# Final Project Formal Languages

Samuel Martinez Arteaga - Laura Andrea Castrillón Fajardo

May 2025

# Contents

# 1 Introduction

This paper presents the fundamentals for building and implementing LL(1) and SLR(1) type parsers, two approaches widely used in the parsing of formal grammars. Throughout the article, the key steps for their development are addressed, including the computation of FIRST and FOLLOW sets, the construction of predictive parsing tables, and the validation of input strings.

In addition, the abstraction of the algorithmic process of each component is incorporated, clearly describing the logical flow guiding our solution. This allows not only to understand the theoretical operation of both parsers, but also their step-by-step practical implementation.

# 2 Understand the objective of the project

The code is organized in several modules that allow:

- Flow to calculate the *FIRST* sets: The process for computing the FIRST set begins by obtaining a production rule. For each rule, iterate through its symbols from the beginning, looking for the first terminal symbol. If the first symbol is terminal, it is added directly to FIRST. If it is a non-terminal symbol, its FIRST set is recursively computed. Several cases must be considered: if the non-terminal symbol can derive the empty string (), then it is continued to the next symbol in the production; if it cannot derive , the process terminates for that production; and if all symbols can derive , then  is included in FIRST. This process is repeated for all rules until no more changes are found in the FIRST sets.

- Flow to calculate the *FOLLOW* sets: To calculate FOLLOW, we start from the rule containing the non-terminal symbol whose FOLLOW set we wish to determine. Each production is analyzed to find positions where this symbol appears. If there is another symbol after the symbol, the FIRST of that symbol is added to the current FOLLOW, excluding . If there is no following symbol or if the following FIRST contains , then the FOLLOW of the symbol generating the production is added. The initial symbol of the grammar is also considered to always include the EOF (end-of-file) symbol in its FOLLOW set. This process is repeated for all productions until the FOLLOW sets no longer change.

- Flow of the construction of the predictive table: The construction of the predictive table is performed using the FIRST and FOLLOW sets previously calculated. For each production rule, FIRST is calculated on the right hand side. For each terminal in FIRST, the corresponding production is inserted into the table for the pair (non-terminal, terminal). If FIRST contains , then for each terminal symbol in FOLLOW of the non-terminal, the production deriving  is inserted into the table. The case of conflicts in the table, which would indicate that the grammar is not LL(1),

must also be handled. The result is a table that allows productions to be selected based on the current input symbol and the non-terminal on the stack.

- Perform parsing of input strings: This point uses all the previous concepts, since we have to start creating the LL(1) and SLR(1) automata, and each automaton has its own process for parsing a string, so we will see the abstraction made in each of these.

# 3 LL(1) Top-Down Parser

The LL(1) parser is a **top-down** parsing method that reads input from left to right, constructing the parse tree from top to bottom using one lookahead symbol to make decisions.

## 3.1 Transition table

The LL(1) transitions table is a matrix that guides top-down parsing. It consists of pairs (non-terminal, input symbol) indicating which production should be applied. Each row corresponds to a non-terminal and each column to a terminal (including the end-of-chain symbol). The cell intersecting a non-terminal and a terminal contains the output to be used if that is the current input symbol. This table allows to deterministically predict which rule to apply, based on the symbol at the top of the stack and the next input symbol.

### 3.1.1 Exceptions/special cases

During the construction of the LL(1) table, special cases also arise that must be handled carefully to ensure that the table is valid and that the grammar can be analyzed predictively. The most important cases are highlighted below:

- **Productions deriving in $\varepsilon$:** When a production rule can derive the empty string ($\varepsilon$), this situation is reflected in the corresponding FIRST set containing the symbol `e`. In this case, the LL(1) table cannot be populated using the FIRST symbols as an index, since $\varepsilon$ is not an input symbol. Instead, the symbols from the FOLLOW set of the nonterminal are used as keys to insert the output that derives $\varepsilon$ into the table
- **Conflicts in the table: ambiguity LL(1):** It is important to consider that if during the construction of the table an attempt is made to insert more than one production in the same cell (i.e. for the same non-terminal and terminal pair), a conflict is generated. This conflict indicates that the grammar is not LL(1) and therefore cannot be analyzed deterministically using this approach.

### 3.2 Process of validating a string - Automata

The parsing process for an input string is performed using a stack and the previously constructed LL(1) table. The parsing starts with an initial stack containing the end-of-input symbol ($) and the initial symbol of the grammar. At each step, the symbol at the top of the stack is compared with the current symbol of the input string. If both match and are terminal, they are removed from the stack and the string, respectively.

If the symbol at the top of the stack is a non-terminal, the output corresponding to the pair (non-terminal, input symbol) is queried in the LL(1) table. That production is inverted (since the stack is LIFO) and added to the stack to continue the analysis. If the ($\varepsilon$) symbol is encountered as a result of a production, it is simply removed from the stack without consuming any input symbols.

