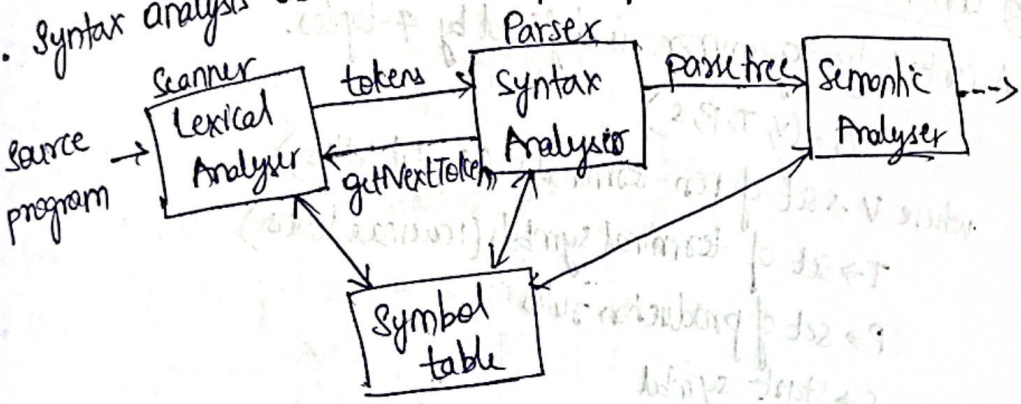
Unit-2

Syntax Analysis

① Role of Syntax Analyser (Parser):

- Syntax analysis is also called parsing.



- The lexical analyser reads the source program character by character and produces a token.
- The symbol table stores the information about tokens (tokenName, token attribute).
- Syntax Analyser (Parser) receives token and sends request (getNextToken) to lexical Analyser. The lexical analyser responds to the request by sending another token.
- Passer for any grammar that takes tokens produced by the lexical analyser as i/p, produces either a parse tree or an error message as output.
- The syntax analyser checks whether tokens are following the language syntax rules or not.
- If they aren't following the language syntax rules, then error handler reports these errors to the user.

- If the tokens follow the language syntax rules then it will generate a parse tree / syntax tree.

② Context-Free Grammars (CFG) and Writing Grammar

A context-free grammar is defined by 4-tuples.

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

where V → set of non-terminals (capital letters)
T → set of terminal symbols (lowercase letters)
P ⇒ set of production rules
S ⇒ start symbol

- A production rule is represented in form of

$$A \rightarrow \alpha$$

where A is non-terminal

Conventions:

- Terminals: Terminals are symbols from which strings are formed.
  - Lowercase letters i.e, a, b, c
  - Operators i.e, +, −, *
  - Punctuation symbols i.e, comma, parenthesis
  - Digits i.e, 0, 1, 2, ..., 9
  - ~~Boldface letters i.e, id, if~~ Boldface letters i.e, id, if.
- Non-terminals: are variables that denote a set of strings.
  - Uppercase letters. i.e, A, B, C
- Start symbol: Start symbol is the head of the production stated first in the grammar.

- Production: Production is of the form LHS → RHS (or) head → body, where head contains only one non-terminal and body contains a collection of terminals and non-terminals.

Derivation:

- A derivation is the process of applying a sequence of production rules to derive a string.
- Process of derivation always starts from start symbol.
- Derivation is classified into two types:
  1. Left-most derivation
  2. Right-most derivation

Leftmost derivation:

- In the left-most derivation, the input is scanned and replaced with the production rule from left to right. So, in left most derivates we read the input string from left to right.

Example: Production rules:

S = S+S
S = S-S
S = a/b/c

Input: a-b+c

The leftmost derivation is

S = S+S
S = S-S+S
S = a-S+S
S = a-b+S
S = a-b+c

Right-most derivation:

In right-most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivation we read the input string from right to left.

Example: Production rules

$S = S+S$
$S = S-S$
$S = a|b|c$

Input: a-b+c

The right-most derivation is:

$S = S-S$
$S = S-S+S$
$S = S-S+c$
$S = S-b+c$
$S = a-b+c$

Parse tree / derivation tree)

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It's the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows these point:
  - All leaf nodes have to be terminals.
  - All interior nodes have to be non-terminals
  - In-order traversal gives original input string.

Yield of the tree:

- The yield of the tree is the collection of leaf nodes from left to right.

Ex: Consider the following grammar.

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

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

$B \rightarrow 0A \mid 1S$

Construct left-most derivation and parse tree for the following.

(i) 0101 (ii) 1100101

Sol:

(i) 0101 (LMD)

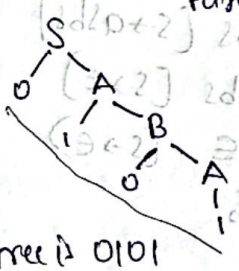
$S \rightarrow 0\underline{A}$

$S \rightarrow 01\underline{B}$

$S \rightarrow 010\underline{A}$

$S \rightarrow 0101$

Parse tree:



Yield of tree is 0101

(ii) 1100101 (LMD)

$S \rightarrow 1\underline{B}$

$S \rightarrow 11\underline{S}$

$S \rightarrow 110\underline{A}$

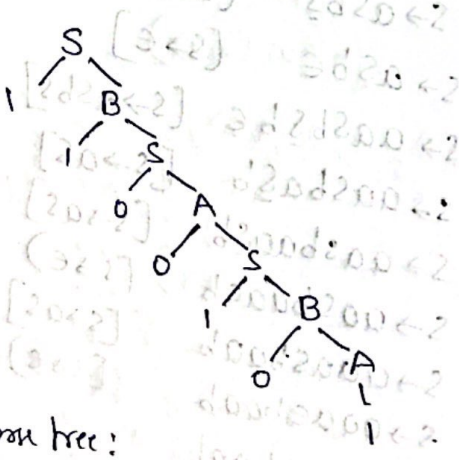
$S \rightarrow 1100\underline{S}$

$S \rightarrow 11001\underline{B}$

$S \rightarrow 1100101\underline{A}$

$S \rightarrow 1100101$

Parse tree:



Yield of parse tree:

1100101

Eg: Design LMD & RMD for the string aaabaab.

grammar is $S \to aS / aSbS / \epsilon$

Check whether the string aabb is accepted or not.

Sol: aaabaab

LMD:

$S \to aS$ [$S \to aS$]

$S \to aaS$ [$\because S \to aS$]

$S \to aaaSbS$ [$\because S \to aSbS$]

$S \to aaa\epsilon bS$ [$S \to \epsilon$]

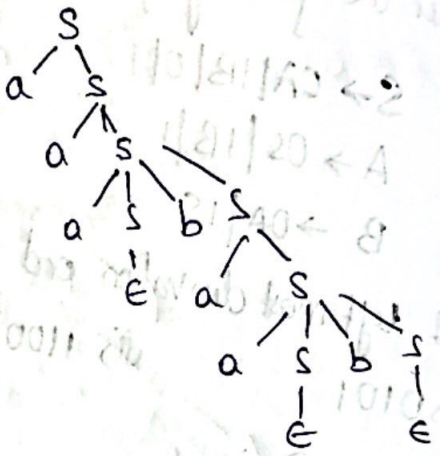
$S \to aaabaS$ [$S \to aS$]

$S \to aaabaaSbS$ [$S \to aSbS$]

$S \to aaabaa\epsilon bS$ [$S \to \epsilon$]

$S \to aaabaaab\epsilon$ [$S \to \epsilon$]

$S \to aaabaab$



aaabaab

RMD:

$S \to aSbS$ [$S \to aSbS$]

$S \to aSb\epsilon$ [$S \to \epsilon$]

$S \to aaSbSb\epsilon$ [$S \to aSbS$]

$S \to aaSbaSb$ [$S \to aS$]

$S \to aaSbaaSb$ [$S \to aS$]

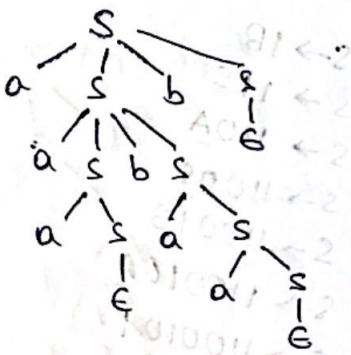
$S \to aaSbaa\epsilon b$ [$S \to \epsilon$]

$S \to aaaSbaab$ [$S \to aS$]

$S \to aaa\epsilon baab$ [$S \to \epsilon$]

$S \to aaabaab$

Parse tree



aaabaab

① check whether the string aabb is accepted or not.

