Compiler Construction Assignment 2 Report

Team: Pink Pony Club
Members:

- Ayaan Khan (22i-0832) Section: K
- Minahil Ali (22i-0849) Section G

Overview

For this assignment, we implemented a Context-Free Grammar (CFG) processor that performs several key operations essential for compiler construction. Our program reads a grammar from a file, applies left factoring and left recursion elimination, computes FIRST and FOLLOW sets, and creates an LL(1) parsing table.

Commands to run the program:
```
g++ sourceCFG.cpp -o a
./a grammar3.txt output3.txt
```

Approach

1. Grammar Representation

We designed data structures to effectively represent the grammar:

```
struct Grammar {
    set<string> terminals;
    set<string> nonTerminals;
    map<string, vector<vector<string>>> productions; // Non-terminal
-> list of alternative productions
    string startSymbol;
};
```

This structure allows us to store terminals, non-terminals, and organize productions by their left-hand side non-terminal. Each production's right-hand side is stored as a vector of symbols, making it easy to process.

2. Grammar Parsing

The grammar is read from a file with careful handling of the production rule syntax:

```
size_t arrowPos = line.find("->");
if (arrowPos == string::npos) continue;

string lhs = line.substr(0, arrowPos);
string rhs = line.substr(arrowPos + 2);

// Trim whitespace
lhs.erase(0, lhs.find_first_not_of(" \t"));
lhs.erase(lhs.find_last_not_of(" \t") + 1);
```

This code identifies the left-hand side and right-hand side of each production by finding the "->" separator. We then handle alternatives (separated by '|') and epsilon productions explicitly.

3. Left Factoring Implementation

Left factoring is crucial for creating predictive parsers. Our implementation iteratively identifies and factors common prefixes:

```
// Map prefix -> list of productions with that prefix
map<string, vector<vector<string>>> prefixMap;
for (const auto& prod : productions) {
    if (prod.empty()) continue;
    string prefix = prod[0];
    prefixMap[prefix].push_back(prod);
}
```

When multiple productions for a non-terminal start with the same symbol, we create a new non-terminal for the remaining parts, thereby eliminating ambiguity in the first prediction step.

4. Left Recursion Elimination

Left recursion is problematic for LL parsers. We implemented a two-phase approach:

```
// Now eliminate immediate left recursion for A_i.
vector<vector<string>> alpha; // Productions of the form A_i -> A_i
α.
vector<vector<string>> beta;  // Productions not starting with A_i.
for (const auto& production : newProds[Ai]) {
    if (!production.empty() && production[0] == Ai) {
        // Remove the leading A_i.
        vector<string> alphaPart(production.begin() + 1,
production.end());
        alpha.push_back(alphaPart);
    } else {
        beta.push_back(production);
    }
}
```

First, we substitute productions to eliminate indirect left recursion. Then, we transform directly left-recursive productions by creating new non-terminals and rewriting the rules.

5. FIRST Set Computation

The FIRST set algorithm is implemented recursively:

```
set<string> CFGProcessor::computeFirstOfString(const vector<string>&
symbols) {
    set<string> firstSet;

    if (symbols.empty()) {
        firstSet.insert("epsilon");
        return firstSet;
    }

    // Initialize allHaveEpsilon to true, will track if all symbols
can derive epsilon
    bool allHaveEpsilon = true;
```

For a string of symbols, we determine which terminals can appear first in any derivation. We handle epsilon productions carefully, as they create special cases where we may need to look at subsequent symbols.

6. FOLLOW Set Computation

FOLLOW sets are computed iteratively until convergence:

```
// If B is the last symbol, add FOLLOW(A) to FOLLOW(B)
size_t beforeSize = followSets[B].size();
followSets[B].insert(followSets[nonTerminal].begin(),
followSets[nonTerminal].end());
if (followSets[B].size() > beforeSize) {
    changed = true;
}
```

We apply the rules: if B is followed by a symbol, add FIRST(that symbol) to FOLLOW(B); if B is at the end of a production or FIRST of the following symbols contains epsilon, add FOLLOW of the left-hand side to FOLLOW(B).

