# LAB SESSION 4: FIRST AND FOLLOW OF A CONTEXT FREE GRAMMAR

**AIM**: To implement a program in python to compute the FIRST and FOLLOW of a given CFG.

**PROBLEM DEFINITION:** Develop a python program to compute the FIRST and FOLLOW of a given Context Free Grammar. Accept the Grammar from a file.

**THEORY:** The First and Follow sets are important in syntax analysis, mainly in parsing. All sets help in making predictive parsers and are integral to identifying how a given grammar can be parsed effectively.

The First Set for a non-terminal symbol represents all possible terminals that can appear at the beginning of any string derived from that non-terminal.
The follow set contains terminals that can appear immediately after a non-terminal in the derivation of the grammar.
All meaning of First and Follow sets is it allows to define which production rules are implemented in different parsing moment.

**What is the First Set?**

The First set of non-terminal in grammar tells us all the possible terminals that can appear at the beginning of any string derived from that non terminal. it gives us a way to know what symbol might come first if we start with a particular non-terminal. For example, if we have a rule like A -> aB | ε, then the First set of A includes the terminal a, as well as ε, which represents the possibility of deriving an empty string. Calculating the First set helps us predict which rules to apply during parsing by knowing which symbols can appear first in a derivation.

**What is a Follow Set?**

The Follow set of a non-terminal contains all terminals that can immediately follow it in any derivation of the grammar. This means that if a non-terminal appears in the middle of a sentence, the Follow set tells us what symbol might come right after it. For example, in a rule like S -> AB, the Follow set of A might include anything that can follow B, depending on other

rules in the grammar. Follow sets are especially useful for parsers when making decisions about which rule to apply next and ensuring that we're interpreting the structure of a sentence correctly, especially for LL(1) parsers where we only look one symbol ahead.

The follow set of the start symbol will always contain "$". Now the calculation of Follow falls under three broad cases:

If a Non-Terminal on the R.H.S. of any production is followed immediately by a Terminal then it can immediately be included in the Follow set of that Non-Terminal.

If a Non-Terminal on the R.H.S. of any production is followed immediately by a Non-Terminal, then the First Set of that new Non-Terminal gets included on the follow set of our original Non-Terminal. In case encountered an epsilon i.e. " # " then, move on to the next symbol in the production.

Note: "#" is never included in the Follow set of any Non-Terminal.

If reached the end of a production while calculating follow, then the Follow set of that non-terminal will include the Follow set of the Non-Terminal on the L.H.S. of that production. This can easily be implemented by recursion.

