LAB SESSION 5: ELIMINATION OF LEFT-RECURSION IN CONTEXT FREE GRAMMAR

DATE:

AIM: To implement a program in python to eliminate left recursion from a given CFG.

PROBLEM DEFINITION: Develop a python program to eliminate left-recursion from a given Context Free Grammar. Accept the Grammar from a file.

THEORY: Context-free grammars play a vital role in compilers and programming languages. CFGs are used to define how valid programs should be organized. We use recursion in parsing, whereby examining a sequence of tokens to its grammatical structure, is greatly dependent on these grammars.

Left Recursion

A context-free grammar is said to be left recursive if it contains a production rule where the non-terminal on the left-hand side of the rule also appears as the first symbol on the right-hand side. In other words, the grammar is trying to define a non-terminal in terms of itself, creating a recursive loop.

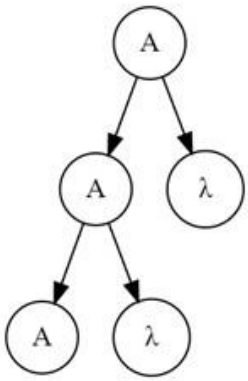
This can be represented formally as: $A \rightarrow A\alpha | \beta$

Where: A is a non-terminal symbol.

α represents a sequence of terminals and/or non-terminals.

β represents another sequence of terminals and/or non-terminals.

The most important part here is the presence of A on both sides of the production rule, with it appearing first on the right-hand side. To visualize this, consider the following parse tree –



It is generated by a left-recursive grammar. As the grammar recursively expands the non-terminal 'A' on the left, the tree grows indefinitely downwards on the left side. This continuous expansion makes it unsuitable for top-down parsing, as the parser could get trapped in an infinite loop, trying to expand 'A' repeatedly.

Problem of Left Recursion for Top-Down Parsing

The top-down parsing works by starting with the start symbol of the grammar and attempting to derive the input string by applying production rules. When encountering a left-recursive rule, the parser keeps expanding the same non-terminal, leading to an infinite loop. This inability to handle left recursion directly is a significant drawback of top-down parsing methods.

Eliminating Left Recursion

To solve this we can eliminate immediate left recursion from a grammar without altering the language it generates. The general approach involves introducing a new non-terminal and rewriting the recursive rules. Consider a simplified arithmetic expression grammar –

$$E \rightarrow E+T|T$$

$$T \rightarrow T*F|F$$

$$F \rightarrow (E)|id$$

Eliminating Left Recursion in `E`: $E \rightarrow TE'$

$$E' \rightarrow +TE'|\varepsilon$$

Eliminating Left Recursion in `T`:

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' | \varepsilon$$

The final transformed grammar, free from left recursion, becomes –

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' | \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' | \varepsilon$$

