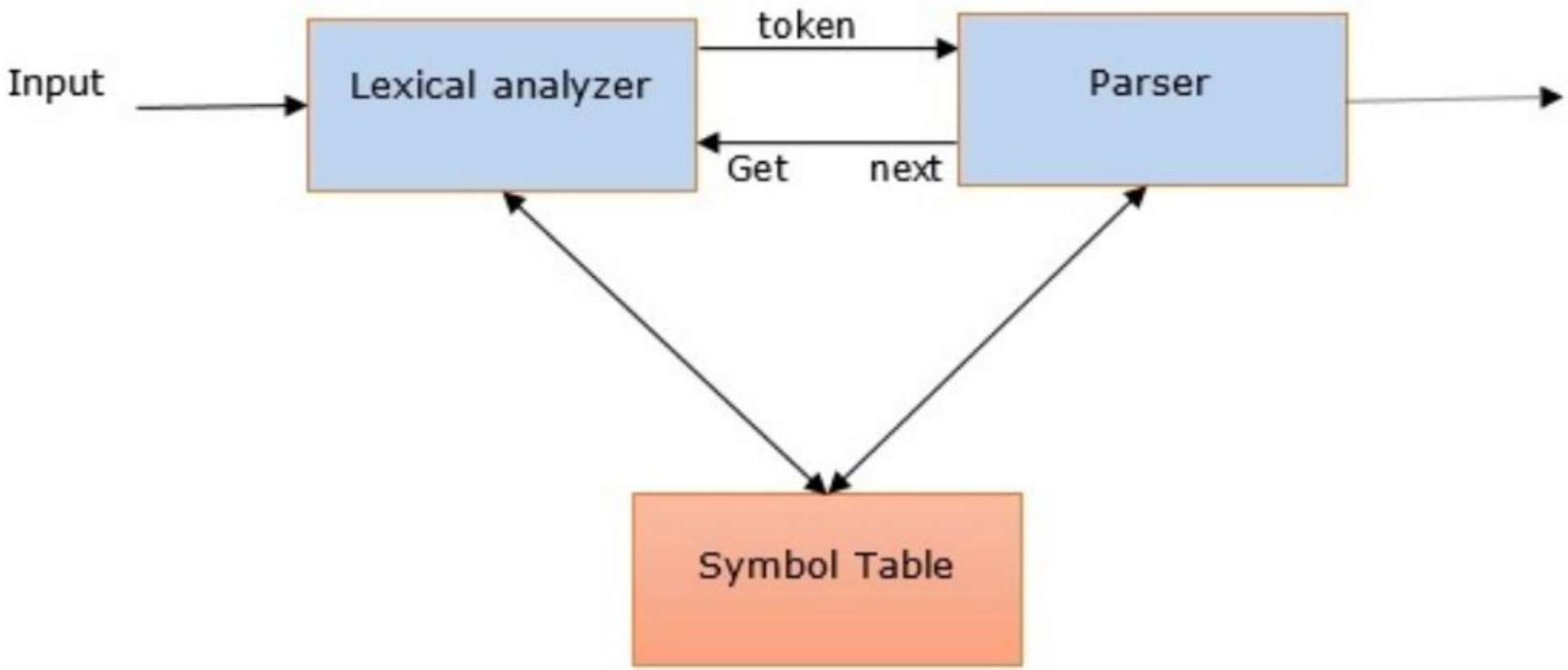
PARSING

Submitted by:-
Kartika
URN-1805192 CRN-1815040 CSA2

PARSING ?

- Also called as Syntactic analysis or syntax analysis
- Syntactic analysis or parsing or syntax analysis is the third phase of NLP
- The word ‘Parsing’ is originated from Latin word ‘pars’ (which means ‘part’)
- Comparing the rules of formal grammar, syntax analysis checks the text for meaningfulness.
- The sentence like “Give me hot ice-cream”, for example, would be rejected by parser or syntactic analyzer.
- The purpose of this phase is to draw exact meaning, or you can say dictionary meaning from the text.

It may be defined as the process of analyzing the strings of symbols in natural language conforming to the rules of formal grammar.


