1.7 PARSING

In computer technology, a parser is a program that's usually part of a compiler. It receives input in the form of sequential source program instructions, interactive online commands, markup tags or some other defined interface.

Parsers break the input they get into parts such as the nouns (objects), verbs (methods), and their attributes or options. These are then managed by other programming, such as other components in a compiler. A parser may also check to ensure that all the necessary input has been provided.

How does parsing work?

A parser is a program that is part of the compiler, and parsing is part of the compiling process. Parsing happens during the analysis stage of compilation.

In parsing, code is taken from the preprocessor, broken into smaller pieces and analyzed so other software can understand it. The parser does this by building a data structure out of the pieces of input.

More specifically, a person writes code in a human-readable language like C++ or Java and saves it as a series of text files. The parser takes those text files as input and breaks them down so they can be translated on the target platform.

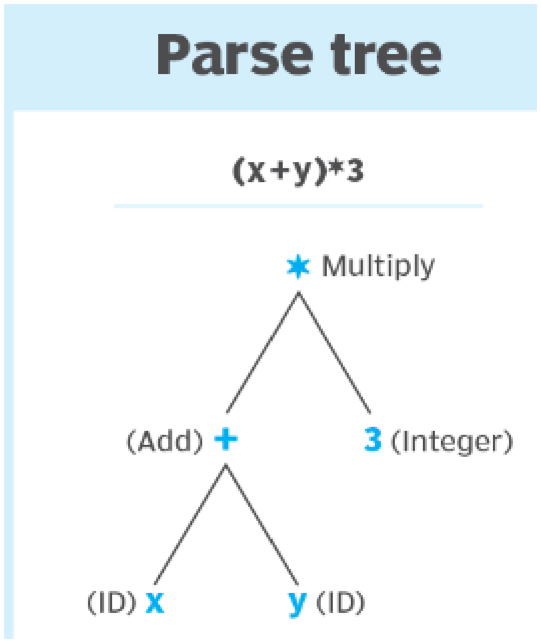
The parser consists of three components, each of which handles a different stage of the parsing process. The three stages are:

Stage 1: Lexical analysis

A lexical analyzer -- or scanner -- takes code from the preprocessor and breaks it into smaller pieces. It groups the input code into sequences of characters called lexemes, each of which corresponds to a token. Tokens are units of grammar in the programming language that the compiler understands.

Lexical analyzers also remove white space characters, comments and errors from the input.

Stage 2: Syntactic analysis



The syntactic analyzer takes (x+y)*3 as input and returns this parse tree, which enables the parser to understand the equation. This stage of parsing checks the syntactical structure of the input, using a data structure called a parse tree or derivation tree. A syntax analyzer uses tokens to construct a parse tree that combines the predefined grammar of the programming language with the tokens of the input string. The syntactic analyzer reports a syntax error if the syntax is incorrect.

Stage 3: Semantic analysis

Semantic analysis verifies the parse tree against a symbol table and determines whether it is semantically consistent. This process is also known as context sensitive analysis. It includes data type checking, label checking and flow control checking.

If the code provided is this:

float a = 30.2; float b = a*20

then the analyzer will treat 20 as 20.0 before performing the operation.

Some sources refer only to the syntactic analysis stage as parsing because it generates the parse tree. They leave out lexical and semantic analysis.

What are the main types of parsers?

When a software language is created, its creators must specify a set of rules. These rules provide the grammar needed to construct valid statements in the language.

The following is a set of grammatical rules for a simple fictional language that only contains a few words:

<sentence> ::= <subject> <verb> <object>
<subject> ::= <article> <noun>
<article> ::= the | a
<noun> ::= dog | cat | person
<verb> ::= pets | fed
<object> ::= <article> <noun>

In this language, a sentence must contain a subject, verb and noun in that order, and specific words are matched to the parts of speech. A subject is an article followed by a noun. A noun can be one of the following three words: *dog*, *cat* or *person*. And a verb can only be *pets* or *fed*.

Parsing checks a statement that a user provides as input against these rules to prove that the statement is valid. Different parsing algorithms check in different orders. There are two main types of parsers:

- **Top-down parsers.** These start with a rule at the top, such as <sentence> ::= <subject> <verb> <object>. Given the input string "The person fed a cat," the parser would look at the first rule, and work its way down all the rules checking to make sure they are correct. In this case, the first word is a <subject>, it follows the subject rule, and the parser will continue reading the sentence looking for a <verb>.

- **Bottom-up parsers.** These start with the rule at the bottom. In this case, the parser would look for an <object> first, then look for a <verb> next and so on.

More simply put, top-down parsers begin their work at the start symbol of the grammar at the top of the parse tree. They then work their way down from the rule to the sentence. Bottom-up parsers work their way up from the sentence to the rule.

Beyond these types, it's important to know the two types of derivation. Derivation is the order in which the grammar reconciles the input string. They are:

- **LL parsers.** These parse input from left to right using leftmost derivation to match the rules in the grammar to the input. This process derives a string that validates the input by expanding the leftmost element of the parse tree.
- **LR parsers.** These parse input from left to right using rightmost derivation. This process derives a string by expanding the rightmost element of the parse tree.

In addition, there are other types of parsers, including the following:

- **Recursive descent parsers.** Recursive descent parsers backtrack after each decision point to double-check accuracy. Recursive descent parsers use top-down parsing.
- **Earley parsers.** These parse all context-free grammars, unlike LL and LR parsers. Most real-world programming languages do not use context-free grammars.
- **Shift-reduce parsers.** These shift and reduce an input string. At each stage in the string, they reduce the word to a grammar rule. This approach reduces the string until it has been completely checked.

What technologies use parsing?

Parsers are used when there is a need to represent input data from source code abstractly as a data structure so that it can be checked for the correct syntax. Coding languages and other technologies use parsing of some type for this purpose.

Technologies that use parsing to check code inputs include the following:

Programming languages. Parsers are used in all high-level programming languages, including the following:

- C++
- Extensible Markup Language or XML
- Hypertext Markup Language or HTML
- Hypertext Preprocessor or PHP
- Java
- JavaScript
- JavaScript Object Notation or JSON
- Perl
- Python

Database languages. Database languages such as Structured Query Language also use parsers.

Protocols. Protocols like the Hypertext Transfer Protocol and internet remote function calls use parsers.

Parser generator. Parser generators take grammar as input and generate source code, which is parsing in reverse. They construct parsers from regular expressions, which are special strings used to manage and match patterns in text.

1.8 RECURSIVE DESCENT PARSER

Recursive Descent Parser uses the technique of Top-Down Parsing without backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a Recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

The major approach of recursive-descent parsing is to relate each non-terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is prescribed by the productions for the equivalent non-terminal.

The recursive procedures can be simply to write and adequately effective if written in a language that executes the procedure call effectively. There is a procedure for each non-terminal in the grammar. It can consider a global variable lookahead, holding the current input token and a procedure match (Expected Token) is the action of recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. Match () is effectively a call to the lexical analyzer to get the next token.

For example, input stream is a + b$.

lookahead == a

match()

lookahead == +

match ()

lookahead == b

……………………….

……………………….

In this manner, parsing can be done.

Example − In the following Grammar, first symbol, i.e., if, while & begin uniquely determine, which of the alternative to choose.

As Statement starting with if will be a conditional statement & statement starting with while will be an Iterative Statement.

Stmt → If condition then Stmt else Stmt

| While condition do Stmt

| begin Stmt end.

Example − Write down the algorithm using Recursive procedures to implement the following Grammar.

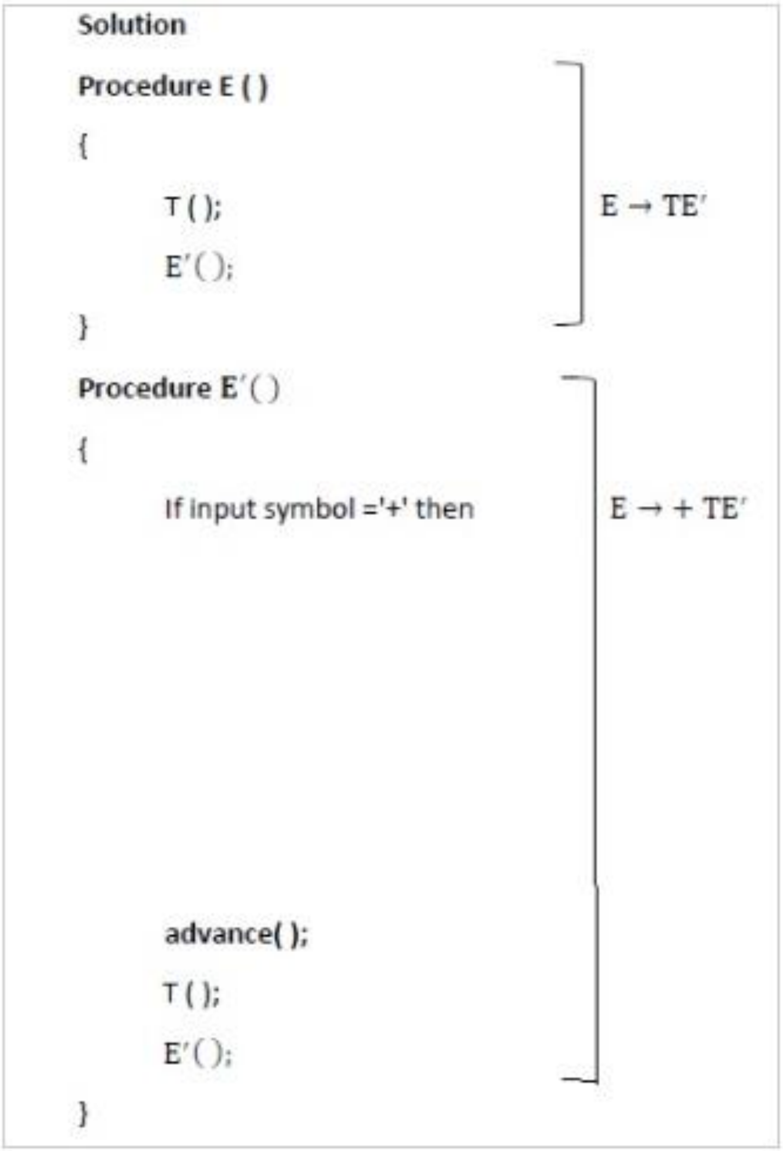
$E \to TE'$

$E' \to +TE'$

$T \to FT'$

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

$F \to (E) \mid id$



Solution

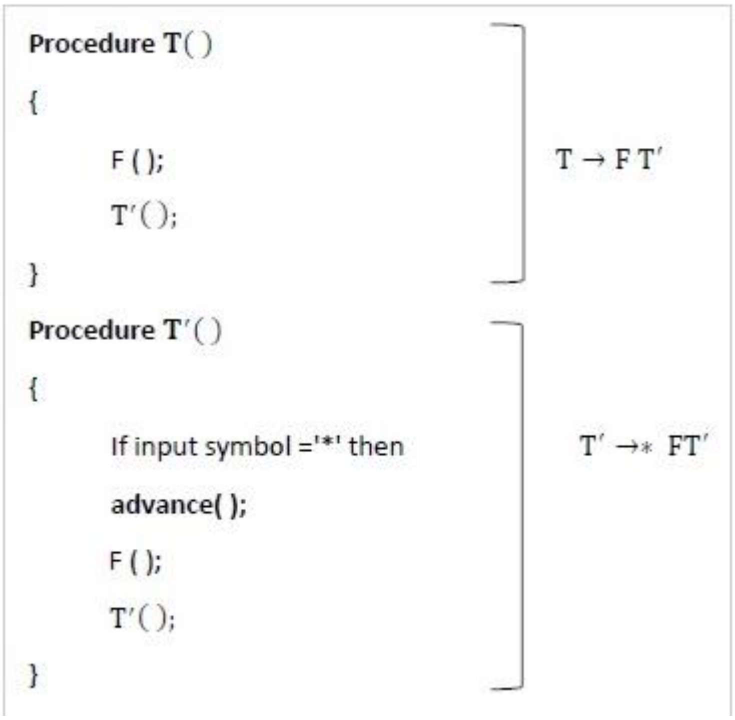
```
Procedure E ( )
{
        T ( );                          E → TE′
        E′( );
}
Procedure E′( )
{
        If input symbol ='+' then       E → + TE′







        advance( );
        T ( );
        E′( );
}
```

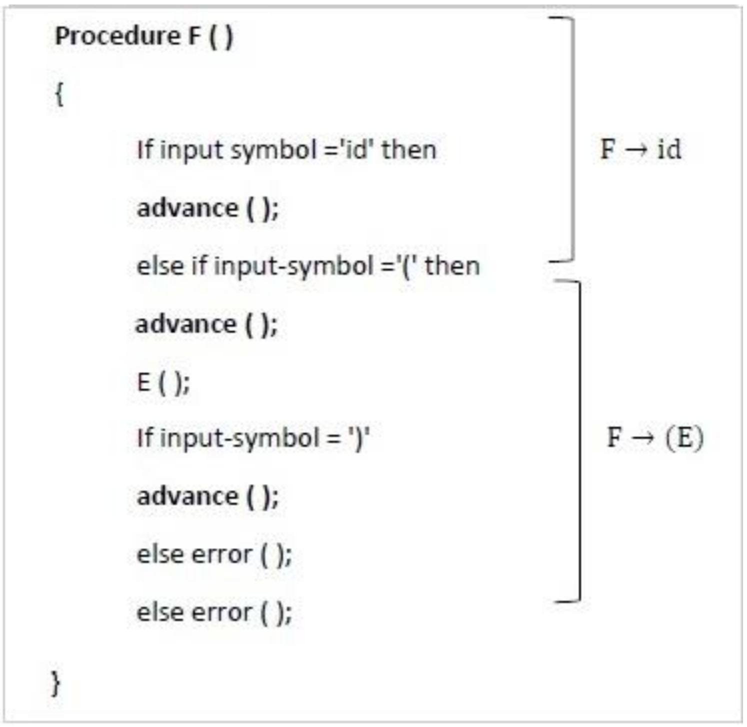


```
Procedure T( )
{
        F ( );
        T′( );
}
```

$T \rightarrow F\,T'$

```
Procedure T′( )
{
        If input symbol ='*' then
        advance( );
        F ( );
        T′( );
}
```

$T' \rightarrow *\ FT'$



```
Procedure F ( )
{
        If input symbol ='id' then
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'
        advance ( );
        else error ( );
        else error ( );
}
```

$F \rightarrow id$

$F \rightarrow (E)$

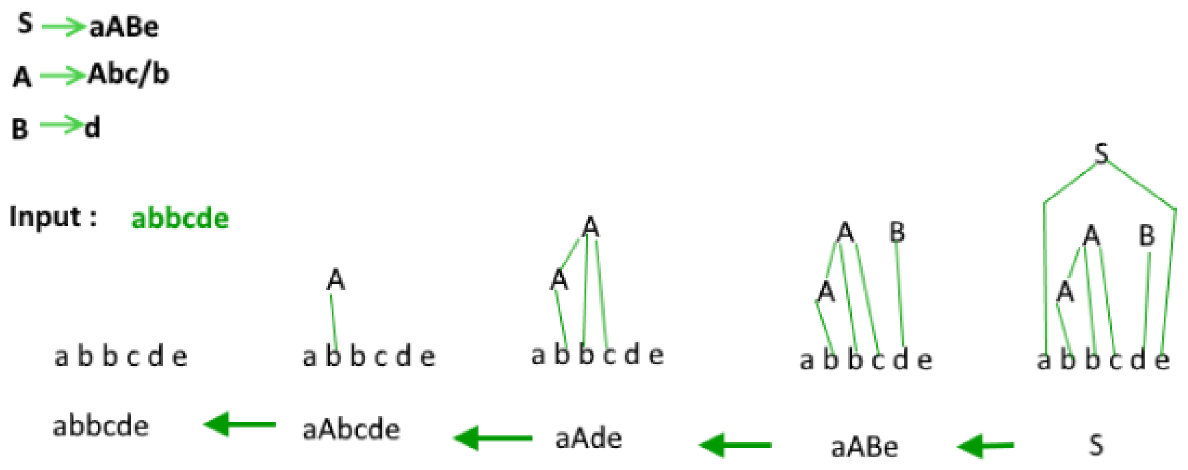
One of major drawback or recursive-descent parsing is that it can be implemented only for those languages which support recursive procedure calls and it suffers from the problem of left-recursion.

1.9 BOTTOM UP PARSING

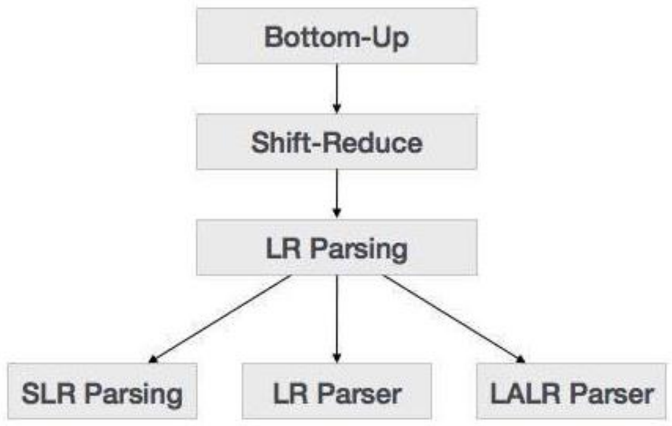
Bottom-up Parsers / Shift Reduce Parsers

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.

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. Eg.



<u>Classification of Bottom-up Parsers:</u>



Shift-Reduce Parsing

Shift-reduce parsing uses 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

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction

- LR(1) – LR Parser:
    - Works on complete set of LR(1) Grammar
    - Generates large table and large number of states
    - Slow construction
- LALR(1) – Look-Ahead LR Parser:
    - Works on intermediate size of grammar
    - Number of states are same as in SLR(1)

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.