**PROGRAM:**

```
from collections import defaultdict

class FirstFollowCalculator:
    def __init__(self, grammar):
        self.grammar = grammar
        self.first_sets = {}
        self.follow_sets = defaultdict(set)
        self.first_in_progress = set()  # To handle left recursion
        self.follow_in_progress = set()  # To handle left recursion
        self.computation_steps = []

    def log_step(self, message):
        """Log computation steps"""
        self.computation_steps.append(message)
        print(message)

    def compute_first(self, symbol):
```

```
    """Compute FIRST set with memoization and detailed steps"""

    # Check if already computed (memoization)
    if symbol in self.first_sets:
      self.log_step(f"✓ FIRST({symbol}) already computed: {self.first_sets[symbol]}")
       return self.first_sets[symbol]

    # Check for left recursion
    if symbol in self.first_in_progress:
       self.log_step(f"⚠ Left recursion detected for {symbol}, returning empty set")
       return set()

    self.first_in_progress.add(symbol)
    self.log_step(f"\n🔄 Computing FIRST({symbol})...")

    first = set()

    # Terminal case
    if symbol not in self.grammar:
       first.add(symbol)
       self.log_step(f"  📝 {symbol} is terminal → FIRST({symbol}) = {{{symbol}}}")
    else:
       # Non-terminal case
              self.log_step(f"  📝 {symbol} is non-terminal with productions: {self.grammar[symbol]}")

       for i, production in enumerate(self.grammar[symbol]):
          self.log_step(f"    Production {i+1}: {symbol} → {' '.join(production)}")

          if production == ["ε"]:
             first.add("ε")
             self.log_step(f"      ✅ Added ε to FIRST({symbol})")
          else:
             # Process each symbol in the production
             all_have_epsilon = True
             for j, char in enumerate(production):
                self.log_step(f"      🔍 Processing symbol {j+1}: {char}")

                # Recursive call with memoization
                first_char = self.compute_first(char)
```

```
                # Add non-epsilon symbols
                before_size = len(first)
                first |= (first_char - {"ε"})
                new_symbols = first - (first - (first_char - {"ε"})) if before_size < len(first)
else set()

                if new_symbols:
                    self.log_step(f"      ➕ Added {new_symbols} from FIRST({char}) to
FIRST({symbol})")

                # Check if epsilon is in FIRST(char)
                if "ε" not in first_char:
                  self.log_step(f"      🛑 {char} doesn't derive ε, stopping here")
                  all_have_epsilon = False
                  break
                else:
                  self.log_step(f"      ⏭️ {char} can derive ε, continuing...")

            # If all symbols can derive epsilon, add epsilon
            if all_have_epsilon:
              first.add("ε")
             self.log_step(f"    ✅ All symbols derive ε → Added ε to FIRST({symbol})")

    # Store result (memoization)
    self.first_sets[symbol] = first
    self.first_in_progress.remove(symbol)
    self.log_step(f" ✅ Final FIRST({symbol}) = {first}")

    return first

  def compute_follow(self, symbol, start_symbol):
    """Compute FOLLOW set with memoization and detailed steps"""

    # Initialize if first time
    if symbol not in self.follow_sets:
      self.follow_sets[symbol] = set()

    # Check for left recursion
    if symbol in self.follow_in_progress:
      self.log_step(f"⚠️ FOLLOW recursion detected for {symbol}, returning current
set")
```

```
        return self.follow_sets[symbol]

    self.follow_in_progress.add(symbol)
    self.log_step(f"\n🔄 Computing FOLLOW({symbol})...")

    initial_set = self.follow_sets[symbol].copy()

    # Start symbol gets $
    if symbol == start_symbol and "$" not in self.follow_sets[symbol]:
        self.follow_sets[symbol].add("$")
      self.log_step(f"  📝 {symbol} is start symbol → Added $ to FOLLOW({symbol})")

    # Look for symbol in all productions
    for lhs in self.grammar:
        for prod_num, production in enumerate(self.grammar[lhs]):
            self.log_step(f"  🔍 Checking: {lhs} → {' '.join(production)}")

            for i, char in enumerate(production):
                if char == symbol:
                    self.log_step(f"    ✓ Found {symbol} at position {i}")

                    # Case 1: There's a symbol after current symbol
                    if i + 1 < len(production):
                        next_symbol = production[i + 1]
                        self.log_step(f"      📍 Next symbol: {next_symbol}")

                        # Get FIRST of next symbol (might trigger computation)
                        if next_symbol not in self.first_sets:
                            self.compute_first(next_symbol)

                        next_first = self.first_sets[next_symbol]
                        before_size = len(self.follow_sets[symbol])
                        self.follow_sets[symbol] |= (next_first - {"ε"})

                        if len(self.follow_sets[symbol]) > before_size:
                            added = (next_first - {"ε"})
                            self.log_step(f"      ➕ Added {added} from FIRST({next_symbol}) to
FOLLOW({symbol})")

                        # If FIRST(next_symbol) contains ε, add FOLLOW(lhs)
                        if "ε" in next_first:
```

```
                        self.log_step(f"        ⚠ {next_symbol} can derive ε → Need FOLLOW({lhs})")
                    if lhs != symbol:  # Avoid infinite recursion
                        before_size = len(self.follow_sets[symbol])
                        follow_lhs = self.compute_follow(lhs, start_symbol)
                        self.follow_sets[symbol] |= follow_lhs
                        if len(self.follow_sets[symbol]) > before_size:
                            added = follow_lhs
                            self.log_step(f"        ➕ Added {added} from FOLLOW({lhs}) to FOLLOW({symbol})")

                # Case 2: Symbol is at the end of production
                else:
                    self.log_step(f"        📍 {symbol} is at end of production")
                    if lhs != symbol:  # Avoid infinite recursion
                        self.log_step(f"        ⚠ Need FOLLOW({lhs})")
                        before_size = len(self.follow_sets[symbol])
                        follow_lhs = self.compute_follow(lhs, start_symbol)
                        self.follow_sets[symbol] |= follow_lhs
                        if len(self.follow_sets[symbol]) > before_size:
                            added = follow_lhs
                            self.log_step(f"        ➕ Added {added} from FOLLOW({lhs}) to FOLLOW({symbol})")

        self.follow_in_progress.remove(symbol)

        # Check if anything was added
        if self.follow_sets[symbol] != initial_set:
            self.log_step(f"    ✅ Updated FOLLOW({symbol}) = {self.follow_sets[symbol]}")
        else:
            self.log_step(f"    ✅ No changes to FOLLOW({symbol}) = {self.follow_sets[symbol]}")

        return self.follow_sets[symbol]

def main():
    grammar = defaultdict(list)
    filename = "grammar.txt"

    # Read grammar from file
    print("📖 Reading grammar from file...")
```

```
    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:
                    prod_symbols = prod.strip().split()
                        prod_symbols = ["ε" if sym.lower() == "epsilon" else sym for sym in
prod_symbols]
                    grammar[lhs].append(prod_symbols)

    print(f"📋 Grammar loaded:")
    for nt in grammar:
        for i, prod in enumerate(grammar[nt]):
            print(f"  {nt} → {' '.join(prod)}")

    # Initialize calculator
    calculator = FirstFollowCalculator(grammar)
    start_symbol = list(grammar.keys())[0]

    # Compute FIRST sets
    print("\n" + "="*60)
    print("🎯 COMPUTING FIRST SETS WITH MEMOIZATION")
    print("="*60)

    for non_terminal in grammar:
        print(f"\n{'='*20} FIRST({non_terminal}) {'='*20}")
        calculator.compute_first(non_terminal)

    # Compute FOLLOW sets
    print("\n" + "="*60)
    print("🎯 COMPUTING FOLLOW SETS WITH MEMOIZATION")
    print("="*60)

    for non_terminal in grammar:
        print(f"\n{'='*20} FOLLOW({non_terminal}) {'='*20}")
        calculator.compute_follow(non_terminal, start_symbol)

    # Final results
    print("\n" + "="*60)
```

```
    print("📊 FINAL RESULTS")
    print("="*60)

    print("\n🔤 FIRST SETS:")
    for nt in grammar:
        print(f"  FIRST({nt}) = {calculator.first_sets[nt]}")

    print("\n🔤 FOLLOW SETS:")
    for nt in grammar:
        print(f"  FOLLOW({nt}) = {calculator.follow_sets[nt]}")

    print("\n📈 COMPUTATION STATISTICS:")
      print(f"  Total FIRST computations avoided by memoization: {len([s for s in
calculator.computation_steps if 'already computed' in s])}")
    print(f"  Total computation steps logged: {len(calculator.computation_steps)}")

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

**grammar.txt:**