$S \to aAB \mid bA \mid \epsilon$

$A \to aAb \mid \epsilon$

$B \to bB \mid \epsilon$

Sol:

$S \to aAB$	$[S \to aAB]$
$S \to aaAbB$	$[A \to aAb]$
$S \to aa\epsilon bB$	$[A \to \epsilon]$
$S \to aabbB$	$[B \to bB]$
$S \to aabb\epsilon$	$[B \to \epsilon]$
$S \to aabb$	is accepted.

② Consider the grammar

$S \to (L) \mid a$

$L \to L,S \mid S$

a) what are the non-terminals, terminals & start symbol

b) find LMD, RMD & parse trees for the following

(i) (a,a)

(ii) (a,(a,a))

② what

Sol: (a) Non-terminals = {S, L}

terminals = {(,), a, ,}

start symbol = S

(b) LMD: (a,a)

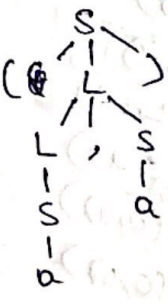
$S \to (L)$

$S \to (L,S)$

$S \to (S,S)$

$S \to (a,S)$

$S \to (a,a)$



(a,a)

PMD: (a,a)

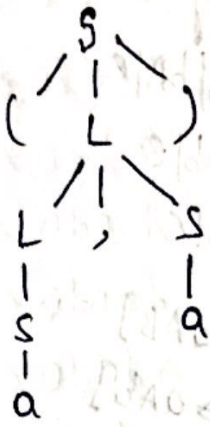
$S \to (L)$

$S \to (L, S)$

$S \to (L, a)$

$S \to (S, a)$

$S \to (a, a)$



(a,a)

(ii) LMD: (a, (a,a))

$S \to (L)$

$S \to (L, S)$

$S \to (S, S)$

$S \to (a, S)$

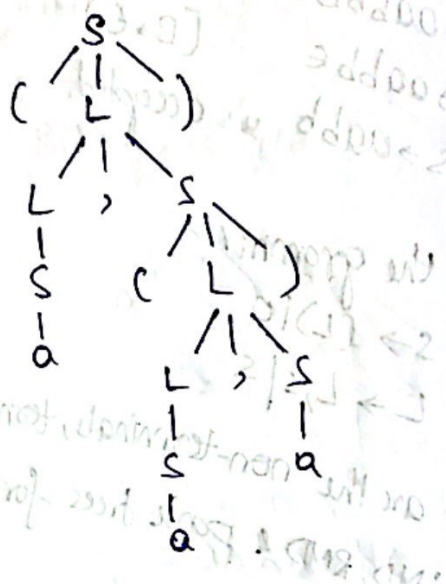
$S \to (a, (L))$

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

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

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

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



(a,(a,a))

RMD: (a, (a,a))

$S \to (L)$

$S \to (L, S)$

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

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

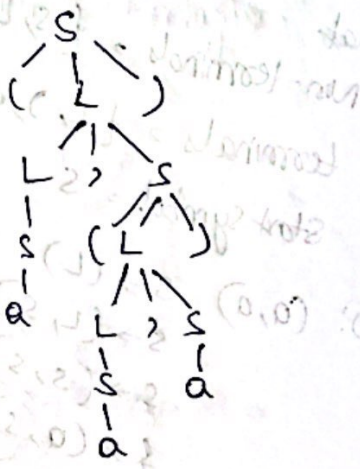
$S \to (L, (L, a))$

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

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

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

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



(a,(a,a))

Ambiguity:

. A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

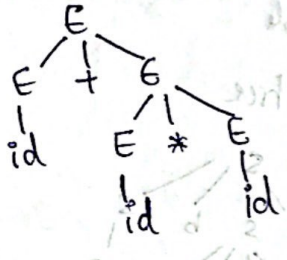
Eg: $E \to E+E \mid E*E \mid (E) \mid id$ Here id is terminal symbol

Sol: string: id+id*id

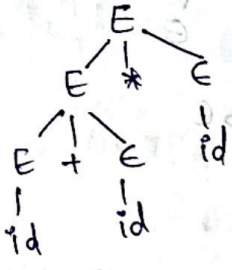
$E \to E+E$

LMD: $E \to E+E$
$E \to id+E*E$
$E \to id+id*E$
$E \to id+id*id$

Parse tree:



RMD: $E \to E*E$
$E \to E*id$
$E \to E+E*id$
$E \to E+id*id$
$E \to id+id*id$



So, is ambiguous grammar.

Eg: Show that the following grammar is Ambiguous

$S \to aSbS$
$S \to bSaS$
$S \to \epsilon$

Sol: LMD:

$S \to aSbS$

$S \to abSaSbS$

$S \to abaSbS$

$S \to ababS$

$S \to ababa$

RMD

$S \to aSbS$

$S \to aSbSbS$

$S \to abaSbS$

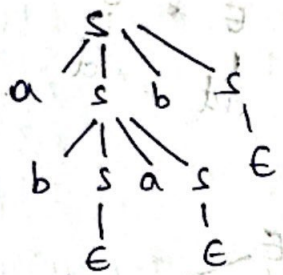
$S \to abaSbS$

$S \to ababS$

$S \to ababa$

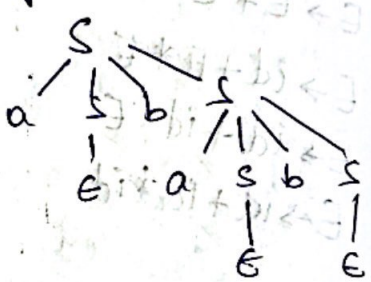
Two parse trees ⇒ Ambiguous grammar.
or possible

LMD parse tree



⇒ abaabab

RMD parse tree



⇒ abaabab

Ex: prove that the following grammar is Ambiguous

$S \to AB$

$B \to ab$

$A \to aa$

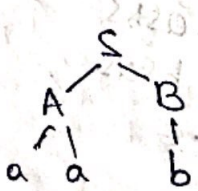
$A \to a$

$B \to b$

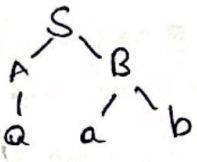
Sol. LMD: $S \to AB$

$S \to aaB$

$S \to aab$



$S \to AB$
$S \to aB$
$S \to aaB$



Two parse trees are possible for the string aab. Hence, it's an ambiguous grammar.

Left Recursion:

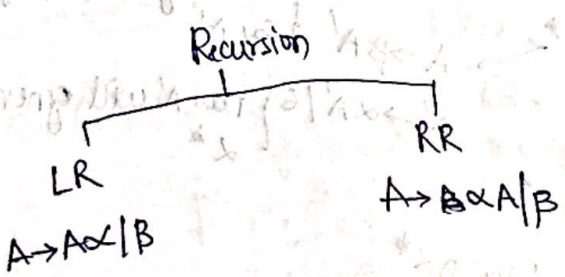
- A production of grammar is said to have left recursion if the leftmost variable of it's RHS is same as variable of its LHS.

  Ex: $A \to A\alpha / \beta$

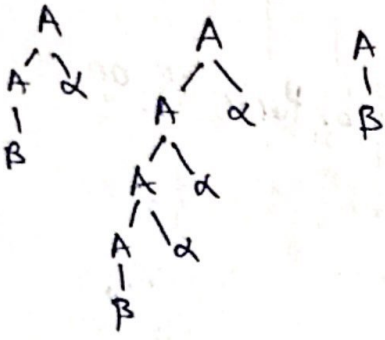
- There are two types of left recursion:
  (i) Direct left recursion Ex: $B \to Ba$
  (ii) Indirect left recursion Ex. $S \to Aa$, $A \to Sm$

Removal of left recursion:

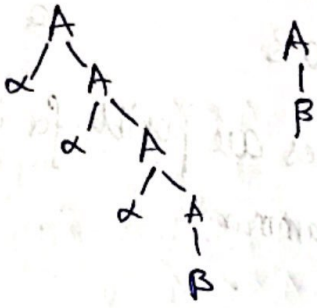
- The top down parsers cannot accept the grammar having left recursion (i.e they donot allow left recursive grammar)
- So, we have to remove left recursion but preserve the language generated by grammar
- Recursion: Recursion could be Left Recursion (LR) or Right Recursion (RR)



LR: $A \to A\alpha / \beta$



RR: $A \to \alpha A / \beta$



Lang = $\beta\alpha^*$

$= \{\beta, \beta\alpha, \beta\alpha\alpha \ldots\}$

Lang = $\alpha^*\beta$

$= \{\beta, \alpha\beta, \alpha\alpha\beta, \ldots\}$

If we write code in terms of function

```
A()
{
   A()
   α;
}
```
(infinite loop)

Left recursion

```
A()
{
   α;
   A();
}
```

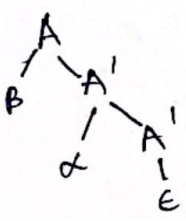
Right recursion

- There is a problem in left recursion, the recursive function calls itself infinite times.
- In right recursion, ($\alpha \Rightarrow$ Base condition) α is evaluated, hence it doesn't lead to infinite loop.
- Hence, we have to remove left recursion.