$$F \rightarrow (E) | id$$

PROGRAM: (python code)

```
from collections import defaultdict

def eliminate_left_recursion(grammar):
    print("\n" + "="*80)
    print("LEFT RECURSION ELIMINATION ALGORITHM")
    print("="*80)

    print("\nFORMULA:")
    print("If we have: A → Aα₁ | Aα₂ | ... | Aαₘ | β₁ | β₂ | ... | βₙ")
    print("Transform to:")
    print("A → β₁A' | β₂A' | ... | βₙA'")
    print("A' → α₁A' | α₂A' | ... | αₘA' | ε")
    print("\nWhere:")
    print("- α represents the part after A in left-recursive productions")
    print("- β represents non-left-recursive productions")
    print("- A' is a new non-terminal")
    print("- ε is epsilon (empty string)")

    new_grammar = defaultdict(list)

    for non_terminal in grammar:
        print(f"\n" + "-"*60)
        print(f"PROCESSING NON-TERMINAL: {non_terminal}")
        print("-"*60)

        alpha = []  # left recursive parts (α)
        beta = []   # non-left recursive parts (β)
```

```
    print(f"\nAnalyzing productions for {non_terminal}:")
    for i, production in enumerate(grammar[non_terminal]):
      prod_str = " ".join(production)
      if production[0] == non_terminal:
        # A → Aα
        alpha_part = production[1:]
        alpha.append(alpha_part)
        alpha_str = " ".join(alpha_part) if alpha_part else "ε"
        print(f"  {non_terminal} → {non_terminal}{alpha_str} [LEFT RECURSIVE - α =
'{alpha_str}']")
      else:
        # A → β
        beta.append(production)
        beta_str = " ".join(production)
            print(f"  {non_terminal} → {beta_str} [NON-LEFT RECURSIVE - β =
'{beta_str}']")

    print(f"\nCollected α parts: {[' '.join(a) if a else 'ε' for a in alpha]}")
    print(f"Collected β parts: {[' '.join(b) for b in beta]}")

    if alpha:
      print(f"\n🔄 LEFT RECURSION DETECTED! Applying transformation...")
      new_nt = non_terminal + "'"
      print(f"Creating new non-terminal: {new_nt}")

      print(f"\nStep 1: Transform {non_terminal} productions")
      print(f"Formula: {non_terminal} → β₁{new_nt} | β₂{new_nt} | ... | βₙ{new_nt}")
      for i, b in enumerate(beta):
        new_prod = b + [new_nt]
        new_grammar[non_terminal].append(new_prod)
        beta_str = " ".join(b)
        new_prod_str = " ".join(new_prod)
        print(f"  β{i+1} = '{beta_str}' → {non_terminal} → {new_prod_str}")

      print(f"\nStep 2: Create {new_nt} productions")
      print(f"Formula: {new_nt} → α₁{new_nt} | α₂{new_nt} | ... | αₘ{new_nt} | ε")
      for i, a in enumerate(alpha):
        new_prod = a + [new_nt]
        new_grammar[new_nt].append(new_prod)
        alpha_str = " ".join(a) if a else "ε"
        new_prod_str = " ".join(new_prod)
```

```
            print(f"  α{i+1} = '{alpha_str}' → {new_nt} → {new_prod_str}")

        # Add epsilon production
        new_grammar[new_nt].append(["ε"])
        print(f"  Adding epsilon: {new_nt} → ε")

        print(f"\n✅ Transformation complete for {non_terminal}")
      else:
        print(f"\n✅ No left recursion found for {non_terminal}")
        print(f"Keeping original productions unchanged")
        new_grammar[non_terminal].extend(beta)

  return new_grammar

def read_grammar_from_file(filename):
  grammar = defaultdict(list)
  with open(filename, "r") as f:
    for line in f:
      if "->" in line:
        lhs, rhs = line.strip().split("->")
        lhs = lhs.strip()
        productions = rhs.strip().split("|")
        for prod in productions:
          grammar[lhs].append(prod.strip().split())
  return grammar

def print_grammar(grammar):
  for nt in grammar:
    rhs = [" ".join(p) for p in grammar[nt]]
    print(f"  {nt} → {' | '.join(rhs)}")
  if not grammar:
    print("  (No productions)")
  print()

# ---------------- MAIN ----------------
def main():
  print("🚀 LEFT RECURSION ELIMINATION DEMONSTRATION")
  print("="*80)

  filename = "grammar.txt"
  print(f"\n📖 Reading grammar from: {filename}")
```

```
    grammar = read_grammar_from_file(filename)

    print("\n📋 ORIGINAL GRAMMAR:")
    print("="*40)
    print_grammar(grammar)

    print(f"\n🔍 CHECKING FOR LEFT RECURSION:")
    print("Left recursion occurs when: A → Aα (A appears as first symbol on RHS)")

    updated_grammar = eliminate_left_recursion(grammar)

    print(f"\n" + "="*80)
    print("🎯 FINAL RESULT - GRAMMAR AFTER ELIMINATING LEFT RECURSION:")
    print("="*80)
    print_grammar(updated_grammar)

    print(f"\n📚 SUMMARY:")
    print("- Left recursion has been successfully eliminated")
    print("- New non-terminals with ' (prime) have been introduced")
    print("- The grammar is now suitable for top-down parsing")
    print("- Epsilon (ε) productions handle the recursive nature")

if __name__ == "__main__":
    main()
```

grammar.txt:

```
A -> A a | A b | c | d
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

OUTPUT:

