Syntax Analysis- II

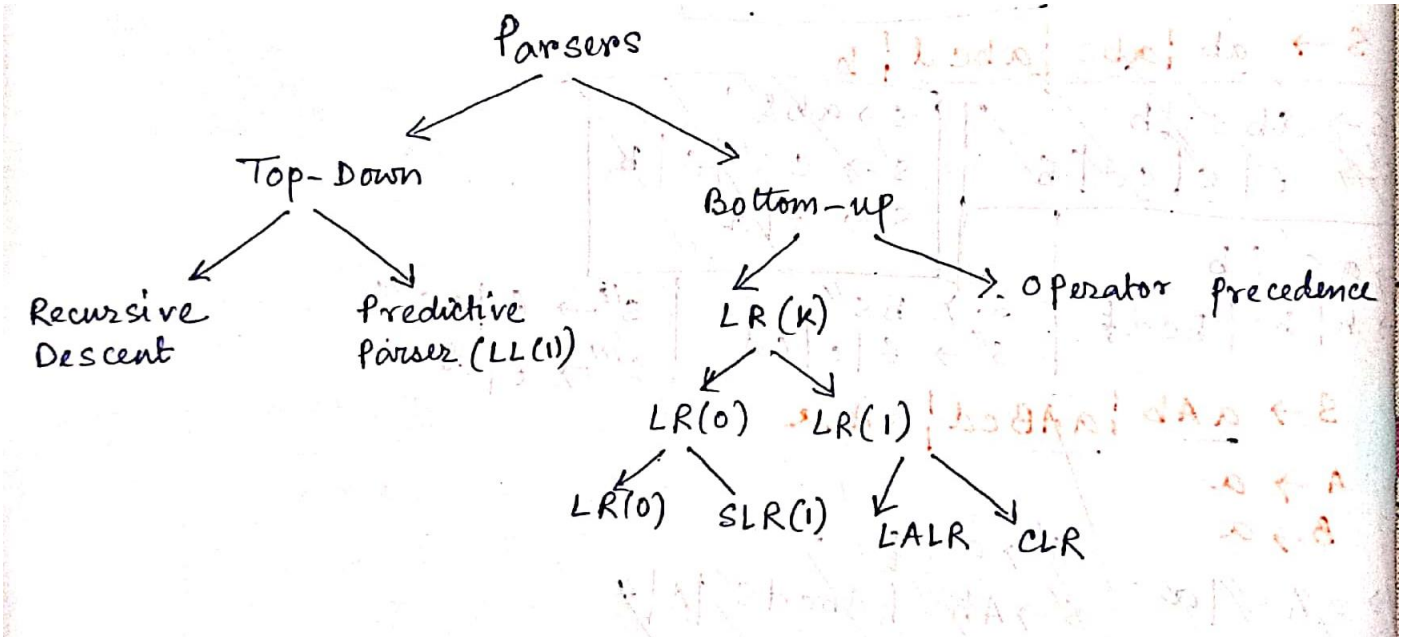
Parsers

Techniques of parsing :

1. Top Down Parsing:- Parsing starts with start symbol and tree is generated using production rules till required leaf nodes are obtained.
*Attempts to find Left most derivation of a non-terminal.

2. Bottom Up Parsing:- Leaf nodes are obtained first from input symbols and step by step reduced till start symbol is achieved.
*Tries to find right most derivation of a non-terminal.

Different types of parsers :



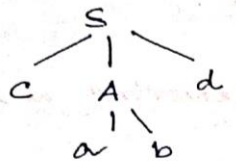
Top Down Parser

A top down parser begins with a start symbol and according to input data, expands non-terminals using given production rules. If the syntax of input string is correct, a syntax tree is generated. Otherwise an error message is issued.