LR: $A \to A\alpha / \beta$

Now, we want $\beta\alpha^*$ $\to$ $A \to \beta A'$ { A will generate β followed by A'

$A' \to \alpha A' / \epsilon$ { now A' will generate α^* }



Lang: $\beta\alpha^*$

Therefore, $A \to \beta A'$, $A' \to \alpha A' / \epsilon$ $\equiv$ $A \to A\alpha / \beta$

This right recursive grammar is equivalent to this left recursive grammar.

So, to eliminate left recursion, we will follow these rules.

Eg: $E \to E+T / T$ // Remove Left recursion

we will compare it with $A \to A\alpha / \beta$

$E \to E+T / T$
$A \quad A \alpha \quad \beta$

Ans: $E \to TE'$, $E' \to +TE' / \epsilon$

Ex: Eliminate left recursion

(i) $S \to SOSIS / 01$

Sol: we know that,

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

$S \to SOSIS / 01$
$A \quad A \alpha \quad \beta$

$S \to 01 S'$
$S' \to OSISS' / \epsilon$

(ii) $S \to (L) / x$ { this isn't not left recursive, as it's left starts with '('}

$L \to L, S / S$ (Left recursive)
$A \quad A \alpha \quad \beta$

Sol:

$L \to SL'$
$L' \to \epsilon / , SL'$

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

(iii) $A \to A\alpha_1 | A\alpha_2 | A\alpha_3 | \cdots$
$| \beta_1 | \beta_2 | \beta_3 \cdots$

Sol) $A \to \beta_1 A' | \beta_2 A' | \beta_3 A' | \cdots$
$A' \to | \epsilon | \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | \cdots$

$A \to A\alpha | \beta$
↓
$A \to \beta A'$
$A' \to \alpha A' | \epsilon$

(iv) $E \to E+T | T$ — ①
$T \to T*F | F$ — ②
$F \to id$ — ③

Sol:- for the 1st production

$E \to E+T | T$
$\overline{A}\ \overline{A}\ \overline{\alpha}\ \overline{\beta}$

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

we know that
$A \to A\alpha | \beta$
↓
$A \to \beta A'$
$A' \to \alpha A' | \epsilon$

Similarly for 2nd production

$T \to T*F | F$
$\overline{A}\ \overline{A}\ \overline{\alpha}\ \overline{\beta}$

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

3rd production - Ok

Final Answer:
$E \to TE'$
$E' \to +TE' | \epsilon$
$T \to FT'$
$T' \to *FT' | \epsilon$
$F \to id$

(IV) $S \to Aa$ [Indirect left recursion]
$A \to Sb/c$

Sol: ~~If we replace with 'A' with Sb/c and 'S' with A~~
If we replace 'S' production:

$A \to Aab/c$ [left recursive]

We know that, $A \to A\alpha/\beta \equiv A \to \beta A'$
$A' \to \alpha A'/\epsilon$

$S \to Aa$
$A \to Sb/c$ [Indirect left recursive]

$A \to Aab/c$
$A \quad A\ \alpha \quad \beta$

Final Ans: $A \to cA'$
$A' \to abA'/\epsilon$

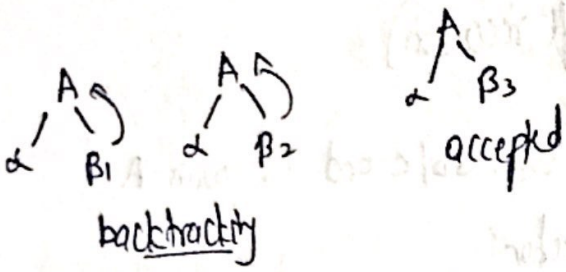
$S \to Aa$
$A \to cA'$
$A' \to abA'/\epsilon$

Left Factoring:

Sometimes, it is not clear which production to choose/to expand a non-terminal because multiple productions begin with the same (terminal/non-terminal)/lookahead. This type of Grammar → Non Deterministic grammar (or) Grammar containing left factoring.

Ex: $A \to \underbrace{\alpha\beta_1/\alpha\beta_2/\alpha\beta_3}_{\text{common prefixes}}$

Suppose we have to access $\alpha\beta_3$



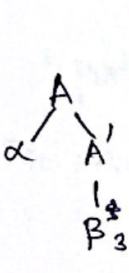
backtracking

- The backtracking is happening because of the common prefixes one or more productions in the RHS are having something common in the prefixes.
- This is also called common prefix problem or non-deterministic grammar.

Reason for backtracking:

- Backtracking occurs because we are making our decision only after seeing α. (i.e which production to choose without seeing full input).
- If input is $\alpha\beta_3$,
  we should make the decision, if β_3 is available then only we will go onto that production.
- So, for

$$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$$ } Non-Deterministic Grammar



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

} we have postponed the decision making

Deterministic grammar

The procedure we used to convert Non-Deterministic grammar to deterministic Grammar is called left factoring.

Problem with Non-Deterministic Grammar:

The problem was most of the top down parser will have to backtrack.

So, we do not need Non-Deterministic grammar.

Eg: $S \to iEtSeS \mid iEtS \mid a$ // Remove left factoring or Remove common prefix problem

$E \to b$

Sol: $S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

(ii) $S \to aSSbS \mid aSasb \mid abb \mid b$

Sol. $S \to aS' \mid b$

$S' \to SSbS \mid Sasb \mid bb$ } again common prefix problem

$S' \to SS'' \mid bb$

$S'' \to SbS \mid asb$

$\therefore A \to aA$, $B \to aB$ } not a common prefix problem

Find ans:

$S \to aS' \mid b$

$S' \to SS'' \mid bb$

$S'' \to SbS \mid asb$

(iii) $S \to bSSaaS \mid bSSasb \mid bSb \mid a$

Sol: $S \to bSS' \mid a$

$S' \to SaaS \mid Sasb \mid b$ } again common prefix prob.

$S' \to SaS'' \mid b$

$S'' \to aS \mid Sb$

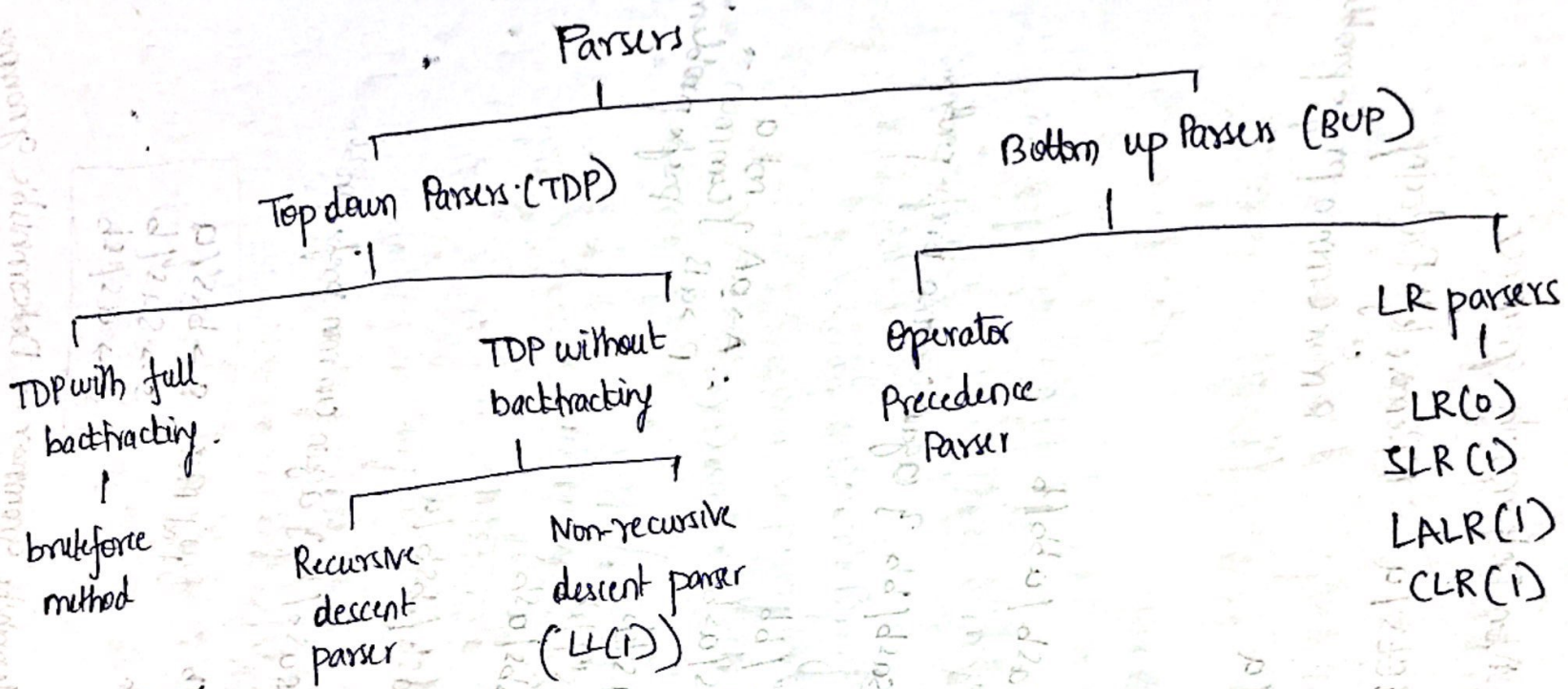
Find Ans:

$S \to bSS' \mid a$

$S' \to SaS'' \mid b$

$S'' \to aS \mid Sb$

Converted Non-Deterministic grammar → Deterministic grammar

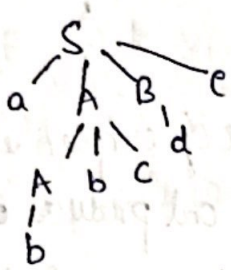


Parser:
. Parser is that phase of compiler which takes input in the form of sequence of tokens and produces output in the form of parse tree. Parser is also known as Syntax Analyser.

Types of Parsers:
. Parser is mainly classified into 2 categories:
1. Top down Parser
2. Bottom up Parser

③ Top-down Parser.
Top down Parser generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals i.e (starts from the start symbol and ends with the terminals)
: It uses left-most derivation.
. The process of constructing the parse tree which starts from the at root and goes down to the leaf is Top-down parsing.
- Top down parsers constructs from the grammar which is free from ambiguity and left recursion.
- Top down parsers uses leftmost derivation to construct a parse tree.
- It allows a grammar which is free from left factoring.

Eg: $S \to aABe$
$A \to Abc \mid b$
$B \to d$



Parse tree

String: abbcde

Yield of parse tree: abbcde

$S \to a\underline{A}Be$ [$S \to aABe$]
$S \to aAbcBe$ [$A \to Abc$]
$S \to abbcBe$ [$A \to b$]
$S \to abbcde$ [$B \to d$]

∴ Left most derivation

- Classification of Top-Down Parsing:
  1. With Backtracking: Brute Force technique
  2. Without Backtracking:
     1. Recursive Descent Parsing
     2. Non-Recursive parsing (or) Predictive Parsing (or) LL(1) Parsing (or) Table Driven Parsing

Top Down Parser with Backtracking (Brute Force Method):

- Here, full backtracking is used to create a parse tree until the correct/given string is generated at leaves.
- In worst case, when string is not given in given language all possible combinations are checked before the failure to construct a parse tree for given string is recognised.

Ex: Given grammar $S \rightarrow aAb$
$A \rightarrow cd/c$

i/p string $w = acb$

Left most derivation

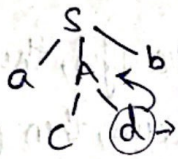
$S \rightarrow aAb$
$S \rightarrow acdb$

$w = acb$
↑ input ptr

$w = acb$
↑ input ptr

$w = acb$
↑ input ptr
no match found in parse tree, backtrack in parse tree and reset i/p pointer to second input symbol.

Parse tree



backtrack
→ point of failure
↑ does not match with given string acb

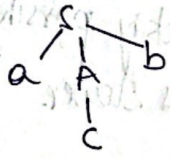
$S \rightarrow aAb$

$w = acb$
↑ input ptr

$S \rightarrow acb$ $(A \rightarrow c)$

$w = acb$
↑ input ptr

Parse tree



As the leaf 'c' matches the second symbol of input string and next leaf 'b' matches third symbol of the input string w, then the parser will stop and announce successful completion of parsing.

. As yield of parser is 'acb' which matches with i/p w = abc.
Hence, string is generated by grammar.

- In top down parsing, every terminal symbol generated by some production of the given grammar (production is predicted) is matched with input string symbol pointed to by the input pointer.
- If the match is successful, then parse tree can continue.
- If the mismatch occurs, then predictions have gone wrong. So, it is necessary to reject (some) previous predictions & input pointer is reset to its position when the rejection production was made. This is known as backtracking.
- Backtracking is one of the major problem of top down parser.
- A backtracking parser will try different production rules to find the match for the input string by backtracking.
- Backtracking parsing is powerful than predictive parsing but it is slower.

Recursive descent parser:

- It is a top-down parsing technique that constructs the parse tree from top and i/p is read from left → right
- A procedure / function is associated with each non-terminal of the grammar.
- A top down parser that uses collection of recursive procedures for parsing the given input string is called as Recursive Descent Parser (RDP).

In RDP, the CFG is used to build the recursive routines.

- The RHS of the production rule is directly converted to a program which is the body of the corresponding non-terminal of LHS.

Basic steps for constructing of Recursive Descent parser:

Step1: Write procedure for start variable production of given grammar. The RHS of the production is directly converted into program code symbol by symbol.

Step2: If the input symbol is Non-terminal, then call procedure of that Non-Terminal.

Eg: $E \rightarrow numT$
$T \rightarrow *numT \mid \epsilon$

```
/* Here num is terminal,
   T is non-terminal */

E()
{
  if (lookahead == num)
  {
    match(num);
    T();
  }
  if (lookahead == '$')
  {
    print ("success");
  }
  else
    print ("error");
}

T()
{
  if (lookahead == '*')
  {
    match('*');
```

```
if (lookahead == num)
{
    match(num);
    T();
}
else
  print("error");
}
else
   return;   // ∈
}

match(char t)          /* t is token */
{
   if(lookahead == t)
   {
      lookahead = getchar();    /*lookahead = next token*/
   }
   else
      print("error");
}

main()
{  E();
}
```

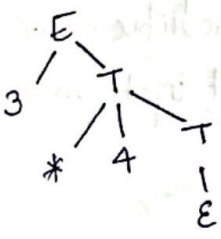
Step3: If the input symbol is terminal then it is matched with the lookahead from the input.

The lookahead pointer has to be advanced on matching the input symbol.

Steps: If the production rule has many alternatives then all these alternatives to be combined into a simple body of the Non-terminal procedure.

Repeat above steps for all non-terminals present in Grammar.

Ex. Let Input String be W = 3*4$ ∴ LA ⇒ Lookahead

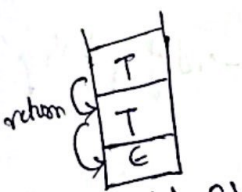


3*4$
↑
LA

3*4$
↑
LA

3*4$
↑
LA

3*4$
↑
LA
success



provided by OS

Advantages of Recursive Descent Parser without backtracking:

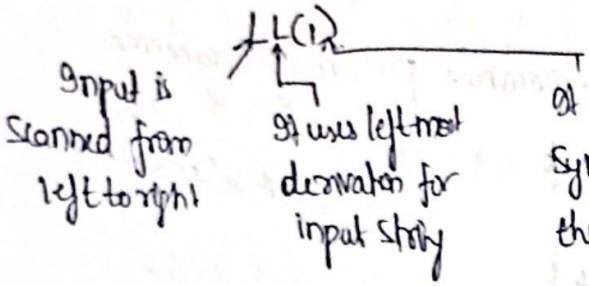
- It's easy to construct
- Overhead associated with backtracking is eliminated.

Drawback / Disadvantages:

- We can not write Recursive descent parser for all types of CFG.
- It can be implemented only for those languages which supports recursive procedure calls.

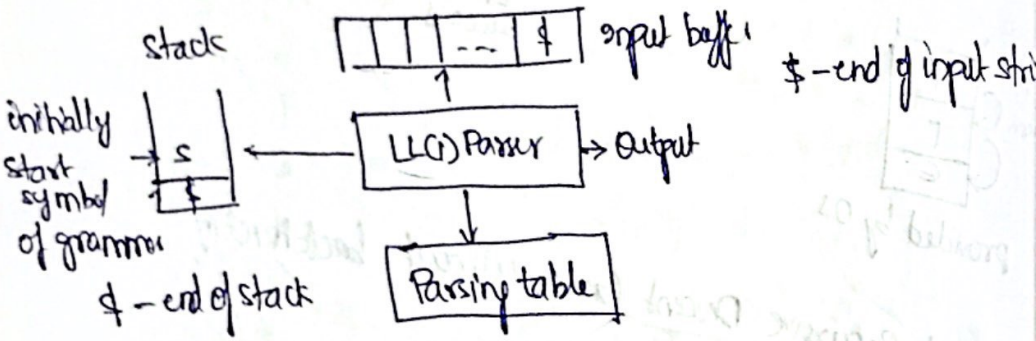
LL(1) Parser:

- LL(1) Parser is a top-down parser of non-recursive type.



