DAYANANDA SAGAR COLLEGE OF ARTS, SCIENCE AND COMMERCE

Shavige Malleshwara Hills, Kumarswamy Layout, Bengaluru – 560078

MASTER OF COMPUTER APPLICATIONS (BU)

PROJECT REPORT

ON

SYNTAX ANALYSIS OF ARITHMETIC EXPRESSIONS USING CONTEXT-FREE GRAMMAR

Submitted To:

Prof. Akshatha

DSCASC

Submitted By:

Neha V(P03CJ24S126062)

Sahana SB (P03CJ24S126084)

Pushpalatha T (P03CJ24S126077)

Veena BB (P03CJ24S126115)

ABSTRACT

This project explores the application of Theory of Computation (TOC) to the design of a syntax analyzer for mathematical expressions. A Python-based program is developed to parse and validate expressions involving arithmetic operations (addition, subtraction, multiplication, division, exponentiation, modulus) and trigonometric functions, demonstrating the use of Context-Free Grammars (CFGs) in defining expression syntax.

TABLE OF CONTENTS

INTRODUCTION

- This project is an investigation into the use of the Theory of Computation (TOC), in the form of Context-Free Grammars (CFGs), in building syntax analyzers.
- TOC gives the formal system for describing the syntax of languages, including programming languages and, as here, arithmetic expressions, and trigonometric notations using CFGs.
- A syntax analyzer (or parser) is a key part of a compiler or interpreter that determines whether the input program or expression conforms to the language's grammar rules.
- By constructing a CFG for arithmetic expressions, we are able to build a system that decides whether an input expression is well-formed based on the rules of arithmetic with addition, subtraction, multiplication, and parentheses.
- This project will entail creating a CFG for these arithmetic operations and writing a simplified syntax analyzer in Python that employs this grammar to check input expressions.
- The implementation will illustrate how theoretical principles of TOC, including grammar rules and parsing, are used in real-world software development for language processing.
- Syntax analysis is essential in constructing more advanced language processing tools, including compilers and interpreters.

Characteristics of TOC in Syntax Analysis:

- Formal Specification of Grammar: TOC CFGs enable an exact and unambiguous specification of the syntactic structure of a language.
- Representation of Hierarchical Structure: The tree-like form obtained from CFG parsing (parse trees) represents the input in a hierarchical form, essential to comprehend the relations between various parts of the expression.
- Foundation for Parsing Algorithms: TOC forms the theoretical foundation for several parsing algorithms (e.g., recursive descent, shift-reduce) employed in syntax analyzers.

TOC AND SYNTAX ANALYSIS

Theory of Computation (TOC) gives us the theoretical basics needed to specify and analyze the syntax of formal languages, which form the basis of constructing syntax analyzers.

Context-Free Grammars (CFGs), a central concept of TOC, are especially well-suited to express the hierarchical nature of programming languages and arithmetic expressions.

A CFG contains a set of production rules used to describe in which ways a non-terminal can be replaced with a terminal (the actual tokens of the language) or an alternative non-terminal.

In syntactic analysis, a CFG defining arithmetic expressions explains how numbers, operators (+, -, *,/,%), trigonometric notations and parentheses must be combined according to the following.

The syntax analysis process is taking an input string (the arithmetic expression) and checking whether it can be derived from the CFG's start symbol using the production rules.

The process of derivation can be represented as a parse tree, which shows the syntactic structure of the expression as per the grammar.

If it's possible to form a parse tree for the input expression using the CFG, then the expression is syntactically correct; otherwise, it's syntactically incorrect

CFG FOR ARITHMETIC EXPRESSIONS

As the first step to construct our little syntax analyzer, we need to define a Context-Free Grammar (CFG) which reflects the syntax of arithmetic expressions involving addition, subtraction, multiplication, and use of parentheses.

One possible CFG to achieve this is as given below:

E -> E + T | E - T | T

T -> T * F | T / F | T % F | F

F -> P ^ F | P

P -> (E) | NUM | FUNC (E)

NUM -> <any number>

FUNC -> sin | cos | tan

Where:

E (Expression): Represents an expression.

- T (Term): Represents a term.
- F (Factor): Represents a factor.
- P (Primary): Represents a primary element (parentheses, number, function).
- NUM: Represents any number.
- FUNC: Represents a trigonometric function.

This grammar specifies the operator precedence (multiplication precedence over addition and subtraction) and the use of parentheses to overrule precedence.

For instance, the term 3 + 5 * (2 - 4) can be generated from this grammar, but not 3 + * 5, since there is no production rule under which a + can follow a * directly.

