Parser

Kit Lao

April 26th, 2020

Goal

Take the generated tokens and build a parse-tree for handling the priorities of different operations. The purpose of the tree is to make evaluation simpler, while correctly following PEMDAS.

Concepts

Parse Tree

A tree where leaves are numbers and all other nodes are operations. The structure will be structured such that the operations on the lower left side will be evaluated first, followed by lower right side, and lastly the parent.

Grammar Rules

$$Expr \rightarrow Term | Term + Term | Term - Term$$

$$Term \rightarrow Factor | Factor * Factor | Factor / Factor$$

$$Factor \rightarrow Number | +Number | -Number | (Expr)$$

Factors

A number with or without the positive or negative sign, or an expression wrapped in left and right parenthesis. Numbers are terminals. These have the highest priority in evaluation so they must be on the bottom of the parse-tree.

Terms

A sequence of factors separated by a multiplication or division symbol, or just by itself. These have the second highest priority so they are on usually on the middle of the parse-tree, except if the terms are within parenthesis.

Expressions

A sequence of terms separated an addition or subtraction symbol or just the term by itself. These have the lowest priority so they are usually on the top of the parse-tree, except if the expressions are within parenthesis.

Design

Nodes

A node will be created for each operation, each type of sign, and for numbers. All operations have 2 nodes, while the other have 1. Nodes for numbers have type float while the rest can have any type.

Parser_

The entire sequence of tokens is an expression. The sub-problem of an expression is either just a term, addition operation on 2 terms, or subtraction operation on 2 terms. To reduce the expression to its sub-problem, iterate through the tokens to find either an addition or subtraction operation, and build term-parse trees for the left and right hand side of the sequence. If there are not addition or subtraction operation, the entire sequence is already reduced to sub-problem. Terms are recursively parsed the same way as expressions. The difference is that terms must reduce their sub-problems to factors, and the tokens will be iterated through to find multiplication and division operators. Factors are problems either reduced to it's base cases or a sub-problem wrapped around parenthesis. The sub-problem can have all types of operations and parenthesis, so it needs to be solved as an expression.

```
function parser(tokens):
    return expr(tokens)

function expr(tokens):
    operand_x = term(tokens)
    while tokens.op is ADD or SUB:
        tokens = tokens.next
        operand_y = term(tokens)
        operand_x = Node(tokens.op, operand_x, operand_y)
    return operand_x

function term(tokens):
    operand_x = factor1(tokens)
    while tokens.op is MULT or DIV:
        tokens = tokens.next
        operand_y = factor1(tokens)
        operand_x = Node(tokens.op, operand_x, operand_y)
    return operand_x

function factor1(tokens):
    fact = tokens
    if fact.op is NUMBER:
        return factor2(tokens)
    else if fact.op is PLUS or MINUS:
        tokens = tokens.next
        return Node(fact.op, factor1(tokens))
    else if fact.op is L_PAREN:
        tokens = tokens.next
        sub_expr = expr(tokens)
        if tokens.op is not R_PAREN:
            error()
        return sub_expr
    else:
        error()

function factor2(tokens):
    num = tokens
    tokens = tokens.next
    if nums.op is not NUMBER:
        error()
    return Node(num.op, num)
```

Resources

This pdf was very helpful. Also used my knowledge from my compilers course at UCSC.

Testing

Samples

```
>>> 1 + 2 * 5
(1.0 + (2.0 * 5.0))
>>> -5 + 8 * (6 + -1) - 5
(((-5.0) + (8.0 * (6.0 + (-1.0)))) - 5.0)
>>> --4
Exception: Invalid syntax
```

Script

In process of working on testing script.