MCA Part III
Paper- XIX

Topic: Parsing

Prepared by: Dr. Kiran Pandey

School of Computer science

Email-Id: kiranpandey.nou@gmail.com

INTRODUCTION

The output of syntax analysis is a parse tree. Parse tree is used in the subsequent phases of compilation. This process of analyzing the syntax of the language is done by a module in the compiler called parser. The process of verifying whether an input string matches the grammar of the language is called parsing. The syntax analyzer gets the string of tokens from the lexical analyzer. It then verifies the syntax of input string by verifying whether the input string can be derived from the grammar or not. If the input string is derived from the grammar then the syntax is correct otherwise it is not derivable and the syntax is wrong. The parser will report syntactical errors in a manner that is easily understood by the user. It has procedures to recover from these errors and to continue parsing action. The output of a parser is a parse tree. The figure below shows the position of a parser in the compiler.

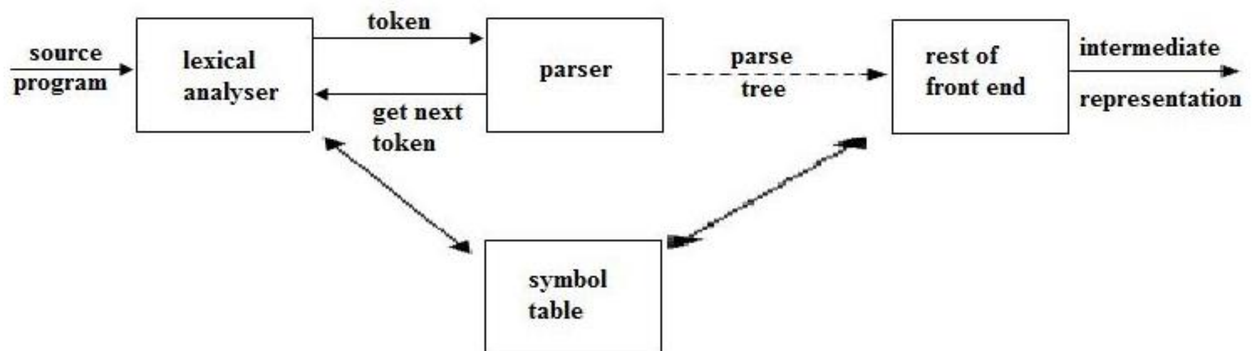


Figure 1: Position of parser in a Compiler.

TYPES OF PARSING

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.

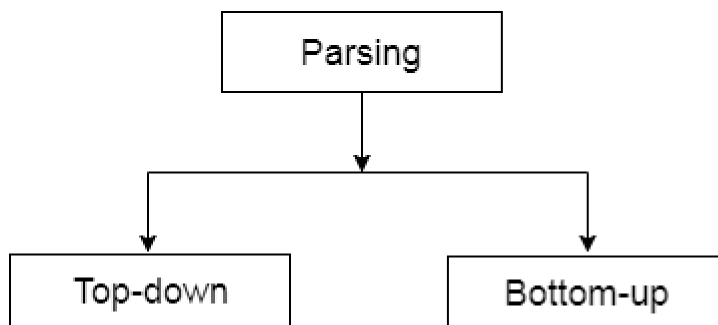
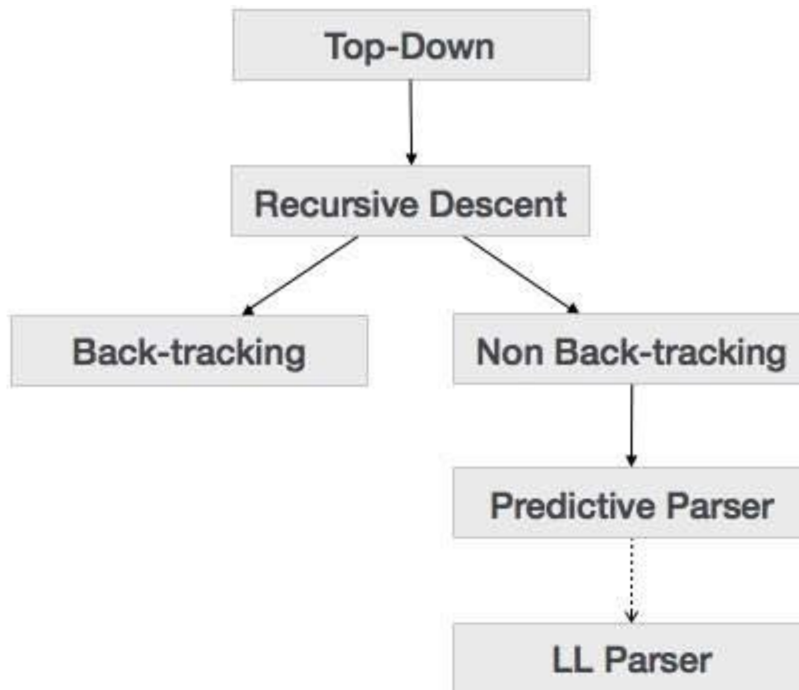


