## **COMPILER DESIGN PROJECT**

## **LL(1) Parser**

Brief Project Report

## **Introduction**

Parser

In order for the code written in human-readable form to be understood by a machine, it must be converted into machine language. This task is usually performed by a translator (interpreter or compiler). The parser is commonly used as a component of the translator that organizes linear text in a structure that can be easily manipulated (parse tree). To do so, it follows a set of defined rules called “grammar”.

Parsers are widely used in the following technologies:

* Java and other programming languages.
* HTML and XML.
* Interactive data language and object definition language.
* Database languages, such as SQL.
* Modeling languages, such as virtual reality modeling language.
* Scripting languages.
* Protocols, such as HTTP and Internet remote function calls.

LL(1) Parser  
In this article, we are going to explore the LL(1) parsing algorithm used to create simple generated parsers.

* The first L indicates that the input is read from left to right.
* The second L says that it produces a left-to-right derivation.
* And the 1 says that it uses one lookahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.)

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

To construct the Parsing table, we have two functions:

**1: 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.  
**2: Follow():** What is the *Terminal Symbol* which follows a variable in the process of derivation.

## Now, after computing the First and Follow set for each *Non-Terminal symbol* we have to construct the Parsing table. In the table Rows will contain the Non-Terminals and the column will contain the Terminal Symbols.

## All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

**Objective**

To implement a LL1 parser which can parse a given stream of tokens according to a given LL(1) grammar.

Languages and tools used: C, C++.

**Input**

The LL(1) parser will read the following two things as input from two different input files.

1. Given LL(1) grammar according to which the parser will work. It must be in the following format.
   1. Each production should be written separately.

For eg. *A -> B | Cb* is invalid.

It should be written as :

*A -> B*

*A -> Cb*

* 1. Each non-terminal should be written as capital alphabets.
  2. Each terminal should be written as small alphabets.
  3. The given grammar must be non-ambiguous and should adhere to the rules of LL(1) grammar.
  4. It should be present in a separate file named **grammar.txt** in the same directory where parser is present.

1. Given the stream of tokens, which the parser will parse according to above grammar. It must be in the following format.
2. Each token should be present separately with a single white-space between them.

For eg. let “*a,b,c,d*” are 4 tokens, then the file can be:

“*a b c d*” or “*c a d b a a d*” etc.

1. It must be present in a separate file named **string.txt** in the same directory where parser is present.

**Output**

Output will be generated every time the program is executed.

Output will be shown in following format:

Case 1: If any of the input files is absent it will simply print an error message and terminate.

Case 2: If the tokens present inside the "string.txt" file are not identified by the grammar then parsing will fail and it will display an error message with sufficient info.

Case 3: If the string is successfully parsed by the parser according to the grammar, then it will print success with sufficient information about the parsing process.

NOTE: Parsing table will be displayed for each execution, however the construction process of the parsing table involving FIRST() and FOLLOW() sets shall remain abstracted.

**Source Code**

[**Parser.cpp**](https://github.com/abhiagg20/LL1-Parser-Implementation/blob/master/parser.cpp)

## **Sample Input/Output** [**Sample Inputs**](https://github.com/abhiagg20/LL1-Parser-Implementation/tree/master/Sample%20Inputs)This folder contains various input files. In order to use the files, rename the required grammar file to “grammar.txt” and the required string file to “string.txt” and replace the already existing files in the root folder with them. [**Output.txt**](https://github.com/abhiagg20/LL1-Parser-Implementation/blob/master/output.txt) This is the format in which the output will be produced by the parser.

## **Runthrough**

Consider the following grammar:

*E -> E + T | T*

*T -> T \* F | F*

*F -> ( E ) | int*

or its longhand equivalent:

*E -> E + T*

*E -> T*

*T -> T \* F*

*T -> F*

*F -> ( E )*

*F -> int*

This represents a grammar for a simple arithmetic language that supports \*, + and ( ) operations. The non-terminals are E, T, and F. The terminals are +, \*, (, ) and int.

The grammar is also left-recursive, which as mentioned earlier, will not do. We're going to manually refactor it and eliminate the left recursion. We get the following grammar after eliminating left recursion from E and T.

*E -> T E'*

*E' -> + T E' |*

*T -> F T'*

*T' -> \* F T' |*

*F -> ( E ) | int*

or longhand:

*E -> T E'*

*E' -> + T E'*

*E' ->*

*T -> F T'*

*T' -> \* F T'*

*T' ->*

*F -> ( E )*

*F -> int*

Note that we didn't do anything to F because it wasn't left-recursive.

Following are the FIRSTs and FOLLOWs for all the Non-Terminals present in the grammar.

**FIRST**(E) = { (, int }

**FIRST**(E') = { +, **nil** }

**FIRST**(T) = { (, int }

**FIRST**(T') = { \*, **nil** }

**FIRST**(F) = { (, int }

**FOLLOW**(E) = { #EOS, ) }

**FOLLOW**(E') = { #EOS, ) }

**FOLLOW**(T) = { +, #EOS, ) }

**FOLLOW**(T') = { +, #EOS, ) }

**FOLLOW**(F) = { \*, +, #EOS,)**}**

We have one column for each terminal, and one row for each non-terminal. In each cell we have a rule, or empty.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **+** | **\*** | **(** | **)** | **int** | **#EOS** |
| **E** |  |  | E-> T E' |  | E-> T E' |  |
| **E'** | E' -> + T E' |  |  | E' -> |  | E' -> |
| **T** |  |  | T-> F T' |  | T-> F T' |  |
| **T'** | T'-> | T'-> \* F T' |  | T'-> |  | T'-> |
| **F** |  |  | F-> ( E ) |  | F-> int |  |

**Github Repo:** [LL1-Parser-Implementation](https://github.com/abhiagg20/LL1-Parser-Implementation)

Submitted By-

Vaibhav Agarwal

2018UCP1581

Abhinav Agarwal

2018UCP1700