Context-Free Grammar Processing and LL(1) Parsing

Compiler Construction

Submitted By:

Anoosha Ali – 22i-1242

Ali Usman – 22i-0926

Section: J

Date of Submission: 23rd March, 2025

1. Introduction

This project implements key techniques for Context-Free Grammar (CFG) transformation and analysis, including Left Factoring, Left Recursion Removal, First & Follow Set Computation, and LL(1) Parsing Table Construction. The primary goal is to transform CFGs to be suitable for LL(1) parsing while maintaining correctness.

2. Implementation Details

This section outlines the transformations and computations performed on the CFG.

2.1 Grammar Preprocessing

The implementation first extracts all symbols from the grammar and separates them into terminals and non-terminals. A key preprocessing step involves handling productions with alternatives:

```
Production* correctProductionFormat(Production* prods, int inputCount, int* outputCount)
```

This function splits productions containing alternatives (using the | operator) into separate productions with the same LHS, simplifying subsequent processing.

2.2 Left Factoring

Left factoring is applied to CFG productions to eliminate common prefixes, ensuring that the grammar is suitable for LL(1) parsing. The implemented function:

- Splits production rules into tokens.
- Identifies the longest common prefix.
- Creates a new non-terminal to handle factored parts uniquely.
- Replaces the original production with the factored version.

2.3 Left Recursion Removal

Left recursion occurs in a CFG when a non-terminal refers to itself as the leftmost symbol in its production, making it unsuitable for top-down parsing. The implemented function:

- Identifies left recursion by analyzing each production rule to check if the left-hand side non-terminal appears at the start of any right-hand side.
- Distinguishes between recursive and non-recursive productions, separating α (recursive part) and β (non-recursive part).
- Eliminates direct left recursion by introducing a new non-terminal and rewriting the production in the form:
  - If `A → Aα | β`, it is transformed into: `A → β A'`
  - `A' → α A' | ε`
  -
- Handles cases where no β exists by ensuring the original non-terminal directly references the newly introduced non-terminal.

- Detects and resolves indirect left recursion by processing non-terminals in a hierarchical manner, replacing indirect recursive references before handling direct left recursion.

2.4 FIRST Set Computation

The algorithm computes FIRST sets for all grammar symbols using a fixed-point iteration approach:

`void computeFirstSets(Production prods[], int numProds)`
For each production `X → α`:

- If `α` starts with terminal `a`, add `a` to `FIRST(X)`.
- If `α` starts with non-terminal `Y`, add `FIRST(Y) - {ε}` to `FIRST(X)`.
- If `Y` can derive `ε` and `α = Yβ`, also consider `FIRST(β)`.
- If all symbols in `α` can derive `ε`, add `ε` to `FIRST(X)`.

The algorithm continues until no more changes occur to any FIRST set.

2.5 FOLLOW Set Computation

Similarly, FOLLOW sets are computed using fixed-point iteration:

`void computeFollowSets(Production prods[], int numProds)`
For each production `A → αBβ`:

- Add `FIRST(β) - {ε}` to `FOLLOW(B)`.
- If `β` can derive `ε` or is empty, add `FOLLOW(A)` to `FOLLOW(B)`.
- The special end marker `$` is added to `FOLLOW(S)` where `S` is the start symbol.

2.6 LL(1) Parsing Table Construction

The parse table is constructed using the standard LL(1) table construction algorithm:

`ParseTable buildParseTable(Production prods[], int numProds)`
For each production `A → α`:

- For each terminal `a` in `FIRST(α)`, add the production to `table[A,a]`.
- If `ε` is in `FIRST(α)`, for each terminal `b` in `FOLLOW(A)`, add the production to `table[A,b]`.
- For `A → ε` productions, add the production to `table[A,b]` for each `b` in `FOLLOW(A)`.

3. Data Structures & Code Organization

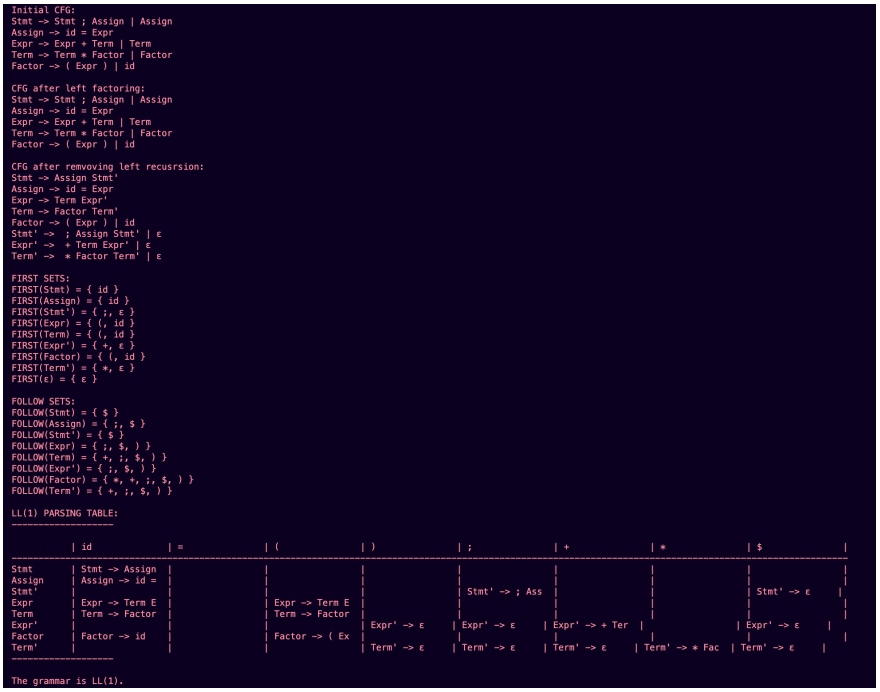
The implementation uses the following key data structures:

- **Production**: Represents a grammar production with left-hand side (LHS) and right-hand side (RHS).
- **SymbolSet**: Stores sets of terminals/non-terminals for FIRST and FOLLOW computations.
- **ParseTable**: Contains entries for each non-terminal and terminal combination.

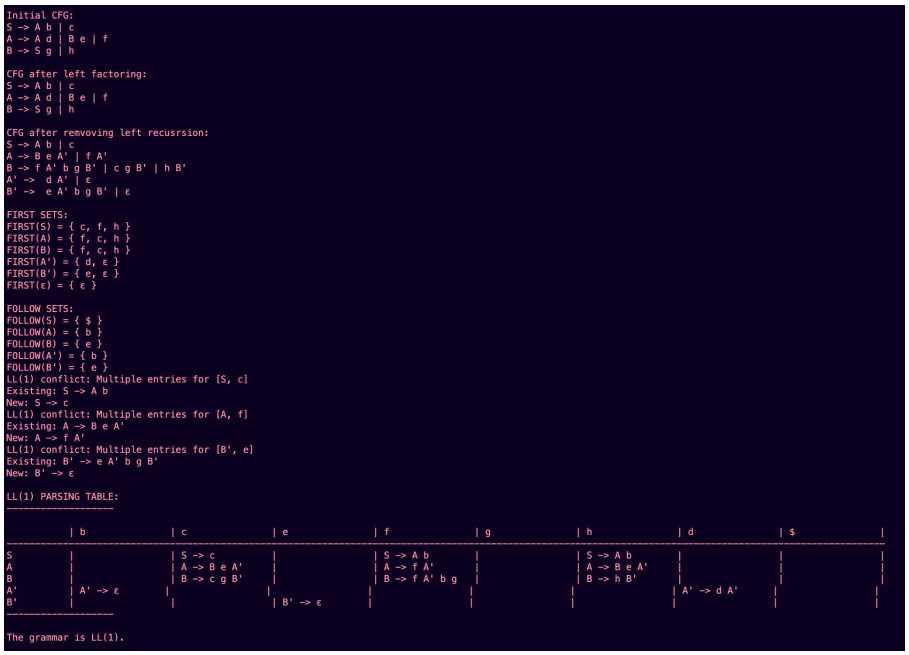
The code is organized into three main components:

- **Grammar Preprocessing (`LL1_parser.h`, `cfg_reader.c`, `transformations.c`)**: Handles reading and transforming the input grammar.
- **Sets Computation (`LL1_parsing.c`)**: Computes FIRST and FOLLOW sets.
- **Parse Table Generation (`parse_table.c`)**: Builds and analyzes the LL(1) parsing table.

4. Sample Input & Output



Example 01



Example 02

5. Challenges & Learnings

5.1 Challenges:

- Handling multiple left-factored cases correctly without conflicts.
- Ensuring dynamically allocated memory is managed properly.
- Maintaining a clear structure in parsing complex CFG transformations.
- Handling indirect left recursion required careful substitution before direct recursion elimination.
- Detecting and reporting conflicts in the parsing table when the grammar is not LL(1).

Solution:

- The implementation includes conflict detection by tracking multiple entries for the same table cell.
- Special case handling for epsilon productions with the function: `bool hasEpsilonInFirstSet(const char* symbol)`
-
- This allows the algorithm to correctly determine when a sequence of symbols can derive the empty string and propagate FOLLOW set information appropriately.

5.2 Learnings:

- Efficient grammar transformation improves parsing efficiency.
- Unique non-terminals prevent conflicts in left factoring.
- Understanding CFG transformations is crucial for compiler design.

6. Verification and Testing

Correctness Verification

To verify correctness of the implementation, the following checks can be performed:

- **FIRST and FOLLOW Set Validation**:
  - Check against manually computed sets for small grammars.
  - Verify key properties (e.g., terminals in their own FIRST sets).
  - Confirm epsilon propagation works correctly.

- **Parse Table Inspection**:
  - Examine table entries for known LL(1) grammars.
  - Verify error handling for non-LL(1) grammars.

- **Conflict Detection**:
  - Test with deliberately ambiguous grammars.
  - Confirm all conflicts are properly detected and reported.

This ensures the correctness and reliability of the CFG transformation and LL(1) parsing implementation.