IMPLEMENTATION AND EXAMPLE

- Python Program for Simple Syntax Analyzer

```
import re
import math

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __repr__(self):
        return f'<{self.type}: {self.value}>'

class MathSyntaxAnalyzer:
    def __init__(self, expression):
        self.expression = expression
        self.input = expression
```

```
        self.tokens = self.tokenize()
        self.pos = 0
        self.lookahead = self.next_token()
        self.functions = {
            'sin': math.sin,
            'cos': math.cos,
            'tan': math.tan
        }

    def tokenize(self):
        tokens = []
        i = 0
        while i < len(self.input):
            char = self.input[i]

            if char.isspace():
                i += 1
                continue

            if re.match(r'\d', char) or char == '.':
                num_str = ''
                while i < len(self.input) and (re.match(r'\d', self.input[i]) or
self.input[i] == '.'):
                    num_str += self.input[i]
                    i += 1
                tokens.append(Token('NUMBER', float(num_str)))
```

```
            continue

        if re.match(r'[a-zA-Z]', char):
            func_str = ''
            while i < len(self.input) and re.match(r'[a-zA-Z0-9]', self.input[i]):
                func_str += self.input[i]
                i += 1
            tokens.append(Token('IDENTIFIER', func_str))
            continue

        if char in '+-*/%^()':
            tokens.append(Token('OPERATOR', char))
            i += 1
            continue

        if char == ',':
            tokens.append(Token('COMMA', char))
            i += 1
            continue

        self.error(f"Invalid character: {char}")
    tokens.append(Token('EOF', None))
    return tokens

def next_token(self):
    if self.pos < len(self.tokens):
```

```
            token = self.tokens[self.pos]
            self.pos += 1
            return token
        return Token('EOF', None)

    def consume(self, expected_type, expected_value=None):
        if self.lookahead.type == expected_type and (expected_value is None or self.lookahead.value == expected_value):
            token = self.lookahead
            self.lookahead = self.next_token()
            return token
        else:
            self.error(f"Expected {expected_type} {expected_value}, but got {self.lookahead}")

    def error(self, message):
        print(f"Syntax Error: {message} at {self.lookahead}")
        exit(1)

    def E(self):
        result = self.T()
        while self.lookahead.type == 'OPERATOR' and self.lookahead.value in ['+', '-']:
            op = self.consume('OPERATOR').value
            right = self.T()
            if op == '+':
                result += right
```

```
            elif op == '-':
                result -= right
        return result

    def T(self):
        result = self.F()
        while self.lookahead.type == 'OPERATOR' and self.lookahead.value in
['*', '/', '%']:
            op = self.consume('OPERATOR').value
            right = self.F()
            if op == '*':
                result *= right
            elif op == '/':
                if right == 0:
                    self.error("Division by zero")
                result /= right
            elif op == '%':
                if right == 0:
                    self.error("Modulus by zero")
                result %= right
        return result

    def F(self):
        result = self.P()
        if self.lookahead.type == 'OPERATOR' and self.lookahead.value == '^':
            self.consume('OPERATOR', '^')
```

```
        right = self.F()
        result = result ** right
    return result

def P(self):
    if self.lookahead.type == 'OPERATOR' and self.lookahead.value == '(':
        self.consume('OPERATOR', '(')
        result = self.E()
        self.consume('OPERATOR', ')')
        return result
    elif self.lookahead.type == 'NUMBER':
        return self.consume('NUMBER').value
    elif self.lookahead.type == 'IDENTIFIER':
        func_name = self.consume('IDENTIFIER').value
        if func_name in self.functions:
            self.consume('OPERATOR', '(')
            arg = self.E()
            self.consume('OPERATOR', ')')
            return self.functions[func_name](arg)
        else:
            self.error(f"Undefined function: {func_name}")
    else:
        self.error("Expected a number, function, or '('")

def parse(self):
    result = self.E()
```

```
        self.consume('EOF')
        return result


if __name__== "__main__":
    expression = input("Enter a mathematical expression:\n")
    analyzer = MathSyntaxAnalyzer(expression)
    try:
        result = analyzer.parse()
        print(f"Result: {result}")
    except Exception as e:
        print(f"Error: {e}")
```

Output

```
= RESTART: C:\Users\DIGITAL -LIBRARY\AppData\Local\Programs\Python\Python313\toc.py
Enter a mathematical expression:
10 + 2 * sin(45) / (1 + cos(60)) - 3 ^ 2 % 4
Result: 44.76200114894897

= RESTART: C:\Users\DIGITAL -LIBRARY\AppData\Local\Programs\Python\Python313\toc.py
Enter a mathematical expression:
2 ^ 3 + 15 % 4 - cos(0) * 5 + 10 / (2 + 1) + sin(90) * 2
Result: 11.12132666053445

= RESTART: C:\Users\DIGITAL -LIBRARY\AppData\Local\Programs\Python\Python313\toc.py
Enter a mathematical expression:
10 + 2 * sin(45) / (1 + cos(60)) - 3 ^ (2 % 4) + sqrt(16) * log(10)
Syntax Error: Undefined function: sqrt at <OPERATOR: (>

= RESTART: C:\Users\DIGITAL -LIBRARY\AppData\Local\Programs\Python\Python313\toc.py
Enter a mathematical expression:
cos(60) * 4 - 1 + 12 % 5 + sqrt(9) + 2 ^ 3 / 4
Syntax Error: Undefined function: sqrt at <OPERATOR: (>

= RESTART: C:\Users\DIGITAL -LIBRARY\AppData\Local\Programs\Python\Python313\toc.py
Enter a mathematical expression:
10 / 2 + 3 * (4 - 1) - 2 ^ (1 + 1) + tan(45) % 3
Result: 11.619775190543862
```

CONCLUSION

The project proved the usage of Context-Free Grammars (CFGs) of the Theory of Computation (TOC) in constructing a basic syntax analyzer for arithmetic expressions.

By specifying a CFG that embodies arithmetic rules involving addition, subtraction, multiplication, and parentheses, we were able to create a simple parser in Python to validate the syntactic correctness of input expressions.

The recursive descent parser that is used here illustrates how production rules of a CFG can be easily mapped to parsing functions.

This exercise underscores the elementary function of TOC in language processing tool design and implementation, for example, for compilers and interpreters, where syntactic correctness checking for input is an essential first step.

Further enhancement might include expanding the grammar to support additional operators, functions, or variables and adding a more comprehensive parsing algorithm to offer more informative error messages and possibly create an abstract syntax tree for subsequent processing.