Compiler Construction
Assignment #03
Report

Submitted By:

- Mohammad Bilal (22i-0806)
- Hassan Imran (22i-0813*)*

1. Introduction

This report details the implementation of Assignment 3 for the Compiler Construction course, which builds on Assignment 2 by extending the LL(1) parser to parse space-separated terminal strings using a stack-based approach. The program, written in C++, processes a context-free grammar (CFG), performs left factoring and left recursion removal, computes FIRST and FOLLOW sets, constructs an LL(1) parsing table, and parses input strings while handling syntax errors with recovery. This report describes the approach, challenges faced, and verification methods used to ensure correctness.

2. Approach

The implementation reuses and extends the C++ code from Assignment 2 (a2.cpp), adapting it to meet Assignment 3's requirements. The key components of the approach are:

2.1. Grammar Processing

- **Input**: The program reads a CFG from grammar.txt in the format NonTerminal -> Symbol1 Symbol2 ... | Symbol3 ..., supporting multi-character symbols (e.g., int, if).
- **Transformations**:
    - **Left Factoring**: Identifies common prefixes among productions and rewrites them using new non-terminals (e.g., A -> αβ | αγ becomes A -> αA', A' -> β | γ).
    - **Left Recursion Removal**: Eliminates direct and indirect left recursion by introducing new non-terminals and restructuring productions.
- **Data Structure**: The Grammar struct uses a map<string, vector<vector<string>>> to store productions and a string for the start symbol, updated from Assignment 2's char-based representation.

2.2. FIRST and FOLLOW Sets

- **FIRST Sets**: Computed recursively using a depth-first search, handling multi-character terminals and epsilon ("e") productions. For each non-terminal, FIRST includes terminals that begin derivable strings.
- **FOLLOW Sets**: Computed iteratively, initializing FOLLOW(start symbol) with "$". For each production, FOLLOW sets are updated based on terminals following non-terminals and epsilon derivations.
- **Implementation**: Updated Assignment 2's functions to use set<string> instead of set<char>.

2.3. LL(1) Parsing Table

- **Construction**: The parsing table is a map<pair<string, string>, string>, mapping (non-terminal, terminal) pairs to productions. For each production A -> α, entries are added for terminals in FIRST(α) and FOLLOW(A) if α derives epsilon.
- **Conflicts**: Detected by appending multiple productions to the same table cell, indicating non-LL(1) grammars.
- **Output**: Displayed as a table with terminals as columns and non-terminals as rows.

2.4. Parsing Stack and Input Processing

- **Stack**: A Stack struct manages parsing symbols using a vector<string>, supporting push, pop, and production expansion (pushing symbols in reverse order).
- **Input**: Reads input.txt line-by-line, splitting each line into space-separated tokens (e.g., int x ; becomes {"int", "x", ";"}).
- **Parsing Algorithm**:
    - Initialize stack with ["$", startSymbol].
    - For each input token:
        - If the stack top is a terminal, match it with the input token (pop and advance if matched, else error).
        - If the stack top is a non-terminal, use the parsing table to select a production and push its symbols.
        - If no table entry or a mismatch occurs, handle the error.
    - Continue until the stack is ["$"] and input is consumed or an error is handled.

2.5. Error Handling and Recovery

- **Error Detection**:
    - Missing table entries (no production for M[A,a]).
    - Mismatched terminals (stack top ≠ input token).
- **Recovery**: Uses panic mode:
    - Skips input tokens until a synchronizing token (from FOLLOW set) is found.
    - Pops the stack to stabilize parsing if necessary.
- **Output**: Errors are logged with line numbers (e.g., Line 3: Syntax Error: Unexpected token ';'), and parsing continues after recovery.

- **Summary**: Reports total errors (e.g., Parsing completed with 2 errors).

2.6. Output

- **Step-by-Step**: Displays a table with columns for step number, stack contents, current input token, and action (e.g., Match int, Expand S -> D, Error: Unexpected token ;).
- **Per-Line Results**: Indicates success (Parsed successfully) or recovery (Parsing continued after error recovery).
- **Final Message**: Summarizes parsing outcome.

