LL(1) and SLR(1) Parser Implementation

Laura Restrepo Johan Rico

1 Introduction

This report documents the implementation of a parser analyzer for context-free grammars. The project supports LL(1) and SLR(1) parsing techniques. The system is composed of four Python files and one grammar input file. The user can add grammars, test whether they are LL(1), SLR(1), or both, and analyze strings with the corresponding parser.

2 Code Structure

- **main.py**: Provides the command-line interface for the user. Allows grammar input, selection, parser generation, and input testing.
- **FIRST_FOLLOW.py**: Contains functions to compute the FIRST and FOLLOW sets of symbols or strings.
- **LL1_PARSER.py**: Checks if a grammar is LL(1), constructs the parsing table, and performs LL(1) parsing with derivation steps.
- **SLR1_PARSER.py**: Checks if a grammar is SLR(1), constructs the parsing table, and performs SLR(1) parsing with reduction steps.
- **grammars.txt**: A plain text file where grammars are saved and loaded from.

3 Functionality

The system allows the user to:

1. Enter and save grammars.
2. Automatically check if a grammar is LL(1), SLR(1), both, or neither.
3. Generate and display parsing tables.
4. Input strings to test whether they belong to the grammar using the corresponding parser.
5. View derivation or reduction steps depending on the selected parser.

4 Implementation Details

4.1 main.py

Functionality

This is the user interface module. It allows users to:

- Input and save grammars.
- Choose which parser to use.
- Analyze strings using LL(1) or SLR(1).

It provides a basic menu system and handles input/output formatting.

4.2 FIRST_FOLLOW.py

Functionality

This module implements the computation of FIRST and FOLLOW sets for a given grammar, which are fundamental concepts in compiler construction. These sets are essential for creating predictive parsers such as LL(1) and SLR(1). The implementation follows algorithms described in the Dragon Book.

1. FIRST Sets

- **Definition**: The FIRST set of a symbol or string represents the set of terminal symbols that could appear at the beginning of any string derived from it.
- **Implementation**:
  - Uses a fixed-point iteration approach that continues until sets stop changing
  - Implements the three rules defined in the Dragon Book

> To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.
>
> 1. If X is a terminal, then FIRST(X) = $\{X\}$.
> 2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.
> 3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).
>
> Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

Figure: FIRST algorithm from "Compilers: Principles, Techniques, and Tools"

  - Properly tracks and handles ε (epsilon) derivations

2. FIRST for Strings

- Calculate the FIRST set for a string $X_1X_2...X_n$ by:
  - Adding all non-ε terminals from FIRST(X_1)
  - If X_1 can derive ε, adding non-ε terminals from FIRST(X_2)
  - Continuing this process as long as each symbol can derive ε
  - Adding ε to the result only if every symbol in the string can derive ε
- Handles special cases like empty strings and single symbols

3. FOLLOW Sets

- **Definition**: The FOLLOW set of a non-terminal A contains all terminal symbols that can appear immediately to the right of A in some sentential form.
- **Implementation**:
  - Uses a fixed-point iteration approach
  - Follows the three rules from the Dragon Book

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Figure: FOLLOW algorithm from "Compilers: Principles, Techniques, and Tools"

  - Relies on previously computed FIRST sets
  - Efficiently handles edge cases like non-terminals at the end of productions

These FIRST and FOLLOW set computations form the foundation for both LL(1) and SLR(1) parsing algorithms, which use these sets to construct parsing tables and determine grammar viability.

4.3 LL1_PARSER.py

Functionality

This file provides the core logic for working with LL(1) grammars. It includes functions to check whether a given grammar is LL(1), to construct the corresponding parsing table, and to simulate the parsing process through a derivation table. Below is a breakdown of each function's role:

1. Check if Grammar is LL(1)
This function checks whether the grammar can be parsed using LL(1) techniques. For that, it verifies that the grammar follows two conditions:

- **No immediate left recursion:** For every production $A \to \alpha$, the first symbol of α cannot be A.
- **Disjoint FIRST sets:** For any two different productions $A \to \alpha$ and $A \to \beta$, where $\alpha \neq \beta$, the intersection of their FIRST sets must be empty:

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset.$$

If both hold, the grammar is considered LL(1).