$S \rightarrow c\,A\,d$
$A \rightarrow ab \mid a$

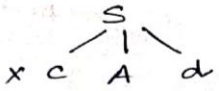
$T = \{a, b, c, d\}$
$V = \{A\}$
$start = S$.

given string $w = c\,a\,b\,d$



∴ string is valid.

string 2: $w = b\,a\,b\,d$

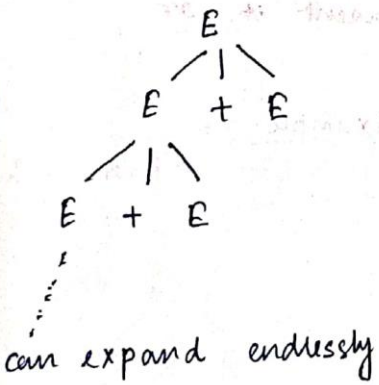


String is invalid, error message generated

Left Recursive Grammar

If for a given grammar, there exists a non-terminal A such that, $A \overset{*}{\Rightarrow} A\alpha$, then it is said to be left recursive.

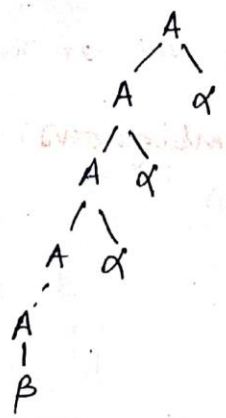
Ex:- $E \rightarrow E + E \mid id$ (Left most symbol in the RHS is equal to LHS)



can expand endlessly

or

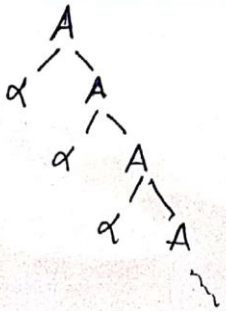
$A \rightarrow A\alpha \mid \beta$



$\beta\alpha^*$

Right Recursive Grammar.

$A \rightarrow \alpha A \mid \beta$.



$\alpha^*\beta$

At first do something then Recursion.

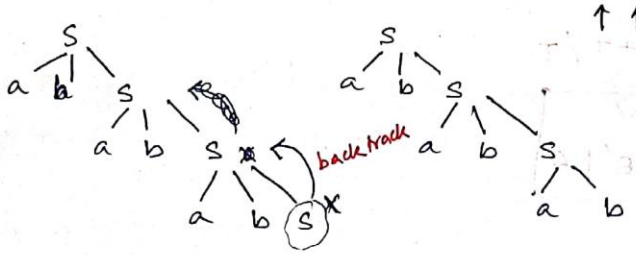
Left Recursion is a problem:-

It allows a parser to expand syntax tree endlessly without moving the input pointer. Thus the i/p string is not analysed but syntax tree keeps growing and is never validated or invalidated.

Right Recursion and why it is not a problem?

It is not a problem because in right recursion, i/p pointer moves through the string α. Thus for an invalid production the string reaches null before syntax tree is completed.

Ex:- $S \rightarrow abS \mid ab$ $\quad w = abababab$



i/p pointer moves.

∴ The string is valid.

Types of Left Recursion

(i) Direct Left recursion
(ii) Indirect Left recursion.

Direct — occurs when non-terminal A has a production rule, $A \rightarrow A\alpha$

Indirect — when non-terminal A does not have a direct production rule $A \rightarrow A\alpha$, but after several substitution there may be a stage when, $A \overset{*}{\Rightarrow} A\alpha$.

Ex:- 1) $S \rightarrow Sab$ shows direct left recursion

2) $A \rightarrow Bab$
$B \rightarrow Cabc$
$C \rightarrow Acab$

$A \rightarrow \underline{B}ab$
$\Rightarrow \underline{C}abcab$
$\Rightarrow Acabcabcab$

Thus it shows indirect left recursion.

Solution :-

1) Algo for removal of DLR :-

For a grammar production rule.

$$A \to A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$$

Left recursion is removed by introducing non-terminal A'.

s.t,

$$\boxed{\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \epsilon \end{aligned}}$$

(A' derives either 0-no of α's or any no of α's).

LRG

$\beta (\alpha^*) \Leftarrow A'$

$\therefore \beta A'$

This a Right Recursive grammar corresponds to Left recursive grammar.

