) = $

3) S → (L) | a
L → SL'
L' → , SL' | ε.

Follow (S) → $
Follow (L) →)
Follow (L') →).

4) S → ACB | CbB | Ba
A → da | BC
B → g | ε
C → h | ε

1.) Follow (C) → b , First (B) , Follow(A)

First (B) → g, ε

So, Follow (C) → b, g, Follow (S), Follow(A)

2.) Follow (A) → First (c) → h, ε

∴ Follow (A) = ~~Foll~~ First (B) ∪ h.

Follow(A) → $, g, h.

Follow (S) → $.

∴ Follow (C) → b, g, h, $

3.) Follow (B) → a, $, First (c)

→ a, $, h, Follow(A)

⇒ a, $, g, h.

5.) E → TE'

E' → +TE' | E

T → FT'

T' → *FT' | E

F → (E) | id

i.) Follow (E) → Follow(E') , Follow (T') ,) , $

Follow (E') = $,) , Follow (T')

Follow (T) → + , (, id

Follow (T') → + , (, id

Follow(F) → First(T') → *, First(E) → (, id, *

Final :-

Follow(E) → $,), +, (, id.

Follow(E') → $,), +, (, id

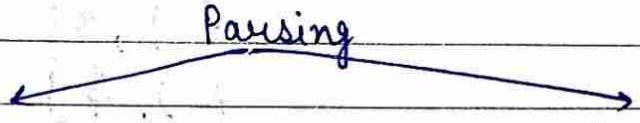
Follow(T) → +, (, id

Follow(T') → +, (, id

Follow(F) → (, id, *

6.) E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Follow(E) → $,)
Follow(E') → $,)
Follow(T) → +, $,)
Follow(T') → +, $,)
Follow(F) → *, +, $,).



Top-Down Parsing
Predictive Parsing
LL(k) Parsing
Left recursion
Left factoring.

Bottom-Up
Shift left reduce
LR(k) Parsing

Predictive Parsing:

1. Construct the parsing table from the given grammar.
2. Apply the predictive parsing algorithm to build the parse tree.

Also to construct parsing table,

For each production $A \to \alpha$ do following steps:

1. For each terminal a in first(α), add $A \to \alpha$ to Table[A, a].
2. If ε is in first(α), then add $A \to \alpha$ to Table[A, b] for each terminal b in Follow(A).
3. If ε in first(α) & \$ in Follow(A), then add $A \to \alpha$ to Table[A, \$].

E.g: Grammar :-

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

	First	Follow
E	{ (, id }	{), \$ }
E'	{ +, ε }	{), \$ }
T	{ (, id }	{ +,), \$ }
T'	{ *, ε }	{ +,), \$ }
F	{ (, id }	{ *, +,), \$ }

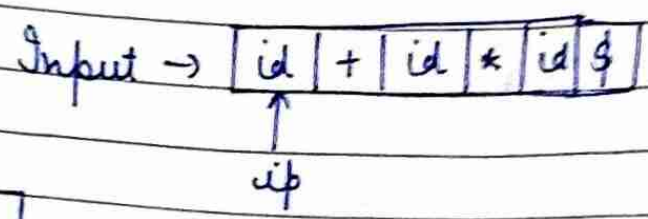
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Apply the predictive parsing algo to construct parse tree.

Set ip to point to first symbol of input string w$.

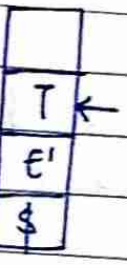
```
repeat
    if Top(Stack) is terminal or $ then
        if Top(Stack) == current(ip) then
            Pop(stack) & ip++.
        else
            Null.
    else
        if Table[X, a] = X → Y1 Y2 Y3 then
        begin
            Pop(Stack);
            Push Y1 Y2 ... Yk onto stack with Y1
            on itop.
            Output the production X → Y1 Y2 .. Yk
```

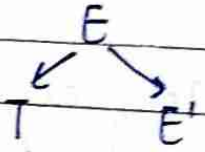
end
else
Null
until (Top(Stack) = $ (i.e. Stack become empty)).



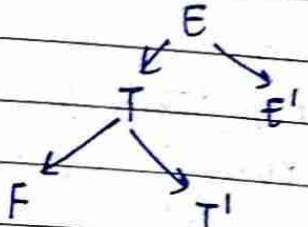
Stack

(i) E is non-terminal,
Table [E, id] = E → TE'

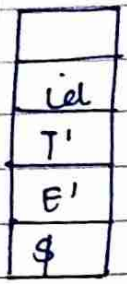


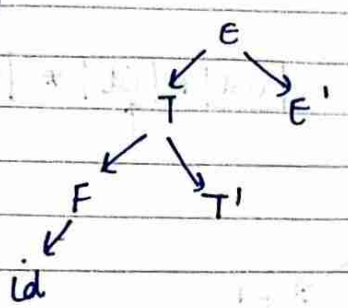


(ii) T is non-terminal
Table [T, id] = T → FT'
Pop (T)
Push (F, T')

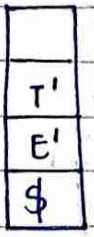


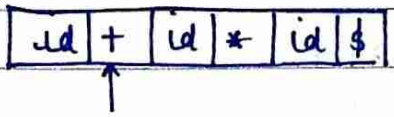
(iii) F is non-terminal,
Table [F, id] = F → id.
Pop(F)
Push(id)



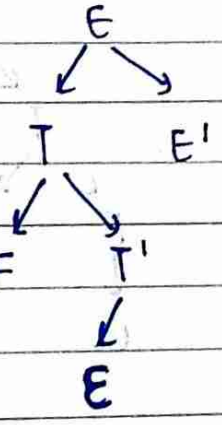


(iv) Top(Stack) = terminal
Top(Stack) = = ip, Yes
Pop(id)
ip++



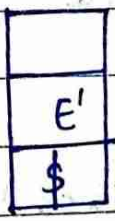


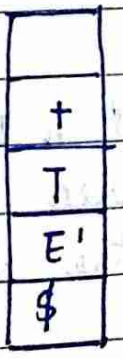
(v) T' is non-terminal
Table [T', +] = T' → ε
Pop(T')
Push(ε)



(vi) ε is non-terminal but null
Table [E, +] = E → $ ε
So, simply Null.

(vii) E' is non-terminal
Table [E', +] → +TE'
Pop(E')
Push(+TE')





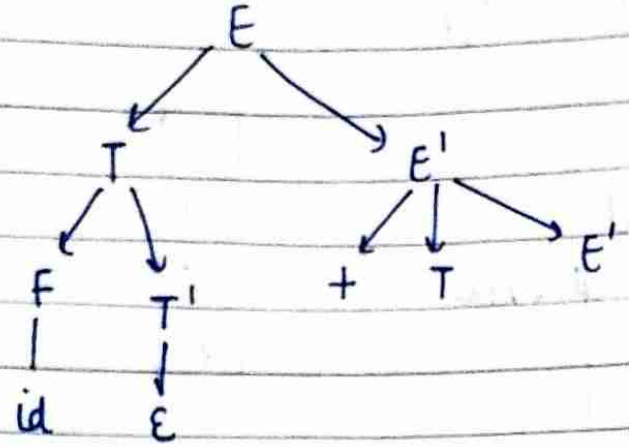
(viii) + is non-terminal

++ == current (ip), Yes

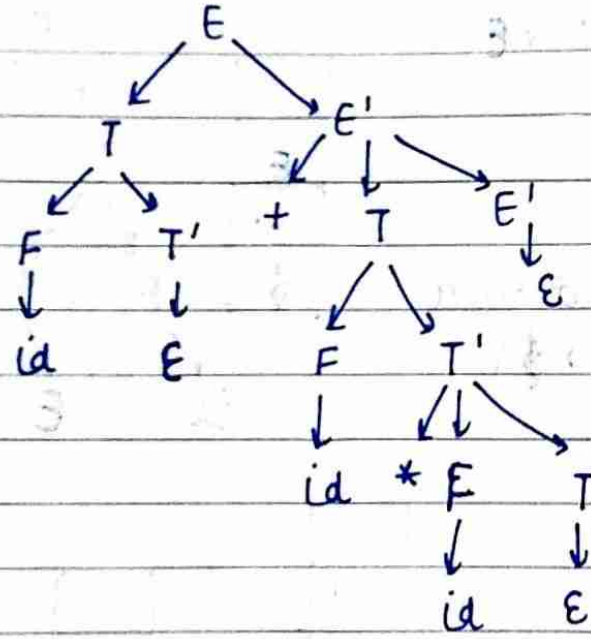
Pop(+)

ip ++

id	+	id	*	id	$



T → FT'



→ Parse tree generated.

When Top (Stack) = $ = input, the parse tree is completed, parser halts & accepts the strings.

Question on predictive Parsing:-

1.) $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow)SL' \mid \varepsilon$

	First	Follow
S	{(, a}	{), $}
L	{(, a}	{)}
L'	{), ε}	{)}

	a	(	)	$
S	S→a	S→(L)		
L	L→SL'	L→SL'		
L'			L'→)SL' L'→ε	L'→ε

Here, we can see that there are 2 productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser.

2.) $S \rightarrow A \mid a$
$A \rightarrow a$

	First ( )	Follow ( )
S	{a}	{$}
A	{a}	{$}

	a	$
S	S → A, S → a	
A	A → a	

This grammar is not feasible for LL(1) Parser.

Note: In above 2 examples, the grammar is ambiguous grammar. So the grammar doesn't satisfy the essential conditions.

LL(k) Parser:

→ This parser parsers from left to right, does a left most derivation. It looks up 'k' symbol ahead to choose its next action.

→ LL(1) grammar is a grammar whose parsing table has no multiply-defined entries.

→ LL(1) grammar is non-ambigous & non-left recursive.

Handle & Handle pruning :

A handle is a substring that matches the body of production i.e. right side of production that can be reduced to left-side of production. So, the reduce of handle represents one step along the reverse of a right most derivation.