Input
Lexical analyzer
token
Parser
Get
next
Symbol Table

ROLE OF PARSER

- To report any syntax error.
- It helps to recover from commonly occurring error so that the processing of the remainder of program can be continued.
- To create parse tree
- To create symbol table.
- To produce intermediate representations (IR).

DEEP VS SHALLOW PARSING

DEEP PARSING	SHALLOW PARSING
In deep parsing, the search strategy will give a complete syntactic structure to a sentence	It is the task of parsing a limited part of the syntactic information from the given task.
It is suitable for complex NLP applications	It can be used for less complex applications
EXAMPLES:- Dialogue Summarisation and summarisation	EXAMPLES:- Information extraction and text mining
It is called Full Parsing	It is called Chunking

TYPES OF PARSING

Parsing is classified into two categories, i.e.

Top Down Parsing and Bottom-Up Parsing.

Top-Down Parsing is based on Left Most Derivation

Bottom Up Parsing is dependent on Reverse rightmost Derivation.

DERIVATION

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

Types of Derivation: Leftmost and rightmost

LEFT MOST DERIVATION

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

Example:

Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c

INPUT

a - b + c

EXAMPLE OF LEFT MOST DERIVATION

1. S = **S** + S
2. S = **S** - S + S
3. S = a - **S** + S
4. S = a - b + **S**
5. S = a - b + c

RIGHT MOST DERIVATION

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

EXAMPLE:-

1. S = S - **S**
2. S = S - S + **S**
3. S = S - **S** + c
4. S = **S** - b + c
5. S = a - b + c

TOP DOWN PARSING

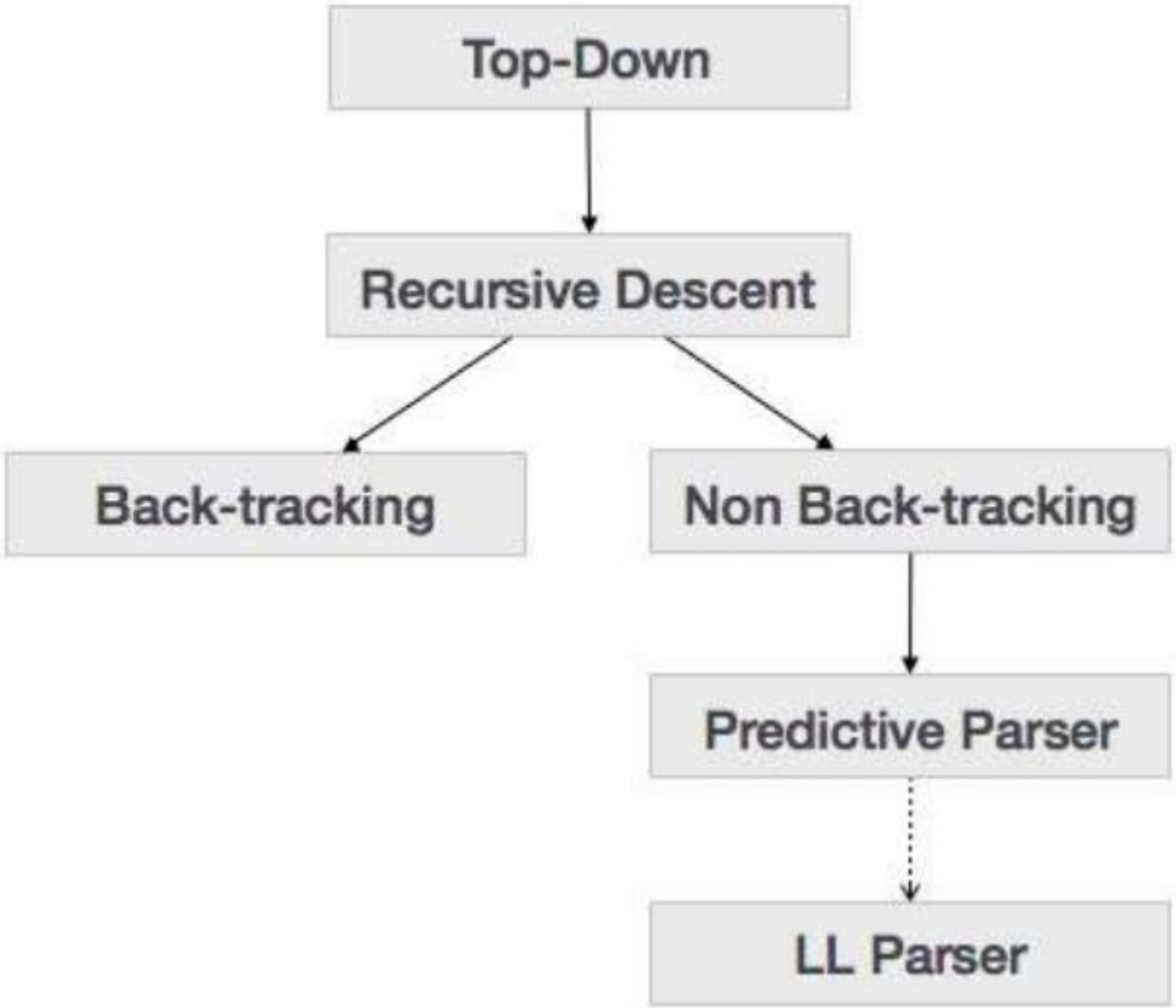
The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.