Figure 2: Types of Parsing

TOP DOWN PARSING,

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



(i) 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.

(a) Back-tracking

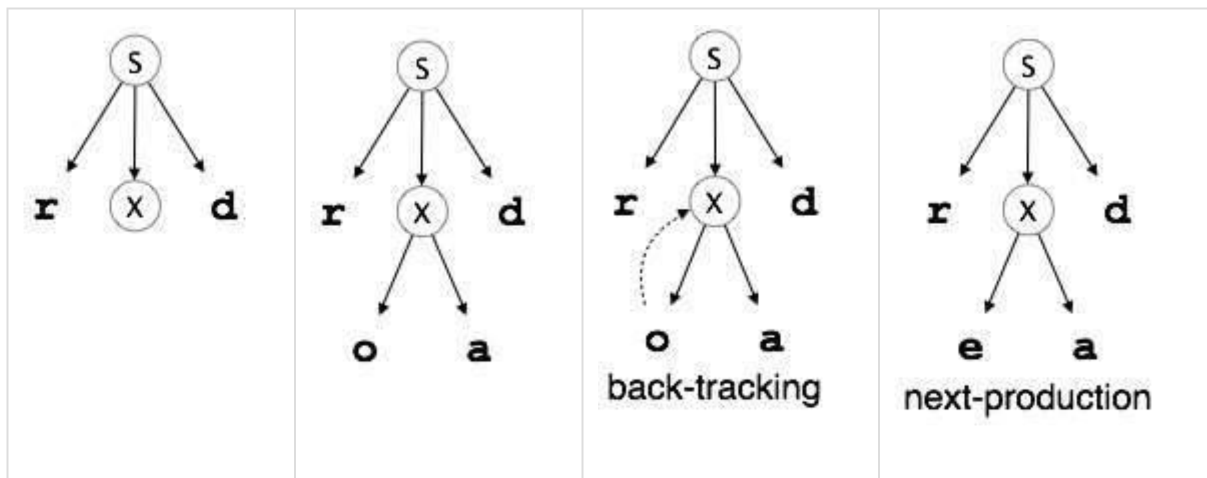
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

```
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).

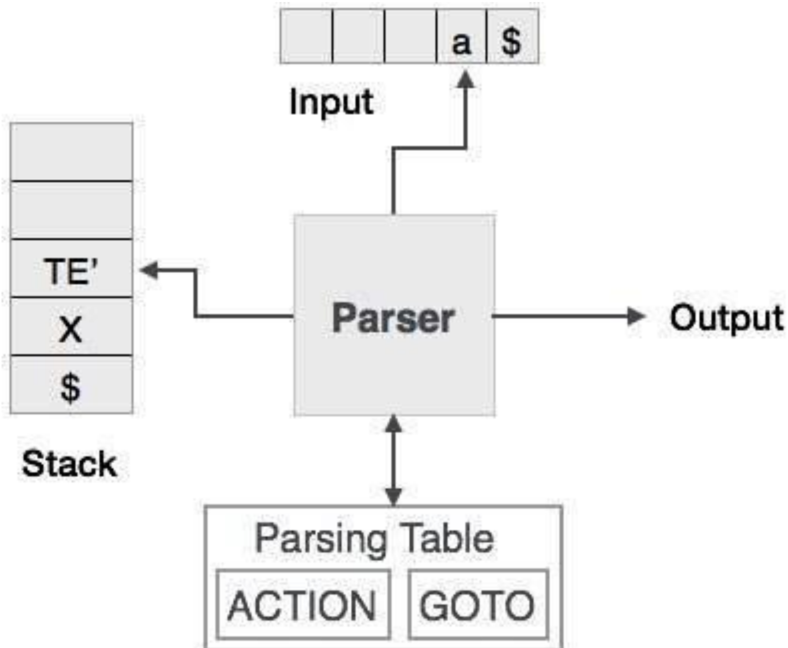
Now the parser matches all the input letters in an ordered manner. The string is accepted.



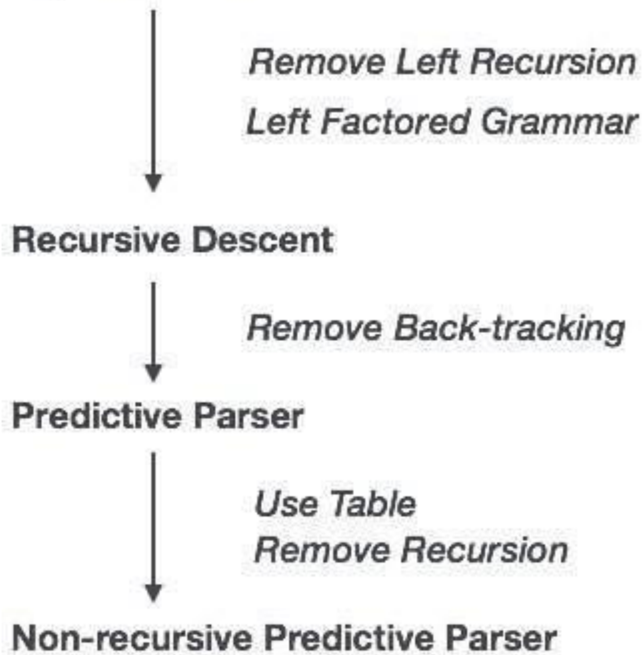
(b) Predictive Parser

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.



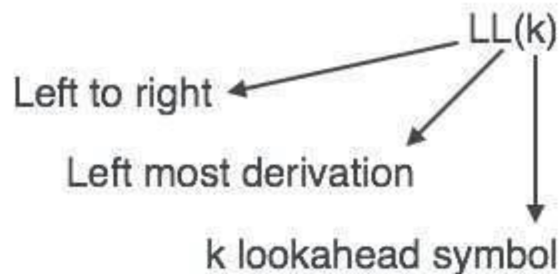
In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most

one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

(c) 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).

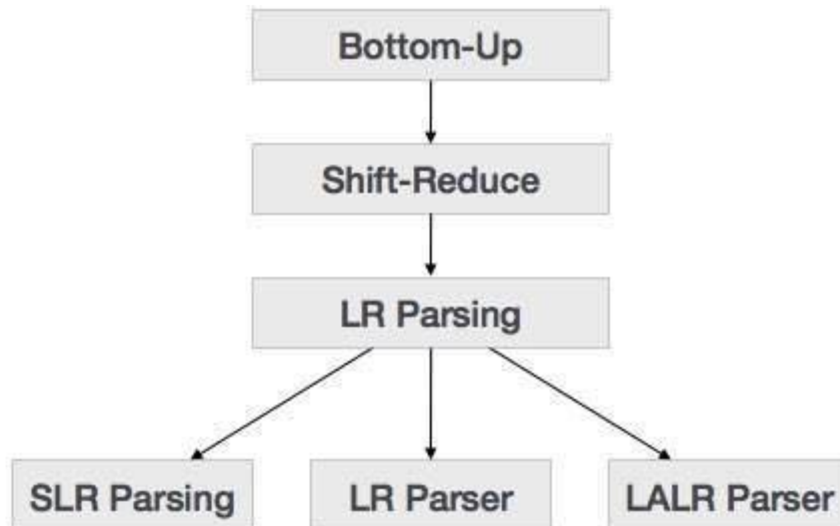


A grammar G is LL(1) if $A \rightarrow \alpha \mid \beta$ are two distinct productions of G:

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

BOTTOM UP PARSING

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. The image given below depicts the bottom-up parsers available.



(i) 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.

(ii) 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

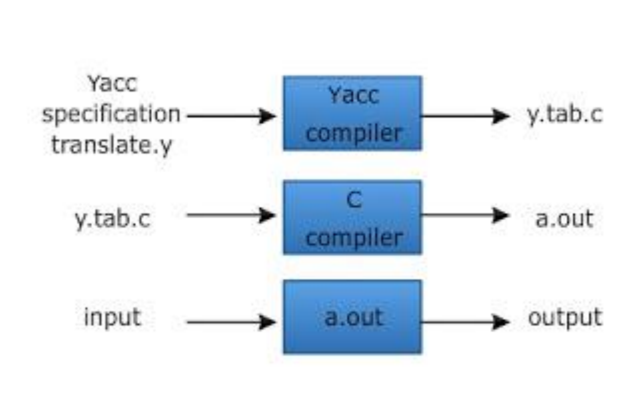
    - Simple and fast construction
- LR(1) – LR Parser:
    - Works on complete set of LR(1) Grammar
    - Generates large table and large number of states
    - Slow construction
- LALR(1) – Look-Ahead LR Parser:
    - Works on intermediate size of grammar
    - Number of states are same as in SLR(1)