Ex:-1) $E \to E+T / T$

$$\boxed{\begin{aligned} E &\to T\ E' \\ E' &\to +T\ E' \mid \epsilon \end{aligned}}$$

2) $S \to S0S1S \mid 01$

$$\boxed{\begin{aligned} S &\to 01\ S' \\ S' &\to 0S1SS' \mid \epsilon \end{aligned}}$$

3) $S \to (L) \mid x$ [NOT Left recursive]

$L \to L,S \mid S$

$$\boxed{\begin{aligned} L &\to S\ L' \\ L' &\to ,S\ L' \mid \epsilon \end{aligned}}$$

Indirect Recursion condition :-

$S \to Aa$

$A \to Sb \mid \epsilon$

So, here indirect left recursion is identified. That is, there is a cycle. $S \rightleftarrows A$

So, we systematically eliminates left recursion from a grammar. It is guaranteed to work, if,

(i) the grammar has no cycle. ($A \Rightarrow A$)

ii) ϵ productions ($A \to \epsilon$)

(2) Algorithm for indirect left recursion

Arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$

for (each i from 1 to n)

{ for (j = 1 to i-1)

{ substitute each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, where, $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j- productions.

}

eliminate the immediate left recursion among the A_i- productions.

}

Ex:

$S \rightarrow aS \mid bA$ --- (1)

$A \rightarrow aS \mid bA$ --- (2)

Remove left Recursion

- No DLR present in this grammar.
- Again, $A \rightarrow aS$ means, no left Recursion
- No Indirect left recursion present here.

Ex: $S \rightarrow aS \mid Ab$, $A \rightarrow aS \mid bA$

Here, $S \rightarrow Ab$, but no $A \rightarrow S\alpha$.

$\therefore$ No ILR present.

Ex:- $S \rightarrow aS \mid Ab$, $A \rightarrow Sa \mid bA$

Here, $S \rightarrow Aa$, $A \rightarrow S\alpha$ cycle. so, indirect present

Ex:-2 $S \to aS \mid aA$, $A \to Sa \mid Ab$ $\Rightarrow$

$S \to aS$ —(i)
$S \to aA$ —(ii)
$A \to Sa$ —(iii)
$A \to Ab$ —(iv)

A_1, A_2, A_3, A_4 $\therefore i = 1, 2, 3, 4$

$V = \{A, S\}$, $T = \{a, b\}$

(Direct Left Recursion)

Step 1: DLR removal.

$A \to A\underset{\alpha}{b} \mid \underset{\beta}{Sa}$

$\therefore A \to SaA' \mid \beta$

$A' \to bA' \mid \epsilon$

$A \to A\alpha \mid \beta$.

$\therefore A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

Step 2: Indirect LRecursion Removal.

Arrange all non-terminal in order, S, A, A'

$S \to aS \mid aA$ —①

$A \to SaA'$ —②

$A' \to bA' \mid \epsilon$ —③

$i = 1$, (for production rule 1)

$j = 0$, Grammar remains same.

$i = 2$, (for production rule 2)

$j = 1$

Now, there is no indirect left recursion, as no production found which is forming cycle.

Ex:- ~~$S \to Aa \mid b$~~ $S \to Aa \mid b$

$A \to A\alpha \mid Sd \mid \epsilon$

Ex:- $A \to Ba \mid Aa \mid c$

$B \to Bb \mid Ab \mid d$.

Ex :-3

$S \to Sab \mid Ab$
$A \to S \mid a$

① Remove DLR

$S \to Abs'$ —①
$S' \to abs' \mid \epsilon$ —②
$A \to S \mid a$ —③

② IDLR removal

In P_1, $S \to Abs'$ } ⇒ S ⇄ A
In P_3, $A \to S$ }

∴ there is a cycle between the non-terminals.

(i) $A_i \to A_j \gamma$ (&) $A_j \to \delta_1 \beta$ } —(ii)
$A \to S$
$A \to a$

S A bs'

∴ substitute

(i) production by (ii)

$A_i \to \delta_1 \gamma$
$S \quad \delta_1 bs'$

∴ $i = 2$.