This process continues recursively until the stack is emptied and the string is completely consumed. If this is achieved without errors, the string is accepted; otherwise, it is rejected. Throughout the process, the stack and input states are recorded so that the history of the step-by-step analysis can be displayed.

## 4 SLR(1) Bottom-Up Parser

The SLR(1) parser is a bottom-up parsing method that reads input from left to right and constructs the parse tree from the leaves (bottom) to the root. It uses one lookahead symbol and considers the **Follow** sets of the grammar for transitions.

### 4.1 Transition table

The SLR(1) transition table is a central structure in bottom-up parsing. It is composed of pairs (state, symbol) that determine the action to be executed during parsing. This action can be a shift, a reduction, a transition between states or an acceptance. The table is generally divided into two sections: *action* section, which is associated with terminal symbols, and *goto* section, that relates to non-terminal symbols. From this table, the parser decides step by step how to process the input string and construct the corresponding derivation.

#### 4.1.1 Actions, reductions, displacements, acceptance status

- **Actions:** Operations that the analyzer performs when reading an input symbol, such as shift, reduce or accept. They are determined from the analysis table.
- **Shift:** Action that consists of moving the current input symbol to the stack and advancing to the next symbol, changing to the indicated state.
- **Reduce:** Action that replaces a sequence of symbols on the stack with a non-terminal, according to a grammar production.

- **Acceptance status:** A situation in which the string has been fully parsed and recognized; it indicates that it belongs to the language of grammar.

### 4.1.2 Exceptions/special cases

During the construction of the SLR(1) table and the processing of the states, certain exceptional or special cases arise that must be handled carefully to ensure that the grammar is valid and that the parser works correctly. The most relevant ones are explained below:

- **Reduction with empty chain ($\varepsilon$):** In some productions, the symbol following the period is a `e` (representing $\varepsilon$). In this case, a shift cannot be performed, since there is no input symbol. Instead, we proceed to insert a reduction directly into the analysis table for all symbols that are in the FOLLOW set of the symbol to the left of the output. This requires having the FOLLOW sets correctly calculated.
- **Conflicts in the table: ambiguities:** It may happen that when trying to insert an action (either reduction or displacement) in the SLR(1) table, there is already an action assigned for the same cell. This is a case of ambiguity indicating that the grammar is not SLR(1). In such situations, the algorithm flags an error and a flag is set to indicate that the analysis cannot continue under the SLR(1) approach. This occurs in two typical cases:
  - When an attempt is made to insert a reduction and an offset already exists.
  - When there are two possible reductions for the same cell.
- **Avoid duplication of states:** During the creation of new states of the LR(0) automaton, a set of rules that were already processed previously can be generated. To avoid creating duplicate states, a check is performed against the already completed flows. If the rules already exist, the corresponding state number is reused instead of creating a new one. This handling prevents redundancies and ensures an accurate representation of the automaton.

## 4.2 Processing a string - Automata

The parsing of a string with an SLR(1) parser is performed using a stack that stores states and an SLR parsing table composed of actions and transitions. Initially, the stack contains state 0. At each iteration, the current input symbol (lookahead) and the state at the top of the stack are observed, and the corresponding action is looked up in the table. If the action is a shift, the symbol and the new state are stacked, and the chain is advanced. If the action is a reduction, the production to be applied is identified, symbols are unstacked according to the length of the right production, and then the transition table (GoTo) is

consulted to determine the new state to be stacked together with the reduced symbol.

This process continues until an accepting action is reached (usually when the current status and symbol $ indicate accept), or until an error occurs when no valid action is found. Throughout the procedure, the stack steps, the processed symbols, and the rest of the input are recorded, allowing the parsing history to be reconstructed. If the entire string is successfully reduced to the initial symbol and the entire input is consumed correctly, the string is accepted; otherwise, it is rejected.

# 5 Diagram

To visualize the derivation process of a string during parsing, a derivation tree was constructed using the **Graphviz** tool. This visualization is useful for both the parser **LLL(1)** and **SLR(1)**, as both generate a derivation history that can be represented as a tree.

The main challenge we faced was learning how to use Graphviz correctly. The initial difficulty was in how to define the unique nodes of each symbol and how to link them properly to represent the hierarchy of derivations of the grammar. In particular, it was necessary to implement logic that avoided node name collisions, since the same symbols can appear multiple times in different parts of the tree.

The algorithm we developed performs the following:

- Create an object `Digraph` To define the graph.
- Uses a list of the derivation history generated during parsing.
- Traverses this history and for each derived symbol, generates a node with a unique label based on its name and a repetition counter.
- Connects each derived node to its parent symbol in the derivation, thus generating the branches of the tree.

Once the graph is constructed, an image file (`.png`) is automatically generated that visually shows the structure of the derivation tree, facilitating the interpretation and understanding of the syntactic process.

This diagram has been essential to visually verify that the derivation rules are being applied correctly, and to debug complex cases during the development of both the LL(1) and SLR(1) parsers.