If $A \rightarrow \alpha$

Then α is a handle that can be reduced to A.

E.g : Grammar → $S \rightarrow aABe$

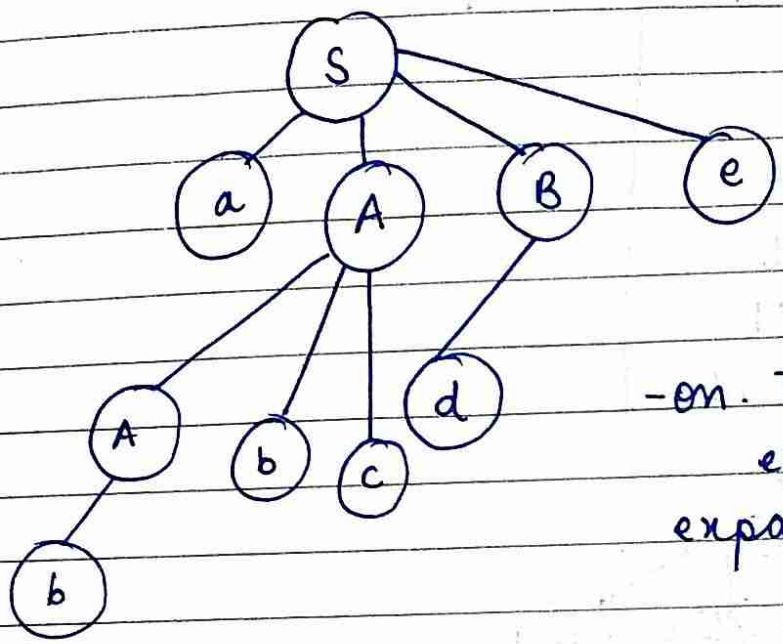
$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input String → abbcde.

Right sentimal form	Handle	Reduction.
a<u>b</u>bcde	b	$A \rightarrow b$
a<u>Abc</u>de	Abc	$A \rightarrow Abc$

a A d e a A B e S	d a A B e	B → d S → a A B e



As, Bottom-up uses right-most derivation. Then firstly, B is expanded & A is expanded.

S
S → a A B e
S → a A d e
S → a A b c d e
S → a b b c d e

So, handle pruning is the process of removing children of left hand side non-terminal from parse.

A right most derivation in reverse can be obtained using handle pruning.

2. E → E+E | E*E | (E) | id

Right Sentimental	Handle	Reduction
id + id * id	id	E → id
E + id * id	id	E → id

E+E * id	E+E	E → E+E
E * id	id	E → id
E * E	E* E	E → E* E
E		

E

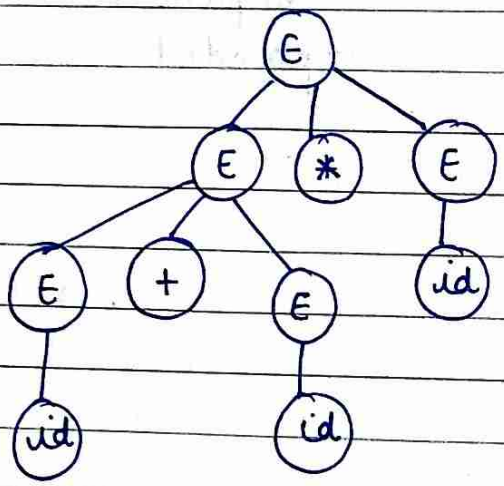
E → E* E

E → E * id

E → E + E * id

E → E + id * id

E → id + id * id



Note :

A certain sentinal form may have many different handles.

Problems :

1. Locate a handle.
2. Decide which production to use (if there are more than two candidate productions)

Shift reduce parsing :

Shift reduce parsing is a process of reducing a string to the start symbol of grammar.
It uses a stack to hold the grammar & an input tape to hold the string.

String $\xrightarrow{\text{reduce to}}$ Start Symbol

Shift reduce parsing performs two actions : Shift & reduce. At shift action, the current symbol in input ~~symbol~~ string is pushed to a stack. At each reduction, the symbols will replaced by the non-terminals. The symbol is right-side of production & non-terminal is left-side of production.

E.g. 1.) $S \rightarrow S+S$
$S \rightarrow S-S$
$S \rightarrow (S)$
$S \rightarrow a$

$w = a1-(a2+a3)$

Stack	Input	Action
$	$a1-(a2+a3)$	Shift
$a1	$-(a2+a3)$	Reduce S→a
$S	$-(a2+a3)$	Shift
$S-	$(a2+a3)$	Shift
$S-(	$a2+a3)$	~~Reduce S→a~~ Shift
$S-(a2	$+a3)$	Shift
$S-(S+	$a3)$	Shift → Reduce S→a

$S-(S+a3	)$	Reduce S→a
$S-(S+S	)$	Shift
$S-(S+S)	$	Reduce S→S+S
$S-(S)	$	Reduce S→(S)
$S-S	$	Reduce S→S-S
$S	$	Accept.

2.) E → E−E
E → E * E
E → id
W = id - id * id

Stack	Input	Action
$	id - id * id $	Shift
$id	- id * id $	Reduce E→id
$E	- id * id $	Shift
$E-	id * id $	~~Redo~~ Shift
$E-id	* id $	Reduce E→id
$E-E	* id $	Reduce E→E-E
$E	* id $	Shift
$E*	id $	Shift
$E*id	$	Reduce E→id
$E*E	$	Reduce E→E*E
$E	$	Accept.

3.) S→TL ; T→ int / float ; L→ L, id / id
W = int id, id

Stack	Input	Action
$	int id, id $	Shift
$int	id, id $	Reduce T→int

$T	id, id$	Shift
$T id	, id $	Reduce L → id
$TL	, id $	Shift
$TL,	id $	Shift
$TL, id	$	Reduce L → L, id
$TL	$	Reduce S → TL
$S	$	Accept.

4. S → 0S0 | 1S1 | 2 W = 10201

Stack	Input	Action
$	10201$	Shift
$1	0201$	Shift
$10	201$	Shift
$102	01$	Reduce S → 2
$10S	01$	Shift
$10S0	1$	Reduce S → 0S0
$1S	1$	Shift
$1S1	$	Reduce S → 1S1
$S	$	Accept.

Operator Percedence:

An operator percedence parser is bottom-up parser that interprets an operator grammar (grammar having restriction that null production is not allowed or no 2 adjacent non-terminals in its right-hand side). Ambigous grammars are not allowed in any parser

except operator percedence.

Rule1 → This parser relies on three percedence relations : $\lessdot$, $\doteq$, $\gtrdot$

1) $a \lessdot b$ This means a "yields percedence to" b.
2) $a \gtrdot b$ This means a " takes percendence over" b.
3) $a \doteq b$ This means a "has same percedence as" b.

Rule 2 : 1.) An identifier is always given the higher percedence than any other symbol.
2.) $ symbol is always given lowest percedence.

Rule 3 : 1.) If two operators have same percedence, then we go by checking their associativity.

Algo steps :

1. Insert $ symbol at beginning & end of input string.
2. Insert percedence operator b/w every two symbols of the string by referring the operatorator percedence table.
3. Start scanning string from L.H.S in forward direction until $\gtrdot$ is encountered.
4. Keep a pointer on that.
5. Start scanning the string from R.H.S in backward direction until $\lessdot$ is encountered.
6. Keep a pointer on that location.

7. Everything that lies in middle of < . > forms the handle.

8. Replace the handle with head of the respective production rule.

Repeat step 3 to 8., until the start symbol is reduced.

E.g.:-

1.) $E \to E+E \mid E*E \mid id$ ⓖ→

$W = id * id + id$

f ↓		id	+	*	$
	id		⋗	⋗	⋗
	+	⋖	⋗	⋖	⋗
	*	⋖	⋗	⋗	⋗
	$	⋖	⋖	⋖	

$ id * id + id $

$ ⋖ id ⋗ * ⋖ id ⋗ + ⋖ id ⋗ $

$ E * E + E $

$ E + E $

$ E $ → $ $

2.) $S \to (L) \mid a$

$L \to L, S \mid S$

$W = (a, (a, a))$

f ↓		a	(	)	,	$
	a		⋗	⋗	⋗	⋗
	(	⋖	⋗	⋗	⋗	⋗
	)	⋖	⋗	⋗	⋗	⋗
	,	⋖	⋖	⋗	⋗	⋗
	$	⋖	⋖	⋖	⋖	

ⓖ →

$(a,(a,a))$

$\$ \lessdot (\lessdot a \gtrdot , (\lessdot a \gtrdot , \lessdot a \gtrdot) \gtrdot) \gtrdot \$$

$\$ \lessdot (S , \lessdot a \gtrdot) \gtrdot \$$

$\$ \lessdot (S , (S , \lessdot a \gtrdot) \gtrdot) \gtrdot \$$

$\$ \lessdot (S , (S , S) \gtrdot) \gtrdot \$$

$\$ \lessdot (S , (S , S) \gtrdot) \gtrdot \$$

$\$ \lessdot (L , (S) \gtrdot) \gtrdot \$$

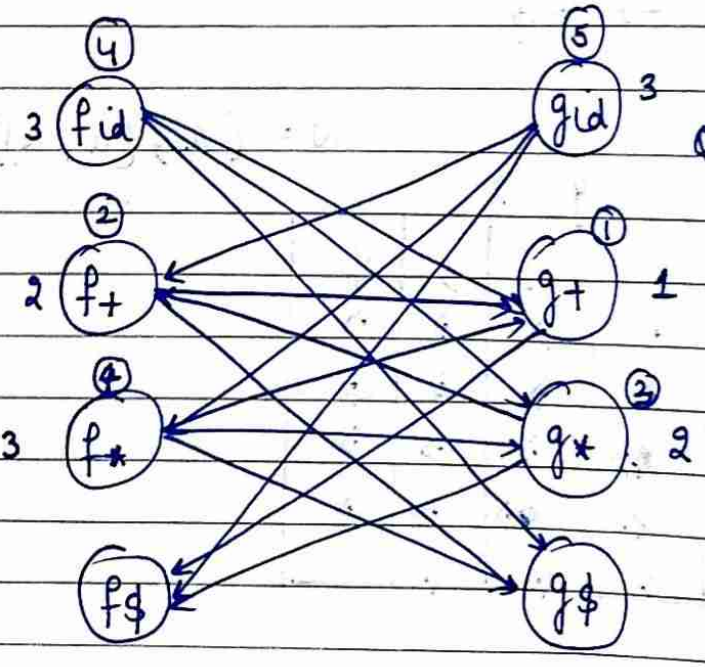
$\$ \lessdot (L , S) \gtrdot \$$

$\$ \lessdot (L) \gtrdot \$$

$\$ \lessdot L \gtrdot \$$

$\$\$$

1.)



