Asian Institute of Technology

AT70.07

Programming Language and Compilers

Professor: Phan Minh Dung

Calculator Assignment : 2024

Phone Myint Zaw - st124359

Github link of the project

https://github.com/phonemz/compiler-calculator

Grammar of the language

Lexer

The lexer defines the tokens our parser will recognize: NUMBER, PLUS, TIMES, LPAREN (left parenthesis), and RPAREN (right parenthesis). It ignores spaces and tabs (' \t') because they don't affect the meaning of the expressions.

Parser

It defines the precedence of operators: PLUS has higher precedence than TIMES, meaning addition will be evaluated before multiplication.

There are rules for different types of expressions:

expr PLUS expr represents addition.

expr TIMES expr represents multiplication.

LPAREN expr RPAREN handles expressions within parentheses.

NUMBER represents individual numbers.

The error method handles syntax errors, providing meaningful error messages.

Example Expression

For example, given the expression 2 + 1 * 3. The addition operation 2+1 will be evaluated first due to its higher precedence. The result of 2+1 (which is 3) will then be multiplied to 3, resulting in the final calculation of 3*3, resulting 9.

```
def postfix(self, p):
    if isinstance(p, tuple):
        op, left, right = p
        return f"{self.postfix(left)} {self.postfix(right)} {op}"
    else:
        return str(p)

def evaluate(self, p):
    if isinstance(p, tuple):
        op, left, right = p
        if op == "+":
            return self.evaluate(                    right)
        elif op == "*":
            return self.evaluate(left) * self.evaluate(right)
    else:
        return p

def error(self, p):
    if p:
        print("Syntax error at token", p.type)
        self.index += 1
    else:
        print("Syntax error at EOF")
```

(variable) left: Any

```
class CalcParser(Parser):
    tokens = CalcLexer.tokens

    precedence = (
        ("left", TIMES),
        ("left", PLUS),
    )

    def __init__(self):
        self.env = {}

    @_("expr PLUS expr")
    def expr(self, p):
        return ("+", p.expr0, p.expr1)

    @_("expr TIMES expr")
    def expr(self, p):
        return ("*", p.expr0, p.expr1)

    @_("LPAREN expr RPAREN")
    def expr(self, p):
        return p.expr

    @_("NUMBER")
    def expr(self, p):
        return int(p.NUMBER)
```

```
def prefix(self, p):
    if isinstance(p, tuple):
        op, left, right = p
        return f"{op} {self.prefix(left)} {self.prefix(right)}"
    else:
        return str(p)

def postfix(self, p):
    if isinstance(p, tuple):
        op, left, right = p
        return f"{self.postfix(left)} {self.postfix(right)} {op}"
    else:
        return str(p)

def evaluate(self, p):
    if isinstance(p, tuple):
        op, left, right = p
        if op == "+":
            return self.evaluate(left) + self.evaluate(right)
        elif op == "*":
            return self.evaluate(left) * self.evaluate(right)
    else:
        return p

def error(self, p):
    if p:
        print("Syntax error at token", p.type)
        self.index += 1
    else:
        print("Syntax error at EOF")
```

Parser Type

In this project, a custom parser has been implemented. This parser is developed using the Sly library, a Python tool designed for creating lexer and parser systems. It is specifically tailored to parse mathematical expressions, employing recursive descent parsing techniques. The parser effectively handles numerical operations such as addition and multiplication, along with parentheses for grouping expressions. Additionally, it incorporates operator precedence rules and offers functionality for converting expressions into prefix and postfix notations, thereby enhancing its utility in parsing mathematical expressions.

Method of translation

The method of translation involves recursively evaluating the parsed expression to calculate the result.

Prefix notation is generated by traversing the expression tree in a pre-order manner, placing operators before operands.

Conversely, postfix notation is generated by traversing the expression tree in a post-order manner, placing operators after operands. This approach ensures that expressions are correctly represented in both prefix and postfix notations.

The integration of the parser and translation

The integration of the parser and translation in the project occurs seamlessly within the parsing process. After parsing an expression, the result is passed to the translation function, which generates equivalent prefix and postfix expressions based on the parsed structure. This integrated approach ensures that parsed expressions are efficiently converted into alternative notations while preserving the grammar-defined structure and semantics.

How to run the application

Install the required dependencies and run python3 main.py command. A pop up GUI box will appear and you can use the calculator app now.

Conclusion

In conclusion, the development of the simple calculator application involved defining a grammar for arithmetic expressions, implementing a SLR(1) parser for parsing and evaluation, generating prefix and postfix notations, and providing a user-friendly interface for input and output. The project successfully demonstrated the application of parsing techniques in building practical software solutions.