CC - Assignment#2

Authors (team @Lang)

- Tauha Imran 22i1239 - cs-g
- Husain Ali Zaidi 22i-0902 - cs-g

1. Overview

This project implements a CFG Processor capable of:

- Parsing a CFG from a file.
- Handling **Left Factoring** and **Left Recursion Elimination**.
- Computing **FIRST** and **FOLLOW** sets for non-terminals.
- Constructing an **LL(1) Parsing Table**.

2. Codebase Overview

The project consists of:

- `CFG.h`: Manages non-terminals, terminals, production rules, and operations like **Left Factoring**, **Left Recursion Elimination**, **FIRST**, and **FOLLOW** sets.
- `CFGReader.h`: Reads CFGs from files and stores them in a `CFG` object.
- `ProductionRule.h`: Handles production rules with a left-hand side (`State`) and a right-hand side (vector of `State`s).
- `State.h`: Represents non-terminals, terminals, epsilon, and the start state.
- `StateSet.h`: Manages **FIRST** and **FOLLOW** sets for non-terminals.
- `ParsingTable.h`: Constructs and displays the **LL(1) Parsing Table**.
- `cfg_processor.cpp`: The main driver file.

3. Implementation Details

CFG Class

```
void leftFactoring();
void eliminateLeftRecursion();
void computeALLFirstSets();
void computeAllFollowSets();
void computeLL1ParsingTable() const;
```

- **Left Factoring** identifies common prefixes and generates new non-terminals.

- **Left Recursion Elimination** replaces recursive rules by introducing auxiliary non-terminals.
- **FIRST & FOLLOW Sets:** Computed via recursion and state set handling.
- **Parsing Table Generation:** Matches terminals and non-terminals based on the computed sets.

StateSet Class

```
void addToFirstSet(const State& state);
void addToFollowSet(const State& state);
void showFirstSet() const;
void showFollowSet() const;
```

- Manages the computation and display of **FIRST** and **FOLLOW** sets.

ParsingTable Class

```
void addEntry(const std::string& nonTerminal, const std::string& terminal, const std::string& production);
void display() const;
```

- Constructs a 2D table associating terminals and non-terminals.

4. Example CFG

Input (input.txt):

```
S -> a A | a B | S b | c
A -> A a | b
```

5. Processed Output

Production Rules After Processing:

```
S -> c S'
S -> a S' S'
S' -> b S'
S' -> epsilon
```

FIRST Sets:

```
FIRST(S) = { epsilon, a }
FIRST(A) = { }
```

FIRST(B) = { }
FIRST(S') = { epsilon }

FOLLOW Sets:

FOLLOW(S) = { $ }
FOLLOW(A) = { }
FOLLOW(B) = { }
FOLLOW(S') = { epsilon, b }

LL(1) Parsing Table:

	a	b	c	$
S	S -> a S' S'		S -> c S'	
A	-	-	-	-
B	-	-	-	-
S'	-	S' -> b S' -		-

6. Instructions for Running on Linux

1. **Ensure you have g++ installed:**

sudo apt-get install g++

2. **Navigate to the directory containing the files:**

cd /path/to/your/files

3. **Compile all `.cpp` files with the necessary headers:**

g++ -o cfg_processor cfg_processor.cpp -std=c++17

4. **Run the program:**

./cfg_processor

5. **View the output:** The results will be saved in a file named `output.txt`.

7. Approach

The approach taken involves breaking down the grammar processing into distinct steps:

- **Reading and Storing CFG:** The program reads a CFG from a file using `CFGReader.h` and stores non-terminals, terminals, and production rules in `CFG` objects.
- **Transformations:** Two transformations are applied -
    - **Left Factoring:** By identifying common prefixes and creating auxiliary non-terminals.
    - **Left Recursion Elimination:** By breaking recursive rules and introducing helper non-terminals.
- **Computing Sets:**
    - `computeALLFirstSets()` recursively generates **FIRST sets** for all non-terminals.
    - `computeAllFollowSets()` computes **FOLLOW sets** by propagating terminal and non-terminal dependencies.
- **Building the Parsing Table:** Using computed sets, a **Parsing Table** is generated by mapping non-terminals to terminals.

8. Challenges Faced

- Managing recursive dependencies while calculating **FIRST and FOLLOW sets**.
- Ensuring all production rules are properly transformed during left factoring and recursion elimination.
- Handling corner cases like epsilon productions and indirect recursion.

9. Verifying Correctness

- Manually cross-checking **FIRST and FOLLOW sets** against theoretical calculations.
- Comparing generated parsing tables with expected results for various CFGs.
- Ensuring successful parsing of valid strings using the generated table.

10. Conclusion

The CFG Processor accurately processes a given grammar by performing necessary transformations, computing **FIRST & FOLLOW sets**, and generating a valid **LL(1) Parsing Table**. The solution is efficient, modular, and suitable for parsing tasks in compiler design. The CFG Processor accurately processes a given grammar by performing necessary transformations, computing **FIRST & FOLLOW sets**, and generating a valid **LL(1) Parsing Table**. The solution is efficient, modular, and suitable for parsing tasks in compiler design.