- To make the parser backtracking free, the predictive parser uses a lookahead pointer, which points to the next input symbols.

Block diagram of LL(1) Parser:



Data structures used by LL(1) Parser

(1) Input Buffer
(2) Stack
(3) Parsing table

- Predictive parsers can be constructed for a class of grammars called LL(1).
- No left recursive or ambiguous grammar can be LL(1).
- Predictive parser has the capability to predict which production is to be used to replace the input string.

- Predictive parser does not suffer from backtracking.
- To remove backtracking, the predictive parser uses a lookahead pointer which points to the next input symbol.
- Predictive parser uses a stack & parsing table to parse the input stored in input buffer & generate a parse tree from top to bottom. (top down approach).
- Both the stack & the input contains an end symbol $, where
  $ in stack - represent empty stack
  $ in input - represent input is consumed.
- The parsing program reads top of the stack & a current input symbol & with the help of these two symbols, the parsing action is determined.
- The parser consults the table M[A, a] (A → top of stack, a - input) each time while taking the actions. Hence, this type of parsing method is called table driven parsing algorithm.

First() and Follow():

- First(A): It is a set of terminal(s) that begin in strings derived from A.

Ex: S → aAB
A → b
B → c

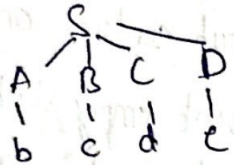
first(S) = {a}, first(A) = {b}
first(B) = {c}

Ex: A → abc | def | ghi
first(A) = {a, d, g}

- Follow(A): Set of terminals that appear immediately to the right of A. Ex: ~~A → BCD~~
- Follow of start symbol is always $.
- for $A \to \alpha B$
  follow(B) = follow(A)

Ex: $S \to ABCD$

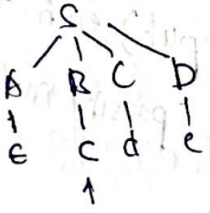
$A \to b$
$B \to c$
$C \to d$
$D \to e$



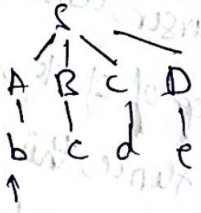
first(S) = first(A) = b

first(B) = c
first(C) = d
first(D) = e

Ex: $S \to ABCD$
$A \to b \mid \epsilon$
$B \to c$
$C \to d$
$D \to e$



(or)



Soln: first(S) = {b, c} as $A \to b \mid \epsilon$

First and Follow rules:

First:

1. If $A \to a\alpha$, α [Combination of terminals & non-terminals]
   then First(A) = {a}
2. If $A \to \epsilon$, then First(A) = {ϵ}
3. If $A \to BC$ then
   First(A) = First(B) if First(B) doesn't contain epsilon
   If First(B) contains epsilon (ϵ) then First(A) = First(B) ∪ First(C)

Follow:

1. If 'S' is start symbol, then Follow(S) = [$]
2. If $A \to \alpha B\beta$ then Follow(B) = [search in RHS]
   Follow(B) = First(β) if First(β) doesn't contain ϵ.
   In Follow, ϵ we cannot place ϵ.
3. If $A \to \alpha B$ then Follow(B) = Follow(A)
4. If $A \to \alpha B\beta$ where $\beta \to \epsilon$
   then Follow(B) = Follow(A)

Ex: First & Follow examples:

	First	Follow
$S \to ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \to a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \to b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \to c$	$\{c\}$	$\{d, e, \$\}$
$D \to d \mid \epsilon$	$\{d, \epsilon\}$	$\{e, \$\}$
$E \to e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$\}$

	First	Follow
$S \to Bb \mid Cd$	$\{a, b, \epsilon, d\}$	$\{\$\}$
$B \to aB \mid \epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \to cC \mid \epsilon$	$\{c, \epsilon\}$	$\{d\}$

	First	Follow
$S \to ACB \mid CbB \mid Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \to da \mid BC$	$\{d, g, h, \epsilon\}$	$\{h, g, \$\}$
$B \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
$C \to h \mid \epsilon$	$\{h, \epsilon\}$	$\{g, \$, b, h\}$

Construction of LL(1) Parser:

Steps involved:

1. Elimination of left recursion
2. Elimination of left factoring
3. Calculation of first and follow
4. Construction of parsing table
5. Check whether input string is accepted by parser or not.

Example:

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

Step1: Elimination of left recursion

If production is of form $A \to A\alpha / \beta$ (left recursive)

Replace it with $A \to \beta A'$

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

$E \to \underset{A}{E} \underset{A}{+} \underset{\alpha}{T} / \underset{\beta}{T}$ $\quad E \to TE'$

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

$T \to \underset{A}{T} \underset{A}{*} \underset{\alpha}{F} / \underset{\beta}{F}$ $\quad T \to FT'$

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

$F \to (E) / id$ (Not left recursive)

After elimination of left recursion, productions are:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

Step 2: Elimination of left factory

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \ldots$ [left factoring]

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3$

There is no left factoring in the production.

Step 3: Calculation of first and follow

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

First(E) = { (, id }

First(E') = { +, ~~ϵ~~ ϵ }

First(T) = { (, id }

First(T') = { *, ϵ }

First(F) = { (, id }

Follow(E) = { $,) }

Follow(E') = { $,) }

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

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

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

Step 4: Construction of 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			$F \to (E)$		$F \to id$	

Rules for construction of parsing table:

$A \to \alpha$

1. If the production is of $A \to \alpha$
   Calculate First(α), if → it produces terminal symbol a.

   then, add $A \to \alpha$ to $M[A, a]$

2. If First(α) prod contains ϵ (or) $A \to \epsilon$ (epsilon)
   then, we have to calculate Follow(A) → produces terminal b.

   then, add $A \to \epsilon$ to $M[A, b]$

3. The remaining entries of the parsing table are filled with ~~error~~ error.

1. $E \to TE'$
   First(TE') = {(, id}
   Add $E \to TE'$ to $M[E, (]$
   $M[E, id]$

2. $E' \to +TE' / \epsilon$
   (a) First($+TE'$) = {+}
   Add $E' \to +TE'$ to $M[E', +]$
   (b) $E' \to \epsilon$
   Follow(E') = {$,)}
   Add $E' \to \epsilon$ to $M[E', \$]$
   $M[E',)]$

3. $T \to FT'$
   First(FT') = {(, id}
   Add $T \to FT'$ to $M[T, (]$
   $M[T, id]$

4. $T' \to *FT' / \epsilon$
   (a) First($*FT'$) = {*}
   Add $T \to *FT'$ to $M[T', *]$
   (b) Follow(T') = {$,), +}
   Add $T \to \epsilon$ to $M[T', \$]$
   $M[T',)]$ $M[T', +]$

5. $F \to (E) / id$
   (a) First((E)) = {(}
   Add $F \to (E)$ to $M[F, (]$
   (b) First(id) = {id}
   Add $F \to id$ to $M[F, id]$

Steps: Check whether the input string is accepted or not.

Let's consider i/p: id+id$

Stack	Input String	Action
$E	id+id$	E→TE'
$E'T	id+id$	T→FT'
$E'T'F	id+id$	F→id
$E'T'id	id+id$	pop (when stack top symbol = input string) & move ptr to one position to right
$E'T'	+id$	T'→ϵ
$E'	+id$	E'→+TE'
$E'T+	+id$	pop
$E'T	id$	T→FT'
$E'T'F	id$	F→id
$E'T'id	id$	pop
$E'T'	$	T'→ϵ
$E'	$	E'→ϵ
$	$	

After performing entire string, if the stack and input string constitutes '$', the string is accepted.

④ Bottom-up Parsing:

- In Bottom-up parsing, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol of grammar.
- The process of bottom-up parsing stops successfully as soon as we reach the start symbol.
- It uses reverse right-most derivation.
- Parse tree is constructed from bottom to top. ie from leaves to root.
- In bottom-up parsing, parser tries to identify the RHS of production rule and replace it by corresponding LHS. This activity is called as Reduction.
- So, the prime task in bottom-up parsing is to find the productions that can be used for reduction.