2. Create Parsing Table

This function constructs the LL(1) parsing table using the grammar's productions, along with their FIRST and FOLLOW sets.

- A table is created with non-terminals as rows and terminals as columns, excluding ε and including the end-of-input symbol \$. Each cell corresponds to a parsing action for a specific pair $(\text{non-terminal}, \text{terminal})$.
- The function iterates through each production $A \to \alpha$.
- For every terminal $a \in \text{FIRST}(\alpha)$, the production is inserted into the cell (A, a).
- If $\varepsilon \in \text{FIRST}(\alpha)$, the production is also added to all cells (A, b) for each $b \in \text{FOLLOW}(A)$.
- The final table maps non-terminals and input symbols to the corresponding production rules, or remains empty when no rule applies.

This table serves as the core decision mechanism for the LL(1) parser, guiding which production to apply based on the current input and stack top.

The following example uses the following grammar:

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid id$$

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figure: Example of parsing table LL(1)

3. Create Derivation Table
This function simulates the LL(1) parsing process and generates a table with three columns: `Stack`, `Input`, and `Action`.

- The `Stack` starts initialized with the end-of-input symbol $, followed by the grammar's start symbol.
- The `Input` begins with the input string concatenated with the end-of-input symbol $.
- At each step, the parser inspects the top symbol of the `Stack` and the current symbol at the front of the `Input`.
- If the top of the stack is a terminal matching the input symbol, both are popped and consumed, and an action "Match" is recorded.
- If the top of the stack is a non-terminal, the parser consults the parsing table to find a production rule to expand it, replaces the non-terminal with the production's right-hand side symbols, and records a "Derive" action.
- If the parser reaches the end-of-input symbol on both stack and input, the input is accepted.
- If at any point no valid action can be found or symbols mismatch, the input is rejected.

This step-by-step process is recorded in the derivation table, which clearly shows the parser's decisions and how it consumes the input string.

4.4 SLR1_PARSER.py

Functionality

This module builds the canonical collection of items and uses the FOLLOW sets to create the SLR(1) parsing table. It performs shift-reduce parsing and shows a reduction table with step-by-step analysis of input strings.

1. Create Parsing Table
The parsing table consists of two main components: the `action` table and the `goto` table.

- The `action` table handles terminals and specifies shift, reduce, or accept actions.
- The `goto` table handles non-terminals and specifies state transitions after reductions.

1.1 Create Items
Items represent productions with a "dot" indicating the current position of the parser. For instance, an item like $A \to \alpha \cdot \beta$ means that α has been recognized and β is expected next.

- The `Item` class stores a production's left-hand side, right-hand side, and the current dot position.
- The `closure` function expands a set of items by adding all items derivable from the non-terminal following the dot.
- The `goto` function computes the resulting item set after moving the dot over a given symbol.
- The `canonical_collection` function repeatedly applies `goto` and `closure` starting from the augmented start production $S' \to S$, where S is the grammar's start symbol, to discover all parser states.

This table serves as the core decision mechanism for the LL(1) parser, guiding which production to apply based on the current input and stack top.

