CS 359 – Programming Paradigms

PEX 2 – Expression Evaluation

Preliminary Submission Due: COB, Lesson M21, Thursday, October 11

Final Submission Due: Taps, Lesson T23, Thursday, October 18

Help Policy:

AUTHORIZED RESOURCES: Any, except another cadet's program.
NOTE:

- Never copy another person's work and submit it as your own.
- Do not jointly create a program unless explicitly allowed.
- You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook).
- **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**

Documentation Policy:

- You must document all help received from any source other than your instructor.
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
- If no help was received on this assignment, the documentation statement must state "NONE."
- If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.
- **Vague documentation statements must be corrected before the assignment will be graded and will result in a 5% deduction on the assignment.**

Turn-in Policies:

- On-time turn-in is at the specific time listed above.
- Late penalties accrue at a rate of 25% per 24-hour period past the on-time turn-in date and time. The late penalty is a cap on the maximum grade that may be awarded for late work.
- There is no early turn-in bonus or extra credit for this assignment.

1. OBJECTIVES

- Expand your range of programming language competence
- Be able to create, test, and debug Scheme functions
- Be able to use the Scheme list structure to represent an Abstract Data Type
- Be able to tokenize, translate, and evaluate mathematical expressions

2. BACKGROUND

Your task for this exercise is to write a collection of Scheme functions that can be used to parse and evaluate a string representing a mathematical expression. The mathematical expressions are to include numeric values; addition, subtraction, multiplication, and division operators; and parentheses to specify order of operations. For example, the user should be able to enter the expression, "(6+2)*5-8/4", and have it evaluate to 38. Yes, many of you have written a PEX very similar to this in CS 220. This is intentional as it allows you to leverage your understanding of the basic task in solving this problem.

Following is a sample interaction with the Scheme interpreter:

```
> (evaluate "4+16*10/4-2") ; standard precedence among operators
42
> (evaluate "4 + 16 * 10 / 4 - 2") ; whitespace is ignored
42
> (evaluate "(4+16)*10/(4-2)") ; parentheses change order of operations
100
```

You should assume all expressions being evaluated are error-free.

Evaluating an expression will be broken down into three steps:

1. Tokenize the expression
2. Convert the expression to postfix notation
3. Evaluate the postfix expression

Implementing each of these steps will require a stack Abstract Data Type. The following Scheme functions use the built-in list structure to implement a stack ADT:

```
; Stack functions.

(define (newStack)
  '())

(define (isEmpty? stk)
  (null? stk))

(define (push item stk)
  (cons item stk))

(define (top stk)
  (if (isEmpty? stk)
    (error "ERROR: Accessed top of empty stack.")
    (first stk)))

(define (pop stk)
  (if (isEmpty? stk)
    (error "ERROR: Popped empty stack.")
    (rest stk)))
```

3. PRELIMINARY EXERCISE

For the preliminary exercise you will write several functions that will be useful in completing the remainder of the programming exercise.

The **`operator?`** function determines if a character is a valid mathematical operator (the valid operators for this assignment, are `+`, `-`, `*`, and `/`). Following is a sample interaction with the Scheme interpreter:

```
> (operator? #\+)
true
> (operator? 42)
false
```

The **`precedence?`** function determines if one operator has precedence over another operator:

```
> (precedence? #\* #\+)
true
> (precedence? #\+ #\*)
false
> (precedence? #\+ #\+)
true
```

The **`calculate`** function applies the given operator to the given operands:

```
> (calculate #\+ 16 32)
48
> (calculate #\* 4 8)
32
```

The **`valueOf`** function returns the value of a numeric character:

```
> (valueOf #\8)
8
```

The **`getNumber`** function returns the value represented by a stack of numeric characters. The top character is the least-significant digit and the bottom character is the most-significant digit:

```
> (getNumber (push #\6 (push #\5 (push #\2 (newStack)))))
256
```

The **`stringToNumber`** function returns the numeric value of a string of digits:

```
> (stringToNumber "256")
256
> (stringToNumber "42")
42
> (stringToNumber "0")
0
```

The `stringToNumber` function must use a stack. The basic algorithm is to push each character onto the stack, from left to right, and then use the **`getNumber`** function to return the value. This is why the bottom character on the stack is the most significant digit in the number.

4. Preliminary Exercise – Helpful Hints

Literal character values in Scheme are written with the character preceded by the escape sequence "**#**". Thus, the character 'a' is written as **#\a** and the plus sign character is written as **#\+**.

The built-in **string->list** function takes a string as a parameter and returns a list of the individual characters in the string:

```
> (string->list "256")
(list #\2 #\5 #\6)
```

5. Programming Exercise

Implement the remaining functionality to evaluate a string containing a mathematical expression in infix notation. The **tokenize** function must process characters one at a time and use the **getNumber** function. The **infixToPostfix** and **evaluatePostifx** functions must use a stack and follow the algorithms outlined in this document.

