Nile University

جامعة النيل

PARSER GENERATOR (CFG-PARSER)

karim ahmed	231001537
ahmed hany	231001623
omar hesham	231000256
alaa sari	221001779

under observation of /
DR.Heba Elnemr

1. Project Overview

The CFG-parser is a robust, interactive **Canonical LR(1) Parser Generator** built using Python and the Streamlit library. The project is designed to enhance the understanding and study of formal parsing algorithms within the scope of Compiler Design and Theory of Computation courses. It provides a complete implementation of the complex LR(1) algorithm, covering every step from computing First/Follow sets to constructing the final parsing table and performing conflict detection.
The tool implements the full **Canonical LR(1) algorithm** (not just SLR or LALR), performing crucial tasks such as **Set Computation** (First and Follow sets) and **Conflict Detection** (automatically identifying and reporting Shift-Reduce and Reduce-Reduce conflicts). The tool is accessible via an interactive web interface for easy grammar input and testing. Its key feature is the ability to generate high-quality **Parse Tree Diagrams** using Graphviz and display the exact, step-by-step **derivation sequence** used to parse an input string, offering clear technical verification and visualization of the process

2. Project description:

The CFG-parser is a robust, interactive **Canonical LR(1) Parser Generator** built using Python and the Streamlit library. The project is designed to enhance the understanding and study of formal parsing algorithms within the scope of Compiler Design and Theory of Computation courses. It provides a complete implementation of the complex LR(1) algorithm, covering every step from computing First/Follow sets to constructing the final parsing table and performing conflict detection.
The tool implements the full **Canonical LR(1) algorithm** (not just SLR or LALR), performing crucial tasks such as **Set Computation** (First and Follow sets) and **Conflict Detection** (automatically identifying and reporting Shift-Reduce and Reduce-Reduce conflicts). The tool is accessible via an interactive web interface for easy grammar input and testing. Its key feature is the ability to generate high-quality **Parse Tree Diagrams** using Graphviz and display the exact, step-by-step **derivation sequence** used to parse an input string, offering clear technical verification and visualization of the process.

2.1 input format

The CFG-parser uses an interactive web interface built with Streamlit to receive two main types of input from the user: the **Context-Free Grammar (CFG)** and the **Test String**.

1. Context-Free Grammar (CFG) Input

The grammar rules must be entered by the user following precise formatting rules:

- **Production Format:** Each rule (production) must be entered on a separate line. The arrow symbol, **->**, is used to separate the left-hand side (Non-Terminal) from the right-hand side (sequence of symbols).
  - *Example:* E -> E + T
- **Alternative Productions:** When a single Non-Terminal has multiple production alternatives, these alternatives must be listed on the same line and separated by the **pipe symbol (|)**.

    - *Example:* T -> T * F | F
- **Symbol Distinction:** The parser relies on convention to distinguish between symbol types:
    - **Non-Terminals:** These are typically represented by uppercase letters or names (e.g., S, E, T, F).
    - **Terminals:** These are the actual tokens and are generally represented by lowercase letters or special symbols (e.g., id, +, (,)).
- **Augmentation:** The user provides the raw grammar; the system then **augments** the grammar automatically by adding a new starting production
- **2. Test String Input**

After the grammar is loaded and the parsing tables have been successfully constructed, the user provides a string of tokens for the parser engine to validate:

- **Content:** The input string must consist exclusively of the **Terminal symbols** defined in the grammar.
- **Structure:** The tokens in the test string must be provided on a single line.
- **Separation:** It is critical that tokens are separated, typically by spaces, to ensure the parser can read them as individual units.
    - *Example:* For an arithmetic grammar, the valid test string would be: id + id * id.

This structured input allows the backend engine to accurately perform the $\{Closure}$ and $\{Goto}$ operations required for LR(1) table construction and subsequent parsing.

2.2 output format

- The generated **ACTION** and **GOTO** parsing tables.
- A detailed report on the detected conflicts (if any).
- A clear **Visual Parse Tree** diagram.
- The step-by-step derivation sequence and final result (Accept/Reject).

2.3 Mechanism Format

The CFG-parser relies on a modular architectural structure that systematically processes the input grammar and transforms it into usable parsing tables via five sequential stages, based on the Canonical LR(1) algorithm.

1. Phase One: Grammar Processing and Augmentation

The initial step is to prepare the user-provided grammar for the parsing process:

- **Structure Validation:** The system ensures that the input productions adhere to the required format (Non-Terminal -> Sequence of Symbols).
- **Augmentation:** The parser automatically adds a new starting production (Augmented Grammar) to guarantee a unique starting symbol where acceptance can terminate. If the original starting symbol is S, the system adds the rule $S' \right arrow S$. This rule signals successful parsing when it is reduced.

2. Phase Two: First and Follow Set Computation
First and Follow sets are critical for determining the necessary lookahead symbols used in the LR(1) items:

- **First Set:** Computed to determine all terminal symbols that can begin any string derived from a particular symbol (terminal or non-terminal).
- **Follow Set:** Computed to determine which terminal symbols can immediately follow a specific non-terminal in any valid sentence derived from the starting production.
- **Algorithm:** The system uses a **fixed-point iteration algorithm** to ensure all possibilities for these sets are comprehensively captured.

3. Phase Three: Canonical Collection of LR(1) Items
This is the most complex phase, involving the construction of the parser's state machine:

- **LR(1) Items:** Production rules are augmented with the position of the dot (.) and a lookahead terminal symbol.
- **Closure Operation:** Applied recursively to a set of LR(1) items to find all grammar rules reachable from the dot's current position.
- **Goto Operation:** Determines the next state the parser transitions to upon reading a specific terminal or non-terminal symbol from the current state.
- **State Machine Construction:** All possible states and the transitions between them are built, forming the complete LR(1) automaton.

4. Phase Four: Parsing Table Construction
Using the generated Canonical Collection of Items, two crucial tables are built:

- **ACTION Table:** Specifies the action to be taken (Shift, Reduce, Accept, or Error) when a specific terminal symbol is encountered as the lookahead in a given state.
- **GOTO Table:** Specifies the next state the parser moves to after performing a Shift or a Reduction and transitioning on a non-terminal.
- **Conflict Detection:** During table filling, the system automatically detects:
  - **Shift-Reduce Conflicts:** Occurs when both a Shift and a Reduce action are possible in the same state for the same lookahead symbol.
  - **Reduce-Reduce Conflicts:** Occurs when more than one Reduce action is possible in the same state for the same lookahead symbol.

5. Phase Five: The Parser Engine
This phase utilizes the constructed tables to analyze the actual input string:

- **Mechanism:** The parser employs a **stack-based Shift-Reduce technique**.
- **Operations:** The engine reads the next input token, uses the current state and the stack to look up the appropriate action from the **ACTION Table**.
- **Output Trace:** All operations (Shifts and Reductions) are tracked and displayed as a step-by-step derivation sequence. Upon reaching the reduction of the augmented rule ($S' \rightarrow S$), the input is declared as accepted.
- **Visualization:** If parsing is successful, the visual parse tree is generated using Graphviz, demonstrating the final syntactic structure.

---

3 *Programming Language, Tools & Libraries Used:*

Programming Language: Python

Core Tools and Libraries:

The project's functionality is built upon a small set of powerful external tools and libraries, enabling both the user interface and the graphical output:

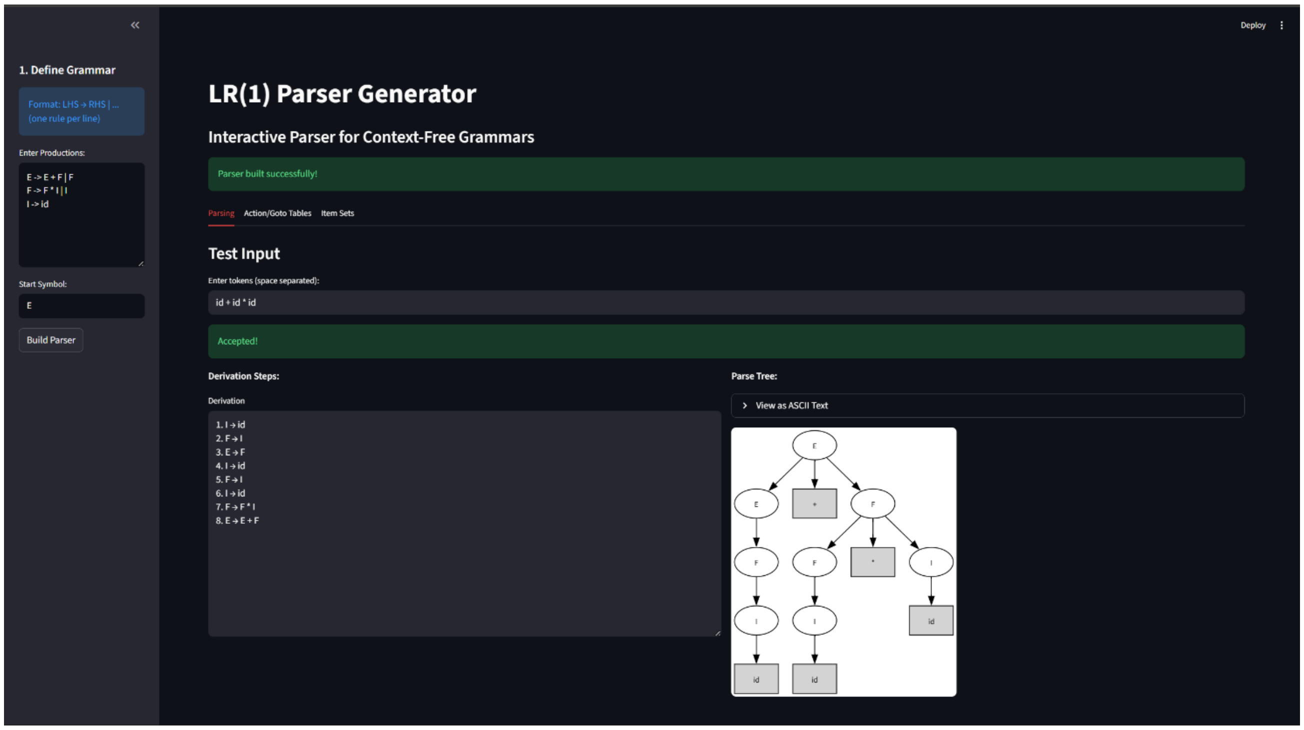
Streamlit:
This is the primary tool used for developing the **interactive web interface (Frontend)**. Streamlit allows the user to easily input the Context-Free Grammar (CFG) rules and test strings, and execute the parser directly through a web browser.

Graphviz:
This is a necessary **system library** utilized for the **Visualization** phase. Its critical role is to generate high-quality, structured graph diagrams, specifically the **Visual Parse Tree**, which maps the successful syntactic derivation of the input string.

Modular Architecture:

While not a library, the project structure itself is designed to be modular, ensuring a clean separation between the core **Backend logic** (the five phases of the LR(1) algorithm) and the **Frontend visualization** components.

4. *Images of the Project with Output:*



Format: LHS → RHS | ...
(one rule per line)

Enter Productions:

```
E -> E + F | F
F -> F * I | I
I -> id
```

Start Symbol:

E

LR(1) Parser Generator

Interactive Parser for Context-Free Grammars

Parser built successfully!

Test Input

Enter tokens (space separated):

id + id * id

Accepted!

Derivation Steps:

Derivation

```
1. I → id
2. F → I
3. E → F
4. I → id
5. F → I
6. I → id
7. F → F * I
8. E → E + F
```

Parse Tree:

5. Conclusion

The **Canonical LR(1) Parser Generator (CFG-parser)** successfully provides a robust and comprehensive tool for studying parsing in Compiler Design.

Key Achievements:

- **Full LR(1) Implementation:** The project implements the entire complex, five-phase **Canonical LR(1) algorithm**.
- **Technical Validation:** It automatically detects and reports **Shift-Reduce** and **Reduce-Reduce** conflicts during table construction.
- **Enhanced Learning:** By integrating a **Streamlit web interface** and using **Graphviz** for visualization, the tool transforms an abstract concept into an accessible, verifiable environment.
- **Detailed Output:** It offers essential tracing, including the **step-by-step derivation sequence** and the **visual parse tree**.

In summary, the CFG-parser is a strong demonstration of compiler theory, offering users a transparent and interactive environment to explore CFG parsing and LR(1) state machine construction.