$A \to S \mid a$
$A \to Abs' \mid a$

∴ Now grammar,

$S \to Abs'$ —①
$S' \to abs' \mid \epsilon$ —②
$A \to Abs' \mid a$ —③

But here, again we have direct left recursion, in ③.

∴

$S \to Abs'$
$S' \to abs' \mid \epsilon$
$A \to aA'$
$A' \to bs'A' \mid \epsilon$

Ex-1: $S \to SaS \mid Ab \mid \epsilon$ —①
$A \to Sa \mid ba$ —②

Step 1: Remove direct left recursion.

$S \to AbS' \mid S'$ —①
$S' \to aSS' \mid \epsilon$ —②
$A \to Sa \mid ba$ —③

$i=1$, $j=0$ ×

$i=2$,
$j=1$ NO production found, $S' \to S\alpha$.

$i=3$
$j=1$

Found, $A \to Sa$. [$S \to A$]
ILR present, remove it, by forward substitution, we get,

$S \to AbS' \mid S'$ — ①
$S' \to aSS' \mid \epsilon$ — ②
$A \to AbS'a \mid S'a \mid ba$ —③

Remove DLR at ③

$S \to AbS' \mid S'$ —①
$S' \to aSS' \mid \epsilon$ —②
$A \to S'aA' \mid baA'$
$A' \to bS'aA' \mid \epsilon$ — ③

Q1. Remove all types of left recursion from given grammar.

$A \rightarrow Aa \mid ab \mid Bc$ — (1)

$B \rightarrow ac \mid cd$ — (2)

$C \rightarrow a \mid b \mid Ab$ — (3)

Step 1: Removing direct left recursion from A

$A \rightarrow abA' \mid BcA'$ — 1

$A' \rightarrow aA' \mid \epsilon$ — 2

$B \rightarrow ac \mid Cd$ — 3

$C \rightarrow a \mid b \mid Ab$ — 4

Step 2: Removing Indirect Left recursion from the production:-

Arrange non-terminals in order A, A', B, C.

for i=1, j - not entered.

Grammar remains same.

for i=2,

j=1, No production of form $A' \rightarrow A\alpha$ found.

∴ Grammar same. (No cycle)

for i=3

j=1, No production found for $B \rightarrow A\alpha$.

j=2, ——— ——— $B \rightarrow A'\alpha$

i=4, (4)

j=1, $C \rightarrow Ab$ production found.

∴ forward substitution, $C \rightarrow abA'b \mid BcA'b \mid a \mid b$ — (4)

i=4, j=2

No production found of the form $C \to A'\alpha$.

i=4, j=3.

production found = $C \to BcA'b$. [C ⇄ B cycle.]

forward substitution apply,

$$C \to abA'b \mid acA'b \mid CdcA'b \mid a \mid b$$

Now, the grammars is,

$A \to abA' \mid BcA'$ ---(1)

$A' \to aA' \mid \epsilon$ ---(2)

$B \to ac \mid cd$ ---(3)

$C \to \underbrace{abA'b}_{\beta_1} \mid \underbrace{accA'b}_{\beta_2} \mid \underbrace{CdcA'b}_{A\alpha} \mid \underbrace{a}_{\beta_3} \mid \underbrace{b}_{\beta_4}$ ---(4)

Now, ~~this is~~, Production (4) is having DLR, remove it,

$$C \to \underbrace{abA'bC'}_{\beta_1 A'} \mid \underbrace{accA'bC'}_{\beta_2 A'} \mid \underbrace{aC'}_{\beta_3 A'} \mid \underbrace{bC'}_{\beta_4 A'} \quad -(4)$$

$$C' \to dcA'bC' \mid \epsilon \quad -(5)$$

∴ Final production:

$A \to abA' \mid BcA'$

$A' \to aA' \mid \epsilon$

$B \to ac \mid cd$

$C \to abA'bC' \mid accA'bC' \mid aC' \mid bC'$

$C' \to dcA'bC' \mid \epsilon$

Left factoring:

CFG

- Deterministic
- Non-deterministic

Non-deterministic CFG :- (common prefix problem)

Grammar with common prefix between atleast 2- different productions from the same LHS.