- 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.

EXAMPLE OF LEFT FACTORING-

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

$$E \rightarrow b$$


Top-Down
Recursive Descent
Back-tracking
Non Back-tracking
Predictive Parser
LL Parser

WORKING OF TOP DOWN PARSER

EXAMPLE:-

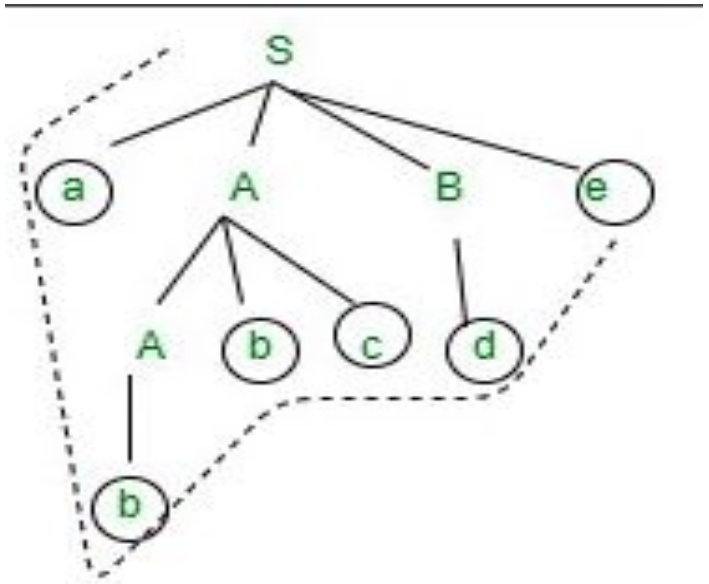
```
S -> aABe

A -> Abc | b

B -> d
```

Input –

```
abbcde
```

Now, you will see that how top down approach works. Here, you will see how you can generate a input string from the grammar for top down approach.

