Project: MU Calculator

CSCI 330: Programming Languages

Objectives

1. Design a solution at various levels to evaluate an expression passed in as a string in a programming language of your choice.
2. Apply the theory of Lexical Analysis (Scanning) to tokenize a string into lexemes representative of operands (numbers) or operators (addition, subtraction, multiplication, division, exponentiation)
3. Apply the theory of Parsing to convert a stream of tokens into an Abstract Syntax Tree representative of an arithmetic expression considering operator precedence and associativity.
4. Evaluate an Abstract Syntax Tree representation of an arithmetic expression to compute the answer

Rules and Guidelines

- You may work in a team with up to three people. All team members must contribute equally to the project. You may also choose to work by yourself or with a pair.
- I will evaluate your program differently based on the language you choose to implement your solution in. Some languages will be given a difficulty score.
- Zero late submissions will be permitted for this assignment. Please observe/check the due date indicated on Autolab.
- You must include comprehensive instructions as to how I build and run your application.

Usage

The application can be a graphical application or a console-based application.

- Upon running your application, the user should be able to enter an expression in some way (either by typing into a prompt or with physical buttons representing every possible character – like an actual calculator).
- There should be some way the user can then calculate the expression's value with the result being displayed. For a graphical application this could be done with an "=", "Calculate" or "Enter" button. For a console/terminal application this could be done by pressing enter.
- If the user enters any invalid symbol or if the expression is malformed, the user should be presented with an error.
- The user should be able to enter another expression to evaluate and display the calculated result.
- Finally, there should be a way for the user to exit or quit the application.

Required Operations / Operands

- **Numbers** can be positive integers or floating-point values.
  - Integers may have any number of leading zeros (e.g. 00001 is equivalent to 1).
  - Floating-point values must have exactly one decimal point and may have any number of digits after the decimal point.
  - For both "integers" and "floating-point" values, they should be stored in a double-precision type.

- **Operations** must be one of:

Description	Symbol	Associativity	Precedence Level
Addition	**+**	Left	0
Subtraction	**-**	Left	0
Multiplication	*****	Left	1
Division	**/**	Left	1
Exponentiation	**^**	Right	2

  - A higher precedence level means the operation should be performed first. For example, given the expression "1+2*3", the multiplication of 2*3 should happen first (rather than the addition of 1+2).

- **Symbols** must be one of:
  - Left parenthesis (
  - Right parenthesis)

  Left and Right Parentheses should also be permitted to force precedence changes. Please note that you must have matching parentheses. For example, (1+2)*3 forces the addition of 1 + 2 to occur before the multiplication.

  Mismatched parentheses (extra parentheses on either side) should result in an error presented to the user. For example ((1+2) would result in an error

- **Constant Variables** must be one of:
  - `pi` representing the floating-point constant value of **`3.14159265358979`**
  - `e` representing the floating-point constant value of **`2.71828182845904`**

- **Whitespace** (extra spaces or tabs) should be ignored

Sample Expressions (and their Answers):

```
1 + 2.5              3.5
2.5 * (3 + 4) / 5    3.5
e ^ (pi - 2)         3.1317521917
pi ^ 2.0             9.8696044011
4 ^ 3 ^ 2            262144
(4 ^ 3) ^ 2          4096
```

Software Design/Architecture

Part 1: Lexical Analyzer

Input: a string (a list of characters)
Output: a list of lexemes

Required Lexemes:

- PLUS + symbol
- MINUS - symbol
- TIMES * symbol
- DIVIDES / symbol
- POWER ^ symbol
- LPAREN (symbol
- RPAREN) symbol
- NUMBER an int/float -- should "hold" the underlying value
- PI represents the "pi" literal
- E represents the "e" literal

Examples:

```
1 + 2            ➔   NUMBER(1.0) ADD NUMBER(2.0)
(pi - 5.2) ^ 4   ➔   LPAREN PI MINUS NUMBER(5.2) RPAREN POWER NUMBER(4)
```

Hints:

- You should be able to scan single character by single character.
- Extracting the lexemes for all operators and parentheses is easy, so do that first.
- Extracting the lexemes for e and pi are probably the next easiest.
- Finally, focus on parsing a number. An integer and floating-point number start the same way (with a sequence of digits [0-9]. Only floating-point values have a "period" while anything else indicates the end of a number.

Visual Representation:

Before Lexical Analysis:

Array of Characters

(	1	2	+	2	.	1	4	)	^	p	i		-		e

After Lexical Analysis:

Stream/Array of Lexemes

LPAREN	NUMBER(12)	PLUS	NUMBER(2.14)	RPAREN	POWER	PI	MINUS	E

Part 2: Expression Parser

Input: a list of lexemes
Output: an Abstract Syntax Tree (expression tree)

Required Grammar:

```
Exp := Exp PLUS Exp
     | Exp MINUS Exp
     | Exp TIMES Exp
     | Exp DIVIDES Exp
     | Exp POWER Exp
     | Term

Term := LPAREN Exp RPAREN
     | Num

Num := NUMBER
     | PI
     | E
```

Required "Tree" Nodes:

- `NumberNode`
  - Should hold the numerical value stored within the NUMBER lexeme
  - You could have PI and E correspond to a number node with its constant value
- `ExpressionNode`
  - Should hold three things:
    - **LHS** – either a `NumberNode` or an `ExpressionNode` that corresponds to the left-hand side of an operation
    - **RHS** – either a `NumberNode` or an `ExpressionNode` that corresponds to the right-hand side of an operation
    - **Operator** – information that contains the operator type (for evaluation)
  - You can have special nodes for each operator type if you want to (e.g. AddNode, SubNode, MulNode, DivNode, ExpNode), but the design is up to you.