```
E -> T X
X -> + T X | epsilon
T -> F Y
Y -> * F Y | epsilon
F -> ( E ) | id
```

**OUTPUT:**

```
PS C:\Users\Joseph\Desktop\compiler design> cd expt4
PS C:\Users\Joseph\Desktop\compiler design\expt4> python expt4a_optimized.py
📖 Reading grammar from file...
📋 Grammar loaded:
  E → T X
  X → + T X
  X → ε
  T → F Y
  Y → * F Y
  Y → ε
  F → ( E )
  F → id
```

```
============================================================
🎯 COMPUTING FIRST SETS WITH MEMOIZATION
============================================================

==================== FIRST(E) ====================

🔄 Computing FIRST(E)...
  📝 E is non-terminal with productions: [['T', 'X']]
  Production 1: E → T X
    🔍 Processing symbol 1: T

🔄 Computing FIRST(T)...
  📝 T is non-terminal with productions: [['F', 'Y']]
  Production 1: T → F Y
    🔍 Processing symbol 1: F

🔄 Computing FIRST(F)...
  📝 F is non-terminal with productions: [['(', 'E', ')'], ['id']]
  Production 1: F → ( E )
    🔍 Processing symbol 1: (

🔄 Computing FIRST(()...
  📝 ( is terminal → FIRST(() = {(}
  ✅ Final FIRST(() = {'('}
    ➕ Added {'('} from FIRST(() to FIRST(F)
    🛑 ( doesn't derive ε, stopping here
  Production 2: F → id
    🔍 Processing symbol 1: id