Example: $E \to E+E$
$E \to id$

Input string: id+id+id

RMD: $E \to E+E$
$E \to E+E+E$
$E \to E+E+id$
$E \to E+id+id$
$E \to id+id+id$

Bottom-up parsing: id+id+id
id + id + E ($E \to id$)
id + E + E ($E \to id$)
id + E ($E \to E+E$)
E + E ($E \to id$)
E ($E \to E+E$)

Handle & Handle Pruning:

Handle: Handle is a substring which matches with right-side of production, then it's replaced with LHS Non-terminal.

Eg: $S \to aABe$
$A \to Abc \mid b$
$B \to d$

Input string "abbcde"

Sol: Input string abbcde

Handles during parsing of abbcde

Right Sentential Form	Handle	Reducing production
a<u>b</u>bcde	b	$A \to b$
a<u>Abc</u>de	Abc	$A \to Abc$
a<u>A</u>d<u>e</u>	d	$B \to d$
a<u>ABe</u>	aABe	$S \to aABe$
<u>S</u>		

Sol: Bottom-up parsing uses right-most derivation in reverse order is called Handle pruning.

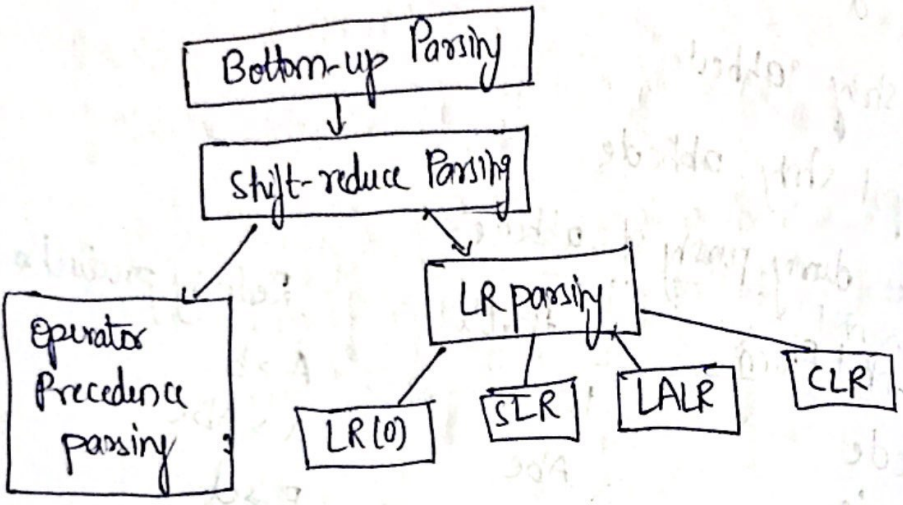
Eg2: $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Input string: "id * id

Right Sentential Form	Handle	Reducing production
id * id	id	$F \to id$
F * id	F	$T \to F$
T * id	id	$F \to id$
T * F	T * F	$T \to T*F$
T	T	$E \to T$
E		

LR Parser:

LR parsing is one type of



Shift-reduce Parser:

- Shift reduce parser attempts for the construction of parse tree in a similar manner as done in bottom up parsing i.e the parse tree is constructed from leaves (bottom) to the root (up). A more general form of shift-reduce parser is LR parser.
- This parser requires some data structures i.e
  - An input buffer for storing the input string.
  - A stack for storing and accessing the production rules.

Basic operations:

- Shift: This involves moving of symbols from input buffer onto the stack.
- Reduce: If the handle appears on top of the stack then its reduction by using appropriate production is done i.e

RHS of the production is popped out of stack and LHS of production rule is pushed onto the stack.

- Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it means successful parsing is done.
- Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Eg: $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Input string: id*id

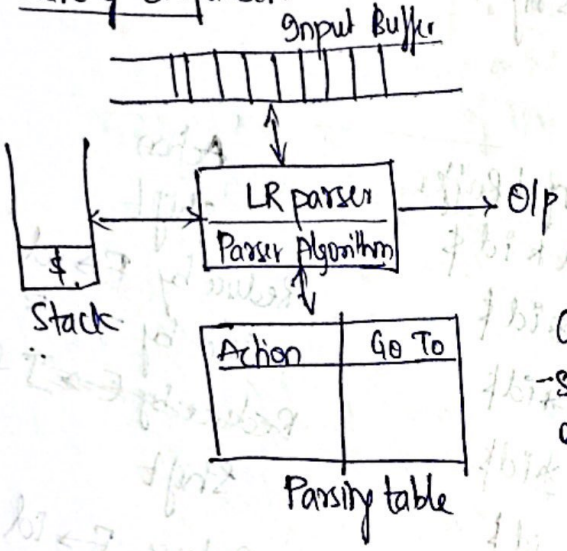
	Stack	Input Buffer	Action
1.	$	id*id$	Shift
2.	$id	*id$	Reduce by F→id
3.	$F	*id$	Reduce by T→F
4.	$T	*id$	~~Reduce by E→T~~ Shift
5.	$T*	id$	Shift
6.	$T*id	$	Reduce F→id
7.	$T*F	$	Reduce T→T*F.
8.	$T	$	Reduce T→E
9.	$E	$	Accepted

1. Shift-reduce conflict: It occurs if the parser has a choice to select both shift action and reduce action. But only one can be selected.
2. Reduce-reduce conflict: It occurs if the parser has more than one reduction is possible.

⑤ LR-Parsing:

- LR parsing is one type of bottom up parsing. It is used to parse the large large class of grammars.
- In the LR parsing, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a rightmost derivation in reverse.
- LR(k), "k" is the no. of input symbols of the lookahead used to make number of parsing decision.

Structure of LR parser:



Action:
- Shift
- Reduce
- Accept,
- Error.

Goto: The function goto takes state and grammar as arguments and produces a state

- There are three widely used algorithms available for constructing an LR parser.
  - SLR(1) – Simple LR
  - CLR(1) – Canonical LR parser
  - LALR(1) – Look ahead LR parser

LR(0):

. An LR(0) item is a production G with dot at some position on the right side of the production.

. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing

. In the LR(0), we place the reduce node in the entire row.

Example:

$S \rightarrow AA$

$A \rightarrow aA/b$

Augmented Grammar: Add Augment production and insert '.' symbol at the first production position of every production

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA$

$A \rightarrow .b$

Steps to solve LR(0) sums:

1. Augment the given grammar
2. Draw the canonical collection of LR(0) item.
3. Number the production
4. Create the parsing table
5. Stack implementation
6. Draw parse tree.

Ex: $E \to BB$
$B \to cB \mid d$

Step1: Augment the given grammar

$E' \to E$
$E \to BB$
$B \to cB$
$B \to d$

Step2: Draw Canonical collection of LR(0) item

I_0 : $E' \to .E$
$E \to .BB$
$B \to .cB$
$B \to .d$

goto(I_0, E): $E' \to E.$: I_1

goto(I_0, B): $E \to B.B$
I_2: $B \to .cB$
$B \to .d$

goto(I_0, c): $B \to c.B$
I_3: $B \to .cB$
$B \to .d$

goto(I_0, d): $B \to d.$: I_4

goto(I_2, B): $E \to BB.$ I_5

goto(I_2, c): $B \to c.B$
I_3: $B \to .cB$
$B \to .d$

goto(I_2, d): $B \to d.$ I_4

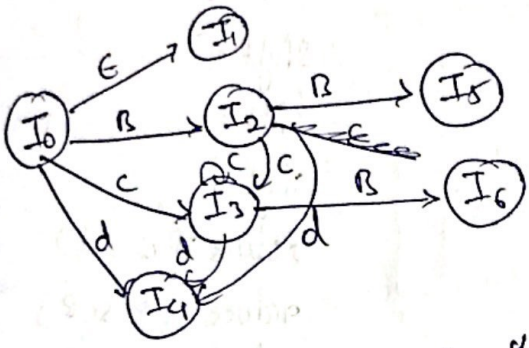
goto(I_3, B): $B \to cB.$: I_6

goto(I_3, c): $B \to c.B$
$B \to .cB$: I_3
$B \to .d$

goto(I_3, d): $B \to d.$ I_4.

~~goto(I_3~~

Step3: Designing of finite automata



Step4: Create Parsing table Number the productions

$E' \rightarrow E - 0$
$E \rightarrow BB - 1$
$B \rightarrow cB - 2$
$B \rightarrow d - 3$

Step5: Create Parsing table:

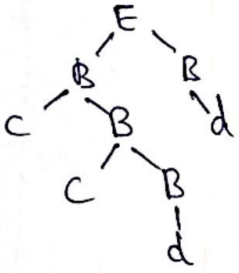
State	Action c	Action d	Action $	Goto E	Goto B
I_0	S3	S4		1	2
I_1			Accepted		
I_2	S3	S4			5
I_3	S3	S4			6
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Steps: Stack implementation, input: ccdd $

Stack	Input	Action
$0	ccdd$	Shift c
$0c3	cdd$	Shift c
$0c3c3	dd$	Shift d
$0c3c3d4	d$	Reduce 3 (B→d)
$0c3c3B6	d$	reduce 2 (B→cB)
$0c3B6	d$	reduce 2 (B→cB)
$0B2	d$	Shift d
$0B2d4	$	reduce 3 (B→d)
$0B2B5	$	reduce 1 (E→BB)
$0E1	$	Accepted

The string is accepted.