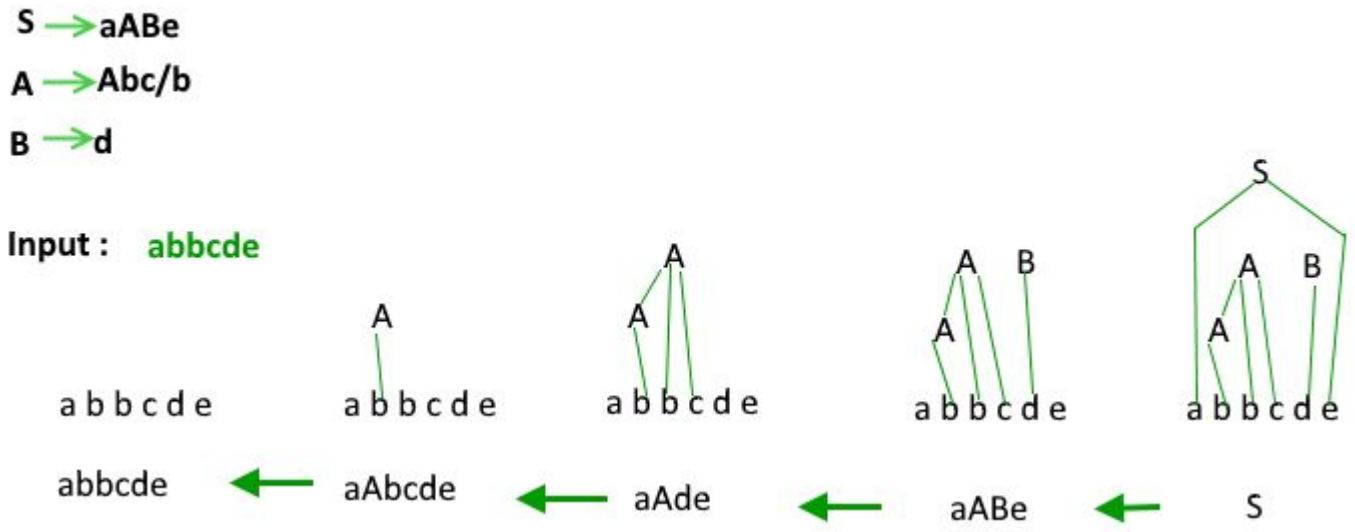
- First, you can start with S -> a A B e and then you will see input string a in the beginning and e in the end.
- Now, you need to generate ~~a~~bbcd~~e~~ .
- Expand A-> Abc and Expand B-> d.
- Now, You have string like aAbcde and your input string is abbcde.
- Expand A->b.
- Final string, you will get **abbcde**.