Tree Representation:

In an Object-Oriented Language, you could have a **Class Hierarchy** where you have a `TreeNode` class that has `NumberNode` and `ExpressionNode` extend the `TreeNode`. By doing so, `ExpressionNode` can just hold onto a `TreeNode`. You will likely need to introduce support for providing an **evaluate** method that is specialized for each node type.

In a Functional Language, you can use a Discriminated Union (Variant Type) to hold onto the corresponding Node type for the left- and right-hand side. You can even model what we did for the Art lab in OCaml.

Tree Construction:

As you scan the sequence of lexemes from left-to-right, you will need to decide what to do based on an operator's precedence (and compare that to the "previous" operator's precedence. Assume the default precedence to be lower than the "lowest" precedence

Pseudocode:

```
expr (prev_precedence=-1):
    lhs ← term()
    while true
        op ← nextLexeme() // ensure that it's an operator
        curr_precedence ← precedence(op)
        if curr_precedence < prev_precedence
           break
        if association(op) == left_to_right
            rhs ← expr(curr_precedence + 1)
        else
            rhs ← expr(curr_precedence)
        lhs = ExpressionNode (lhs, op, rhs)
    return lhs

term():
    val ← nextLexeme()
    if val is NUMBER
        return NumberNode (val.value)
    else if val is PI
        return NumberNode (3.14159…)
    else if val is E
        return NumberNode (2.71828…)
    else if val is LPAREN
        node ← expr()
        assert (nextLexeme() is RPAREN)
        return node
    else
        // failure – expected number but got something else
```

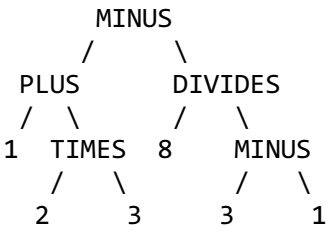
Example 1:

Lexeme Array	Prior Op Precedence	Tree
NUM(1) PLUS NUM(2) MINUS NUM(3)	-1	NumNode(1)
NUM(1) PLUS NUM(2) MINUS NUM(3)	-1	AddNode / \ NumNode(1) *
NUM(1) PLUS NUM(2) MINUS NUM(3)	0	AddNode / \ NumNode(1) NumNode(2)
NUM(1) PLUS NUM(2) MINUS NUM(3)	0	MinusNode / \ AddNode * / \ NumNode(1) NumNode(2)
NUM(1) PLUS NUM(2) MINUS NUM(3)	0	MinusNode / \ AddNode NumNode(3) / \ NumNode(1) NumNode(2)

Example 2:

1+2*3-8/(3-1)

NUM(1) PLUS NUM(2) TIMES NUM(3) MINUS NUM(8) DIVIDES LPAREN NUM(3) MINUS NUM(1) RPAREN



Part 3: Expression Evaluator

Input: an Abstract Syntax Tree (expression tree)
Output: a number

This is actually the easiest part of the assignment!

Assuming you have an expression tree, for all ExpressionNodes, recursively evaluate the LHS and RHS, then apply the operation. Finally, return the result.

Pseudocode:

```
evaluate(node)
    if node is ExpressionNode
        lhs ← evaluate(node.lhs)
        rhs ← evaluate(node.rhs)
        return lhs (node.op) rhs
    else
        return node.value // it's a NumberNode!
```

Evaluation Criteria

This project is worth **200** points total.

- **[50 pts] Lexical Analysis**
  - [10 pts] Token Definitions
  - [20 pts] String Parsing
  - [10 pts] Error Detection of invalid lexemes or extra lexemes at the end
  - [10 pts] Style / Creativity / Design
- **[50 pts] Expression Parsing**
  - [25 pts] Node Definitions
  - [20 pts] Parsing Algorithm Correctness
  - [5 pts] Style / Creativity / Design
- **[25 pts] Expression Evaluation**
  - [20 pts] Correctness
  - [5 pts] Style / Creativity / Design
- [**50 pts] Program Entrypoint**
  - [10 pts] User input / exiting
  - [10 pts] Lexing error reporting
  - [10 pts] Parsing error reporting
  - [10 pts] Calculated answer reporting
  - [5 pts] Style / Creativity / Design
- **[25 pts] Documentation and Directions**
  - [10 pts] Directions on Building / Running your Application
  - [15 pts] Documentation of Classes, Algorithms, etc

Extra Credit Opportunities

Language Selection

Choosing a less familiar language (or one that we've extensively used in this class) will earn you extra points. Whatever score you earn I will multiply it by the multiplicative factor specified. If you want to program in a language which is not listed, please let me know and I will tell you which tier the language falls under.

Tier	Multiplicative Factor	Languages
0	1.00x	Java C++
1	1.05x	Python Javascript PHP C C#
2	1.10x	Ruby Kotlin Scala Swift Go
3	1.15x	OCaml F# Rust Lisp

User Interface

A traditional command-line application is what we are used to for most assignments so far throughout our MU CS student career; however, I am giving everyone an opportunity to develop a graphical user interface-based application. If you choose to create a GUI, I will add additional multiplicative factor:

- **1.00x** for a command line interface
- **1.05x** for a graphical interface that has a text box and a place for an answer to appear
- **1.15x** for a graphical interface that has buttons for every character that can be entered along with a display of user input and a place for the calculator "output"