Q: Find operator percedence operate functions.

Resulting percedence functions are:-

	id	+	*	$
f	4	2	4	0
g	5	1	2+1	0

→ [refer to longest path from a symbol to $ sign].

Left factoring:-

If R.H.S of more than one production starts with the same symbol, then such a grammar is called grammar with common prefixes. This kind of grammar creates a problem for top down parsers because they can't decide which production rule must be chosen to parse the string. So, we use left factoring.

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for top down parser.

How?

1. We make one production for each common prefixes. The common prefix may be terminal non-terminal or combination of both.
2. Rest of derivation is added by new productions

E.g.:- 1.) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

→ $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$.
$E \rightarrow b$

2.) $A \rightarrow aAB \mid aAC \mid aBc$

$\rightarrow$ $A \rightarrow aA'$
$A' \rightarrow AB \mid Ac \mid Bc$. # Again a left factored grammar
$A' \rightarrow AA_1 \mid Bc$
$A_1 \rightarrow B \mid c$

3.) $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$\rightarrow$ $S \rightarrow bSS' \mid a$
$S' \rightarrow SaaS \mid SaSb \mid b$
$S' \rightarrow SaS_1 \mid b$.
$S_1 \rightarrow aS \mid aSb$

4.) $E \rightarrow T+E \mid T$
$T \rightarrow int \mid int * T \mid (E)$

$\rightarrow$ $E \rightarrow TT'$
$T' \rightarrow +E \mid \varepsilon$.
$T \rightarrow intT_1 \mid (E)$
$T_1 \rightarrow *T \mid \varepsilon$.

5.) $S \rightarrow A$
$A \rightarrow aB \mid aC \mid ab \mid AC$
$B \rightarrow bBC \mid f$
$C \rightarrow g \mid \varepsilon$
$D \rightarrow d \mid \varepsilon$.

→ S → A
A → aA' | AC
A' → B | C | b
B → bBC | f
C → g | ε
D → d | ε.

6.) S → A#0
A → Ad | aB | aC
C → c
B → bBc | r

→ S → A#0
A → Ad | aA'
A' → B | C
C → c
B → bBc | r.

7.) S → abc | abd | ae | f

↳ S → aS' | f
↳ S' → bc | bd | e

Ans: S → aS' | f
S' → bS1 | e
S1 → c | d.

Left recursion : A Grammar G is left recursive if it has production in the form,

A → Aα | β

The above grammar is left recursive because the left of production occuring at first position is also present at right hand side of production.

It can eliminate left recursion by replacing a pair of production with

$$A \to A\alpha / \beta$$

$$\to A \to \beta A'$$
$$A' \to \alpha A' / \varepsilon.$$

E.g: 1. $A \to ABd / Aa / a$

$$B \to Be / b.$$

$$\to A \to aA'$$
$$A' \to BdA' / aA' / \varepsilon$$
$$B \to bB'$$
$$B' \to eB' / \varepsilon.$$

2. $E \to E+E / E*E / a$

$$\to E \to aE'$$
$$E' \to +EE' / *EE' / \varepsilon.$$

3. $E \to E+T / T$
$T \to T*F / F$
$F \to id$

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

$T \rightarrow FT'$

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

$F \rightarrow id$

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

$L \rightarrow L, S / S.$

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

$L \rightarrow SL'$

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

5. $S \rightarrow SOS1S / 01$

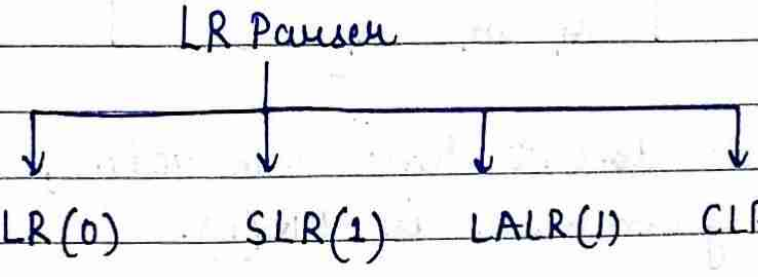
$S \rightarrow 01S'$

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

LR Parser:-

LR(k) parser is type of bottom-up parser. It is used to parse large class of grammars.

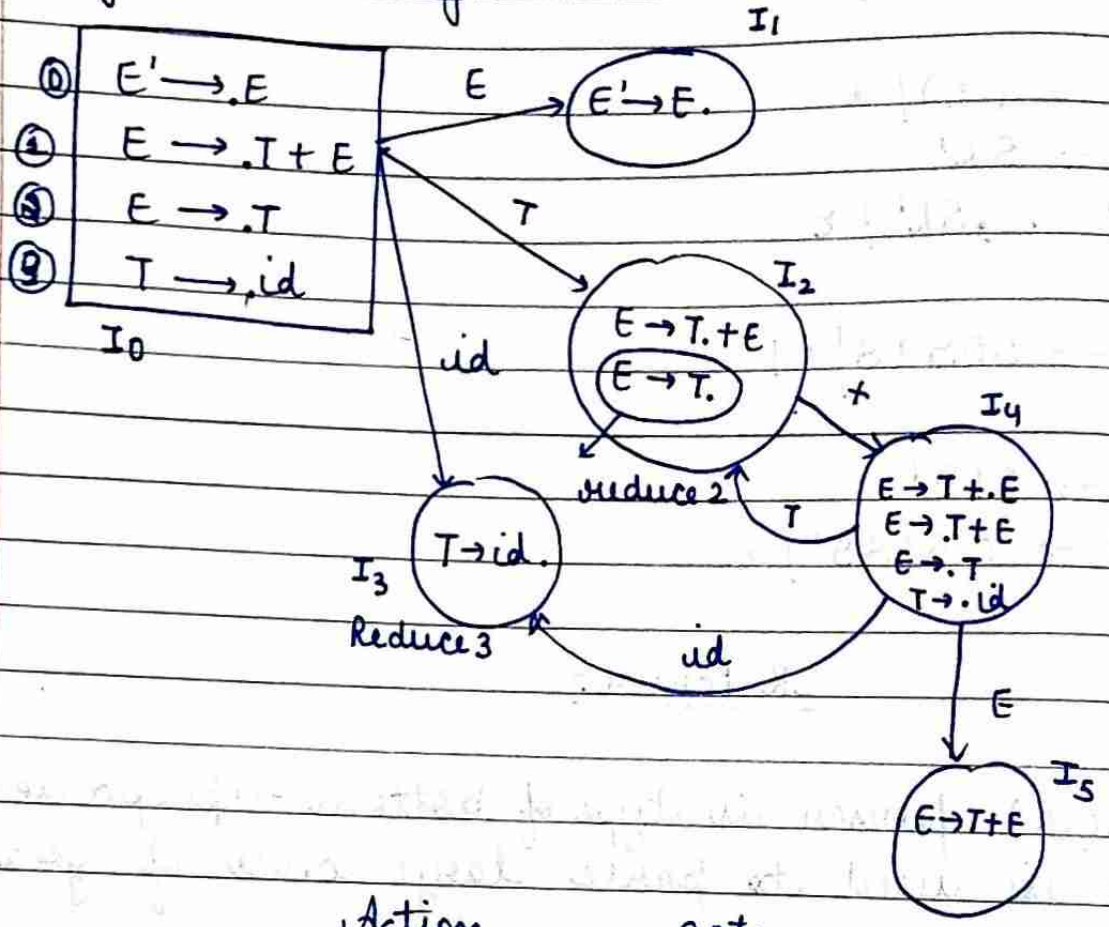
In LR(k) parsing, "L" stands for left-to-right scanning of the input. "R" stands for constructing right-most derivation. "K" stands for number of input symbols of look ahead used to make number of parsing decision.



