**LL(1) PARSER**

**COMPILER MINI PROJECT**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TITLE** | **PAGE NO** |
| 1. | **ABSTRACT** | 3 |
| 2. | **INTRODUCTION** | 4 |
| 3. | **OBJECTIVE** | 5 |
| 4. | **MODULES AND IMPLEMENTATION** | 6 |
| 5. | **CONCLUSION** | 9 |

# **ABSTRACT**

The objective of this project is to implement an efficient and robust LL(1) parser using a high level programming language, that is capable of correctly parsing any input fed to it. LL(1) Parser is a Predictive parser.

The LL(1) Parser is a Top-Down Parser that does not require any back-tracking. It is also called a Non-Recursive Parser.

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 indicates that the grammar uses a look ahead of one source symbol- that is the prediction to be made is determined by the next source symbol.

The project includes a set of predefined grammar which is evaluated by LL(1) parser program and the parsing table is created and displayed. The implementation is pretty straightforward and simple. Then it would take any input string belonging to the grammar language and would find the FIRST and FOLLOW sets of the grammar.

# **INTRODUCTION**

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.

The goal of parsing is to check the validity of a source string and to determine its syntactic structure. For an invalid string the parser issues diagnostic messages reporting the cause and nature of error in the string. For a valid string it builds a parse tree to reflect the sequence of derivations or reductions performed during parsing.

In top-down parsing, the parse tree is generated from top to bottom, i.e., from root to leaves & expand till all leaves are generated.

It generates the parse tree containing root as the starting symbol of the Grammar. It starts derivation from the start symbol of Grammar & performs leftmost derivation at each step.

The LL(1) Parser is a Top Down Parser that does not require any back-tracking. It is also called a Predictive or Non-Recursive Parser.

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.

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.

# **OBJECTIVE**

The objective of this project is to implement an efficient and robust LL(1) parser using a high level programming language, that is capable of correctly parsing any input fed to it. LL(1) Parser is a Predictive parser.

The project includes a set of predefined grammar which is evaluated by LL(1) parser program and the parsing table is created and displayed. The implementation is pretty straightforward and simple. Then it would take any input string belonging to the grammar language and would find the FIRST and FOLLOW sets of the grammar.

# **MODULES AND IMPLEMENTATION**

Verifying whether a given grammar is LL(1) includes the following steps:

* 1. Computation of First and Follow Sets
  2. Constructing the Predictive Parsing Table
  3. Checking if the Parsing Table has multiple entries in any given cell.

**Following are the steps to perform Predictive Parsing Table**

* Elimination of Left Recursion
* Left Factoring
* Computation of FIRST & FOLLOW
* Construction of Predictive Parsing Table
* Parse the Input String

**Construction of LL(1) Parsing Table**

**Algorithm**

**Input** − Context-Free Grammar G

**Output** – LL(1) Parsing Table

**Method**

*Step 1*: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

*Step 2*: Calculate First() and Follow() for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

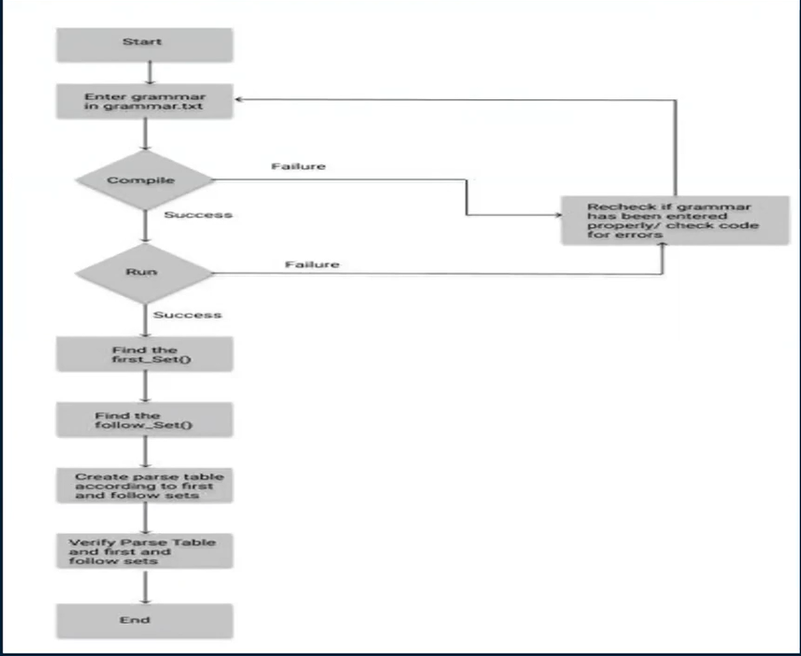
*Step 3*: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

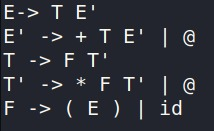
If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

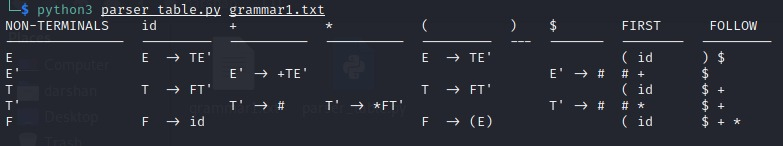
***Working of LL(1) Parser***



# **INPUT**



# **OUTPUT**



# **CONCLUSION**

The grammar for the parser is assumed i.e. fixed in the initial phase itself and the user can use any of the predefined sample grammars. The parser would generate an LL(1) set of items and display that. Then it would generate a parsing table given when the generated item set is fed on to it.

Then the parsing of any user input is shown step by step. But the only static thing in the project is the set of pre-defined grammar. The add-on to the project would be to generalize any grammar that is accepted by the user and then generate LL(1) set of tokens on it and then parse the input string.