Report: Grammar Processing and LL(1) Parsing Table Construction

Authors: Awais Khan (22i-0997), Shayan Memon (22i-0773)
Date: March 22, 2025

---

1. Introduction

This report outlines the development of a C program designed to process a context-free grammar through multiple stages: parsing, left factoring, left recursion removal, computation of First and Follow Sets, construction of an LL(1) Parsing Table, and logging the output to a text file. The program takes a grammar from an input file (`input.txt`) and produces a detailed log (`output_log.txt`) showing the results at each stage. This report describes the approach taken, challenges encountered during implementation, and the methods used to verify the program's correctness.

---

2. Approach

The program was developed iteratively, building upon standard compiler design techniques. The approach is broken down into the following stages:

2.1 Grammar Parsing

- **Input:** A text file (`input.txt`) containing production rules in the format `A -> α | β`.
- **Method:** Read the file line by line, split each line at `->` to separate the left-hand side (LHS) non-terminal from the right-hand side (RHS) alternatives, and further split RHS by `|` to handle multiple alternatives. Store productions in a dynamic array of `Production` structures.
- **Output:** An in-memory representation of the grammar.

2.2 Left Factoring

- **Purpose:** Remove common prefixes from production alternatives to simplify parsing.
- **Method:** Identify common prefixes among RHS alternatives for each non-terminal. If found, introduce a new non-terminal (e.g., `A'`) to factor out the common part, updating the grammar accordingly.
- **Output:** Logged to `output_log.txt` under "After Left Factoring".

2.3 Left Recursion Removal

- **Purpose:** Eliminate left recursion to make the grammar suitable for top-down parsing.

- **Method:** For a recursive production `A → Aα | β`, transform it into:
  - `A → βA'`
  - `A' → αA' | ε` where `A'` is a new non-terminal.
- **Output:** Logged to `output_log.txt` under "After Left Recursion Removal".

2.4 First Sets Computation

- **Purpose:** Compute the set of terminals that can begin strings derived from each non-terminal.
- **Method:** Iteratively apply rules:
  - For `A → α`, add `First(α)` to `First(A)`, considering nullable non-terminals to propagate First Sets.
  - Stop when no changes occur.
- **Output:** Logged to `output_log.txt` under "First Sets".

2.5 Follow Sets Computation

- **Purpose:** Compute the set of terminals that can follow each non-terminal in a derivation.
- **Method:**
  - Initialize `Follow(S) = {$}` for the start symbol `S`.
  - For each production `B → αAβ`:
    - Add `First(β) \ {ε}` to `Follow(A)`.
    - If `β` is nullable, add `Follow(B)` to `Follow(A)`.
  - Iterate until convergence.
- **Output:** Logged to `output_log.txt` under "Follow Sets".

2.6 LL(1) Parsing Table Construction

- **Purpose:** Build a table for LL(1) parsing, mapping non-terminals and terminals to productions.
- **Method:**
  - For each production `A → α`:
    - Add `A → α` to `T[A, a]` for each `a ∈ First(α)`.
    - If `α` is nullable, add it to `T[A, b]` for each `b ∈ Follow(A)`.
  - Detect conflicts if a cell is overwritten.
- **Output:** Logged to `output_log.txt` under "LL(1) Parsing Table".

2.7 Output Logging

- **Purpose:** Record all stages in a single text file.
- **Method:** Open `output_log.txt` in write mode, pass the file pointer to output functions, and use `fprintf` to write results with stage headers and separators.
- **Output:** A single file, `output_log.txt`, containing all processing stages.

---

3. Challenges Faced

3.1 Memory Management

- **Challenge:** Dynamic allocation for productions, non-terminals, terminals, and sets risked memory leaks or overflows.
- **Solution:** Used `malloc` and `realloc` with capacity doubling for dynamic arrays, paired with `free_grammar` for cleanup.

3.2 Fixed Set Capacities

- **Challenge:** First and Follow Sets had a fixed capacity (e.g., 10), leading to possible overflow.
- **Solution:** Added a warning when capacity is exceeded. Considered but did not implement dynamic resizing.

3.3 Order of Transformations

- **Challenge:** Left factoring before recursion removal yielded no changes due to recursion hiding common prefixes.
- **Solution:** Kept the order (factoring before recursion removal) but noted that swapping them might improve results.

3.4 Table Formatting

- **Challenge:** Displaying the LL(1) Parsing Table in a readable format within a fixed-width text file.
- **Solution:** Used a fixed 8-character width per column (`%-8s` in `fprintf`).

3.5 Conflict Detection

- **Challenge:** Ensuring the LL(1) table correctly identified conflicts.
- **Solution:** Encoded production indices uniquely and checked for overwrites, printing conflict messages to the console.

---

4. Verification of Correctness

4.1 Step-by-Step Comparison

Method: Compared each stage's output in `output_log.txt` against manually computed results for the example grammar:

E → E + T | T

T → T * F | F

F → (E) | id

-
- **Results:**
    - First and Follow Sets matched expected values.
    - LL(1) table entries were correct, with no conflicts.

4.2 Theoretical Validation

- **Method:** Applied LL(1) conditions: `First` sets must be disjoint, and nullable productions must not cause overlap between `First` and `Follow` sets.
- **Result:** The transformed grammar satisfied these conditions.

4.3 Test Case Execution

- **Method:** Ran the program and compared `output_log.txt` with textbook examples.
- **Result:** Matched expected results, confirming correctness.

4.4 Edge Case Consideration

- **Method:** Considered empty grammars and malformed inputs.
- **Result:** Basic error handling was implemented but could be improved.

---

5. Conclusion

The C program successfully processes a grammar through all required stages, producing a detailed log in `output_log.txt`. Standard algorithms were leveraged, with modifications for file output. Challenges like memory management and table formatting were addressed, though some limitations remain. Correctness was verified through manual comparison, theoretical checks, and test execution. Future enhancements could include dynamic set resizing, improved error handling, and parsing simulation using the LL(1) table.