Compiler Construction Assignment Report

Implementation Approach for CFG Processing

This report describes the approach taken to implement a C++ program that processes a Context-Free Grammar (CFG) by performing left factoring, removing left recursion, computing FIRST and FOLLOW sets, and constructing an LL(1) parsing table.

Overview of the System Architecture

The implementation follows a modular design with separate functions for each major transformation or analysis step. The system is structured as follows:

1. **Grammar Representation**: Using a struct with a map of non-terminals to their productions and identifying the start symbol
2. **Input Processing**: Reading and parsing the grammar from a file
3. **Transformation Steps**: Applying left factoring and removing left recursion
4. **Analysis Steps**: Computing FIRST and FOLLOW sets
5. **Parsing Table Construction**: Building the LL(1) parsing table

Detailed Description of Implementation Steps

1. Grammar Representation

The grammar is represented using a structure:

```
struct Grammar {
    map<char, vector<vector<char>>> productions;
    char startSymbol;
};
```

This representation allows for: - Efficient lookup of productions by non-terminal - Easy modification during transformation steps - Natural representation of the grammar's structure

2. Input Processing

The grammar is read from a file with each line representing a production rule in the format `A-> | |` :

- The first character is identified as the non-terminal
- The string after the arrow (->) is split by the pipe symbol (|) to obtain individual productions
- Each production is stored as a vector of characters
- The first non-terminal encountered is considered the start symbol

3. Left Factoring

Left factoring is performed to eliminate non-determinism in the grammar by identifying common prefixes:

1. For each non-terminal's set of productions, common prefixes are identified
2. When a common prefix is found:
   - A new non-terminal is created
   - The original productions are replaced with a single production leading to the new non-terminal
   - The new non-terminal derives the suffixes of the original productions
3. The process is repeated until no more common prefixes are found

The algorithm prioritizes longer prefixes and handles cases where multiple productions share the same prefix.

4. Left Recursion Removal

Both direct and indirect left recursion are addressed:

1. **Indirect Left Recursion**:
   - Non-terminals are processed in a specific order
   - For each Ai, productions of earlier non-terminals Aj are substituted
   - This converts indirect left recursion to direct left recursion
2. **Direct Left Recursion**:
   - Productions are split into those with left recursion (Ai -> Ai) and those without (Ai ->)
   - A new non-terminal Ai' is introduced
   - The grammar is rewritten as:
     - Ai -> Ai'
     - Ai' -> Ai' |

This transformation ensures that the resulting grammar is free from left recursion while preserving the language generated by the original grammar.

5. FIRST Set Computation

The FIRST set for each non-terminal is computed using a recursive approach:

1. For terminals and , FIRST is the symbol itself
2. For non-terminals A, FIRST(A) includes:
   - FIRST of the first symbol in each production, except
   - If the first symbol can derive , then FIRST of the next symbol is also included
   - This process continues until a symbol that cannot derive is encountered
3. If all symbols in a production can derive , then is added to FIRST(A)

The implementation uses a depth-first search approach to compute these sets, carefully handling left-recursive productions to prevent infinite recursion.

6. FOLLOW Set Computation

FOLLOW sets are computed using the following rules:

1. FOLLOW(S) includes $ for the start symbol S
2. For a production A -> B , FOLLOW(B) includes:
   - FIRST( ) - { }
   - If can derive or is empty, then FOLLOW(A) is included in FOLLOW(B)

The implementation iteratively applies these rules until no more changes occur, ensuring all necessary terminals are included in the FOLLOW sets.

7. LL(1) Parsing Table Construction

The LL(1) parsing table is constructed by determining which production to use for each non-terminal and lookahead terminal:

1. For each production A -> :
   - If terminal a is in FIRST( ), add A -> to M[A,a]
   - If is in FIRST( ), for each terminal b in FOLLOW(A), add A -> to M[A,b]

The implementation checks for conflicts (multiple entries in the same cell), which would indicate that the grammar is not LL(1).

Challenges Faced and Solutions

1. Left Recursion Removal

Challenge: Handling both direct and indirect left recursion correctly while preserving the language.

Solution: Implemented a systematic approach following the algorithm from compiler theory: - First eliminate indirect left recursion by substituting productions in a specific order - Then eliminate direct left recursion by introducing new non-terminals

2. FIRST Set Computation for Complex Grammars

Challenge: Computing FIRST sets for grammars with complex dependencies and potential left recursion.

Solution: Developed a recursive approach with careful handling of potential infinite recursion: - Using a depth-first search approach - Tracking visited symbols to prevent cycles - Special handling for -productions

3. Accurate FOLLOW Set Computation

Challenge: Correctly computing FOLLOW sets, especially when non-terminals can derive .

Solution: Implemented an iterative approach that repeatedly applies the rules until convergence: - For each non-terminal appearance in a production, compute the terminals that can follow it - Handle special cases like end of production and -derivation carefully - Continue iterations until no more changes occur

4. LL(1) Table Construction

Challenge: Correctly filling the parsing table and identifying conflicts.

Solution: - Calculated the FIRST set of each production right-hand side - Handled -productions by considering FOLLOW sets - Detected and reported conflicts as indications of non-LL(1) grammars

Verification of Correctness

The program's correctness was verified using several approaches:

1. Test Cases

Multiple test grammars were used, including: - Simple grammars without left recursion or left factoring needs - Grammars with direct left recursion - Grammars with indirect left recursion - Grammars requiring left factoring - Ambiguous grammars that should generate conflicts in the LL(1) table

2. Manual Verification

For each test case, the outputs were manually verified: - Transformed grammars were checked to ensure they generate the same language - FIRST and FOLLOW sets were compared with hand-calculated results - The LL(1) parsing table was validated against expected entries

3. Step-by-Step Verification

Each transformation step was independently verified: - Left factoring was checked by confirming common prefixes were correctly extracted - Left recursion removal was validated by confirming no left recursion remained - FIRST sets were checked to contain all required terminals - FOLLOW sets were verified against the grammar's structure - The LL(1) table was checked for correct production placement and conflict detection

Limitations and Potential Improvements

Current Limitations

1. **Single-character Symbols**: The current implementation only supports single-character terminals and non-terminals.
2. **Error Reporting**: Limited detailed error reporting for issues like ambiguous grammars.

3. **Efficiency**: Some algorithms could be optimized for better performance on large grammars.

Potential Improvements

1. **Support for Multi-character Symbols**: Extend the implementation to handle symbols of arbitrary length.
2. **Enhanced Error Diagnostics**: Provide more detailed information about grammar issues.
3. **Algorithm Optimization**: Improve the efficiency of the FOLLOW set computation.
4. **Parser Generation**: Extend the system to generate a working parser from the LL(1) table.
5. **Visualization**: Add graphical representation of the grammar and parsing process.

Conclusion

The implemented system successfully processes context-free grammars through all the required transformation and analysis steps. It correctly applies left factoring, removes left recursion, computes FIRST and FOLLOW sets, and constructs LL(1) parsing tables.

The modular design ensures each step is handled independently, making the code maintainable and extensible. While there are some limitations in the current implementation, the system provides a solid foundation for further development of compiler front-end tools.

The challenges faced during implementation provided valuable insights into the theoretical aspects of parsing and grammar manipulation, reinforcing the importance of a systematic approach to compiler construction.