LR(0) :-

1.) $E \rightarrow T+E \mid T$
$T \rightarrow id$

Augument the grammar :-

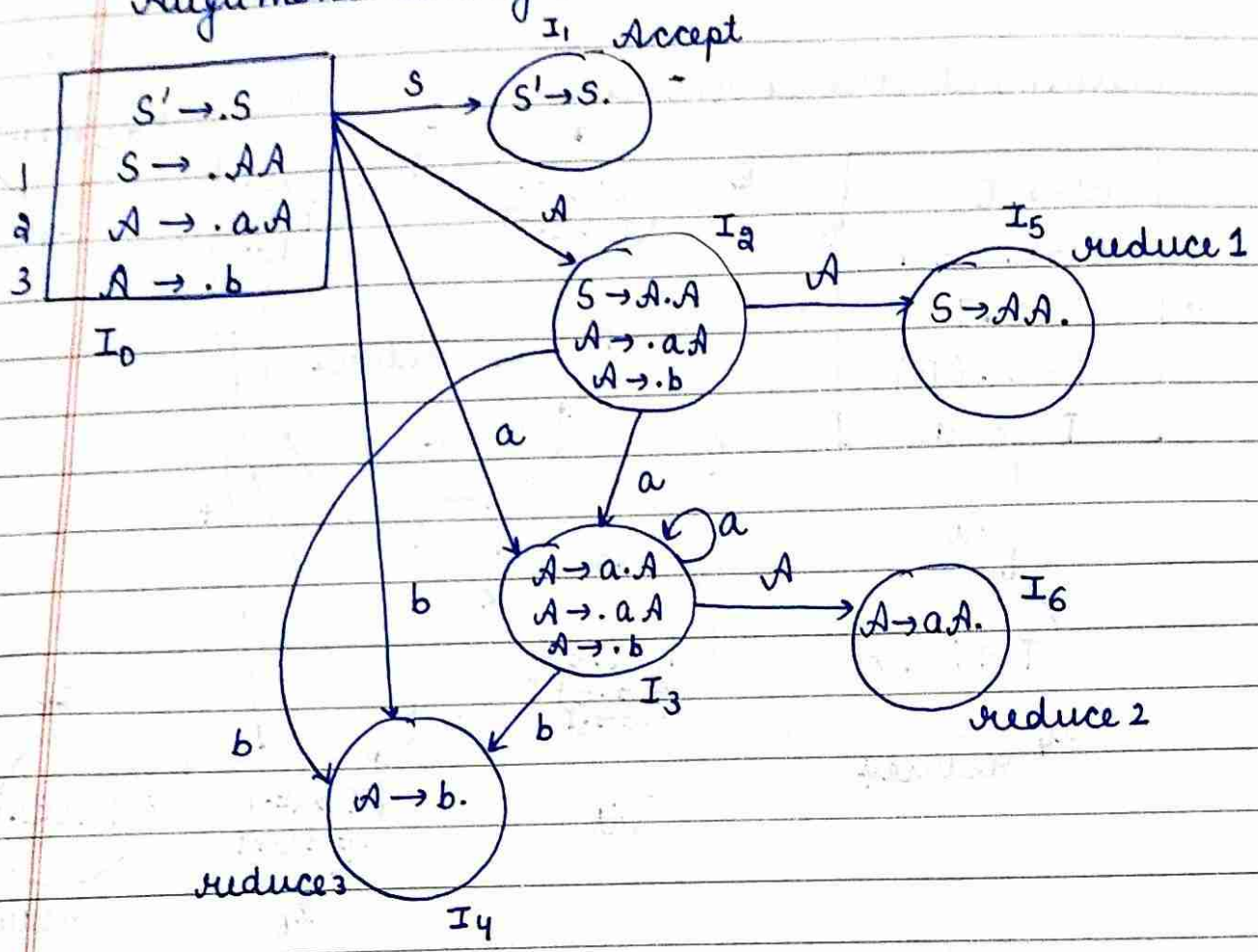


State	Action			goto	
	id	+	$	E	T
0	S_3			1	2
1		Accept			
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4				5	2
5	r_1	r_1	r_1		

Each cell doesn't have one value, hence the given grammar is LR(0).

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

Augument the grammar:

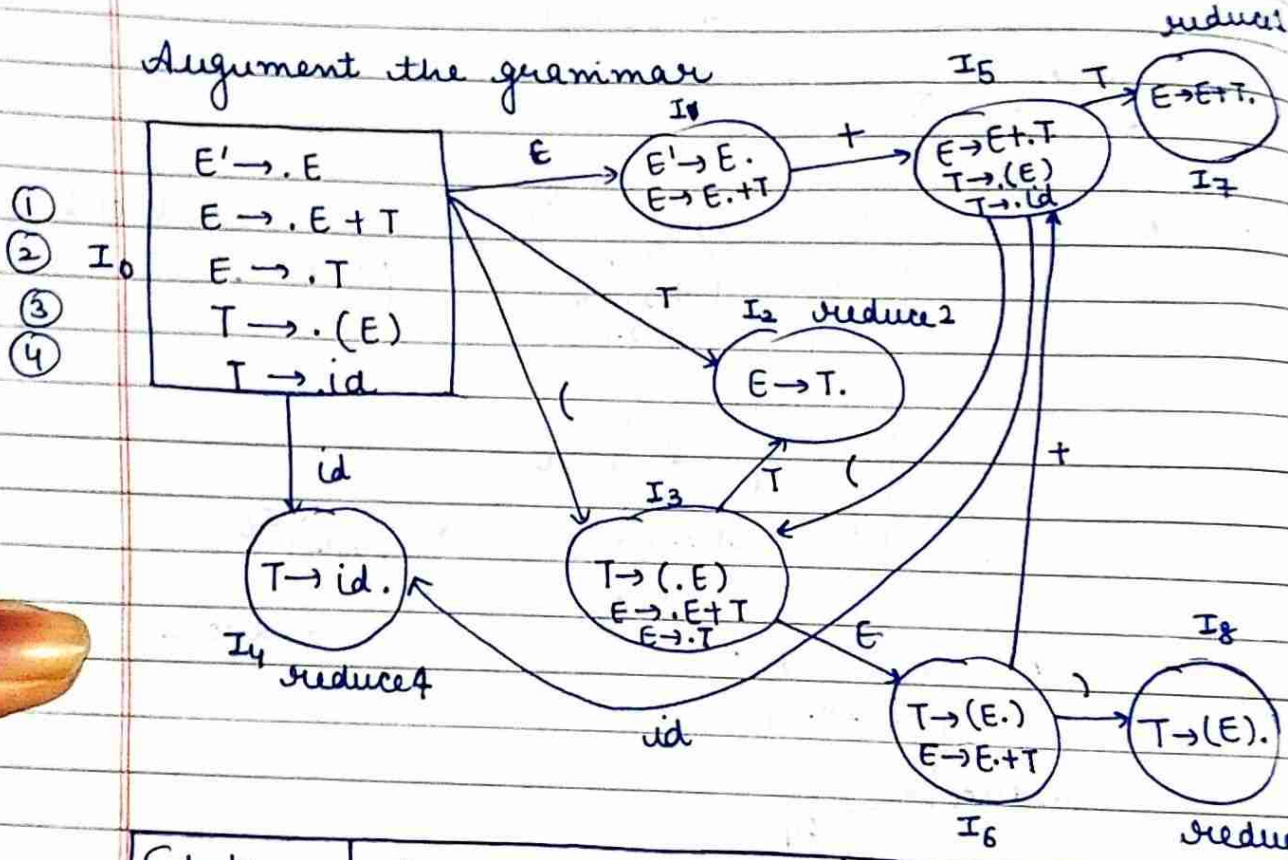


State	Action			goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1	-	Accept	-	-	-
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

The given grammar is LR(0).

3. $E \to E+T$
$E \to T$
$T \to (E)$
$T \to id$

Augument the grammar



State	(	+	)	id	$	E	T
0	S_3			S_4		1	2
1		S_5					
2	r_2	r_2	r_2		r_2		
3						6	2
4	r_4	r_4	r_4		r_4		
5	S_3			S_4			7
6		S_5	S_8				
7	r_1	r_1	r_1		r_1		
8	r_3	r_3	r_3		r_3		

Action goto

Augmented grammar:

If G is a grammar with starting symbol S, then G' (augmented grammar) is a grammar with a new starting symbol S' & productions S → S'. The purpose of this new starting production is to indicate the parser when it should stop parsing. The '.' symbol indicates 2 things : the left side of '.' has been read by a complier & the right side of '.' is yet to be read by a complier.

Steps for constructing LR parsing table :

1. Write augmented grammar.
2. LR(0) collection of items to be found.
3. Defining 2 functions : goto (list of non-terminals) and action (list of terminals) in the parsing table.

Note: LR(0) cononical item is a production G with dot at some position on right side of production. LR(0) items are useful to indicate how much of the input has been scanned up to a given point in process of parsing.

LR(0) Table:

1. If a state is going to some other state on terminal then it corresponds to a shift move.
2. If a state is going to some other state

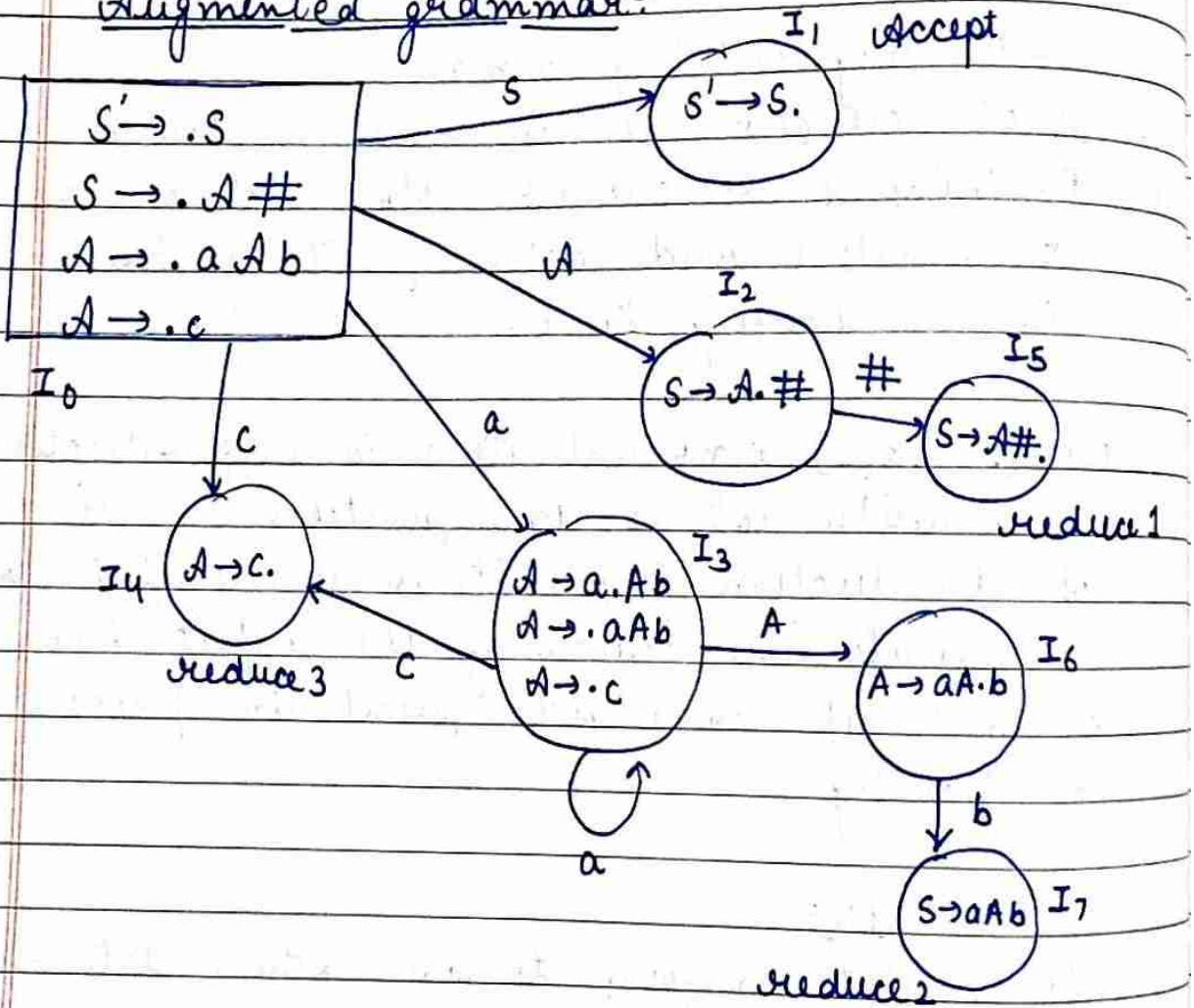
on variable then it corresponds to go to move.