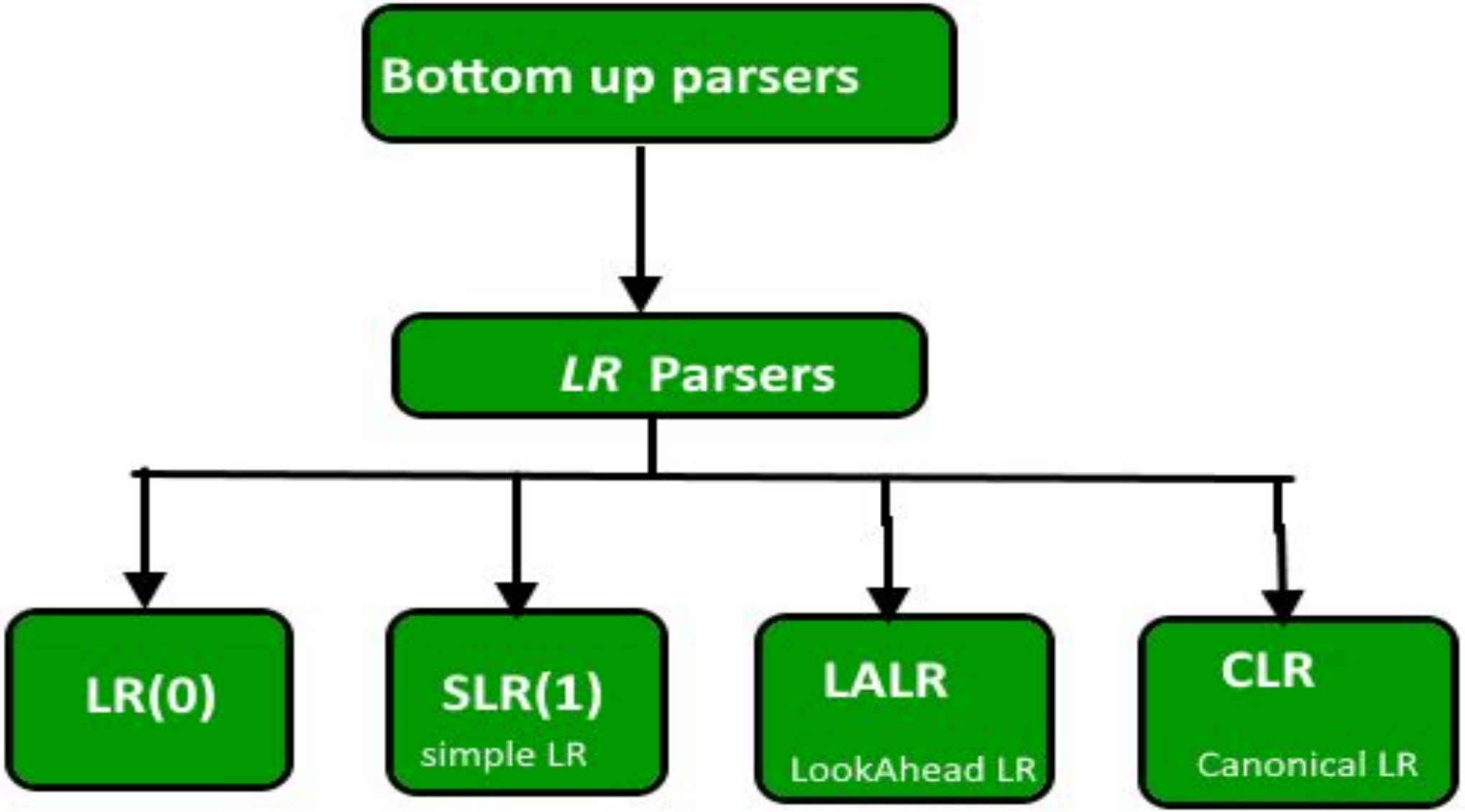


BOTTOM UP PARSER OR SHIFT REDUCE PARSERS

Bottom Up Parsers / Shift Reduce Parsers

Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.




Bottom up parsers
LR Parsers
LR(0)
SLR(1)
simple LR
LALR
LookAhead LR
CLR
Canonical LR

BOTTOM UP PARSER

There are two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR PARSER

A general shift reduce parsing is LR parsing. The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse.

Benefits of LR parsing:

1. Many programming languages using some variations of an LR parser. It should be noted that C++ and Perl are exceptions to it.
2. LR Parser can be implemented very efficiently.
3. Of all the Parsers that scan their symbols from left to right, LR Parsers detect syntactic errors, as soon as possible.

S.No	Top Down Parsing	Bottom Up Parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempts to reduce the input string to start symbol of a grammar.
3.	In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner.	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
4.	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
5.	It's main decision is to select what production rule to use in order to construct the string.	It's main decision is to select when to use a production rule to reduce the string to get the starting symbol.

ISSUES IN BASIC PARSING

✓ Never explores trees that aren't potential solutions, ones with the wrong kind of root node.

X But explores trees that do not match the input sentence (predicts input before inspecting input).

X Naive top-down parsers never terminate if G contains recursive rules like X → X Y (left recursive rules).

X Backtracking may discard valid constituents that have to be re-discovered later (duplication of effort)

EARLEY ALGORITHM

The Earley Parsing Algorithm:

an efficient top-down parsing algorithm that avoids some of the inefficiency associated with purely naive search with the same top-down strategy

In naive search, top-down parsing is inefficient because structures are created over and over again.

Intermediate solutions are created only once and stored in a chart (dynamic programming).

Left-recursion problem is solved by examining the input.

Earley is not picky about what type of grammar it accepts, i.e., it accepts arbitrary CFGs.

```
function Earley-Parse(words,grammar) returns chart
    Enqueue((γ → •S, [0,0]),chart[0])
    for i ← from 0 to Length(words) do
        for each state in chart[i] do
            if Incomplete?(state) and Next-Cat(state) is not POS  then
                Predictor(state)
            elseif Incomplete?(state) and Next-Cat(state) is POS then
                Scanner(state)
            else
                Completer(state)
            end
        end
    return(chart)
```

A state consists of:

1) A subtree corresponding to a grammar rule S → NP VP
2) Info about progress made towards completing this subtree S → NP • VP
3) The position of the subtree wrt input S → NP • VP, [0, 3]
4) Pointers to all contributing states in the case of a parser

A dotted rule is a data structure used in top-down parsing to record partial solutions towards discovering a constituent.

Earley: fundamental operations

1) Predict sub-structure (based on grammar)
2) Scan partial solutions for a match
3) Complete a sub-structure (i.e., build constituents)

How to represent progress towards finding an S node?

