CC Assignment-2: CFG Processing Program

Mohib Ullah Iftikhar 22i-1044, Zain Asad 22i-1240

March 21, 2025

1 Introduction

This report outlines the development of a C program designed to process a context-free grammar (CFG) by applying left factoring, eliminating left recursion, computing FIRST and FOLLOW sets, and generating an LL(1) parsing table. The program integrates multiple components, each implemented in separate header files (`leftFactoring.h`, `leftRecursion.h`, `first_follow.h`, `ll1_table.h`), with a main driver in `main.c`. The goal was to transform an input CFG into a form suitable for LL(1) parsing and verify its correctness.

2 Approach

The program follows a modular, sequential approach to CFG processing:

2.1 Left Factoring (`leftFactoring.h`)

- **Objective**: Remove common prefixes from production rules to eliminate ambiguity.
- **Method**: The `readCFG` function reads the grammar from `input.txt`. The `leftFactoring` function iteratively identifies common prefixes in productions using `needsFactoring`, factors them out into new non-terminals with `factorProduction`, and updates the grammar. The result is written to `grammar.txt`.

2.2 Left Recursion Elimination (`leftRecursion.h`)

- **Objective**: Remove both immediate and non-immediate left recursion to ensure compatibility with top-down parsing.
- **Method**: The `inputData` function reads the grammar from `grammar.txt`. The `applyAlgorithm` function iteratively applies `solveImmediateLR` for immediate recursion (e.g., $A \to A\alpha|\beta$) and `solveNonImmediateLR` for non-immediate recursion (e.g., $A \to B\gamma$, $B \to A\delta$), introducing new non-terminals as needed. The transformed grammar is output to `grammar_output.txt`.

2.3 FIRST and FOLLOW Sets (`first_follow.h`)

- **Objective**: Compute the FIRST and FOLLOW sets for LL(1) table construction.
- **Method**: The `readGrammar` function loads the grammar from `grammar_output.txt`. `computeFirst` iteratively builds the FIRST sets by analyzing production rules, handling terminals, non-terminals, and ϵ. `computeFollow` computes FOLLOW sets by propagating terminals and the end marker \$. Results are saved to `first_follow_output.txt`.

2.4 LL(1) Parsing Table (`ll1_table.h`)

- **Objective**: Construct an LL(1) parsing table for predictive parsing.
- **Method**: The `buildLL1Table` function uses the FIRST and FOLLOW sets to populate the table. For each production $A \to \alpha$, terminals in $FIRST(\alpha)$ (excluding ϵ) and $FOLLOW(A)$ (if $\epsilon \in FIRST(\alpha)$) determine table entries. The table is written to `ll1_table_output.txt`.

3 Challenges Faced

Several challenges arose during development:

1. **Memory Management**: Dynamic allocation in `leftRecursion.h` (e.g., `realloc` for rules) required careful handling to avoid memory leaks. The `freeGrammar` function was critical for cleanup.
2. **String Parsing**: Parsing multi-character symbols (e.g., S') in `first_follow.h` and `ll1_table.h` was complex due to inconsistent tokenization across modules. Assumptions about terminals vs. non-terminals varied (e.g., uppercase letters in `isTerminal`).
3. **Algorithm Convergence**: Ensuring termination in `applyAlgorithm` and `leftFactoring` required tracking changes and avoiding infinite loops.
4. **File I/O**: Coordinating input/output across files (e.g., `grammar.txt` to `grammar_output.txt`) risked data loss if file operations failed.

4 Verification of Correctness

The program's correctness was verified through:

- **Unit Testing**: Each module was tested with sample grammars:
  - Left Factoring: $A \to ab|ac \to A \to aA', A' \to b|c$
  - Left Recursion: $S \to Sa|b \to S \to bS', S' \to aS'|\epsilon$
  - FIRST/FOLLOW: Validated against hand-calculated sets.
  - LL(1) Table: Checked for single entries per cell (LL(1) property).
- **Output Inspection**: Intermediate files (`grammar.txt`, `first_follow_output.txt`, `ll1_table_output.txt`) were manually reviewed for expected transformations.
- **Error Handling**: Added checks (e.g., file opening failures) to ensure robustness.

5 Conclusion

The program successfully processes a CFG through left factoring, left recursion elimination, FIRST/FOLLOW computation, and LL(1) table generation. Despite challenges with memory, parsing, and convergence, the modular design and verification steps ensured a reliable implementation. Future improvements could include better symbol standardization and automated testing.