3. Challenges Faced

Several challenges were encountered during implementation:

1. **Multi-Character Terminals**:
    - **Challenge**: Assignment 2 used single-character symbols (char), but Assignment 3 required strings (e.g., int, if).
    - **Solution**: Updated Grammar to use string and modified all functions (readGrammar, applyLeftFactoring, etc.) to handle string comparisons and storage. Input parsing was adapted to split space-separated symbols.
2. **Error Recovery**:
    - **Challenge**: Implementing panic mode recovery to continue parsing after errors (e.g., x = 5 + ;) was complex, as it required skipping tokens or popping stack symbols without breaking the parser.
    - **Solution**: Used FOLLOW sets to identify synchronizing tokens and implemented a recovery loop that skips invalid tokens or pops non-terminals, ensuring subsequent lines are parsed.
3. **Output Formatting**:
    - **Challenge**: Producing a clear, tabular output for parsing steps required careful string formatting to align columns and handle variable-length symbols.
    - **Solution**: Used setw from <iomanip> to format the output table, ensuring readability for stack contents, input tokens, and actions.
4. **Integration with Assignment 2**:
    - **Challenge**: Adapting Assignment 2's code to support Assignment 3's requirements while maintaining modularity was time-consuming.
    - **Solution**: Retained the modular structure (separate functions for each task) and systematically updated data types and logic, ensuring compatibility with the new parsing functionality.

4. Verification of Correctness

The program's correctness was verified through multiple approaches:

1. **Test Cases**:
    - **Valid Input**: int x ; (parses as S -> D -> int V ;).
    - **Invalid Input**: x = 5 + ; (triggers error due to unexpected =, recovers by skipping tokens).
    - **Complex Input**: if (x > num) { x = x - 1 ; } (parses partially, detects errors in the block, recovers).

Used the sample grammar:
S -> D | I
D -> int V ;
I -> if (E) { S }
V -> x

    - E -> x > num
2. **Manual Verification**:
    - Compared parsing steps (stack transitions, actions) with hand-calculated LL(1) parsing traces for valid inputs.
    - Verified error messages and recovery actions against expected behavior (e.g., Line 3: Syntax Error: Unexpected token ';').
    - Checked FIRST and FOLLOW sets against manual computations to ensure accurate table construction.
3. **Step-by-Step Output**:
    - Inspected the tabular output to confirm correct stack operations, token matching, and production expansions.
    - Ensured error messages included line numbers and specific details, as required.
4. **Edge Cases**:
    - Tested empty lines and malformed inputs (e.g., int x) to ensure robust error handling.
    - Verified that the parser continues after errors, matching the example output's behavior (Parsing continued after error recovery).

5. Limitations and Potential Improvements

- **Limitations**:
    - The grammar assumes space-separated symbols, which may not handle complex tokenization (e.g., no spaces between operators).
    - Error recovery is basic (panic mode), which may skip valid tokens unnecessarily.
    - The program supports only LL(1) grammars, limiting its applicability to ambiguous or non-LL(1) CFGs.
- **Improvements**:
    - Add support for tokenization without spaces using a lexer.
    - Implement more sophisticated error recovery (e.g., inserting missing tokens).
    - Extend the parser to generate parse trees or support other parsing techniques (e.g., LR parsing).

6. Conclusion

The Assignment 3 implementation successfully extends the LL(1) parser from Assignment 2 to parse space-separated terminal strings using a stack-based approach. By updating the grammar representation for multi-character symbols, implementing a robust parsing stack, and adding error handling with recovery, the program meets all requirements. Challenges such as handling string-based symbols and error recovery were addressed through careful design and testing. The verification process confirmed the program's correctness across various test cases, ensuring it is a solid foundation for further compiler development.

Sample Output (Excerpt):

```
Parsing Input:
Step | Stack              | Input         | Action
-----|--------------------|---------------|----------------------------
0    | [$ S]              | int           | Expand S -> int V ;
1    | [$ ; V int]        | int           | Match int
...
Line 2: Syntax Error: Unexpected token '='
Line 2: Parsing continued after error recovery.
...
Parsing completed with 2 errors.
```

Attachments:

- Source code: a3.cpp
- Sample input: input.txt
- Sample grammar: grammar.txt
- Output log: output.txt