The following example uses the following grammar:

$$E' \to E$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

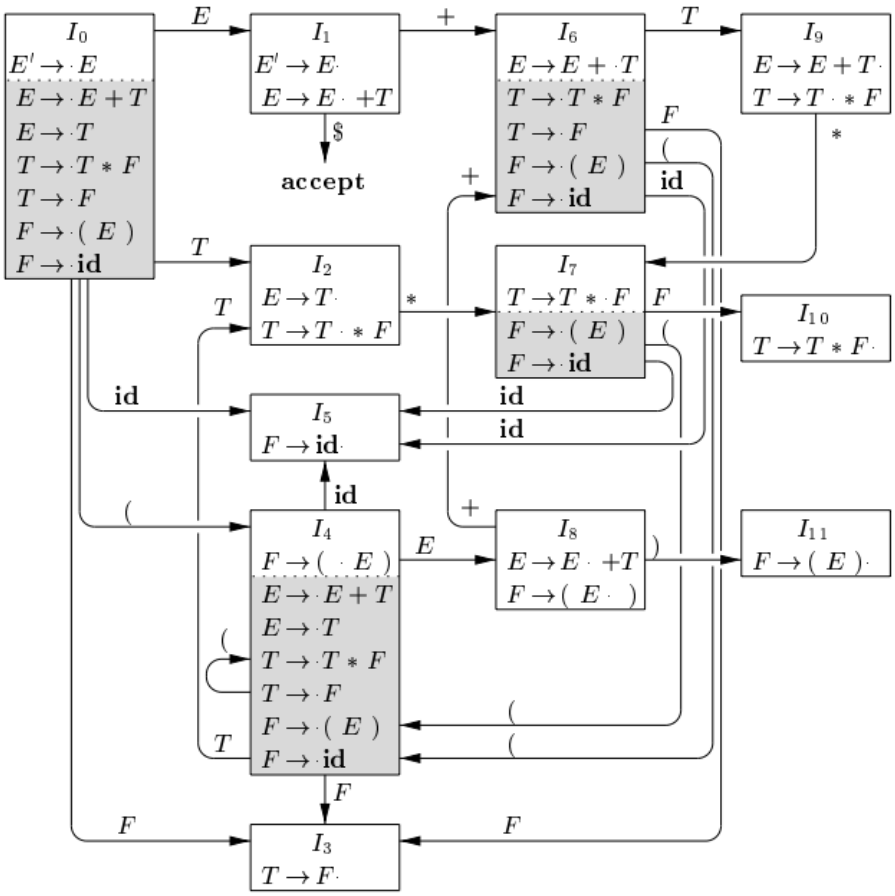

Figure: Automaton for the expression grammar

1.2 Build Parsing Tables

Once all item sets (states) are created, the parsing tables are built as follows:

- For each item in each state:
  - If the dot is before a terminal, a shift action is added to the `action` table for that terminal.
  - If the dot is at the end of a production, reduce actions are added for all symbols in the FOLLOW set of the production's left-hand side.
  - If the item corresponds to the completed augmented start production, an accept action is added for the end-of-input symbol ($).

- If the symbol after the dot is a non-terminal, the corresponding transition is added to the `goto` table.

- Any shift or reduce conflicts are identified and stored for later grammar validation.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure: Parsing table

2. Create Reduction Table
The parser uses the constructed tables to analyze input strings using:

- A `stack` of parser states, initially containing only state 0.

- A `symbols` stack to track grammar symbols, initially containing only the end-of-input symbol $.

- The input string, with the end-of-input symbol $ appended.

The parsing process proceeds as follows:

- The parser looks up the current state and next input symbol in the `action` table.

- If the action is *shift*, it pushes the symbol and next state onto the stacks and consumes the input symbol.

- If the action is *reduce*, it pops symbols and states based on the production's right-hand side, then uses the `goto` table to determine the next state and pushes it along with the non-terminal.

- If the action is *accept*, the input string is successfully parsed.
- If there is no valid action, the string is rejected as invalid.

A detailed table is generated to log each step, showing `States`, `Symbols`, `Input`, and the `Action` taken.

3. SLR(1) Grammar Check
Before parsing, the grammar is checked for conflicts during the parsing table creation. A *conflict* occurs when two distinct actions are assigned to the same cell in the parsing table. If any conflicts exist, the grammar is not SLR(1), and parsing cannot proceed without ambiguity.