Comparing LL and LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.

Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

YACC- AUTOMATIC PARSER GENERATOR



Creating an Input/Output Translator with YACC

- YACC is a automatic tool that generates the parser program
- YACC stands for Yet Another Compiler Compiler. This program is available in UNIX OS
- The construction of LR parser requires lot of work for parsing the input string. Hence, the process must involve automation to achieve efficiency in parsing an input
- Basically YACC is a LALR parser generator that reports conflicts or uncertainties (if at all present) in the form of error messages
- The typical YACC translator can be represented as shown in the image

YACC Specification:

The YACC specification file consists of three parts

(i) **Declaration section:** In this section, ordinary C declarations are inserted and grammar tokens are declared. The tokens should be declared between %{ and %}

(ii) Translation rule section

It includes the production rules of context free grammar with corresponding actions

Example:

Rule-1 action-1

Rule-2 action-2
:
:
Rule n action n
If there is more than one alternative to a single rule then those alternatives are separated by '|' (pipe) character. The actions are typical C statements. If CFG is
LHS: alternative 1 | alternative 2 | …… alternative n
Then
LHS: alternative 1 {action 1}
| alternative 2 {action 1}
:
:
alternative n {action n}

(iii) **C functions Section:** this consists of one main function in which the routine yyparse() is called. And it also contains required C functions

The specification file comprising these sections can be written as:

```
%{

/ * declaration section */

%}

/* Translation rule section */

%%

/* Required C functions */
```

Example:
YACC Specification of a simple desk calculator:

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line: expr '\n'     { printf("%d\n", $1); }
          ;
expr : expr '+' term        { $$ = $1 + $3; }
          | term
          ;
term : term '*' factor    { $$ = $1 * $3; }
          | factor
          ;
```

```
factor : '(' expr ')'          { $$ = $2; }
         | DIGIT
         ;
%%
yylex()  {
                int c;
                c = getchar();
                if(isdigit(c)
                {
                        yylval = c-'0';
                        return DIGIT;
                }
return c;
}
```

Ambiguous Grammars in YACC

YACC declarations resolve shift/reduce and reduce/reduce conflicts using operator precedence and operator associativity information. YACC has default methods for resolving conflicts. However, it is better to find out what conflicts arose and how they were resolved using the '-v' option. The declarations provide a way to override YACCs defaults. Productions have the precedence of their rightmost terminal, unless otherwise specified by "%prec" element.

The declaration keywords %left, %right and %nonassoc inform YACC that the following tokens are to be treated as left-associative (as binary + - * & / commonly are), right-associative (as exp often is), or non-associative (as binay < & > often are). -The order of declarations informs YACC that the tokens should be accorded increasing precedence

QUESTIONS

1. **Define Parsing.**
2. **What are different types of parsing?**
3. **Explain the difference between top-down parsing and bottom up parsing.**
4. **Discuss different types of bottom up parsing.**
5. **Define Yacc. Explain it using an example.**