Data Structures and Objects
CSIS 3700

Fall Semester 2024 — CRN 41145

Project 2 — Fraction Calculator
Due date: Monday, Octber 21, 2024

Goal

Develop a program that implements a four-function calculator that performs all arithmetic with fractions.

Details

Your program will read a list of arithmetic expressions, evaluate them and display their results. All numbers in the expression will be integers; however, the results of calculations will be fractions.

Your program must be able to process a sequence of valid arithmetic expressions that include the following:

- Nonnegative integer numbers
- The four basic arithmetic operations
- Parentheses
- Variable names, up to 100 variables; names follow C++ naming rules
- Assignment in the form $var = expression$; note that assignment is also an operator; its "result" is the expression on the right side of the =.

For each expression, evaluate it, display the result and store the result in the appropriate variable, if necessary.

Input an expression consisting of just # to end the program.

In addition, if the user adds the option `-t` to the command line (*e.g.*, `./project2 -t`), instead of evaluating the expressions, the program should list each *token* (number, name, operator, parenthesis) seen in the input. Each token should appear by itself on a line, surrounded by square brackets, and have no additional whitespace before or after. This is a key debugging tool, as it lets you verify that the program is at least reading the input and identifying the input components properly before testing the arithmetic functions.

▹*Required Objects*

Since operands can either be variables or fractions, it will be necessary to create a basic structure with two fields: a **string** and a Fraction. If the string is empty, it is assumed that the Fraction holds the structure's value. If the string is not empty, it is assumed to be a valid variable name.

Side note: If assignment was not treated as an operator, this would not be necessary; the variable on the left side of an assignment would be handled in a different manner than variables on the right side, where the variable's value is used instead of the name.

A calculator needs two **Stack** objects — one to store numbers / variables and one to store operators. In this program, the number stack — the *numStack* — will store the structures describe above and the operator stack — the *opStack* — will store characters.

In order to store and retrieve variable values, a **Dictionary** object will be necessary. The keys are strings and the values are **Fraction**s. The exact implementation of the variable dictionary does not matter.

▹*Calculator Algorithm*

The program must read multiple lines from the standard input. Each line contains an arithmetic expression which may include assignment to a variable. An algorithm for processing such a line follows in Algorithm 1.

Algorithm 1 Calculator evaluation algorithm

```
1:  procedure EVALUATE(string s)
2:      Clear numStack
3:      Clear opStack
4:      Push $ onto opStack
5:      first ← 0

6:      while first < s.length do
7:          if s[first] is a digit then
8:              Convert digit sequence to Fraction
9:              Store Fraction object in structure and push onto numStack
10:             Advance first to first character past digit sequence
11:         else if s[first] is a letter then
12:             Extract name into string
13:             Store name in structure and push onto numStack
14:             Advance first to first character past name
15:         else if s[first] is ( then
16:             Push ( onto opStack
17:             Increment first
18:         else if s[first] is ) then
19:             while top of opStack is not ( do
20:                 Perform top operation
21:             end while
22:             Pop ( from top of numStack
23:             Increment first
24:         else if s[first] is an operator then
25:             while top of opStack has precedence over s[first] do
26:                 Perform top operation
27:             end while
28:             Push s[first] onto opStack
29:             Increment first
30:         else
31:             Increment first
32:         end if
33:     end while

34:     while top of opStack is not $ do
35:         Perform top operation
36:     end while

37:     output top of numStack
38: end procedure
```

To process an operator, pop the **opStack** into a variable. Then, pop two structures from the **numStack** and store them. The first value popped is the right operand, the second value is the left operand.

If the operator is =, then the left side must have a variable name. Use the name as a key, and the right operand is the value; store the key-value pair in your dictionary, or update if the key is already there. Push the right operand onto the **numStack** when done.

If the operator is an arithmetic operator, perform the given operation and push the answer onto

the **numStack**. If either operand is a variable, search for it in the dictionary and use its value in the calculation.

If the expression is well-formed, then at line 37 of Algorithm 1, the **opStack** will only have $ and the **numStack** will have only one value which is the result of evaluating the expression. If the expression is not well-formed, an exception might be thrown or one of the stacks will have more than one value. In these cases, output an error message.

To determine if the top of the **opStack** has precedence over the current input symbol, use the following table:

↓top / input→	* /	+ −	=
* /	Yes	Yes	Yes
+ −	No	Yes	Yes
= ($	No	No	No

Pro tip: Write a separate function that determines precedence. It takes two characters as parameters and returns a boolean value. Use `if-else-if` to determine which value to return.

What to turn in

Turn in your source code and **`Makefile`**. If you use an IDE, turn in a tarball (compressed folder) of your project directory.

Example

Note: "calc:" is the prompt from the program.

```
1     calc: 3 * (2 + 4/105)
2     214 / 35
3     calc: a = 3 * (2 + 4/105)
4     214 / 35
5     calc: a
6     214 / 35
7     calc: a - 4/35
8     6 / 1
9     calc: b = a - 4/35
10    6 / 1
11    calc: b
12    6 / 1
13    calc: #
14
15    Process finished with exit code 0
```

Here is an example with the `-t` option:

```
    calc: abc = 3 * (2 + 4/105)
    [abc]
    =
    [3 / 1]
    *
    (
    [2 / 1]
    +
    [4 / 1]
    /
    [105 / 1]
    )
    calc: #

    Process finished with exit code 0
```