Context-Free Grammar Processing and LL(1) Parsing

Compiler Construction

Submitted By:

Anoosha Ali – 22i-1242

Ali Usman – 22i-0926

Section: J

Date of Submission: 23rd March, 2025

1. Introduction

This project implements key techniques for CFG transformation and analysis, including Left Factoring, Left Recursion Removal, First & Follow Set Computation, and LL(1) Parsing Table Construction.

2. Implementation Details

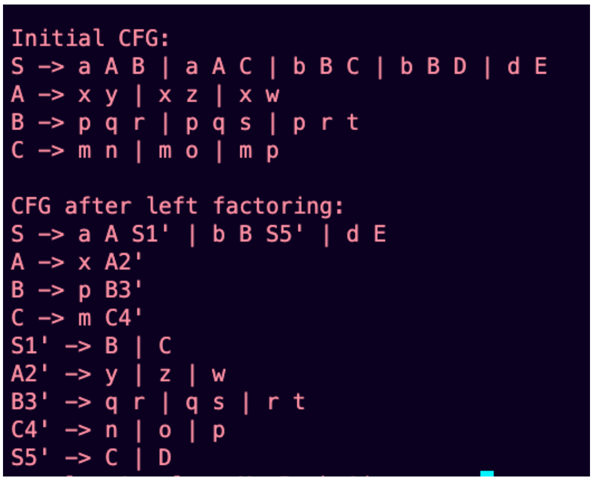
This section outlines the transformations and computations performed on the CFG.

2.1 Left Factoring

Left factoring is applied to CFG productions to eliminate common prefixes, ensuring that the grammar is suitable for LL(1) parsing. The implemented function:

- Splits production rules into tokens.
- Identifies the longest common prefix.
- Creates a **new non-terminal** to handle factored parts uniquely.
- Replaces the original production with the factored version.

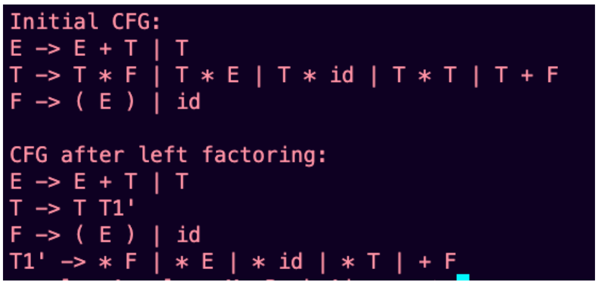
Example Transformation:



```
Initial CFG:
S -> a A B | a A C | b B C | b B D | d E
A -> x y | x z | x w
B -> p q r | p q s | p r t
C -> m n | m o | m p

CFG after left factoring:
S -> a A S1' | b B S5' | d E
A -> x A2'
B -> p B3'
C -> m C4'
S1' -> B | C
A2' -> y | z | w
B3' -> q r | q s | r t
C4' -> n | o | p
S5' -> C | D
```

Example 01



```
Initial CFG:
E -> E + T | T
T -> T * F | T * E | T * id | T * T | T + F
F -> ( E ) | id

CFG after left factoring:
E -> E + T | T
T -> T T1'
F -> ( E ) | id
T1' -> * F | * E | * id | * T | + F
```

Example 02

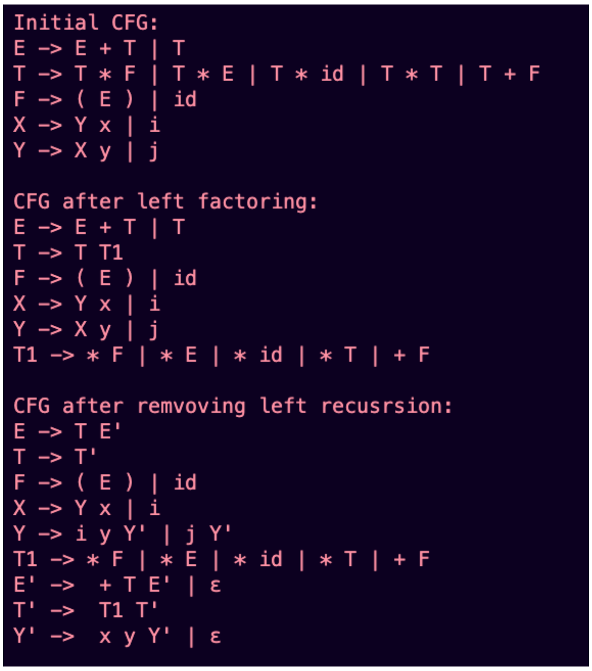
2.2 Left Recursion Removal

Left recursion occurs in a CFG when a non-terminal refers to itself as the leftmost symbol in its production, making it unsuitable for top-down parsing. The implemented function:

- **Identifies left recursion** by analyzing each production rule to check if the left-hand side non-terminal appears at the start of any right-hand side.
- **Distinguishes between recursive and non-recursive productions**, separating α (recursive part) and β (non-recursive part).

- **Eliminates direct left recursion** by introducing a new non-terminal and rewriting the production in the form:
  - If `A → Aα | β`, it is transformed into:
    - `A → β A'`
    - `A' → α A' | ε`
- **Handles cases where no β exists** by ensuring the original non-terminal directly references the newly introduced non-terminal.
- **Detects and resolves indirect left recursion** by processing non-terminals in a hierarchical manner, replacing indirect recursive references before handling direct left recursion.

Example Transformation:

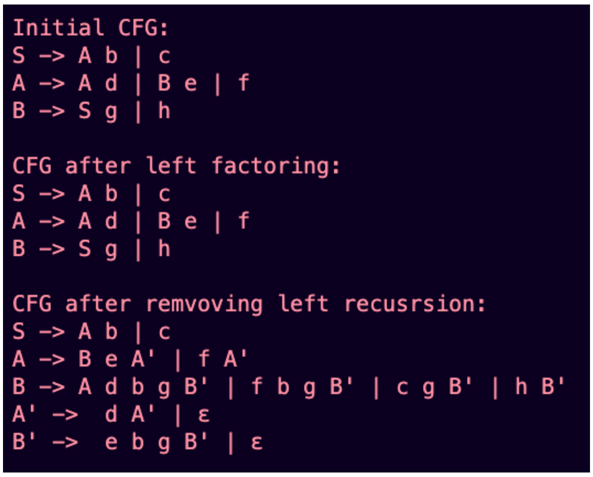


```
Initial CFG:
E -> E + T | T
T -> T * F | T * E | T * id | T * T | T + F
F -> ( E ) | id
X -> Y x | i
Y -> X y | j

CFG after left factoring:
E -> E + T | T
T -> T T1
F -> ( E ) | id
X -> Y x | i
Y -> X y | j
T1 -> * F | * E | * id | * T | + F

CFG after remvoving left recusrsion:
E -> T E'
T -> T'
F -> ( E ) | id
X -> Y x | i
Y -> i y Y' | j Y'
T1 -> * F | * E | * id | * T | + F
E' ->  + T E' | ε
T' ->  T1 T'
Y' ->  x y Y' | ε
```

Example 01



```
Initial CFG:
S -> A b | c
A -> A d | B e | f
B -> S g | h

CFG after left factoring:
S -> A b | c
A -> A d | B e | f
B -> S g | h

CFG after remvoving left recusrsion:
S -> A b | c
A -> B e A' | f A'
B -> A d b g B' | f b g B' | c g B' | h B'
A' ->  d A' | ε
B' ->  e b g B' | ε
```

Example 02

2.3 First & Follow Set Computation

2.4 LL(1) Parsing Table Construction

3. Data Structures & Code Organization

- **Grammar Representation:** Stored using a structure where each production has a **LHS (non-terminal)** and **RHS (productions separated by** '|').

- **Tokenization & Processing:** String operations (e.g., `strtok`) are used to split and process production rules.

- **Transformation Storage:** Modified rules are stored dynamically, ensuring correctness.

4. Sample Input & Output

5. Challenges & Learnings

5.1 Challenges:

- Handling multiple left-factored cases correctly without conflicts.
- Ensuring dynamically allocated memory is managed properly.
- Maintaining a clear structure in parsing complex CFG transformations.
- Handling indirect left recursion required careful substitution before direct recursion elimination.

5.2 Learnings:

- Efficient grammar transformation improves parsing efficiency.
- Unique non-terminals prevent conflicts in left factoring.
- Understanding CFG transformations is crucial for compiler design.