3. If a state contain final item in a particular row then write the reduce node completely.

e.g.:

4. $S \rightarrow A\#$
   $A \rightarrow aAb \mid c$

Augmented grammar:



State	a	c	b	#	$	S	A
0	S_3	S_4				1	2
1			Accept				
2				S_5			
3	S_3	S_4					6
4	R_3	R_3	R_3	R_3	R_3		
5	R_1	R_1	R_1	R_1	R_1		
6			S_7				
7	R_2	R_2	R_2	R_2	R_2		

SLR(1) :- (Simple LR)

1. SLR is simple LR. It is the smallest class of grammar having few number of states.
2. SLR is very easy to construct & it is similar to LR parsing. The only diff. between SLR(1) and LR(0) parser is that in LR(0) parsing table, there is chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminals of state but in SLR 'reduce' is entered only in "Follow" of L.H.S of production in terminating state. This is called SLR(1) collection of items

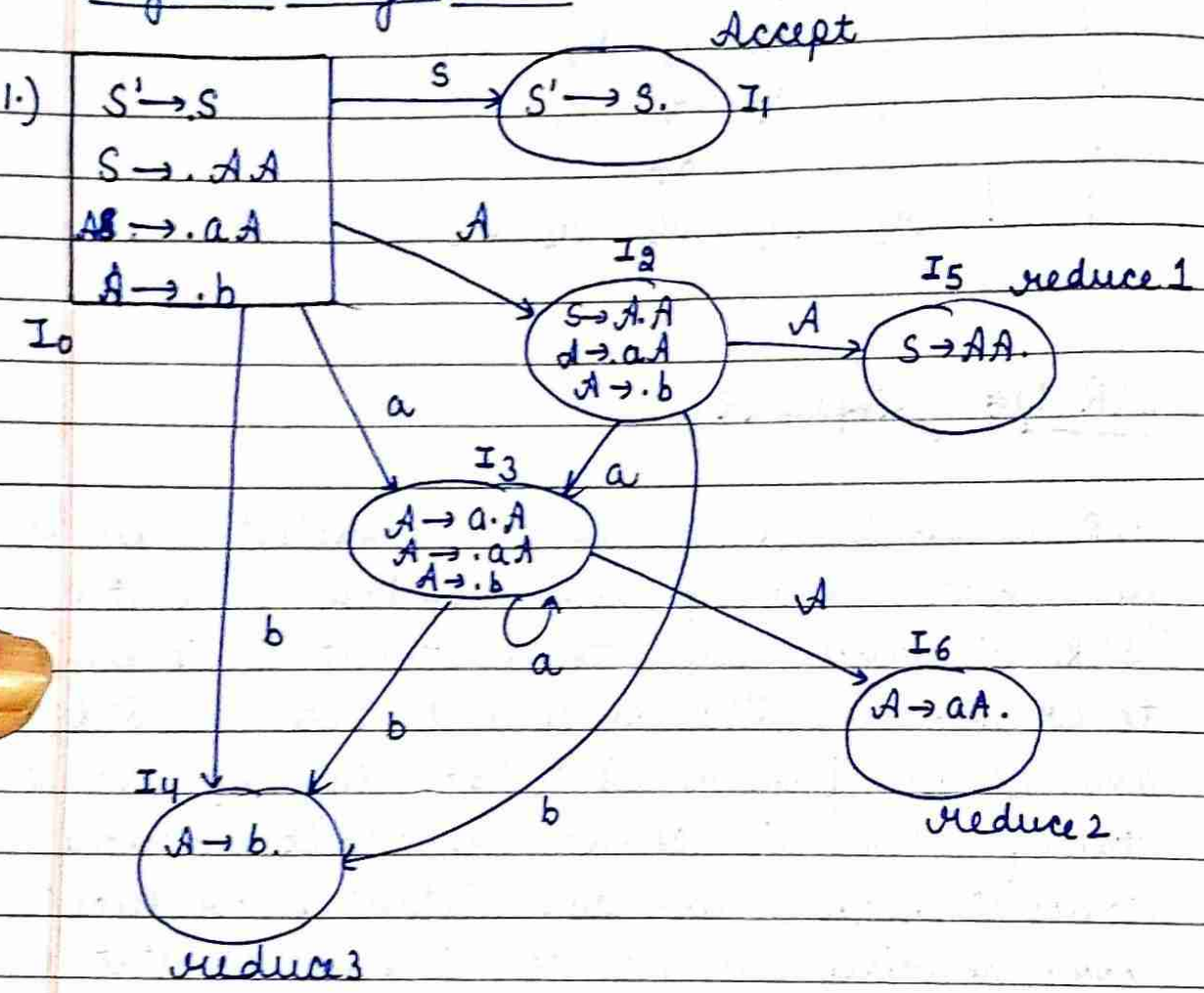
Steps :-

1. Writing augmented grammar.
2. LR(0) collection of items to be found.
3. Find Follow of L.H.S of production.
4. Defining 2 functions :- 1.) goto [list of non-terminals], 2.) action [terminals].

e.g. :

	Follow
$S \to AA$	$S \to \{\$\}$
$A \to aA \mid b$	$A \to \{\$, a/b\}$

Augmented grammar :

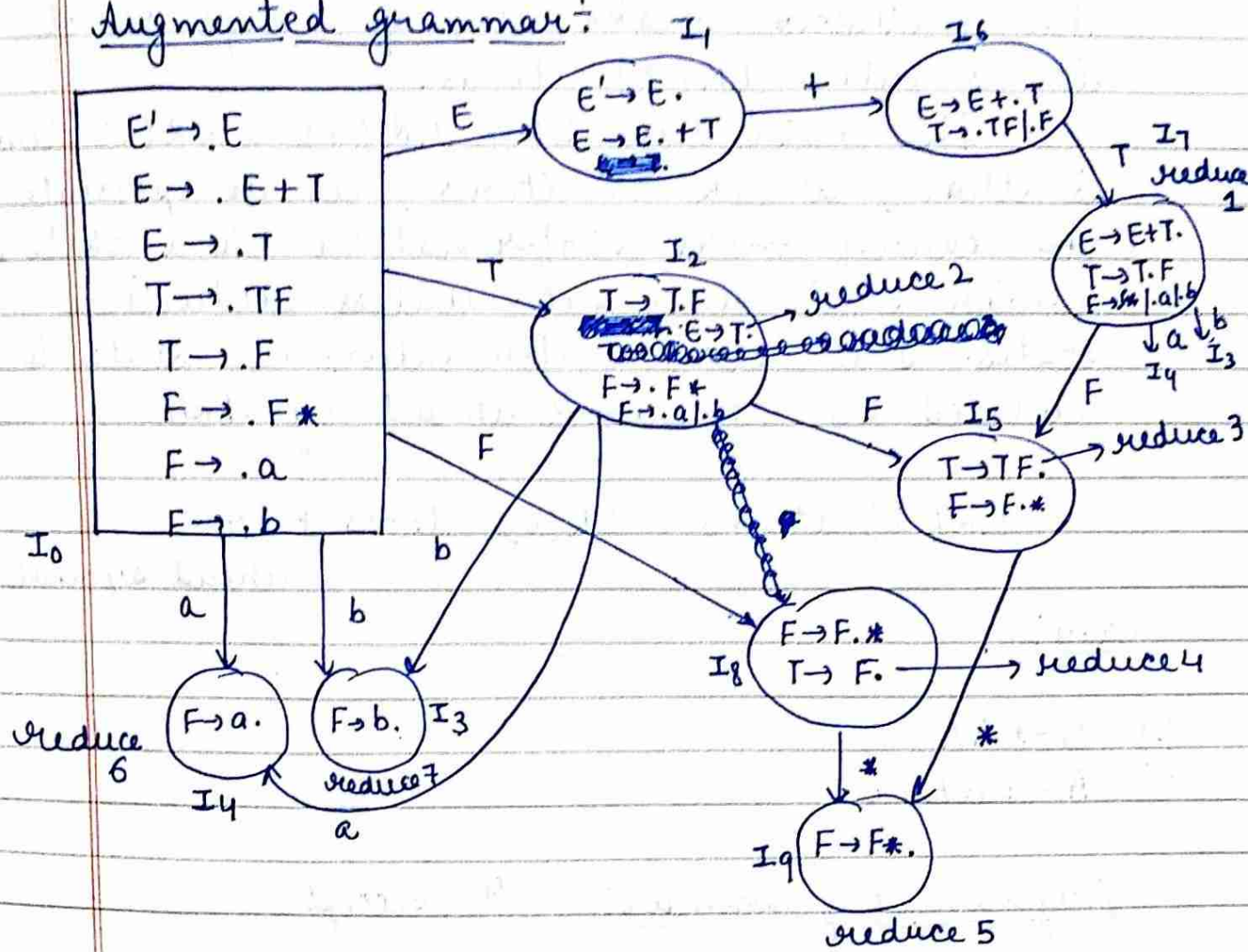


State	a	b	$	S	A
0	S_3	S_4		1	2
1	-	Accept	-	-	-
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5			R_1		
6	R_2	R_2	R_2		

2. $E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b$

Follow : $E \to \{+, \$\}$
$F \to \{*, \$, a, b\}$
$T \to \{\$, a, b\}$

Augmented grammar :



State	+	*	a	b	$	E	T	F
0			S_4	S_3		1	2	8
1	S_6							
2	r_2		S_4	S_3	r_2			5
3	r_7	r_7	r_7	r_7	r_7			
4	r_6	r_6	r_6	r_6	r_6			
5	S_9	r_3	r_3	r_3	r_3			
6							7	
7	r_1	S_8	S_4	S_3	r_1			5
8	S_9	r_4	r_4	r_4	r_4			
9	r_5	r_5	r_5	r_5	r_5			

CLR(1) parser:-

CLR parser stands for ~~com~~ canonical LR parser. It is more powerful LR parser. It makes use of lookahead symbols. This methods make use of large set of items called LR(1) items.

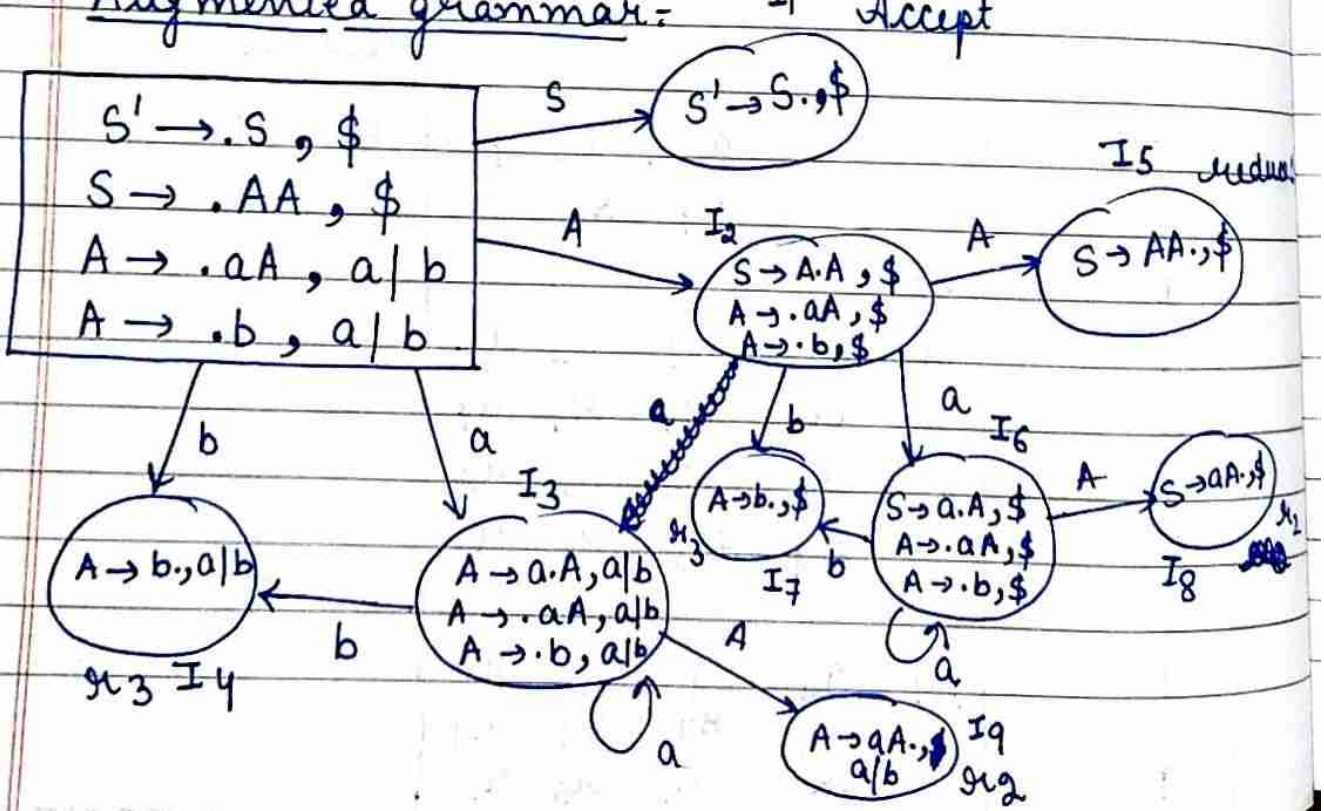
The main diff. b/w LR(0) & LR(1) items is that, in LR(1) items, it is possible to carry more information in a state, which will rule out useless reduction states. This extra information in state is carried by the look ahead symbol.

LR(1) items = LR(0) items + look ahead symbol

e.g:-

1.) $S \rightarrow AA$
$A \rightarrow aA \mid b$

Augmented grammar:-



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

Note :- [The reductions will be written in the corresponding look ahead symbols of that state.]

2.) $S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c$

$B \to c$

Augmented grammar:

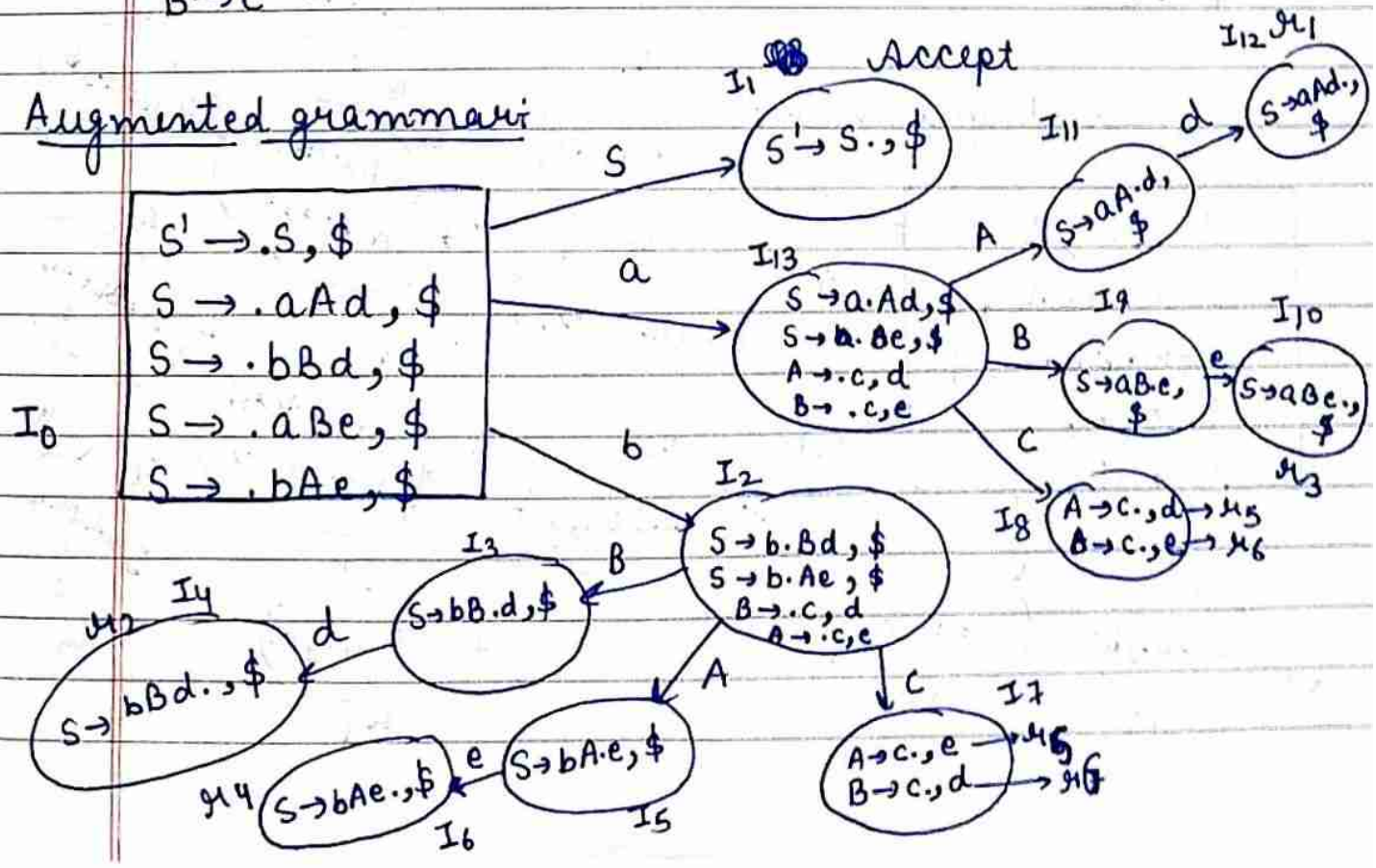
$S' \to .S, \$$

$S \to .aAd, \$$

$S \to .bBd, \$$

$S \to .aBe, \$$

$S \to .bAe, \$$



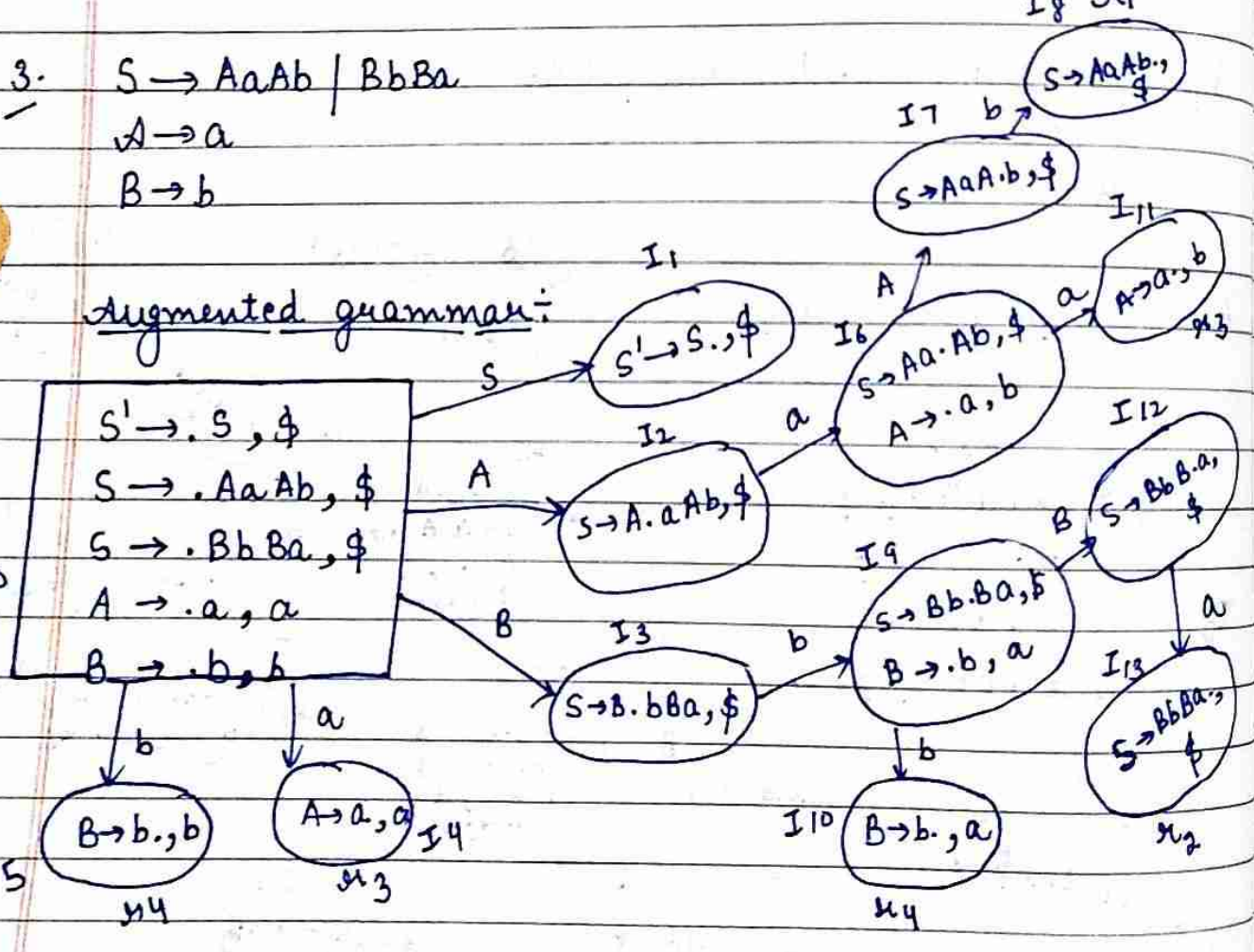
State	a	b	c	e	d	$	S	A	B
0	S_{13}	S_{12}					1		
1	-	-	Accept	-	-	-	-	-	-
2			S_7					5	3
3				S_4					
4						r_2			
5			S_6						
6						r_4			
7			r_5		r_6				
8			r_6		r_5				
9			S_{10}						
10						r_3			
11				S_{12}					
12						r_1			
13			S_8				11		9

3. $S \rightarrow AaAb \mid BbBa$

$A \rightarrow a$

$B \rightarrow b$

Augmented grammar:



State	a	b	$	S	A	B
0	S_4	S_5		1	2	3
1	-	-	Accept	-	-	-
2	S_6					
3	~~r_3~~	~~S_9~~				
4	r_3	~~r_3~~				
5		r_4				
6	S_{11}				7	
7		S_8				
8			r_1			
9		S_{10}				12
10	r_4					
11		r_3				
12	S_{13}					
13			r_2			

LALR(1) :-

LALR is lookahead LR parser. It is powerful parser which can ~~large~~ handle large class of grammar. The size of CLR table is quite large as compared to other parsing table. LALR reduced the size of this table. otherwise LALR is similar to CLR. The only difference is, it combines the similar states of CLR parsing table into one single state.

So, LALR also makes use of LR(1) canonical items.

LR(1) = LR(0) + look ahead symbols.

For example,

In last example, I have done in CLR. The state I_4 & I_{11} is same i.e. both are reduced 3 state but the difference is in their look aheads. So, we can combine these states like.

$I_{411} \rightarrow A \rightarrow a., a/b.$

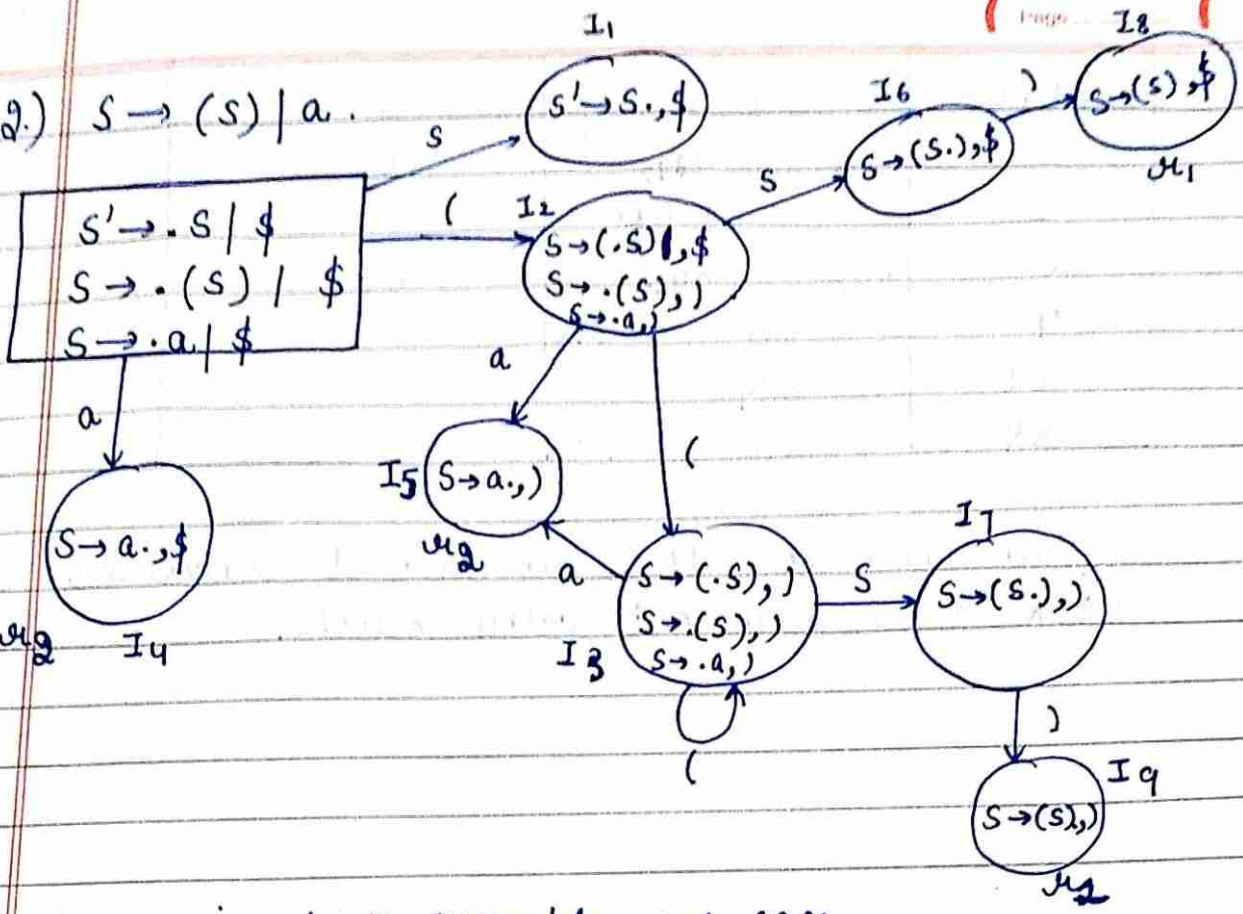
Similarly, the state I_5 & I_{10} can be combined to form,

$I_{510} \rightarrow B \rightarrow b., b/a.$

Table of LALR :-

State	a	b	$	S	A	B
0	S_{411}	S_{510}		1	2	3
1	-	Accept	-	-	-	-
2	S_6					
3		S_9				
411	r_3	r_3				
510	r_4	r_4				
6	S_{411}				7	
7		S_8				
8			r_1			
9		S_{510}				12
12	S_{13}					
13			r_2			

2.) $S \rightarrow (S) \mid a$.



Now, in above example, we can see.

→ I_2 & I_3 are same except lookahead symbols. So, they can be combined.

I_{23}

$S \rightarrow (.S), \$ \mid)$
$S \rightarrow .(S),)$
$S \rightarrow .a,)$

→ I_4 & I_5 are both r_2 states. So,

$S \rightarrow a., \$ \mid)$.