```
PS C:\Users\Joseph\Desktop\compiler design\expt5> python expt5a.py
🚀 LEFT RECURSION ELIMINATION DEMONSTRATION
================================================================================
```

📖 Reading grammar from: grammar.txt

📋 ORIGINAL GRAMMAR:
==========
A → A a | A b | c | d
E → E + T | T
T → T * F | F
F → (E) | id

🔍 CHECKING FOR LEFT RECURSION:
Left recursion occurs when: A → Aα (A appears as first symbol on RHS)

==========
==========
LEFT RECURSION ELIMINATION ALGORITHM
==========
==========

FORMULA:
If we have: A → $A\alpha_1$ | $A\alpha_2$ | ... | $A\alpha_m$ | β_1 | β_2 | ... | β_n
Transform to:
A → $\beta_1 A'$ | $\beta_2 A'$ | ... | $\beta_n A'$
A' → $\alpha_1 A'$ | $\alpha_2 A'$ | ... | $\alpha_m A'$ | ε

Where:
- α represents the part after A in left-recursive productions
- β represents non-left-recursive productions
- A' is a new non-terminal
- ε is epsilon (empty string)

PROCESSING NON-TERMINAL: A

Analyzing productions for A:
A → Aa [LEFT RECURSIVE - α = 'a']
A → Ab [LEFT RECURSIVE - α = 'b']
A → c [NON-LEFT RECURSIVE - β = 'c']
A → d [NON-LEFT RECURSIVE - β = 'd']

Collected α parts: ['a', 'b']

Collected β parts: ['c', 'd']

🔁 LEFT RECURSION DETECTED! Applying transformation...
Creating new non-terminal: A'

Step 1: Transform A productions
Formula: $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
β1 = 'c' → A → c A'
β2 = 'd' → A → d A'

Step 2: Create A' productions
Formula: $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$
α1 = 'a' → A' → a A'
α2 = 'b' → A' → b A'
Adding epsilon: A' → ε

✅ Transformation complete for A

PROCESSING NON-TERMINAL: E

Analyzing productions for E:
E → E+ T [LEFT RECURSIVE - α = '+ T']
E → T [NON-LEFT RECURSIVE - β = 'T']

Collected α parts: ['+ T']
Collected β parts: ['T']

🔁 LEFT RECURSION DETECTED! Applying transformation...
Creating new non-terminal: E'

Step 1: Transform E productions
Formula: $E \rightarrow \beta_1 E' \mid \beta_2 E' \mid \ldots \mid \beta_n E'$
β1 = 'T' → E → T E'

Step 2: Create E' productions
Formula: $E' \rightarrow \alpha_1 E' \mid \alpha_2 E' \mid \ldots \mid \alpha_m E' \mid \varepsilon$
α1 = '+ T' → E' → + T E'
Adding epsilon: E' → ε

✅ Transformation complete for E

PROCESSING NON-TERMINAL: T

Analyzing productions for T:
T → T* F [LEFT RECURSIVE - α = '* F']
T → F [NON-LEFT RECURSIVE - β = 'F']

Collected α parts: ['* F']
Collected β parts: ['F']

🔄 LEFT RECURSION DETECTED! Applying transformation...
Creating new non-terminal: T'

Step 1: Transform T productions
Formula: $T \rightarrow \beta_1 T' \mid \beta_2 T' \mid \ldots \mid \beta_n T'$
β1 = 'F' → T → F T'

Step 2: Create T' productions
Formula: $T' \rightarrow \alpha_1 T' \mid \alpha_2 T' \mid \ldots \mid \alpha_m T' \mid \varepsilon$
α1 = '* F' → T' → * F T'
Adding epsilon: T' → ε

✅ Transformation complete for T

PROCESSING NON-TERMINAL: F

Analyzing productions for F:
F → (E) [NON-LEFT RECURSIVE - β = '(E)']
F → id [NON-LEFT RECURSIVE - β = 'id']

Collected α parts: []
Collected β parts: ['(E)', 'id']

✅ No left recursion found for F
Keeping original productions unchanged

==========
==========
🎯 FINAL RESULT - GRAMMAR AFTER ELIMINATING LEFT RECURSION:
==========
==========
A → c A' | d A'
A' → a A' | b A' | ε
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

📚 SUMMARY:
- Left recursion has been successfully eliminated
- New non-terminals with ' (prime) have been introduced
- The grammar is now suitable for top-down parsing
- Epsilon (ε) productions handle the recursive nature
PS C:\Users\Joseph\Desktop\compiler design\expt5>

CONCLUSION:

This experiment demonstrated the essential process of eliminating left-recursion from context-free grammars to enable top-down parsing. We learned that left-recursion causes infinite loops in recursive descent parsers and must be systematically removed using the transformation formula ($A \rightarrow A\alpha \mid \beta$ becomes $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$). The practical implementation revealed how to algorithmically identify recursive and non-recursive parts, create new non-terminals, and handle epsilon productions while preserving language semantics. This foundational knowledge is crucial for building efficient parsers in compiler tools, IDEs, and language processors, bridging theoretical grammar concepts with practical compiler implementation techniques.