7. Parse Table Construction

```
The LL(1) parsing table is constructed using the FIRST and FOLLOW
sets:
// For each terminal 'a' in FIRST(α), add A -> α to M[A, a]
for (const auto& terminal : firstAlpha) {
    if (terminal != "epsilon") {
        parseTable[{nonTerminal, terminal}] = production;
    }
}

// If epsilon is in FIRST(α), for each terminal 'b' in FOLLOW(A), add
A -> α to M[A, b]
if (firstAlpha.find("epsilon") != firstAlpha.end()) {
    for (const auto& terminal : followSets[nonTerminal]) {
        parseTable[{nonTerminal, terminal}] = production;
    }
}
```

This is the core of the LL(1) parsing table algorithm, determining which production to use when a specific terminal is encountered.

Challenges Faced

1. Handling Epsilon Productions

Epsilon productions required special treatment throughout the code. We needed to handle them carefully in FIRST set computation, FOLLOW set computation, and parse table generation to ensure correct behavior.

2. Left Recursion Elimination

Implementing the left recursion elimination algorithm was challenging, particularly handling indirect left recursion. We had to carefully track substitutions and new production formations to avoid introducing errors.

3. Computation of FOLLOW Sets

Computing FOLLOW sets correctly proved challenging, especially when epsilon productions were involved:

```
// If epsilon is in FIRST(beta), add FOLLOW(A) to FOLLOW(B)
if (firstBeta.find("epsilon") != firstBeta.end()) {
    size_t beforeSize = followSets[B].size();
    followSets[B].insert(followSets[nonTerminal].begin(),
followSets[nonTerminal].end());
    if (followSets[B].size() > beforeSize) {
        changed = true;
    }
}
```

We needed multiple iterations to ensure that these sets converged correctly, particularly when there were complex dependencies between non-terminals.

4. Indirect Left Recursion

Handling indirect left recursion required careful implementation of the algorithm to substitute one non-terminal's productions into another:

```
// For each A_j with j < i, substitute productions of A_j into
productions of A_i.
for (size_t j = 0; j < i; j++) {
    string Aj = origNonTerminals[j];
    vector<vector<string>> updated;
    // For each production of A_i.
    for (const auto& production : newProds[Ai]) {
        // If the production begins with A_j, replace it.
        if (!production.empty() && production[0] == Aj) {
            // Remove A_j from the beginning.
            vector<string> gamma(production.begin() + 1,
production.end());
            // For each production A_j -> delta, form A_i -> delta
gamma.
            for (const auto& delta : newProds[Aj]) {
                vector<string> newProduction;
                newProduction.insert(newProduction.end(),
delta.begin(), delta.end());
                newProduction.insert(newProduction.end(),
gamma.begin(), gamma.end());
                updated.push_back(newProduction);
            }
        } else {
            updated.push_back(production);
        }
    }
    newProds[Ai] = updated;
}
```

This process is order-dependent, requiring us to process non-terminals in a specific sequence to ensure all indirect left recursion is eliminated.

5. Handling Ambiguous Grammars

Detecting and resolving ambiguities in the grammar was a significant challenge. When constructing the parse table, we needed to check for conflicts:

```
// For each terminal 'a' in FIRST(α), add A -> α to M[A, a]
for (const auto& terminal : firstAlpha) {
    if (terminal != "epsilon") {
        // Check if there's already an entry in the table
        if (parseTable.find({nonTerminal, terminal}) !=
parseTable.end()) {
            // Handle ambiguity here
        }
        parseTable[{nonTerminal, terminal}] = production;
    }
}
```

These conflicts indicate that the grammar is not LL(1), and additional transformations would be needed to make it suitable for predictive parsing.

Verification of Correctness

We verified our implementation through several methods:

1. **Test Cases**: We tested the program with various grammars, including those with left recursion, common prefixes, and epsilon productions.
2. **Manual Verification**: For smaller grammars, we manually calculated the expected FIRST and FOLLOW sets and compared them with the program's output.
3. **Recursive Descent Parser Generation**: We checked that our resulting grammar was suitable for a recursive descent parser by ensuring it was LL(1) compliant.
4. **Comparison with Textbook Algorithms**: We compared our results with the algorithms described in compiler construction textbooks.
5. **Edge Case Testing**: We verified behavior with special cases like empty productions and grammars with a single production rule.

Conclusion

Our CFG processor successfully performs the transformations necessary to prepare a grammar for use in an LL(1) parser. The program reliably handles left factoring, left recursion elimination, and computes FIRST and FOLLOW sets correctly. The parse table generation provides a clear visual representation of the parsing strategy.

The project reinforced our understanding of the theoretical aspects of compiler construction while giving us practical experience implementing these algorithms. The modular design allows for future expansion, such as adding support for semantic actions or symbol table generation.