Compiler Construction

Assignment #3 Report

Group Members

Huzaifa Nasir	22i-1053
Maaz Ali	22i-1042
Section	CS-A

Introduction

This report documents the implementation of a stack-based LL(1) parser. The parser, written in C, processes space-separated terminal symbols from an input file based on a given grammar, producing detailed parsing steps and error messages. This report provides a brief overview of our approach, the challenges encountered during development, and the methods used to verify the correctness of the program.

Approach

The implementation leveraged the LL(1) parser developed in Assignment 2, integrating a stack-based parsing mechanism to process input strings from input.txt according to the grammar in grammar.txt. The program was structured to meet the assignment's requirements for modularity, reusability, and clear output. Below is a detailed overview of the approach:

1. Grammar Processing

- **Input Handling**: The grammar was read from grammar.txt using the readGrammarFromFile function, which parsed productions into a GrammarRule structure (storing non-terminals and their productions).
- **Grammar Transformation**:
    - Applied left factoring (leftFactorGrammar) to eliminate common prefixes in productions.
    - Removed left recursion (removeLeftRecursion) to ensure the grammar was LL(1)-compliant.
- **Parsing Table Construction**:
    - Computed First and Follow sets using computeFirstSets and computeFollowSets to identify viable terminals for each non-terminal.
    - Built the LL(1) parsing table with buildLL1Table, mapping non-terminals and terminals to productions.
- **Output**: Displayed the initial grammar, transformed grammar, First/Follow sets, and LL(1) table in output.txt for verification.

2. Input Parsing

- **Input Processing**: The parseInputFile function read input.txt line by line. Each line, containing space-separated terminals (e.g., k i ;, i = (i + n) * i ;), was tokenized into an array using tokenize, with $ appended as the end marker.

- **Stack-Based Parsing**: The parseLine function implemented the LL(1) parsing algorithm:
    - Initialized a stack with the start symbol (S) and $.
    - For each input token:
        - If the stack top was a terminal, it was matched against the current input token. On a match, the stack was popped, and the input advanced.
        - If the stack top was a non-terminal, the LL(1) parsing table was consulted to select a production, which was pushed onto the stack in reverse order.
        - If no valid action was possible (e.g., missing table entry), an error was reported, and the input token was skipped.
    - At each step, the stack contents, remaining input, and action (e.g., “match 'k'”, “use S->D”, “ERROR: expected ';'”) were printed in a tabular format.
- **Output**: Each line’s parsing steps were logged, followed by a success message (“Parsed successfully”) or error summary. The output was redirected to output.txt.

3. Error Handling

- **Detection**: Errors were identified in cases such as:
    - Mismatched terminals (stack top did not match the input token).
    - Missing LL(1) table entries for the current non-terminal and input token.
    - Unexpected end of input.
- **Recovery**: The parser skipped the erroneous token, logged a descriptive error message (e.g., “ERROR: expected ';', discarding '+'”), and continued parsing to detect additional errors.
- **Reporting**: Errors were reported with the step number, stack state, input, and a clear explanation. A summary of errors per line and total errors was included in the output.

4. Code Design

- **Modularity**: The code was organized into distinct functions for grammar processing (readGrammarFromFile, buildLL1Table), set computation (computeFirstSets, computeFollowSets), and parsing (parseInputFile, parseLine).
- **Reusability**: The parser was designed to handle any LL(1) grammar, avoiding hardcoding specific rules.
- **Documentation**: Comments explained key functions, data structures (e.g., GrammarRule, Sets), and constants (e.g., MAX_STACK, EPSILON).

- **Output Formatting**: Helper functions (printStack, printRemaining) ensured consistent, readable output tables.

Challenges Faced

1. **Robust Error Recovery**:
    - Designing an error recovery mechanism that allowed continued parsing after errors was challenging. Early attempts to skip erroneous tokens caused synchronization issues, leading to incorrect parsing steps. We resolved this by skipping only the problematic token and resuming with the next input, ensuring accurate error detection.
    - Crafting precise error messages required tracking the expected and actual tokens, which was complex for nested constructs.
2. **Stack Management**:
    - Correctly managing the parsing stack for productions with multiple symbols was difficult. Pushing productions in reverse order was critical to maintain the correct parsing sequence, and errors in this logic caused incorrect stack states. We debugged this by tracing stack operations for small inputs.
    - Ensuring the stack could handle long inputs without overflowing required setting an appropriate MAX_STACK limit and validating bounds.
3. **Output Formatting**:
    - Producing a clear, aligned table for parsing steps (stack, input, action) was time-consuming. We used fixed-width columns and helper functions to achieve consistency, but aligning dynamic content (e.g., variable-length stack contents) required multiple iterations.
    - Matching the assignment's example output format (e.g., specific phrasing for actions) needed careful attention to detail.
4. **Integration with Assignment 2**:
    - Bugs in the Assignment 2 parser, such as inaccuracies in First/Follow sets, initially caused parsing failures. We retested the grammar processing functions to ensure correctness before adding the stack-based parsing logic.
    - Adapting the parser to handle space-separated terminals (instead of a tokenized stream) required rewriting the input processing logic to correctly tokenize and feed terminals to the parser.

Verification of Correctness

We employed a rigorous verification process to ensure the program's correctness:

1. **Test Cases**:
    - **Valid Inputs**: Tested all five lines in input.txt, covering declarations (k i ;), assignments (i = i + n ;), if-statements (f (i > n) { i = i + n ; }), and expressions (i = (i + n) * i ;). All lines parsed successfully with zero errors, as verified in output.txt.
    - **Invalid Inputs**: Created test cases with syntax errors (e.g., i = i + ;, f (i > n {) to validate error detection and recovery. The parser correctly identified issues (e.g., unexpected tokens, missing symbols) and continued parsing.
    - **Edge Cases**: Tested empty lines, lines with only spaces, and unrecognized tokens. The parser skipped empty lines and reported errors for invalid tokens without crashing.
2. **Grammar and Parsing Table Validation**:
    - Manually computed First and Follow sets for the grammar and compared them with the program's output to confirm accuracy.
    - Inspected the LL(1) parsing table to ensure no conflicts existed, verifying that the grammar was LL(1)-compliant after left factoring and recursion removal.
3. **Output Verification**:
    - Compared output.txt with the assignment's requirements, confirming that parsing steps, stack contents, actions, and error messages were correctly formatted.
    - Validated that success messages ("Parsed successfully") and error summaries (e.g., "Parsing completed with 0 total errors") were accurate and consistent with the parsing results.
4. **Debugging and Tracing**:
    - Added temporary logging to trace stack operations, parsing table lookups, and token processing, helping identify and fix issues in stack updates.
    - Manually traced the parsing process for small inputs to verify that stack operations and production selections aligned with the LL(1) algorithm.
5. **Genericity Testing**:

- Tested the parser with an alternative LL(1) grammar to ensure it was not hardcoded to grammar.txt. The program correctly processed the new grammar, generating accurate First/Follow sets, parsing tables, and parsing outputs.

Conclusion

The stack-based LL(1) parser successfully processed input strings, producing detailed parsing logs and handling errors robustly. The modular C implementation, with clear documentation and reusable functions, effectively met the project's goals. Challenges in error recovery, stack management, and output formatting were overcome through debugging and iterative refinement. Comprehensive testing, including valid inputs, error cases, and alternative grammars, confirmed the program's correctness and reliability, reinforcing our understanding of compiler parsing techniques.