Step 6: Draw parse tree



If the grammar is not LR(0), there are multiple entries in single cell.

⑥ SLR (Simple LR) parser:

Eg: Consider the following grammar

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

Construct SLR parse table for the grammar. Show the actions of the parser for the input string "abab".

Sol: Steps involved are:

1. Calculate canonical collection of LR(0) items.
2. Designing of finite Automata
3. Construction of SLR parse table
4. Parsing the input string abab

Augmented Grammar:

In augmented grammar, a new production is added. [∵ Start symbol' → start symbol]

$S' \rightarrow S$

$S \rightarrow AS$

$S \rightarrow b$

$A \rightarrow SA$

$A \rightarrow a$

After writing the Augmented Grammar, above steps are to be followed:

Step 1: Calculating canonical collection of LR(0) items.

- If there is a (.) dot symbol at RHS of production, then it's called item.
- If there is a Non-terminal after (.) dot symbol, then we have to write the productions of the non-terminal.

I_0 : $S' \to .S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

[After(.) Non-terminal ⇒ so A productions need to write]

goto (I_0, S) : (after(.) dot)

'S' is observed on RHS in two productions. Whenever goto is applied, (.)dot position is shifted to one position right.

goto(I_0, S): $S' \to S.$
I_1 : $A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

[If there is a non-terminal after(.)dot then we need to write productions of that non-terminal]

[After(.)dot there is a non-terminal]

goto (I_0, A) :

I_2 : $S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_0, b) :

I_3 : $S \to b.$ [dot at end ⇒ final item]

goto (I_0, a)

I_4 : $A \to a.$

goto (I_1, A):

I_5: $A \to SA.$
$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$
~~$A \to SA$~~

goto (I_1, S):

I_6: $A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_1, a):

I_4: $A \to a.$

goto (I_1, b):

I_3: $S \to b.$

goto (I_2, S)

I_7 $S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to .a$
~~$S \to AS$~~
$S \to .AS$
$S \to .b$

goto (I_2, A):

I_2: $S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_2, a):

I_4: $A \to a.$

goto (I_2, b)

I_3: $S \to b.$

goto (I_5, S):

I_7: $S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_5, A):

I_2: $S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_5, a):

I_4: $A \to a.$

goto (I_5, b):

I_3: $S \to b.$

goto (I_6, A):

I_5: $A \to SA.$
$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_6, S):

I_6: $A \to S\ A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_6, a):

I_4: $A \to a.$

goto (I_6, b):

I_3: $S \to b.$

goto (I_7, A)

I_5: $A \to SA.$
$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

goto (I_7, S):

I_6: $A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

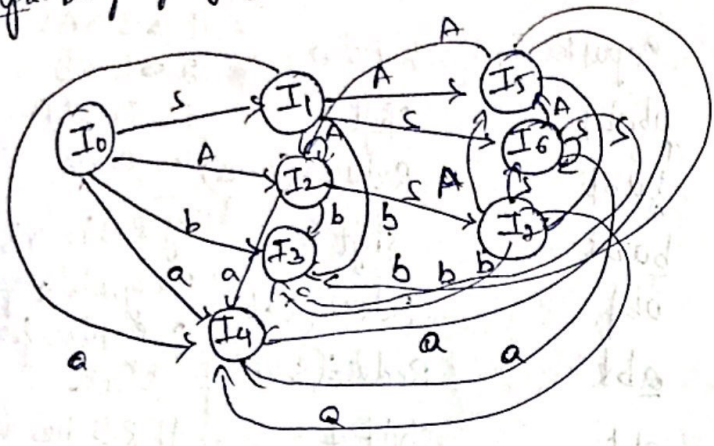
goto (I_7, a):

I_4: $A \to a.$

goto (I_7, b)

I_3: $S \to b.$

Step2: Designing of finite automata:



Step1: Construction of SLR parse table

$S \rightarrow AS/b$

$A \rightarrow SA/a$

First (S) = {b, a}

First (A) = {a, b}

Follow(S) = {$, a, b}

Follow(A) = {a, b}

	Action			Goto	
I	a	b	$	S	A
0	S4	S3		1	2
1	S4	S3	Accepted	6	5
2	S4	S3		7	2
3	r2	r2	r2		
4	r4	r4			
5	S4/r3	S3/r3		7	2
6	S4	S3		6	5
7	S4/r1	S3/r1	r1	6	5

I_3: $S \rightarrow b.$ (similarly, do is for every final item)

Add r2 to the follow(S) = {$, a, b}

Sol Shift/r-reduce

- Some cells contain multiple entries [shift & reduce].

Hence, its not SLR(1). [each cell → single entry → SLR(1)]

Step: Parsing of string abab

Stack	Input Buffer	Action
$0	abab$	Shift 4
$0a4	bab$	Reduce 4 (A→a)
$0A2	bab$	Shift 3
$0A2b3	ab$	Reduce 2 (S→b)
$0A2S7	ab$	Reduce (S→AS)
$0S1	ab$	Shift 4
$0S1a4	b$	Reduce 4 (A→a)
$0S1A5	b$	Reduce 3 (A→SA)
$0A2	b$	shift 3
$0A2b3	$	Reduce 2 (S→b)
$0A2S7	$	Reduce 1 (S→AS)
$0S1	$	Accepted

∴ The input string is accepted by parser.

0 ⇒ S' → S
1 ⇒ S → AS
2 ⇒ S → b
3. A → SA
4- A → a

If RHS has only one symbol, pop 2 symbols from stack.

If RHS has two symbol, pop 4 symbols

Apply 'a' on A

Difference between LR(0) and SLR(1):

The SLR(1) has the extra ability to help decide what action to take when there are conflicts. Because of this, any grammar that can be parsed by LR(0) parser can be parsed by an SLR (1) parser.

- However SLR(1) parser can parse a large number of grammars than LR(0).

LALR(1) Parser:

- LALR refers to the lookahead LR. To construct the LALR(1) parsing table, we use the canonical collection of LR(1) items.
- In LALR(1) parsing, the LR(1) items which have same productions but different lookahead are combined to form a single set of items.
- LALR(1) parsing is same as the CLR(1) parsing, only difference in the parsing table.

CLR(1) Parsing:

- CLR refers to canonical lookahead. CLR parsing use the collection of LR(1) items to build the CLR(1) parsing table.
- CLR(1) parsing table produces the more number of states as compare to the SLR(1) parsing.
- In CLR(1), we place the reduce node only in the lookahead symbols.
- Various steps involved in the CLR(1) Parsing:

- For the given input string, write a CFG.
- Check the ambiguity of the grammar
- Add Augment production in the given grammar.
- Create Canonical collection of LR(0) items.
- Draw a data flow diagram (DFA)
- Construct a CLR(1) parsing table

LR(1) item = LR(0) item + lookahead

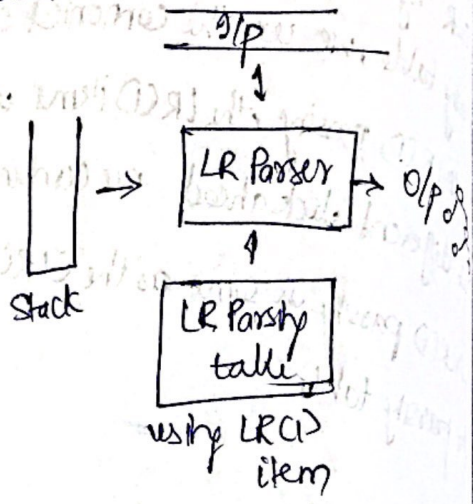
CLR(1) & LALR(1)

LR(1) item = LR(0) item + lookahead

LR(0) ⇒ Put reduce in full row

SLR(1) ⇒ Put reduce in follow of P

CLR(1) ← LALR(1) ⇒ Put reduce only on lookahead



Procedure to find lookahead:

E → BB

B → cB / d

Augmented grammar

E' → E

E → BB

B → cB

B → d

E' → .E, $

E → .BB, $ for the first production, lookahead is $

B → .cB, c/d First(B, $) = First(B) = (c, d)

B → .d, c/d

6: Construct LALR parser for

$S \to CC$

$C \to aC / b$ parse the string bb

Sol: Augmented grammar:

$S' \to S$

$S \to CC$

$C \to aC$

$C \to b$

Step1: Calculation of LR(1) items.

