Top Down Parsing

Introduction:

- In computer science, **top-down parsing** is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. LL parsers are a type of parser that uses a top-down parsing strategy.
- Top-down parsing is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.
- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.
- Simple implementations of top-down parsing do not terminate for left-recursive grammars, and top-down parsing with backtracking may have exponential time complexity with respect to the length of the input for ambiguous CFGs. However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially exponential number of parse trees as shown in Fig 1.

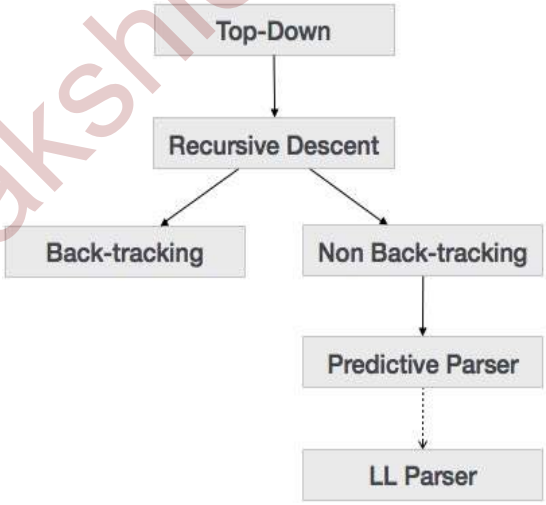


Fig 1

- A parser is top-down if it discovers a parse tree top to bottom. A top-down parse corresponds to a preorder traversal of the parse tree. A leftmost derivation is applied at each derivation step.
- Top-down parsers come in two forms

- Predictive Parsers- Predict the production rule to be applied using lookahead tokens.
- Backtracking Parsers-Will try different productions, backing up when a parse fails. Predictive parsers are much faster than backtracking ones. Predictive parsers operate in linear time. Backtracking parsers operate in exponential time.

- Two kinds of top-down parsing techniques will be studied. Recursive-descent parsing and LL parsing.
- Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars.
  e.g., conditional statement defined by a rule such as;
  if S_1 and S_2 are statements and E is an expression, then
  "**if** E **then** S_1 **else** S_2" is a statement
- This form of conditional statement cannot be specified using the notation of regular expressions.
- CFG consists of terminals, nonterminals, a start symbol, and productions.
  1. **Terminals** are basic symbols from which strings are formed.
  e.g., in programming language; **if**, **then**, and **else** is a terminal.
  2. **Non-terminals** are syntactic variables that denote set of strings.
  e.g., in production: stmt → if expr then stmt else stmt, expr and stmt are nonterminals.
  The non terminals define sets of strings that define the language generated by the grammar. They also impose hierarchical structure on the language that is useful for both syntax analysis and translation. In a grammar, one non terminal is distinguished as the **start symbol**, and the set of strings it denotes is the language defined by the grammar.
  3. The **production** of a grammar specifies the manner in which the terminals and nonterminals are combined to form strings.
  Each production consists of a nonterminal, followed by an arrow, followed by a string of nonterminals and terminals.

Exercise 1:

In the following grammar find terminals, nonterminals, start symbols, and productions:

expr → expr op expr
expr → (expr)
expr → - expr
expr → **id**
op → +
op → -
op → *
op → /

op → ^

Notational Conventions:

To avoid having to state that "these are terminals" and "these are nonterminals", the following **notational conventions** with regard to grammars are used:

1. These symbols are terminals:
   - Lower-case letters early in the alphabet such as a, b, c
   - Operator symbols such as +, -, etc.,
   - Punctuation symbols such as parenthesis, comma, etc.,
   - Boldface strings such as **id** or **if**
2. These symbols are nonterminals:
   - Upper-case letters early in the alphabet such as A, B, C.
   - The letter S, when it appears, is usually the start symbol.
   - Lower-case italic names such as expr or stmt.
3. Upper-case letters late in the alphabet, such as X, Y, Z represent grammar symbols, i.e., either nonterminals or terminals.
4. Lower-case Greek letters, α, β, γ, for example, represent strings of grammar symbols.
5. If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …, $A \rightarrow \alpha_k$ are all productions with A on the left (A-productions), then we can write as; $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$. ($\alpha_1, \alpha_2, \ldots, \alpha_k$, the alternatives for A).

e.g., A sample grammar:

E → E A E | (E) | - E | id

A → + | - | * | / | ^

Derivations:

- Derivation gives a precise description of the top-down constructions of a parse tree.
- In derivation, a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production.

  e.g. E → E + E | E * E | - E | (E) | **id**

  tell us we can replace one instance of an E in any string of grammar symbols by E + E or E * E or – E or (E) or id

  E => - E read as "E derives – E"
- Take single E and repeatedly apply productions in any order to obtain a sequence of replacements.

  e.g., E => - E => - (E) => - (**id**)

  Such a sequence of replacements is called as derivation of – (**id**) from E.

  Symbol => means "derive in one step".

  Symbol $\overset{*}{\Rightarrow}$ means "derives in zero or more step"

  Symbol $\overset{+}{\Rightarrow}$ means "derives in one or more steps"

Given a grammar G with start symbol S, then α relation to define L(G), the language generated by G.

Strings in L(G) may contain only terminal symbols of G.

When string of terminals w is in L(G) if and only if S(w). The string w is called a **sentence** of G.

- A language that can be generated by a grammar is said to be **context-free language.**
  If two grammar generated by the same language, the grammars are said to be **equivalent.**
- If S (α), where α may contain nonterminals, then α is a **sentential form** of G.
  A sentence is a sentential form with no nonterminals.
  Example: using the grammar, the string – (id + id) is a sentence of grammar**.**

E => - E => - (E) => - (E + E) => - (id + E) => - (id + id),

The strings E, - E, - (E + E), - (id + E), - (id + id) appearing in the derivation are all sentential form of the grammar

- It can be written as $E \overset{*}{\Rightarrow}$ - (id + id) to indicate – (id + id) can be derived from E.
  There are two types of derivations: **leftmost derivation** and **rightmost derivation**;
  The derivation in which only the leftmost nonterminal in any sentential form is replaced at each step. Such derivation is called **leftmost derivation**.
  Example: $E \underset{lm}{\Rightarrow}$ - E $\underset{lm}{\Rightarrow}$ - (E) $\underset{lm}{\Rightarrow}$ - (E + E) $\underset{lm}{\Rightarrow}$ - (id + E) $\underset{lm}{\Rightarrow}$ - (id + id)

  If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a **left-sentential form** of the grammar at hand.

  The derivation in which only the rightmost nonterminal in any sentential form is replaced at each step. Such derivation is called **rightmost derivation** or **canonical derivation.**
  Example: $E \underset{rm}{\Rightarrow}$ - E $\underset{rm}{\Rightarrow}$ - (E + E) $\underset{rm}{\Rightarrow}$ - (E + id) $\underset{rm}{\Rightarrow}$ - (id + id)

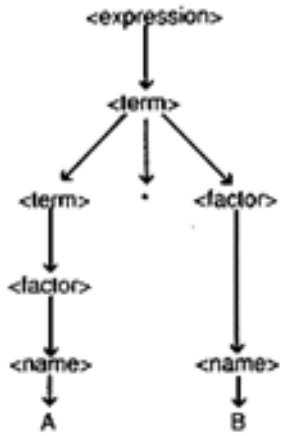
  If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, then α is a right-sentential form of the grammar at hand.

Finding the structure of an expression

Using the grammar above, the string

A * B

can be diagramed:



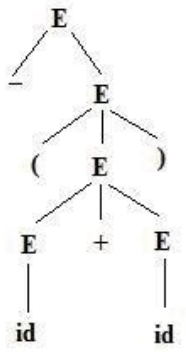
Parse Trees and Derivations:

A **parse tree** may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.

Each interior node of a parse tree is labeled by some nonterminal A, and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this A was replaced in the derivation.

The leaves of the parse tree are labeled by nonterminals or terminals and, read for left to right; they constitute a sentential form, called the yield or frontier of the tree.

Example 1: parse tree for – (id + id),



Parse tree for - (id + id)

Ambiguity:

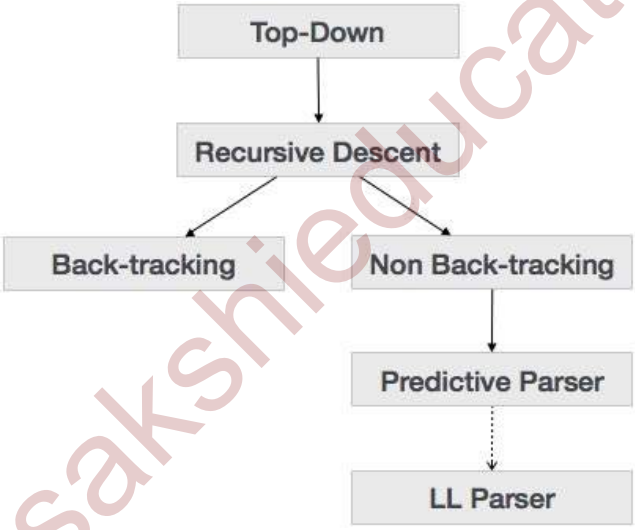
A grammar that produces more than one parse tree for some sentence is said to be **ambiguous.**

An **ambiguous grammar** is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

Example: from the grammar example 1, the sentence id + id * id has two distinct leftmost derivations.

Top-Down Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of n tokens. Top-down parsers build parse trees from the top (root) to the bottom (leaves).



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking

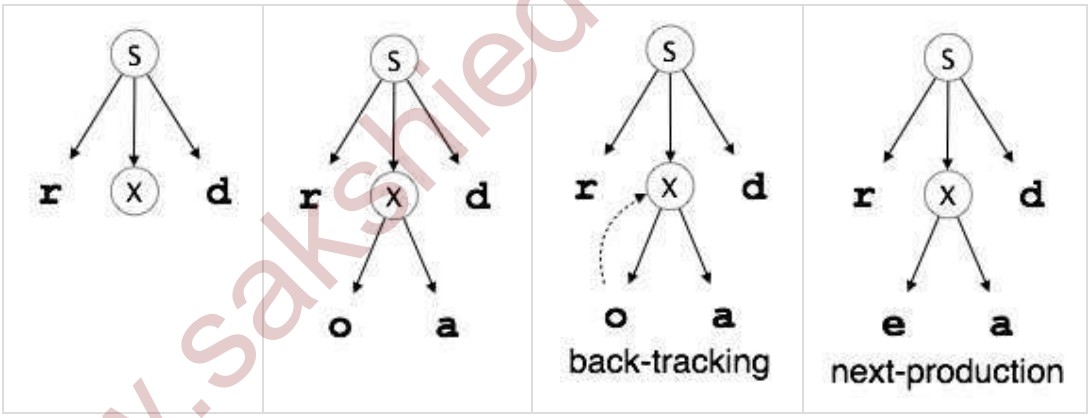
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd
X → oa | ea
Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

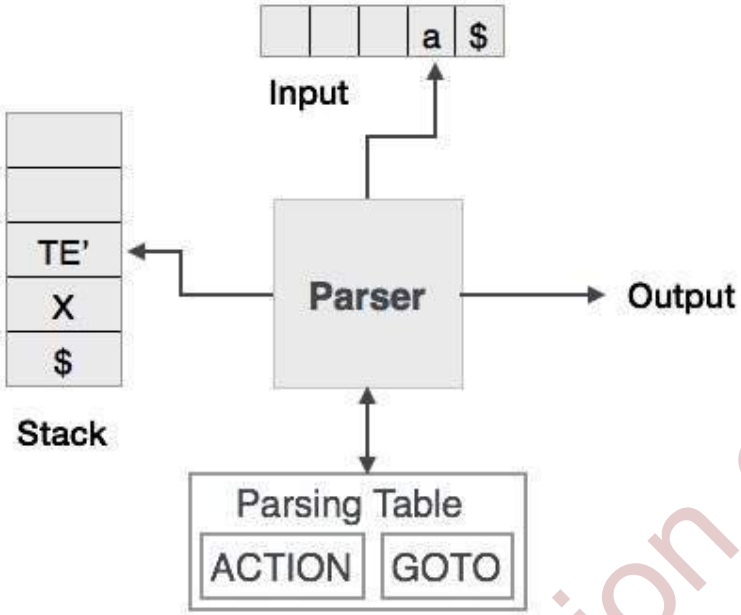
Now the parser matches all the input letters in an ordered manner. The string is accepted.



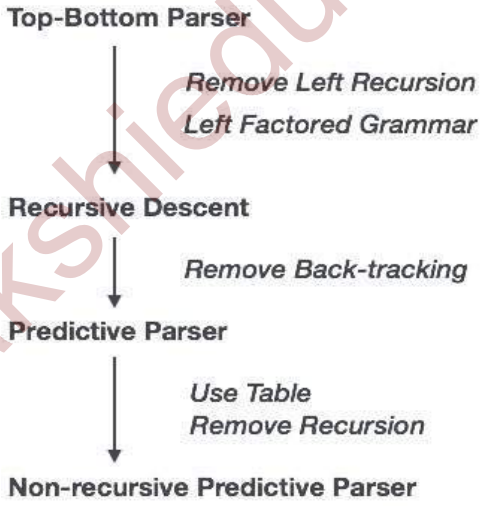
Predictive Parser

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

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



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



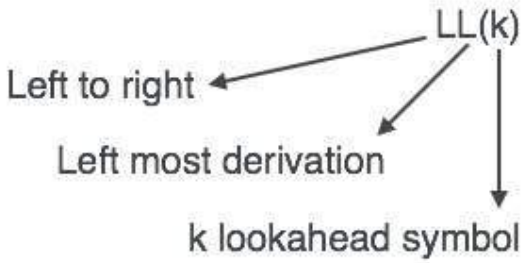
In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure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

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy

implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



LL Parsing Algorithm

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

Input:
string ω
parsing table M for grammar G

Output:
If ω is in L(G) then left-most derivation of ω,
error otherwise.

Initial State : $S on stack (with S being start symbol)
ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat
let X be the top stack symbol and a the symbol pointed by ip.

if X∈ Vt or $
if X = a
POP X and advance ip.
else
error()

```
                endif

        else    /* X is non-terminal */
                if M[X,a] = X → Y1, Y2,... Yk
                        POP X
                        PUSH Yk, Yk-1,... Y1 /* Y1 on top */
                        Output the production X → Y1, Y2,... Yk
                else
                        error()
                endif
        endif
until X = $     /* empty stack */
```

A grammar G is LL(1) if A → α | β are two distinct productions of G:

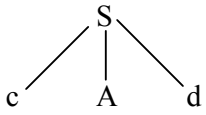
- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

Recursive - Descent Parsing

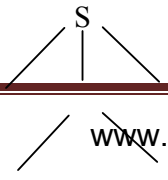
This is general form of top-down parsing , called recursive descent parsing where backtracking may be involved. This is a bad type of parsing which involves repeated trying to get the correct output.This can also be termed as brute-force type of parsing. Presently, this type of parsing is outdated ,just because there are much better methods of parsing which we will be discussing later.

Consider the grammar:
S -> cAd | bd
A -> ab | a
and the input string is “cad”.

To construct the tree, we create an initial tree of just one node S.
The input pointer points to c , and we use the first production, for s to get the expanded tree.



The leftmost leaf labeled c matches the fist symbol of the input and hence we advance the pointer to the second symbol of the input which is a. we now expand A by its first production to obtain the following tree.

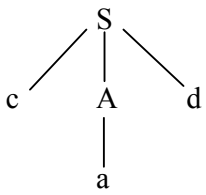


c A d

a b

Now we have a match for the second symbol of the input and hence advabce the pointer to d , and compare it with the next leaf b, which does not match , we report failure and go back to see whether there is an alternative production for A.

In going back to A ,we must backtrack the input pointer to a. Finding another production , we try out the next configuration.



Now the leaf a matches with the second symbol of the input and the the third leaf d matches with the third symbol of the input.

And because the input string is consumed , we halt and denote the successful completion of parsing.

Predictive Parsing:

This is a top down parsing method where we execute a set of recursive set of procedures to process the input. A procedure is associated with a nonterminal of a grammar. Here the lookahead symbol unambiguously determines the procedure selected for each nonterminal . The sequence of procedures called in processing the input implicitly defines a parse tree for the input.

Consider the grammar:

S -> cAd | bd

A -> ab | e

PSEUDO CODE for a predictive parser

```
function match(token t)
{
if lookahead = t          then
        lookahead = nexttoken()
 else error
}

function S
{
if lookahead is in { c }
        match(c) , A(),match (d);
```

```
else if lookahead is in {b }
        match(b) , match(d);
else if lookahead is in {a,e}
        A();
else error
}

function A
{
if lookahead is in { a }
        match(a) , match(b);
else if lookahead is in { e }
        match(e);
else if error
}
```

input string: “ced”

The function match() compares the current lookahead symbol with the argument token and if matched changes the lookahead symbol by advancing the input pointer.

Parsing begins with a call to the procedure for the starting nonterminal S in our grammar. Because the lookahead 'c' is in the set { c } , the function S executes the code:

```
if lookahead is in { c }
match(c) , A(),match (d);
```

once it matched 'c' , the function A() is called and checks out that the next input symbol 'e' is then in the set { e } , it executes the code :

```
else if lookahead is in { e }
  match(e);
```

After the matching of 'e' is over it returns from the function A() and matches the next token with 'd'.