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$$

① $S \rightarrow aSb \mid aA \mid b$
$A \rightarrow aB \mid a$
$B \rightarrow b$

② $S \rightarrow aSb \mid bSa \mid \epsilon$

③ $S \rightarrow AaB \mid BA$
$A \rightarrow a \mid b$
$B \rightarrow d \mid e$

Deterministic CFG — Grammar without any common prefix in any of the different productions from the same LHS.

→ during parsing non-deterministic grammar requires lots of backtracking (time consuming)

→ to make grammar suitable for predictive or top-down parsing, we need to convert non-deterministic grammars into deterministic grammar, process is known as Left factoring.

Left factoring :-

Often, when backtracking, we have to collapse and rebuild same syntax because of a common superexpression.

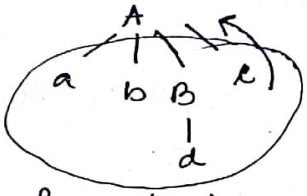
Left factoring reduces backtracking overheads by finding these subexpression and turning them into a seperate non-terminal.

$A \to abBc \mid abc \mid abBd$

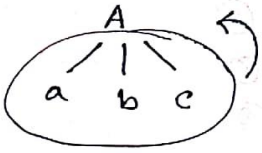
$B \to a \mid b \mid d$

$w = abdd$

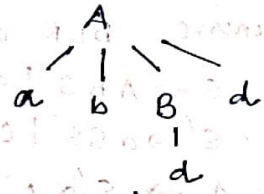
syntax tree:



Back track



Backtrack



string Matched

Rule: $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$$\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \mid \beta_2 \mid \beta_3 \end{aligned}$$

It is conversion of ND-CFG to D-CFG ⇒ Left factoring
(α = common prefix)

Ex:- $A \to abBc \mid abc \mid abBd$

$B \to a \mid b \mid d$

⇒ $A \to abA'$

$A' \to Bc \mid c \mid Bd$

$B \to a \mid b \mid d$ (Ans)

Ex: $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

⇒ $S \to iEtSS' \mid a$

~~$S' \to \epsilon \mid eS$~~

$S' \to \epsilon \mid eS$

$E \to b$

Ex:- $S \to aSSbS \mid aSaSb \mid abb \mid b$

~~$S \to aSS'$~~ $S \to aS' \mid b$.

$S' \to SSbS \mid SaSb \mid bb$

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

$S'' \to SbS \mid aSb$

$S \to aS' \mid b$

Ex:- $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \to bSS' \mid a$

$S' \to SaaS \mid SaSb \mid b$

$S \to bSS' \mid a$

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

$S'' \to aS \mid Sb$

Ex:- $S \to aSb \mid abS \mid ab$

Ex:- $S \to \mid abc \mid abcd \mid b$

Ex:- $S \to aAb \mid aABc \mid aABcd \mid aA \mid a$

$A \to a$

$B \to a$

General Top-Down Parser / Recursive Descent Parser:

- Constructs from the Grammar which is free from ambiguity and left recursion.
- Uses leftmost derivation to construct a parse tree.
- May or may not require *Backtracking* to find correct *A*-production.
- Allows a grammar which is free from Left Factoring.
- Consists of set of procedures for each non-terminal.
- Execution begins with start symbol.
- The parser may have more than one production to choose from for a single instance of input.

```
void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Example:

E→iE′

E′→+ i E′ | ϵ

```
// Definition of E, as per the given production
E()
{
    if (l == 'i') {
        match('i');
        E'();
    }
}

  // Definition of E' as per the given production
  E'()
  {
      if (l == '+') {
          match('+');
          match('i');
          E'();
      }//The second condition of E'
      else if ( l == 'e' )
      {
        match('e');
      }
          return ();
  }
```

```
// Match function
match(char t)
{
    if (l == t) {
        l = getchar();
    }
    else
        printf("Error");
}


int main()
{
    // E is a start symbol.
    E();

    // if lookahead = $, it represents the end of the string
    // Here l is lookahead.
    if (l == '$')
        printf("Parsing Successful");
}
```