I_0: $S' \to .S, \$$ — 0
$S \to .CC, \$$ — 1
$C \to .aC, a/b$ — 2
$C \to .b, a/b$ — 3

goto(I_0, S):

I_1: $S' \to S., \$$

goto(I_0, C):

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

goto(I_0, a):

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

goto(I_0, b):

I_4: $C \to b., a/b$

goto(I_2, C):

I_5: $S \to CC., \$$

goto(I_2, a):

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

goto(I_2, b):

I_7: $C \to b., \$$

goto(I_3, C):

I_8: $C \to aC., a/b$

goto(I_3, a):

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

goto(I_3, b):

I_4: C → b., a/b

goto(I_6, C):

I_9: C → aC., $

goto(I_6, a):

I_6: C → a.C, $
C → .aC, $
C → .b, $

goto(I_6, b):

I_7: C → b., $

We can combine I_3 & I_6 → I_{36} [Same production but different lookahead]

C → a.C, a/b/$
C → .aC, a/b/$
C → .b, a/b/$

Similarly I_4 & I_7 → I_{47}

C → b., a/b/$

Similarly I_8 & I_9 → I_{89}

C → aC., a/b/$

Conflict

Construction of LALR parse table.

I	a	b	$	S	C
0	S36	S47		1	2
1			Accepted		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Stack implementation:

Stack	I/p string	Action
$0	bb$	S47
$0b47	b$	r(C→b)
$0C2	b$	S47
$0C2b47	$	r(C→b)
$0C2C5	$	r(S→CC)
$0S1	$	Accepted

The input string is accepted by parser

2. Construct CLR parser table for the grammar

$S \to CC$
$\epsilon \to aC$
$C \to d$

parse ddd$

Augmented grammar:
$S' \to S$ — 0
$S \to CC$ — 1
$C \to aC$ — 2
$C \to d$ — 3

LR(1) Items:

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

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

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

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

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

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

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

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

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

goto(I_6, C)
I_9 : $C \to aC., a/d$

	Action			goto	
state	c	d	$	S	C
0	S3	S4		1	2
1			Accepted		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

stack implementation:

Stack	Input string	Action
$0	dd$	shift 4
$0d4	d$	reduce (c→d)
$0c2	d$	shift 7
$0c2d7	$	reduce (c→d)
$0c2c5	$	reduce (s→cc)
$0s1	$	Accepted

The input string is accepted.

⑥ Using Ambiguous Grammars:

Only operator precedence parser accepts ambiguous grammars.

Operator grammar:

- A grammar that is used to define mathematical operators is called an operator grammar (or) operator precedence grammar.
- A grammar is said to be operator precedence grammar if it has 2 properties:
  (i) no RHS of any production has a ∈(epsilon)
  (ii) no two non-terminals are adjacent on RHS.

Ex: $E \to E+E \mid E*E \mid id$ // operator grammar

Ex: $S \to SAS \mid a$
$A \to bSb \mid b$ // not operator grammar
[no two non-terminals can be adjacent]

We can convert it into operator grammar

$S \to SbSbS \mid a \mid SbS$ // operator grammar
~~$A \to bSb \mid b$~~ not required

There are 3 operator precedence relations:

1) $a \cdot\!> b$ → means 'a has higher precedence than the terminal b'
   (or) 'a' takes precedence over 'b'

2) $a <\!\cdot b$ → means 'a yields precedence to 'b (or)
   a has lower precedence than terminal b

3) $a \doteq b$ ⇒ means 'a' has same precedence as 'b'.

Operator Precedence Parser:

- Bottom up parser that interprets an operator grammar.
- This parser is only used for operator grammar.
- Ambiguous grammar are not allowed in any parser except operator precedence parser

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

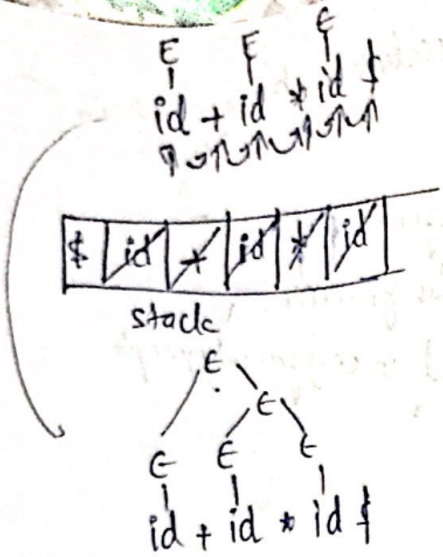
Operator precedence relation table

	id	+	*	$
id	-	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	-

} defined precedence rules

- Two id's will never be compared 'becoz they will never come side by side.
- Identifier will be given highest precedence compared to any other operator.
- $ has least precedence compared to any other operator

Using this operator precedence table, parser will parse the i/p.



if top of stack of the precedence is higher perform pop operation else perform push operation

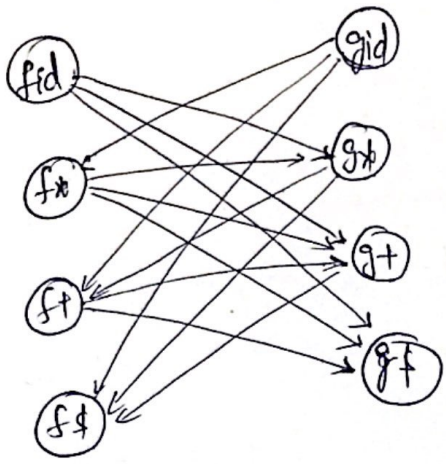
(parse tree generated)

Disadvantage of operator Relation Table:
When no. of entries are large, size of table will be n^2.
for N operators → $O(n^2)$

So, to decrease the size of the table, we use operator function for table.

f \ g	id	+	*	$
id	-	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	-

Now, we will construct a graph.
Now, for every operator, we will have fid, gid, f+, g+, f*, g*, and f$, g$



If we found any cycle in graph, we will stop there because then we cannot construct an operator function table

To make function table:

We have to find out for every node that what is the length of the longest path starting from that node.

fid → g* → f+ → g+ → f$

gid → f* → g* → f+ → g+ → f$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Advantage:

⇒ Size is less.

⇒ O(2n) ⇒ Space complexity

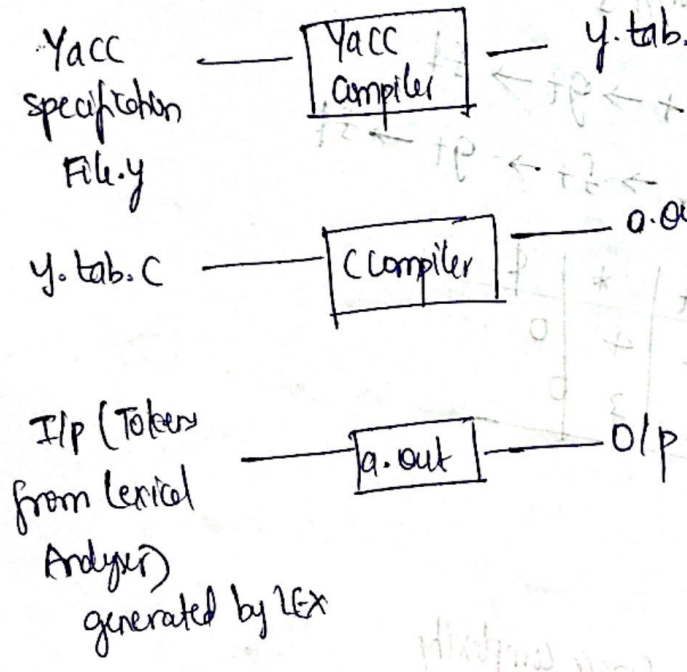
Disadvantage:

Error detection capability of function table is less compare to operator relation table.

(7) Parser Generator (YACC in compiler design):

- It stands for Yet Ano
- It stands for Yet Another Compiler - Compiler
- It is a tool for generating Look-Ahead Left-to-Right (LALR) parser
- It takes i/p from the lexical analyzer and generates parse tree
- Syntax Analyser / parser is the 2nd phase of the compiler which takes i/p as tokens and generates a parse tree.

Working (3 steps) / Block diagram:



Working:

1) I/p to the Yacc compiler will be a file with .y extension. It will contain desired grammar in yacc format. YACC compiler will convert it into a Ccode in the form of y.tab.c file.

2) This y.tab.c file will be given as a input to the ccompiler and the output will be LALR (i.e a.out).

3) Tokens generated by the lexical analyser (using the lex tool) will be given as input to a.out, i.e LALR parser we will get the parse tree as o/p.