Add a dummy rule to grammar: γ → • S

This seeds the chart as the base case for recursion.

Earley's dot notation: given a production X → αβ, the notation X → α • β represents a condition in which α has already been parsed and β is expected.

Chart[0]:	• astronomers saw stars with ears partial solutions ...
Chart[1]:	astronomers • saw stars with ears partial solutions ...
Chart[2]:	astronomers saw • stars with ears partial solutions ...
Chart[3]:	astronomers saw stars • with ears partial solutions ...
Chart[4]:	astronomers saw stars with • ears partial solutions ...
Chart[5]:	astronomers saw stars with ears •

Assumed indexing scheme:

$\bullet_0$ *astronomers* $\bullet_1$ *saw* $\bullet_2$ *stars* $\bullet_3$ *with* $\bullet_4$ *ears* $\bullet_5$

CYK ALGORITHM

CYK algorithm is a parsing algorithm for context free grammar.

In order to apply CYK algorithm to a grammar, it must be in Chomsky Normal Form. It uses a dynamic programming algorithm to tell whether a string is in the language of a grammar.

Algorithm :

Let w be the n length string to be parsed. And G represent the set of rules in our grammar with start state S.

1. Construct a table DP for size n × n.
2. If w = e (empty string) and S -> e is a rule in G then we accept the string else we reject.
3.
```
For i = 1 to n:
  For each variable A:
     We check if A -> b is a rule and b = w_i for some i:
        If so, we place A in cell (i, i) of our table.
```

4.
```
For l = 2 to n:
  For i = 1 to n-l+1:
       j = i+l-1
        For k = i to j-1:
           For each rule A -> BC:
        We check if (i, k) cell contains B and (k + 1, j) cell contains C:
             If so, we put A in cell (i, j) of our table.
```

5. We check if S is in (1, n):
   If so, we accept the string
   Else, we reject.

Example –

Let our grammar G be:

```
S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a
```

CYK Algorithm:- Check whether a string 'abbb' is a valid member of following CFG

* CYK applicable on CNF only (Chomsky Normal Form)
  CNF, $A \to BC$ or $A \to a$
* CYK is universal (applicable on all CNF)

Time complexity $\to O(n^3)$
Space Complexity $\to O(n^2)$

Example

$S \to AB$
$A \to BB \mid a$
$B \to AB \mid b$

Input string $\Rightarrow$ 'abbb'
1 2 3 4

		(4) b	(3) b	(2) b	(1) a
(1)	a	(S,B)	A	(B,S)	A
(2)	b	(S,B)	(A)	B	
(3)	b	A	B		
(4)	b	B			

On (12) cell or (ab) concatenate (11) and (22)
$= aa + bb$
$= AB$
Now S and B derives AB.

For (23) = (22)(33) = BB
For (34) = (33)(44) = BB
For (13) = (11)(23) = AA = ϕ
(12)(33) = (B S)(B)
(B B)(S B)
A ϕ = A

For 24 $\to$ 2 3 4
(22) (34)
B A $\Rightarrow$ BA $\Rightarrow \phi$
(23)(44)
A B $\Rightarrow$ S, B

For $\to$ 14 $\Rightarrow$ 1 2 3 4

(11)(24) | (12)(34) | (13)(44)

(11)(24): A S B; AS, AB; ϕ, S, B

(12)(34): S B A; SA, BA; ϕ, ϕ

(13)(44): A B; S, B.

At the (1,4) cell i.e (1→4) S is there. if S is there in (1,4) cell then the string is a valid member of CFG. (abbb, ab, bbb) are valid.

