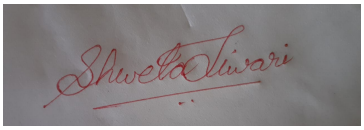
CD: COMPILER DESIGN
CD: UNIT-2 09/2022

SEPTEMBER 2022 / IT-3rd year, Vth semester
FALL SEMESTER, YEAR (Vth, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College

CD: COMPILER DESIGN

INSTRUCTOR: Ms. SHWETA TIWARI
shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com



September 2022

TOPIC On : UNIT-2 BASIC PARSERS

By SHWETA TIWARI

Under On: Basic Parsing Techniques

TOPIC On : UNIT-2 BASIC PARSERS

II. PARSER

SYNTAX ANALYSIS:

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant advantages to both language designers and compiler writers.

- Grammar gives a precise, yet easy-to-understand. Syntactic specification of a programming language.
- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.
- A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors. Tools are available for converting grammar-based descriptions of translations into working programs.

Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

ROLE OF THE PARSER:

- Parser for any grammar is a program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w

is not a valid sentences of given grammar.

- The goal of the parser is to determine the syntactic validity of a source string is valid; a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue a diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbols to a non-terminal.

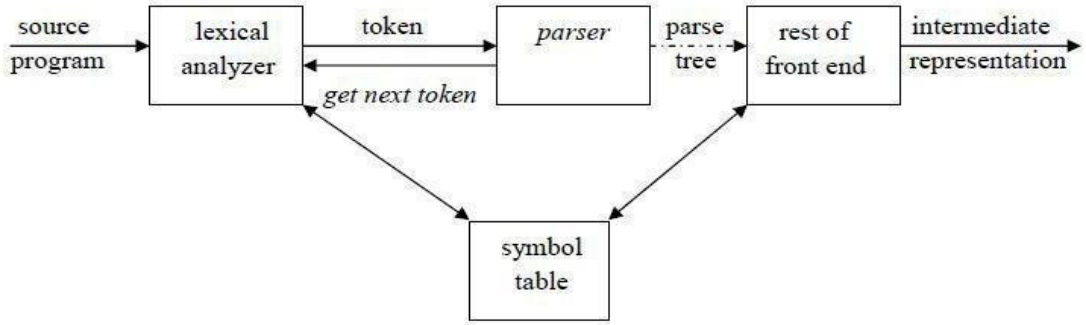


Fig 1 Role of Parser

CONTEXT FREE GRAMMARS

A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammar for programming languages.
2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
3. 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.
4. The productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings. Each production consists of a non

terminal, followed by an arrow, followed by a string of non terminals and terminals.

Inherently recursive structures of a programming language are defined by a context-free Grammar. In context-free grammar, we have four triples G(V,T,P,S). Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic- variables).P is a finite set of productions rules in the following form A → α where A is a non- terminal and α is a string of terminals and non-terminals (including the empty string).S is a start symbol (one of the non-terminal symbol).

L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff S ω where ω is a string of terminals of G. If G is a context-free grammar (G) is a context-free language. Two grammars G1 and G2 are equivalent, if they produce the same grammar.

Consider the production of the form S α, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

Example: Consider the grammar for simple arithmetic expressions:

expr → expr op expr

expr → (expr)

expr → - expr

expr → **id**

op → +

op → -

op → *

op → /

op → ^

Terminals : id + - * / ^ ( )

Non-terminals : expr , op

Start symbol : expr

Notational Conventions:

1. These symbols are terminals:

i. Lower-case letters early in the alphabet such as a, b, c.

ii. Operator symbols such as +, -, etc.

iii. Punctuation symbols such as parentheses, comma etc.

iv. Digits 0,1,...,9.

v. Boldface strings such as id or if (keywords)

2. These symbols are non-terminals:

   i. Upper-case letters early in the alphabet such as A, B, C..

   ii. The letter *S*, when it appears, is usually the start symbol.

   iii. 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, that is either terminals or non-terminals.

4. Greek letters α , β , γ represent strings of grammar symbols.

   e.g a generic production could be written as $A \rightarrow \alpha$.

5. If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ... , $A \rightarrow \alpha_n$ are all productions with A , then we can write A $\rightarrow \alpha_1 \mid \alpha_2 \mid . . . . \mid \alpha_n$, (alternatives for A).

