Compiler Construction

Assignment # 3

Prepared By:
Umer Farooq | 22I-0891 | CS-6D
Muhammad Irtaza Khan | 22I-0911 | CS-6D

Date: 26th April, 2025

of Contents

LL(1) Parser Generator and String Parser 2
Project Overview 2
Input 2
Output 2
File Structure and Purpose 2
Detailed Process Flow 3
1. Grammar Transformation 3
2. FIRST and FOLLOW Set Computation 3
3. LL(1) Parsing Table Construction 3
String Parsing Implementation (Core Focus) 3
Parsing Stack Management 3
String Parsing Process 4
Error Recovery and Reporting 5
Verification and Testing 5
Conclusion 5

LL(1) Parser Generator and String Parser

Project Overview

This project implements a complete LL(1) parser generator and string parser for context-free grammars. The system takes a grammar as input, transforms it into a suitable form for LL(1) parsing by removing left recursion and left factoring, computes FIRST and FOLLOW sets, generates an LL(1) parsing table, and uses this table to parse input strings. The parser reports whether the input strings conform to the grammar rules and provides detailed error information when parsing fails.

Input

- A context-free grammar file (grammar.txt) defining production rules
- A file of input strings (input.txt) to be parsed according to the grammar

Output

- Transformed grammar after left factoring and left recursion removal
- FIRST and FOLLOW sets for each non-terminal
- LL(1) parsing table
- Step-by-step parsing process for each input string
- Summary of parsing results, including detailed error reporting

File Structure and Purpose

1. **main.c**: The entry point that coordinates the overall parsing process
2. **utils.c**: Contains utility functions for grammar manipulation and basic set operations
3. **leftFactoring.c**: Implements algorithms for left factoring the grammar
4. **leftRecursion.c**: Implements algorithms for removing left recursion from the grammar
5. **LL1Parser.c**: Contains functions to compute FIRST and FOLLOW sets and construct the parsing table
6. **parsingStack.c**: Implements the parsing stack data structure used during string parsing
7. **stringParser.c**: Contains functions to parse input strings using the LL(1) parsing table

Detailed Process Flow

1. Grammar Transformation

The system first loads the grammar from the input file and applies two important transformations:

- **Left Factoring**: Identifies common prefixes in production rules and factors them out to ensure deterministic parsing
- **Left Recursion Removal**: Eliminates direct left recursion by introducing new non-terminals, making the grammar suitable for top-down parsing

2. FIRST and FOLLOW Set Computation

After transforming the grammar, the system computes:

- **FIRST sets**: Determines the set of terminals that can appear as the first symbol in strings derived from each non-terminal
- **FOLLOW sets**: Identifies terminals that can appear immediately to the right of each non-terminal in valid derivations

3. LL(1) Parsing Table Construction

Using the FIRST and FOLLOW sets, the system builds an LL(1) parsing table that maps (non-terminal, terminal) pairs to production rules. This table guides the parser in selecting the appropriate production during string parsing.

String Parsing Implementation (Core Focus)

Parsing Stack Management

The parsing stack plays a central role in the LL(1) parsing algorithm. The parsingStack.c file implements a stack data structure with specialized operations:

- **Stack Initialization**: initStack() prepares an empty stack for parsing
- **Symbol Management**:
  - push() and pop() handle single character symbols
  - pushSymbol() and popSymbol() handle multi-character symbols (like "id")
- **Stack Inspection**: peek() and peekSymbol() examine the top of the stack without modifying it
- **Stack Display**: displayStack() shows the current state of the stack during parsing

A key innovation in this implementation is the handling of multi-character tokens, particularly the "id" token, which is treated as a single logical unit despite consisting of multiple characters.

String Parsing Process

The stringParser.c file implements the core parsing algorithm:

1. **Stack Initialization**:
    - The stack is initialized with the end marker '$' and the grammar's start symbol
    - This represents the initial goal of the parser: to derive a string that matches the input
2. **Parsing Loop**:
    - The parser repeatedly compares the top of the stack with the current input symbol
    - Based on this comparison, it takes one of three actions:
        - **Match**: If the top of stack is a terminal matching the current input, both are consumed
        - **Expand**: If the top of stack is a non-terminal, the parser consults the LL(1) table to replace it with an appropriate production
        - **Error Recovery**: If no valid action exists, the parser reports an error and attempts to continue parsing
3. **Multi-character Token Handling**:
    - The parser has special logic for handling "id" tokens, treating them as a single unit
    - This requires careful coordination between the input scanning and stack manipulation
4. **Error Reporting**:
    - When errors are detected, they are collected in a LineErrors structure
    - For each line, up to 10 detailed error messages can be stored
    - The parser continues attempting to parse even after errors, up to a limit of 20 errors per line
5. **Parsing Visualization**:
    - Each step of the parsing process is displayed, showing:
        - Current input symbol
        - Current stack contents
        - Production being applied (if any)

        - Matching operations
6. **Summary Generation**:
    - After parsing all input strings, a comprehensive summary is displayed
    - This includes the parsing status of each line and detailed error information

Error Recovery and Reporting

The parser implements a simple but effective error recovery strategy:

1. When a syntax error is detected, the parser reports the error with a descriptive message
2. The parser then skips the problematic symbol and continues parsing
3. This approach allows the parser to detect multiple errors in a single pass

The error messages are specific and helpful, indicating:

- Expected symbols that were not found
- Unexpected symbols encountered
- Missing productions in the parsing table

Verification and Testing

The correctness of the parser was verified using:

1. Test cases with both valid and invalid input strings
2. Step-by-step tracing of the parsing process
3. Comparison of parsing results with expected outcomes

The sample grammar (E → E+T | T, T → T*F | F, F → (E) | id) was used as a primary test case, as it demonstrates both left recursion and the need for precedence handling, which are addressed correctly by the system.

Conclusion

This LL(1) parser generator successfully transforms context-free grammars into a form suitable for top-down parsing, generates the necessary parsing tables, and performs detailed syntax analysis on input strings. The implementation handles multi-character tokens, produces comprehensive error reports, and provides clear visualization of the parsing process. The modular design makes it easy to understand and extend the system for more complex parsing requirements.