“abbb”

S->AB,A->BB|a, B->AB |b

Length 1 strings:

a,b,b,b

a =>(11) or (aa) , a can be derived from A

b=>(44) or (33) or (22), b can derived from B

Length 2 strings

ab =>(11)or (22) ,(11)=>**A** and (22)=>**B ,concatenate** them

ie **AB** ,**AB** can be derived from {**S,B}**

bb =>22 or 33 ,22=>**B** and 33=>**B, concatenate** them ie **BB** can be derived from { **A}**

“abbb”

S->AB,A->BB|a, B->AB |b

abb (11) and (23) or 12 and 33

AA or (SB) and B=> SB and BB

“bbb”=>234

22 and 34 or 23 and 44

BA or AB =>S and B

“abbb” 1234

11+24 and 12+34 and 13+44

{S,B}

S->AB,A->BB|a, B->AB |b Input_string=”abbb”

	b (4)	b (3)	b (2)	a (1)
a (1)	{B,S}	{A}	{S,B}	{A}
b (2)	{B,S}	{A}	{B}	
b (3)	{A}	{B}		
b (4)	{B}			

S->AB|BC , A->BA|a, B->CC|b,C->AB|a “baaba”

	b (1)	a (2)	a(3)	b (4)	a(5)
b (1)					
a (2)					
a (3)					
b (4)					
a (5)					

ample 2

$S \rightarrow AB \mid BC$

$A \rightarrow BA \mid a$

$B \rightarrow CC \mid b$

$C \rightarrow AB \mid a$

'baaba' → input string
12345

	1) b	2) a	3) a	4) b	5) a
1) b	{B}	{S,A}	∅	∅	{C,S,A}
2) a		{A,C}	{B}	{B}	{C,S,A}
3) a			{A,C}	{S,C}	{B}
4) b				{B}	{S,A}
5) a					{A,C}

12
ba
(11)(22)
(B)(AC)
BA, BC
⇓ ⇓
A S

23
aa
(22)(33)
(aa)(aa)
(AC)(AC)
AC, AA, CC
⇓ ⇓ ⇓
∅ ∅ B

34
(33)(44)
{AC}{B}
AB BC
⇓
S, C

24 ⇒ aab
234
(22)(34) | (23)(44)
(AC)(SC) | (B)(SA)
(AS, AC, CC) | (BS, BA)
⇓ | BB
B | ⇓
∅

35, aba
(33)(45) | (34)(55)
(AC)(SA) | (SC)(A,C)
(AC, AA, CS) | SA, SC, CA
⇓ | CC
∅ | ⇓
B.

14
1234
(11)(24) | (12)(34) | (13)(44)
(B)(B) | (SA)(SC) | ∅ {S,A}
BB ⇒ ∅ | SS, AS, AC | = SA ⇒ ∅
⇓
∅

(25). aaba
2345.
⇒ (22)(35)
(AC) B ⇒ {C,S}
AB, BC.
⇓ ⇓
S, C S
→ (23)(45)
(B)(SA)
(BS, BA) ⇒ A
→ (24)(45)
(B) SA) ⇒ A

15
(11)(25) or
(12)(35) or
(13)(45) or
(14)(55)

45
(44)(55)
(B)(AC)
BA, BC
⇓ ⇓
A S

13 → 123
(11)(23) | (12)(33)
(B)(B) | (SA)(AC)
BB | {SA, SC, AA, AC}
⇓ | ⇓
∅ | ∅

Since there is S, Start Symbol in (1,5) cell it is valid member

We check if **baaba** is in L(G):

1. We first insert single length rules into our table.

	b	**a**	**a**	**b**	**a**
b	{B}				
a		{A,C}			
a			{A,C}		
b				{B}	
a					{A,C}

Time and Space Complexity :

- **Time Complexity –**

  $O(n^3 . |G|)$

  Where $|G|$ is the number of rules in the given grammar.

- **Space Complexity –**

  $O(n^2)$

A Probabilistic Context-Free Grammar (PCFG)

S	⇒	NP	VP	1.0
VP	⇒	Vi		0.4
VP	⇒	Vt	NP	0.4
VP	⇒	VP	PP	0.2
NP	⇒	DT	NN	0.3
NP	⇒	NP	PP	0.7
PP	⇒	P	NP	1.0

Vi	⇒	sleeps	1.0
Vt	⇒	saw	1.0
NN	⇒	man	0.7
NN	⇒	woman	0.2
NN	⇒	telescope	0.1
DT	⇒	the	1.0
IN	⇒	with	0.5
IN	⇒	in	0.5

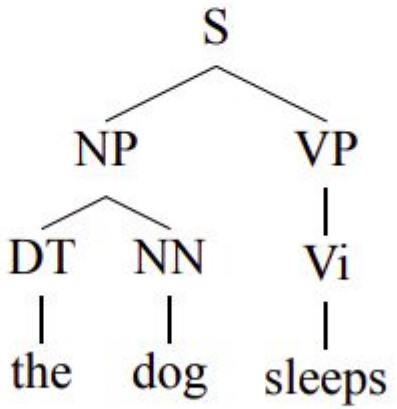
- Probability of a tree t with rules

$$\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n$$

is $p(t) = \prod_{i=1}^{n} q(\alpha_i \rightarrow \beta_i)$ where $q(\alpha \rightarrow \beta)$ is the probability for rule $\alpha \rightarrow \beta$.

$$\sum_{\alpha \rightarrow \beta \in R : \alpha = \texttt{VP}} q(\alpha \rightarrow \beta) = q(\texttt{VP} \rightarrow \texttt{Vi}) + q(\texttt{VP} \rightarrow \texttt{Vt NP}) + q(\texttt{VP} \rightarrow \texttt{VP PP})$$
$$= 0.3 + 0.5 + 0.2$$
$$= 1.0$$

To calculate the probability of any parse tree t, we simply multiply together the q values for the context-free rules that it contains. For example, if our parse tree t is



then we have

$$p(t) = q(\texttt{S} \rightarrow \texttt{NP VP}) \times q(\texttt{NP} \rightarrow \texttt{DT NN}) \times q(\texttt{DT} \rightarrow \texttt{the}) \times q(\texttt{NN} \rightarrow \texttt{dog}) \times q(\texttt{VP} \rightarrow \texttt{Vi}) \times q(\texttt{Vi} \rightarrow \texttt{sleeps})$$

Properties of PCFGs

- Assigns a probability to each *left-most derivation*, or parse-tree, allowed by the underlying CFG
- Say we have a sentence s, set of derivations for that sentence is $\mathcal{T}(s)$. Then a PCFG assigns a probability $p(t)$ to each member of $\mathcal{T}(s)$. i.e., *we now have a ranking in order of probability.*
- The most likely parse tree for a sentence s is

$$\arg\max_{t \in \mathcal{T}(s)} p(t)$$

Parsing using Probabilistic Context Free Grammars

Each parse tree t_i is a sequence of context-free rules: we assume that every parse tree in our corpus has the same symbol, S, at its root. We can then define a PCFG (N, Σ, S, R, q) as follows:

- N is the set of all non-terminals seen in the trees $t_1 \ldots t_m$.
- Σ is the set of all words seen in the trees $t_1 \ldots t_m$.
- The start symbol S is taken to be S.
- The set of rules R is taken to be the set of all rules $\alpha \rightarrow \beta$ seen in the trees $t_1 \ldots t_m$.
- The maximum-likelihood parameter estimates are

$$q_{ML}(\alpha \rightarrow \beta) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

  where $\text{Count}(\alpha \rightarrow \beta)$ is the number of times that the rule $\alpha \rightarrow \beta$ is seen in the trees $t_1 \ldots t_m$, and $\text{Count}(\alpha)$ is the number of times the non-terminal α is seen in the trees $t_1 \ldots t_m$.

For example, if the rule $\texttt{VP} \rightarrow \texttt{Vt NP}$ is seen 105 times in our corpus, and the non-terminal VP is seen 1000 times, then

$$q(\texttt{VP} \rightarrow \texttt{Vt NP}) = \frac{105}{1000}$$

REFERENCES

Basics of PCFG:-https://www.youtube.com/watch?v=wSONlMwa9rE&list=PLlQBy7xY8mbKypSJe_AjVtCuXXsdODiDi&index=3

https://www.youtube.com/watch?v=DjwH9wzCFzg&list=PLlQBy7xY8mbKypSJe_AjVtCuXXsdODiDi&index=4

CYK ALGORITHM:-

https://www.youtube.com/watch?v=xRMn6HK84io

https://www.geeksforgeeks.org/cyk-algorithm-for-context-free-grammar/

PARSING:-

https://www.tutorialspoint.com/compiler_design/compiler_design_types_of_parsing.htm

Documents: http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/pcfgs.pdf