Compiler Construction: Assignment #02

Design and Implementation for CFG Processor

Azfar Bilal 22i0950

Sahrish Mustafa 22i0977

Table of Contents

Introduction

This report describes the approach, challenges, and verification steps for a program that processes context-free grammars (CFGs). The program **reads a CFG from a file**, performs **left factoring** (with recursive factoring for multiple alternatives), removes **left recursion** (both direct and indirect), computes **FIRST and FOLLOW** sets, and finally constructs an **LL(1) parsing table.**

Approach

1. Left Factoring:
    - The program first tokenizes the production rules, then groups alternatives based on their common prefixes.
    - A helper function (`factorRuleSingle`) computes the longest common prefix for alternatives and factors them out.
    - Recursive calls are used to ensure that if a production has more than two alternatives or additional common factors after initial factoring, these are processed further.
    - The new nonterminals are generated with an apostrophe suffix (e.g., A', A'') to maintain uniqueness.
2. Left Recursion Removal:
    - The program first builds a grammar map while preserving the order of nonterminals.
    - Indirect left recursion is removed by substituting earlier nonterminals into later ones.
    - Immediate left recursion is eliminated using the standard transformation:
    - For a production A -> Aα | β, it is rewritten as A -> β A' and A' -> α A' | ε (with epsilon represented as %).
    - The removal process handles both indirect and direct recursion, updating the grammar map and the nonterminal order accordingly.
3. FIRST and FOLLOW Sets Computation:
    - FIRST sets are computed recursively for nonterminals only. If a symbol is a terminal, it is returned immediately.
    - FOLLOW sets are computed by scanning each production and using the FIRST sets to decide which tokens follow a nonterminal.
    - Epsilon is represented as % in FIRST sets, whereas $ remains as the end-of-input marker in FOLLOW sets.

    - The code avoids computing FIRST/FOLLOW for terminal symbols by checking membership in the terminals set.

4. LL(1) Parsing Table Construction:
    - The LL(1) parsing table is constructed as a mapping from nonterminals to a mapping of terminal symbols to the production rule.
    - FIRST sets for individual productions are computed, and if epsilon is in the FIRST set, the FOLLOW set of the left-hand nonterminal is used.

Challenges Faced

Left Factoring:
Indirect left factoring was tricky because it required a recursive code to check whether all cases of left factoring are handled. Since recursion is something we hadn't practiced in a while, implementing it proved to be initially challenging.

Differentiating Symbols:
A major challenge was distinguishing between the epsilon symbol and the end-of-input marker. The code was updated so that epsilon is represented as % while the end-of-input marker remains $ in the FOLLOW sets.

Ensuring Correctness in FIRST/FOLLOW Computation:
The recursive computation of FIRST and FOLLOW sets needed proper base-case handling to prevent incorrect propagation of symbols, especially when terminals were mistakenly processed as nonterminals.

Verification

The correctness of the program was verified using several test cases including:

- **Simple Grammar with Common Prefixes:**
  For example, testing productions like A -> ax | ay should factor to A -> a A' and A' -> x | y.

- **Multiple Alternatives:**
  Grammars with more than two alternatives (e.g., A -> abc | abd | aef) were used to ensure recursive factoring correctly handled all common prefixes.

- **Left Recursion Cases:**
  Both direct and indirect left recursion examples were processed to ensure that the grammar was correctly transformed into a form suitable for LL(1) parsing.

- **FIRST and FOLLOW Sets:**
  The computed FIRST and FOLLOW sets were compared against known examples and manual derivations.

- **LL(1) Table:**
  Finally, the LL(1) parsing table was generated and manually verified for consistency with the grammar.

Conclusion

The program successfully integrates left factoring, left recursion removal, FIRST/FOLLOW set computation, and LL(1) table construction into a single CFG processing pipeline. Verification through a diverse set of test cases confirmed the correctness of the approach. Future work might include extending the parser to handle more complex grammars and incorporating better error handling.