🔄 Computing FIRST(id)...
  📝 id is terminal → FIRST(id) = {id}
  ✅ Final FIRST(id) = {'id'}
    ➕ Added {'id'} from FIRST(id) to FIRST(F)
    🛑 id doesn't derive ε, stopping here
  ✅ Final FIRST(F) = {'(', 'id'}
    ➕ Added {'(', 'id'} from FIRST(F) to FIRST(T)
    🛑 F doesn't derive ε, stopping here
  ✅ Final FIRST(T) = {'(', 'id'}
```

➕ Added {'(', 'id'} from FIRST(T) to FIRST(E)
🛑 T doesn't derive ε, stopping here
✅ Final FIRST(E) = {'(', 'id'}

==========  FIRST(X)  ==========

🔄 Computing FIRST(X)...
📝 X is non-terminal with productions: [['+', 'T', 'X'], ['ε']]
Production 1: X → + T X
🔍 Processing symbol 1: +

🔄 Computing FIRST(+)...
📝 + is terminal → FIRST(+) = {+}
✅ Final FIRST(+) = {'+'}
➕ Added {'+'} from FIRST(+) to FIRST(X)
🛑 + doesn't derive ε, stopping here
Production 2: X → ε
✅ Added ε to FIRST(X)
✅ Final FIRST(X) = {'+', 'ε'}

==========  FIRST(T)  ==========
✓ FIRST(T) already computed: {'(', 'id'}

==========  FIRST(Y)  ==========

🔄 Computing FIRST(Y)...
📝 Y is non-terminal with productions: [['*', 'F', 'Y'], ['ε']]
Production 1: Y → * F Y
🔍 Processing symbol 1: *

🔄 Computing FIRST(*)...
📝 * is terminal → FIRST(*) = {*}
✅ Final FIRST(*) = {'*'}
➕ Added {'*'} from FIRST(*) to FIRST(Y)
🛑 * doesn't derive ε, stopping here
Production 2: Y → ε
✅ Added ε to FIRST(Y)
✅ Final FIRST(Y) = {'*', 'ε'}

```
==================== FIRST(F) ====================
✓ FIRST(F) already computed: {'(', 'id'}

============================================================
🎯 COMPUTING FOLLOW SETS WITH MEMOIZATION
============================================================

==================== FOLLOW(E) ====================

🔄 Computing FOLLOW(E)...
  📝 E is start symbol → Added $ to FOLLOW(E)
  🔍 Checking: E → T X
  🔍 Checking: X → + T X
  🔍 Checking: X → ε
  🔍 Checking: T → F Y
  🔍 Checking: Y → * F Y
  🔍 Checking: Y → ε
  🔍 Checking: F → ( E )
   ✓ Found E at position 1
     📍 Next symbol: )

🔄 Computing FIRST())...
  📝 ) is terminal → FIRST()) = {)}
  ✅ Final FIRST()) = {')'}
     ➕ Added {')'} from FIRST()) to FOLLOW(E)
  🔍 Checking: F → id
  ✅ Updated FOLLOW(E) = {'$', ')'}

==================== FOLLOW(X) ====================

🔄 Computing FOLLOW(X)...
  🔍 Checking: E → T X
   ✓ Found X at position 1
     📍 X is at end of production
     ⚠ Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
  🔍 Checking: E → T X
```

🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
➕ Added {'$', ')'} from FOLLOW(E) to FOLLOW(X)
🔍 Checking: X → + T X
✓ Found X at position 2
📍 X is at end of production
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ Updated FOLLOW(X) = {'$', ')'}

========== FOLLOW(T) ==========

🔄 Computing FOLLOW(T)...
🔍 Checking: E → T X
✓ Found T at position 0
📍 Next symbol: X
➕ Added {'+'} from FIRST(X) to FOLLOW(T)
⚠ X can derive ε → Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y

🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
➕ Added {'$', ')'} from FOLLOW(E) to FOLLOW(T)
🔍 Checking: X → + T X
✓ Found T at position 1
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(X)

🔄 Computing FOLLOW(X)...
🔍 Checking: E → T X
✓ Found X at position 1
📍 X is at end of production
⚠ Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found X at position 2
📍 X is at end of production
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε

🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(X) = {'$', ')'}
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ Updated FOLLOW(T) = {'$', '+', ')'}

========== FOLLOW(Y) ==========

🔄 Computing FOLLOW(Y)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
✓ Found Y at position 1
📍 Y is at end of production
⚠ Need FOLLOW(T)

🔄 Computing FOLLOW(T)...
🔍 Checking: E → T X
✓ Found T at position 0
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )

🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found T at position 1
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(X)

🔄 Computing FOLLOW(X)...
🔍 Checking: E → T X
✓ Found X at position 1
📍 X is at end of production
⚠ Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found X at position 2
📍 X is at end of production
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(X) = {'$', ')'}
🔍 Checking: X → ε
🔍 Checking: T → F Y

🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(T) = {'$', '+', ')'}
➕ Added {'$', '+', ')'} from FOLLOW(T) to FOLLOW(Y)
🔍 Checking: Y → * F Y
✓ Found Y at position 2
📍 Y is at end of production
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ Updated FOLLOW(Y) = {'$', '+', ')'}

========== FOLLOW(F) ==========

🔄 Computing FOLLOW(F)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
✓ Found F at position 0
📍 Next symbol: Y
➕ Added {'*'} from FIRST(Y) to FOLLOW(F)
⚠ Y can derive ε → Need FOLLOW(T)

🔄 Computing FOLLOW(T)...
🔍 Checking: E → T X
✓ Found T at position 0
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y

🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found T at position 1
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(X)

🔄 Computing FOLLOW(X)...
🔍 Checking: E → T X
✓ Found X at position 1
📍 X is at end of production
⚠ Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found X at position 2
📍 X is at end of production
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )

🔍 Checking: F → id
✅ No changes to FOLLOW(X) = {'$', ')'}
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(T) = {'$', '+', ')'}
➕ Added {'$', '+', ')'} from FOLLOW(T) to FOLLOW(F)
🔍 Checking: Y → * F Y
✓ Found F at position 1
📍 Next symbol: Y
⚠ Y can derive ε → Need FOLLOW(Y)

🔄 Computing FOLLOW(Y)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
✓ Found Y at position 1
📍 Y is at end of production
⚠ Need FOLLOW(T)

🔄 Computing FOLLOW(T)...
🔍 Checking: E → T X
✓ Found T at position 0
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε

🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found T at position 1
📍 Next symbol: X
⚠ X can derive ε → Need FOLLOW(X)

🔄 Computing FOLLOW(X)...
🔍 Checking: E → T X
✓ Found X at position 1
📍 X is at end of production
⚠ Need FOLLOW(E)

🔄 Computing FOLLOW(E)...
🔍 Checking: E → T X
🔍 Checking: X → + T X
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
✓ Found E at position 1
📍 Next symbol: )
🔍 Checking: F → id
✅ No changes to FOLLOW(E) = {'$', ')'}
🔍 Checking: X → + T X
✓ Found X at position 2
📍 X is at end of production
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id

✅ No changes to FOLLOW(X) = {'$', ')'}
🔍 Checking: X → ε
🔍 Checking: T → F Y
🔍 Checking: Y → * F Y
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(T) = {'$', '+', ')'}
🔍 Checking: Y → * F Y
✓ Found Y at position 2
📍 Y is at end of production
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ No changes to FOLLOW(Y) = {'$', '+', ')'}
🔍 Checking: Y → ε
🔍 Checking: F → ( E )
🔍 Checking: F → id
✅ Updated FOLLOW(F) = {'$', '+', ')', '*'}

==========
📊 FINAL RESULTS
==========

🔤 FIRST SETS:
FIRST(E) = {'(', 'id'}
FIRST(X) = {'+', 'ε'}
FIRST(T) = {'(', 'id'}
FIRST(Y) = {'*', 'ε'}
FIRST(F) = {'(', 'id'}

🔤 FOLLOW SETS:
FOLLOW(E) = {'$', ')'}
FOLLOW(X) = {'$', ')'}
FOLLOW(T) = {'$', '+', ')'}
FOLLOW(Y) = {'$', '+', ')'}
FOLLOW(F) = {'$', '+', ')', '*'}

📈 COMPUTATION STATISTICS:

```
  Total FIRST computations avoided by memoization: 2
  Total computation steps logged: 373
PS C:\Users\Joseph\Desktop\compiler design\expt4>
```

**CONCLUSION:**

We successfully implemented FIRST and FOLLOW set computation algorithms for context-free grammars, demonstrating essential concepts in parser construction. The program correctly computed sets for an arithmetic expression grammar, properly handling epsilon productions and recursive dependencies. Key learnings include understanding how FIRST sets determine valid starting tokens, how FOLLOW sets identify lookahead symbols, and how these algorithms form the foundation for LL (1) predictive parsing. This experiment bridges theoretical compiler concepts with practical implementation, providing crucial knowledge for syntax analysis and parser design.