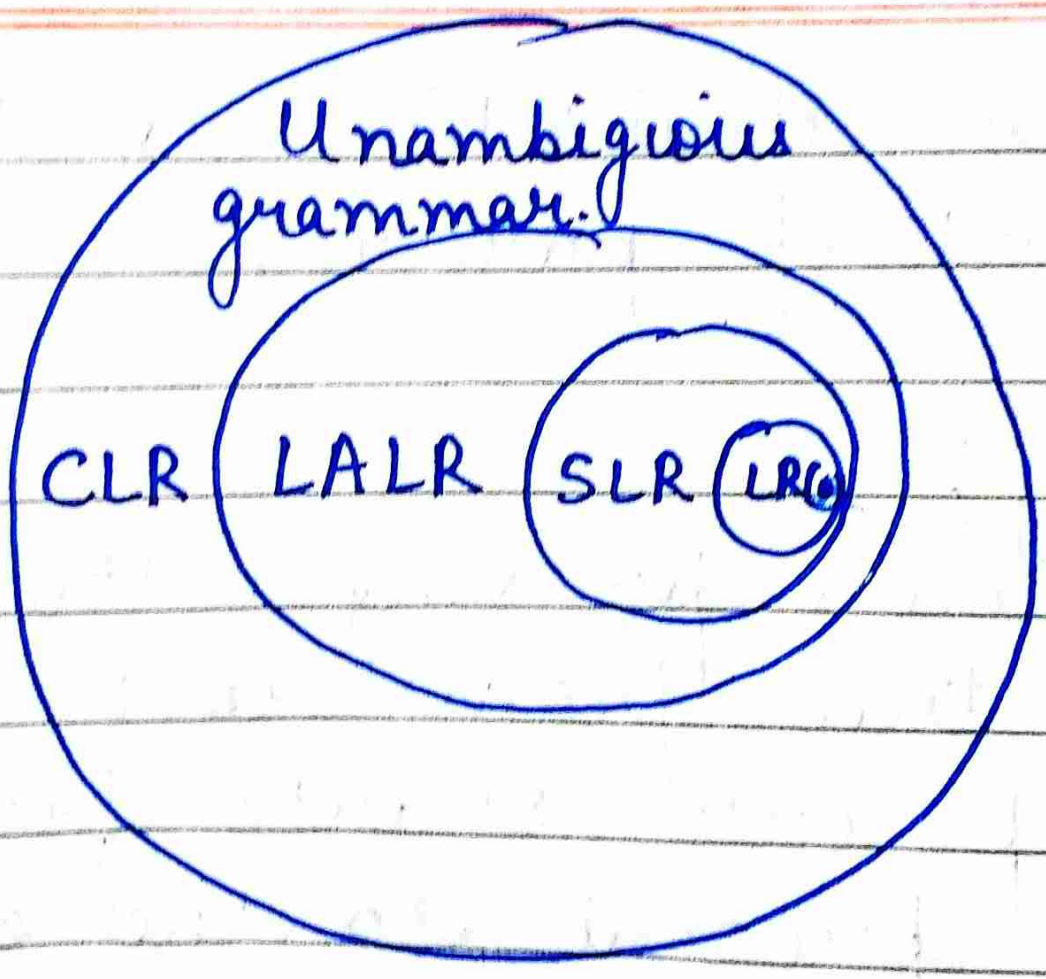
→ I_6 & I_7 are same,

$S \rightarrow (S.), \$ \mid)$

→ I_8 & I_9 are both r_1 states. So, combined state : $S \rightarrow (S)., \$ \mid)$.

State	(	)	a	$	S
0	S_{23}		S_{45}		1
1	-	-	Accept	-	-
23	S_{23}		S_{45}		67
45		r_2		r_2	
67		S_{89}			
89		r_1		r_1	

So, we can see the number of states in LALR has reduced very much.


Unambiguous grammar.
CLR
LALR
SLR
LR(0)

Error recovery techniques:

Error is the operation which cause an abnormal change or problems in a program.

In the phase of compilation, the errors are sent in the form of messages. This is called error handling process. Error handler performs all the functions that are supposed to performed while detecting an error in a complier. It perform 3 operations:

- Error detection
- Error reporting
- Error recovery

Types of errors:

- Compile-time errors
- Runtime error
- Logical error

Compile-time errors occur while compiling a program & they occur due to incorrect syntax, like when we miss a semicolon at end of program or when we miss putting brackets at end of any loop or conditional statement etc.

Runtime errors occurs during execution of a program. These are the errors which arise due to invalid user input. For e.g. → An code

Panic mode Eg: int a, $gch, b, $b;

// After int a, $gch, b, $b; // parser discards input symbol at a time

to take input of type integer, but the user entered the character input gives an error at runtime.

logical errors are types of errors in which the given code is unreachable or there is a presence of infinite loop.

Error recovery techniques:

1. Panic mode:

This is the most basic & easiest way of error recovery. In this, as soon as the parser detects an error anywhere in the program it immediately discard the rest of the remaining part of code from that particular place where error is found by not processing it.

Advantages: 1. It is easy to implement.
2. It never goes in an infinite loop.

Disadvantage:
1. A huge amount of input is discarded without processing that & it does not detect any more errors as it is not processing it further.

2. Statement mode / Phase level recovery:

In this mode, on discovering an error,

it performs recovery by correcting the remaining input so that remaining input statement allows parser to move further. So, the parser can easily continue its job. In this correction includes, removing extra semicolon, fitting missing semicolons, replacing the comma with semicolon etc. This type of correction is decided by the complier developer.

E.g. →

```
int a, b
// After recovery
int a, b; // Semicolon is added by complier
```

Advantage:- 1. It is widely used in many error reparing complier.

Disadvantage:- 1. While doing replacement in the program, it may fall into infinite loop.

3. Error production:-

If a complier designer has some knowledge of errors that are very common then these types of errors are recovered by augmented (adding extra productions) grammar for corresponding language with error productions that produce incorrect results. If error production is used

during parsing, we can generate appropiate error message to indicate error has been recognized in input.

It is very difficult to maintain, because if we change grammar, then it become necessary to change corresponding productions.

E.g.:- Input string → abcd

Grammar:- $S \rightarrow A$

$A \rightarrow aA \mid bA \mid a \mid b$

$B \rightarrow cd$

The input string is not obtained by above grammar, so we need augmented grammar.

$S' \rightarrow SB$

$S \rightarrow A$

$A \rightarrow aA \mid bB \mid a \mid b$

$B \rightarrow cd.$

Now, string can be obtained.

Advantages:-

1. Syntatic phase errors are generally recovered by error production.

Disadvantage:-

1. It is difficult to maintain.

4. Global correction:-
In this method, the parser judges or considers the full program & try to find the closest matching of this which is free from errors. When an incorrected statement is tackled, it creates a parse tree for error free statement which is closest to input. Global corrections increase time & space requirement at parsing time, so it can't be implemented pratically.

Advantage:- It makes very few changes in processing an incorrect input.

Disadvantage:- It is theoretical concept.

5. Symbol table:-
In semantic errors, errors are recover-ed by using ~~counts~~ symbol table for corresponding identifier & if data types of two operands is not compatible, automati-cally type conversion is done by complier.

Advantage:- It allows basic type-conversion, which we generally do in real-life.

Disadvantage:- Only implicit type conversion is possible.

YACC: (Yet another complier complier)

YACC is a computer program for the Unix operating system. It is an LALR(1)[look-ahead, left-to-right, right most derivation producer with the 1 lookahead] parser generator. YACC was originally designed for being complemented by lex.

The input of YAAC is rule or grammar & the output is C program.

Structure of YAAC program:

Declarations
%%
rules
%%
Auxillairy functions
%%

1. Declarations section consists of two parts :
1.) C declarations.
2.) YACC declarations.
C declarations contains all the declarations u
-sed for the C code in the auxilliary sec
YACC copies the contents of this section in
generated y.tab.c file without any
modification.
The following abstract outline of structu
of declaration part :

```
/* Beginning of declaration part */

%{
    // C declarations
%}

    // YAAC declarations
/* End of declaration part */
%%
```

The YACC declarations part comprises of declarations of token (returned by lexical analyzer). The parser reads the tokens by invoking yylex() function.

2. Rules consists of 2 parts i) the production part and ii) action part. The syntax of any programming language will be specified in form of Context Free Grammar.
A rule in YACC is of the form :

production_head : production_body { action in C};

%%

/* Rules section begins */

/* Rules section ends */

%%

Example of a production:

expr : expr '+' expr } → We can give names to these token which are defined in declaration section.

head body

expr → non-terminals

'+' → terminal

2. Action part of rule consists of C statements enclose within { and }. These statements are executed when the input is matched with body of production & a reduction takes place.

E.g.

expr: DIGIT{ printf ("NUM %d", pos);}

3. The auxiliary functions section contains the difinations of three mandatory functions main(), yylex(), yyerror() & can also contain user-defined functions. The main function must invoke yyparse() to parse the input.

Date
Page

For e.g. YACC program:

```
%{
    /* Declaration part */

    #include <stdio.h>
    #include <conio.h>

    /* function declaration */
     void print_operator (char op)

    /* Variable declaration */
       int pos = 0;
%}

    /* YACC declarations */
%token DIGIT
%left '+'
%left '*'
%%

    /* Rules section */

Start : expr '\n'
 { exit (1); }
        ;
expr: expr '+' expr
 { print_operator ('+'); }
|expr '*' expr
 { print_operator ('*'); }
```

```
    | DIGIT
{ printf ("NUM %d", pos); }
    ;
%%

/* Auxiliary function section */

void print_operator(char c)
 switch (c)
 {
  case '+':
       printf ("Plus");
       break;
  Case '*':
       printf ("MULTIPLY");
       break;
 }
 return;
}

yyerror( char const *s)
{
  printf ("error %s", s);
}
yylex ()
 {
  char c;
  c = getchar();
  if ( isdigit (c)) {
       pos++;
       return DIGIT;
  }
```

```
    else
      {
        return c;
      }
  }

main()
{
  yyparse();
   return 1;
}
```

Function :