6. Unless otherwise stated, the left side of the first production is the start

symbol. Using the shorthand, the grammar can be written as:

E → E A E | (E) | - E | **id**

A → + | - | * | / | ^

Derivations:

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

$\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in $\geq$ one step

$\overset{*}{\underset{G}{\Rightarrow}}$ indicates that the derivation utilizes the rules of grammar G

To create a string from a context-free grammar:

- Begin the string with a start symbol.
- Apply one of the production rules to the start symbol on the left-hand

side by replacing the start symbol with the right-hand side of the production.

- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

In general a derivation step is αAβ αγβ in sentential form and if there is a production rule A→γ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols α1 α2... αn (αn derives from α1 or α1 derives αn). There are two types of derivation:

1. Leftmost Derivation (LMD):

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.
- The sentential form derived by the left-most derivation is called the left-sentential form.

2. Rightmost Derivation (RMD):

- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
- The sentential form derived from the right-most derivation is called the right-sentential form.

Example:

Consider the G,

E → E + E | E * E | (E) | - E | **id**

Derive the string **id + id * id** using leftmost derivation and rightmost derivation.

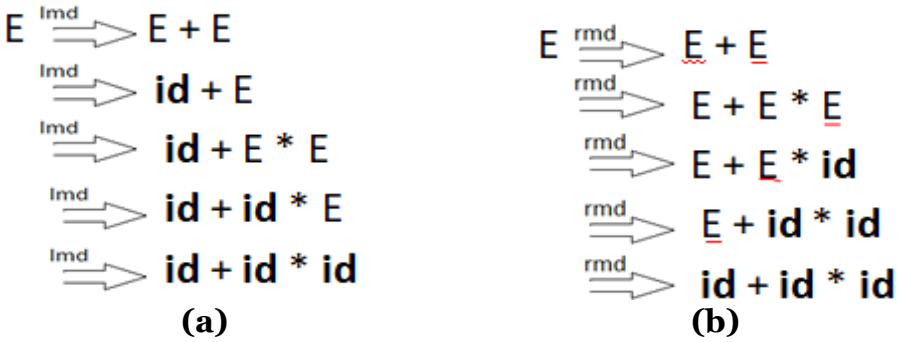


Fig 2. a) Leftmost derivation b) Rightmost derivation

Strings that appear in leftmost derivation are called left sentential forms. Strings that appear in rightmost derivation are called right sentential forms.

Sentential Forms:

Given a grammar G with start symbol S, if S => α , where α may contain non-terminals or terminals, then α is called the sentential form of G.

Parse Tree:

A parse tree is a graphical representation of a derivation sequence of a sentential form. In a parse tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

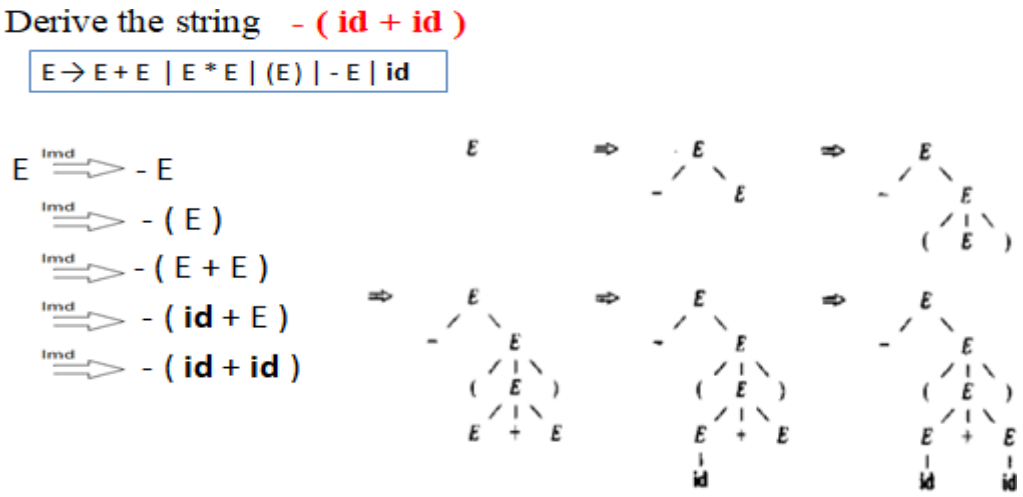


Fig 3 Build the parse tree for string **–(id+id)** from the derivation

Yield or frontier of tree:

Each interior node of a parse tree is non-terminal. The children of nodes can be a terminal or non-terminal of the sentential forms that are read from left to right. The sentential form in the parse tree is called yield or frontier of the tree.

Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar. i.e. An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E lmd⇒ E + E
lmd⇒ **id** + E
lmd⇒ **id** + E * E
lmd⇒ **id** + **id** * E
lmd⇒ **id** + **id** * **id**

E lmd⇒ E * E
lmd⇒ E + E * E
lmd⇒ **id** + E * E
lmd⇒ **id** + **id** * E
lmd⇒ **id** + **id** * **id**

The two corresponding parse trees are:

Fig 4 Two Parse tree for id+id*id

Consider another example,

stmt → if expr then stmt | if expr then stmt else stmt | other

This grammar is ambiguous since the string if E1 then if E2 then S1 else S2 has the following Two parse trees for leftmost derivation :

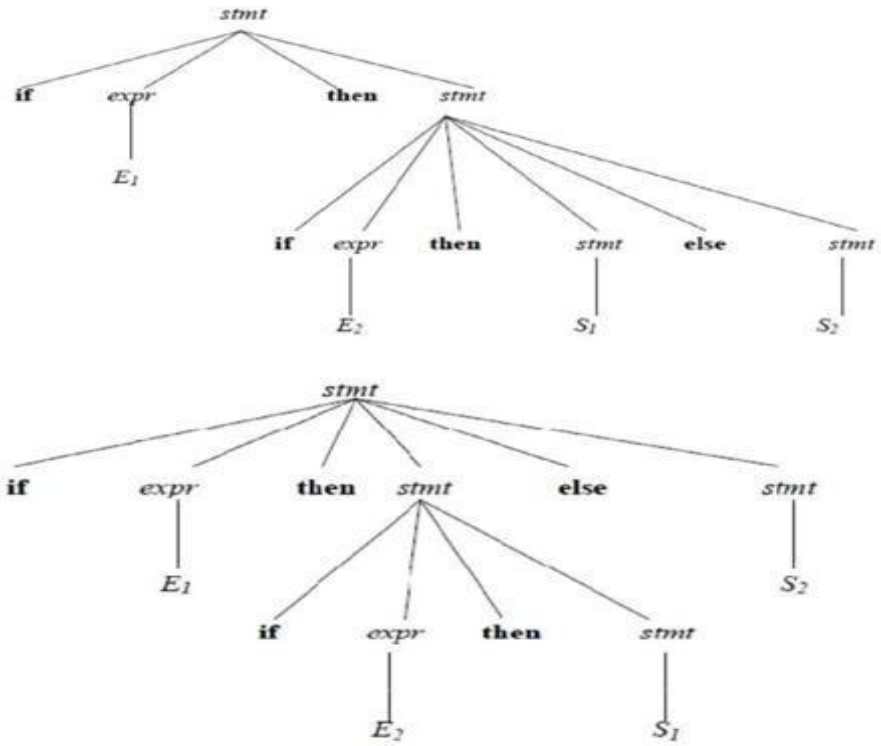


Fig 5 Two Parse tree for **if E1 then if E2 then S1 else S2**

S2 Eliminating Ambiguity:

An ambiguous grammar can be rewritten to eliminate the ambiguity. e.g.

Eliminate the ambiguity from **"dangling-else"** grammar,

stmt → **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| other

Match each else with the closest previous unmatched then. This disambiguation rule can be incorporated into the grammar.

stmt → matched_stmt | unmatched_stmt

matched_stmt →if expr then matched_stmt else matched_stmt

| other

unmatched_stmt → if expr then

stmt

| if expr then matched_stmt else unmatched_stmt

This grammar generates the same set of strings, but allows only one parsing for string.

Table 1 Ambiguous grammar vs. Unambiguous grammar

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.

Removing Ambiguity by Precedence & Associativity Rules:

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules:

Rule-1:

- The level at which the production is present defines the priority of the operator contained in it.
  - The higher the level of the production, the lower the priority of the operator.
  - The lower the level of the production, the higher the priority of the operator.

Rule-2:

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

Example: Consider the ambiguous Grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \mathbf{id}$$

Introduce new variable / non-terminals at each level of precedence,

- an expression **E** for our example is a sum of one or more terms. (+,-)
- a term **T** is a product of one or more factors. (*, /)
- a factor **F** is an identifier or parenthesised expression.

The resultant unambiguous grammar is:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

Trying to derive the string **id+id*id** using the above grammar will yield one unique derivation.

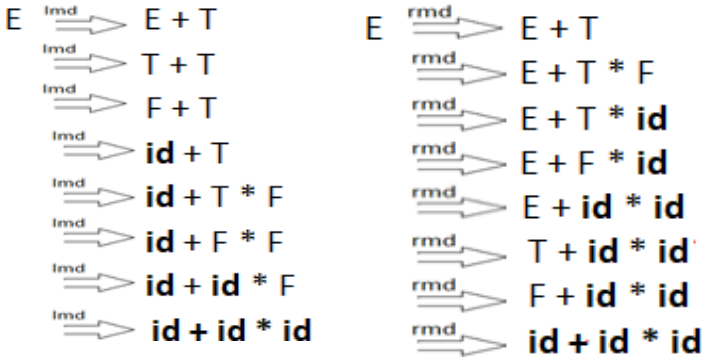


Fig 6 Distinct Leftmost and Rightmost derivation

Regular Expression vs. Context Free Grammar:

- Every construct that can be described by a regular expression can be described by a grammar.
- NFA can be converted to a grammar that generates the same language as

recognized by the NFA.

- Rules:
  - For each state *i* of the NFA, create a non-terminal symbol A_i .
  - If state *i* has a transition to state *j* on symbol ***a***, introduce the production $A_i \rightarrow a\ A_j$
  - If state *i* goes to state *j* on symbol ε, introduce the production $A_i \rightarrow A_j$
  - If *i* is an accepting state, introduce $A_i \rightarrow \varepsilon$
  - If *i* is the start state make A_i the start symbol of the grammar.

Example: The regular expression (a|b)*abb, consider the NFA

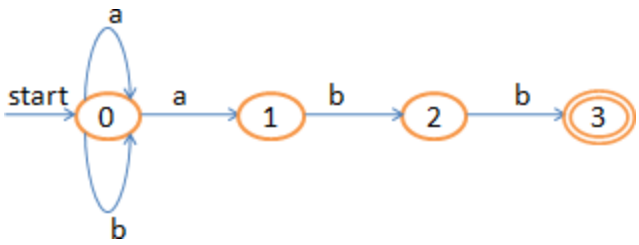


Fig 7 NFA for (a|b)*abb

Equivalent grammar is given by:

$A_0 \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1\ A_1 \rightarrow b\ A_2$

$A_2 \rightarrow b\ A_3\ A_3$

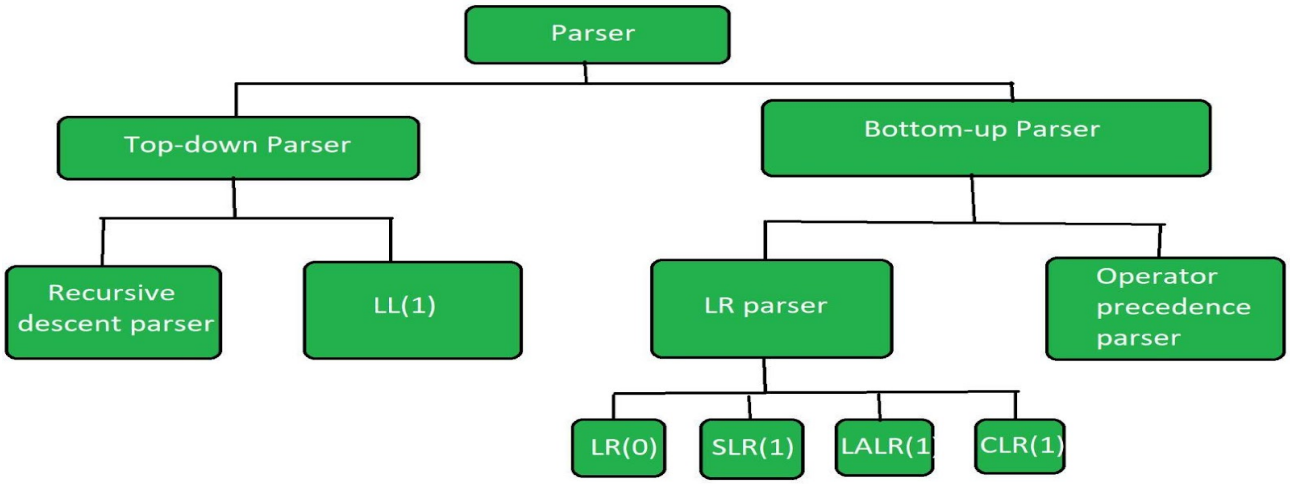
$\rightarrow \varepsilon$

****

****

****

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation (IR). The parser is also known as *Syntax Analyzer*.



Classification of Parser

Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

Top-Down Parser:

The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses the leftmost derivation.

Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

1. **Recursive descent parser** is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

- **LR parser** is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.
  LR parser is of four types:

(a)LR(0)
(b)SLR(1)
(c)LALR(1)

(d)CLR(1)

- **<u>Operator precedence parser</u>** generates the parse tree from given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears on the right-hand side of any production.
- The operator precedence parsing techniques can be applied to **Operator grammars.**
- **Operator grammar:** A grammar is said to be operator grammar if there does not exist any production rule on the right-hand side.

  1. as ε(Epsilon)

  2. Two non-terminals appear consecutively, that is, without any terminal between them operator precedence parsing is not a simple technique to apply to most the language constructs, but it evolves into an easy technique to implement where a suitable grammar may be produced.

****
****
****

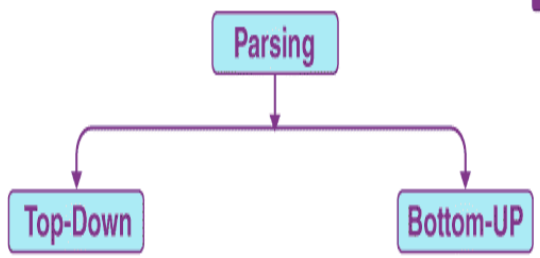
What is Parsing in Compiler Design?

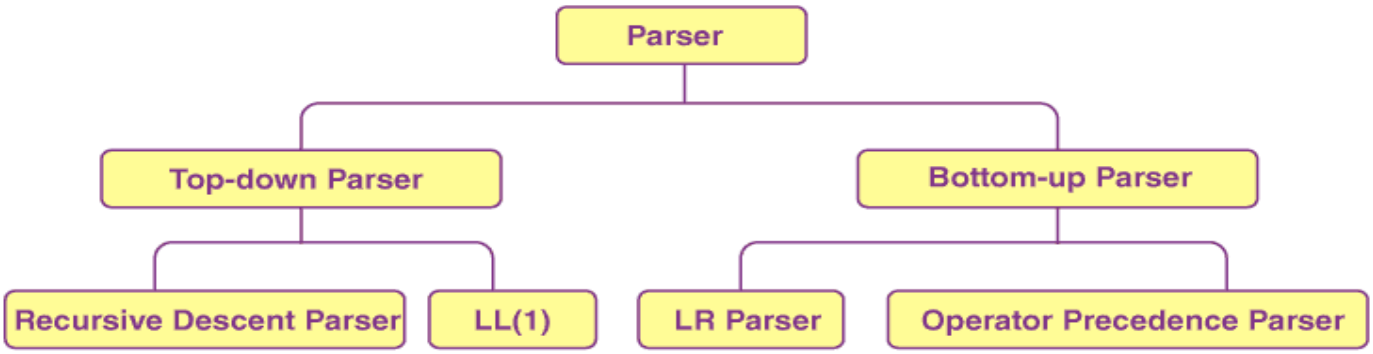
The process of transforming the data from one format to another is called Parsing. This process can be accomplished by the parser. The parser is a component of the translator that helps to organize linear text structure following the set of defined rules which is known as grammar.

Types of Parsing:

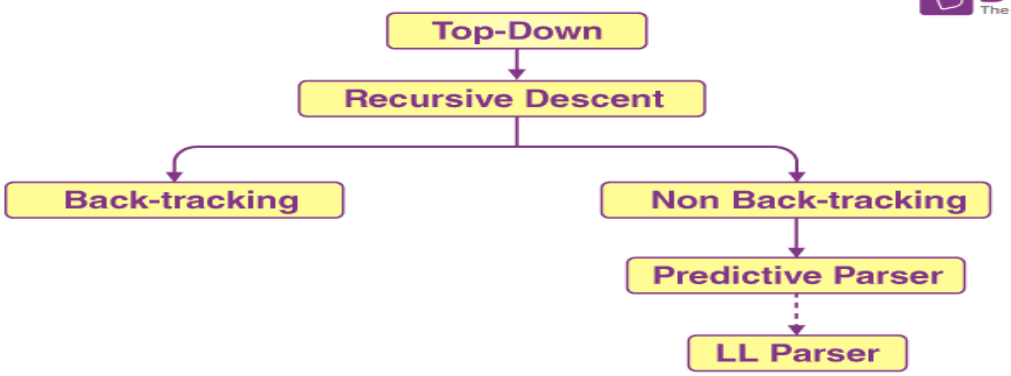
There are two types of Parsing:

- The Top-down Parsing
- The Bottom-up Parsing

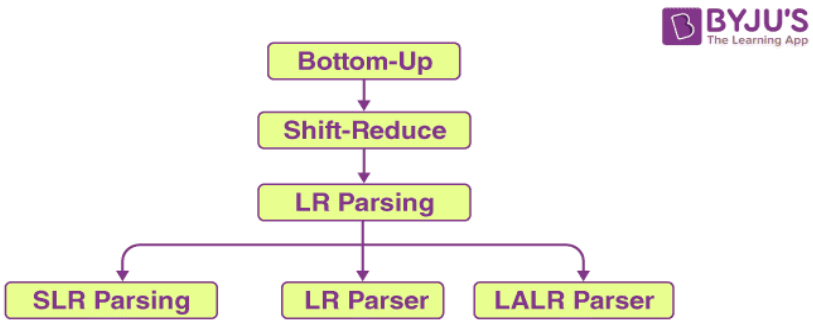




Top-down Parsing: When the parser generates a parse with top-down expansion to the first trace, the left-most derivation of input is called top-down parsing. The top-down parsing initiates with the start symbol and ends on the terminals. Such parsing is also known as predictive parsing.



- **Recursive Descent Parsing:** Recursive descent parsing is a type of top-down parsing technique. This technique follows the process for every terminal and non-terminal entity. It reads the input from left to right and constructs the parse tree from right to left. As the technique works recursively, it is called recursive descent parsing.
    - **Back-tracking:** The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules.



Bottom-up Parsing: The bottom-up parsing works just the reverse of the top-down parsing. It first traces the rightmost derivation of the input until it reaches the start symbol.

- **Shift-Reduce** Parsing: Shift-reduce parsing works on two steps: Shift step and Reduce step.
    - **Shift step:** The shift step indicates the increment of the input pointer to the next input symbol that is shifted.
    - **Reduce Step:** When the parser has a complete grammar rule on the right-hand side and replaces it with RHS.
- **LR Parsing:** LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to right tracing, and R stands for the right to left tracing.

Why is parsing useful in compiler designing?

In the world of software, every different entity has its criteria for the data to be processed. So parsing is the process that transforms the data in such a way so that it can be understood by any specific software.

The Technologies Use Parsers:

- The programming languages like Java.
- Database languages like SQL.
- The Scripting languages.
- The protocols like HTTP.
- The XML and HTML.

Practice Problem Related To Parsing in Compiler Design

Q. Which one of the following kinds of derivation is used by LR parsers?

- (A) Leftmost
- (B) Leftmost in reverse
- (C) Rightmost
- (D) Rightmost in reverse

Q. Which of the following statements about parser is/are CORRECT?

I. Canonical LR is more powerful than SLR

II. SLR is more powerful than LALR.

III. SLR is more powerful than Canonical LR.

- (A) I only
- (B) II only
- (C) III only
- (D) II and III only

Frequently Asked Questions on Parsing in Compiler Design

What is LR Parsing?

LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to right tracing, and R stands for the right to left tracing.

What is parsing and its types?

Parser is a compiler that is used to break the data into smaller elements coming from the lexical analysis phase.

What is parsing in programming?

To parse, in computer science, is where a string of commands – usually a program – is separated into more easily processed components, which are analyzed for correct syntax and then attached to tags that define each component.

The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

**********THE END**********