An expression may be typed with or without whitespace between numbers, operators, and parentheses. Thus, the first task is to tokenize the expression, creating a list containing numbers and characters.

The **tokenize** function converts a string into a list of tokens:

```
> (tokenize "4+16*10/4-2")
(list 4 #\+ 16 #\* 10 #\/ 4 #\- 2)
> (tokenize "4 + 16 * 10 / 4 - 2")
(list 4 #\+ 16 #\* 10 #\/ 4 #\- 2)
> (tokenize "(4+16)*10/(4-2)")
(list #\( 4 #\+ 16 #\) #\* 10 #\/ #\( 4 #\- 2 #\))
```

The **infixToPostfix** function converts a list of tokens in infix notation to a list of tokens in postfix notation:

```
> (infixToPostfix '(4 #\+ 16 #\* 10 #\/ 4 #\- 2))
(list 4 16 10 4 #\/ #\* #\+ 2 #\-)
> (infixToPostfix '(#\( 4 #\+ 16 #\) #\* 10 #\/ #\( 4 #\- 2 #\)))
(list 4 16 #\+ 10 4 2 #\- #\/ #\*)
```

The **evaluatePostfix** function evaluates a list of tokens in postfix notation:

```
> (evaluatePostfix '(4 16 10 4 #\/ #\* #\+ 2 #\-))
42
> (evaluatePostfix '(4 16 #\+ 10 4 2 #\- #\/ #\*))
100
```

Finally, the **evaluate** function puts everything together to evaluate an expression written as a string:

```
> (evaluate "4+16*10/4-2") ; standard precedence among operators
42
> (evaluate "4 + 16 * 10 / 4 - 2") ; whitespace is ignored
42
> (evaluate "(4+16)*10/(4-2)") ; parentheses change order of operations
100
```

6. PROGRAMMING EXERCISE – HELPFUL HINTS

The **`stringToNumber`** function written for the prelim will not be used directly in completing the programming exercise. Code similar to it will be needed in the **`tokenize`** function and the **`getNumber`** function will be used, but the **`stringToNumber`** function itself will not. It was written on the preliminary exercise for practice using a stack.

Infix to Postfix Translation Algorithm

- Push a left parenthesis (as a string) onto the stack.
- Append a right parenthesis to the end of the infix expression.
- While the stack is not empty,
  - Get the next token from the input string.
  - If the current token is a left parenthesis, push it onto the stack.
  - Else if the current token is an operator,
    - Pop all operators (if there are any) from the top of the stack that have equal or higher precedence than the current operator and append them to the postfix expression.
    - Push the current operator onto the stack.
  - Else if the current token is a right parenthesis,
    - Pop all operators (if there are any) from the top of the stack until a left parenthesis is reached and append them to the postfix expression.
    - Pop (and discard) the left parenthesis from the stack.
  - Else // The current token must be a number.
    - Append the current token to the postfix expression.

Postfix Expression Evaluation Algorithm

- For each token in the postfix expression,
  - If the current token is a number,
    - Push it onto the stack.
  - Else // The current token must be an operator
    - Pop the top value on the stack into the variable x.
    - Pop the next value on the stack into the variable y.
    - Calculate y operator x.
    - Push the result of the calculation onto the stack.
- When finished with all tokens in the postfix expression, pop the top value from the stack. This is the result.

7. SUBMISSION REQUIREMENTS

- **FILL IN YOUR NAME AT THE TOP OF THE PROVIDED SOURCE FILE!**
- **FILL IN THE APPROPRIATE DOCUMENTATION STATEMENT (PRELIM/PEX) AT THE TOP OF YOUR SOURCE FILE!**
- Change the name of the source file to be your own last name.
- Use the website to submit a **single source file** (not zipped). **Do not email your project to me!**
- **All function names must exactly match those in this document as this will be auto-graded.**

CS 359 – PEX 2 Prelim – Grade Sheet Name:_____

Criteria		Points Earned	Points Available
(operator? character)			**3**
(precedence? operator1 operator2)			**3**
(calculate operator operand1 operand2)			**3**
(valueOf character)			**3**
(getNumber stack)			**9**
(stringToNumber string)			**9**
	Subtotal:		**30**
Adjustments	**All code meets specified standards:**		**– 3**
	Vague/Missing Documentation:		**– 2**
	Submission Requirements Not Followed:		**– 2**
	Late Penalties:		**25/50/75%**
	Total w/adjustments:		

Comments from Instructor:

CS 359 – PEX 2 – Grade Sheet Name:_____

Criteria		Points Earned	Points Available
(tokenize infixString)			**12**
(infixToPostfix tokenList)			**12**
(evaluatePostfix postfixTokenList)			**12**
(evaluate infixString)			**12**
Overall quality of software design			**12**
	Subtotal:		**60**
Adjustments	**All code meets specified standards:**		**– 6**
	Vague/Missing Documentation:		**– 3**
	Submission Requirements Not Followed:		**– 3**
	Late Penalties:		**25/50/75%**
	Total w/adjustments:		

Comments from Instructor: