Assignment 2 Report

Compiler Construction

By: Hassaan Afzal (22i-0918)

Muhammad Asjad (22i-1227)

1. Introduction

This report provides an overview of the implementation of an LL(1) parser, detailing the approach used, challenges encountered, and the verification methods employed to ensure the correctness of the program.

2. Approach Taken

The implementation of the LL(1) parser followed a structured approach consisting of the following steps:

2.1 Reading and Storing the Grammar

- The program reads a context-free grammar (CFG) from a file and stores it in an appropriate data structure.
- Each production rule is parsed and stored using maps and vectors for efficient retrieval.

2.2 Left Factoring

- Left factoring was applied to remove common prefixes in production rules to make the grammar suitable for predictive parsing.
- Example transformation:
- `A -> X Y | X Z  -->  A -> X A'`
- `A' -> Y | Z`

2.3 Removing Left Recursion

- Both **direct** and **indirect** left recursion were handled.
- Example transformation:
- `A -> A α | β  -->  A -> β A'`
- `A' -> α A' | ε`
- Indirect left recursion required substitution of non-terminals before applying the direct recursion removal algorithm.

2.4 Computing FIRST and FOLLOW Sets

- **FIRST sets:** Determined the set of terminals that could appear at the beginning of derivations for each non-terminal.
- **FOLLOW sets:** Identified terminals that could appear immediately after a non-terminal in derivations.
- FOLLOW set computation ensured correct propagation of end-of-input ($) symbols.

2.5 Constructing the LL(1) Parsing Table

- The LL(1) parsing table was built using computed FIRST and FOLLOW sets.
- If a conflict was found (i.e., multiple entries for a single cell), the grammar was deemed **not LL(1)-compliant**.

2.6 Table Formatting Improvements

- Adjusted column widths dynamically based on terminal lengths.
- Ensured that long productions were truncated appropriately to maintain a uniform table layout.
- Handled multi-line formatting issues to avoid shifting of columns.

3. Challenges Faced

3.1 Handling Indirect Left Recursion

- Unlike direct left recursion, indirect recursion required detecting dependencies between non-terminals and performing recursive substitutions before elimination.
- Solution: Implemented a recursive algorithm to substitute recursive definitions before applying direct recursion removal.

3.2 Nullable Productions and FOLLOW Set Propagation

- Handling nullable productions (ε) properly was crucial for correct FOLLOW set propagation.
- Solution: Ensured that if a production was nullable, FOLLOW was inherited from its parent.

3.3 Formatting the LL(1) Parsing Table

- The initial implementation had issues where long productions caused column misalignment.
- Solution: Implemented dynamic column width adjustment and text wrapping to ensure a structured display.

4. Verification & Testing

The correctness of the program was verified through the following methods:

4.1 Sample Grammars Used for Testing

- **Basic Arithmetic Grammar:**
- `Expr -> Term Expr'`
- `Expr' -> + Term Expr' | ε`
- `Term -> id | num`
- **Conditional Statements Grammar:**
- `Stmt -> if ( Cond ) { StmtList } Stmt' | id = Expr ;`
- `Stmt' -> else { StmtList } | ε`
- **Indirect Left Recursion Test Case:**
- `A -> B y | z`
- `B -> S w | u`
- `S -> A x`

4.2 Comparing Computed FIRST and FOLLOW Sets

- Manually derived FIRST and FOLLOW sets were compared with program output to confirm correctness.

4.3 Validating LL(1) Parsing Table Construction

- The parsing table was checked for conflicts to verify that the grammar adhered to LL(1) rules.
- Formatting issues were fixed based on visual inspection.

5. Conclusion

The LL(1) parser implementation successfully handled **left recursion elimination, left factoring, FIRST and FOLLOW computation, and table formatting improvements**. The program was rigorously tested against multiple grammars to verify its correctness. The final implementation ensures efficient parsing of LL(1)-compliant grammars and outputs a well-structured parsing table.