Assignment 2: Calculator Brain

Suggested Completion Date: February 23

Objective

We'll take the calculator you built in Assignment 1 and continue to add features to it. First, you'll update it (if you haven't already) with the Brain we built in lecture, then we'll add the ability to use variables (which we will later use for graphing) as well as to see what has been entered so far.

This assignment is significantly harder than the first assignment. Some of the extra challenges are quite challenging indeed, but if you do them I guarantee that you will learn a lot.

Required Tasks

1. Your calculator needs to be updated to include all of the changes covered in lecture (through getting the Model fully implemented). The first walkthrough posted covers this, as do the first three Stanford videos.
2. You are not allowed to change the public API of the CalculatorBrain unless noted in the assignment.
3. Your UI (the View) should always be in sync with the Model.
4. The extra item from the first assignment about turning `displayValue` into a `Double`? (an Optional) is now required. `displayValue` should return `nil` when the contents cannot be interpreted as a Double. Setting it to `nil` should clear the display (unless you take on error reporting).
5. Add the capability to allow pushing variables onto the internal stack. In this assignment we will only explicitly use one variable, but you should make your code general enough that it could hold multiple variables. To do this implement the following API:
   ```
   func pushOperand(symbol: String) -> Double?
   func setVariable(symbol: String, value: Double)
   var variableValues: Dictionary<String,Double>
   ```
   These should do what you would think. The first pushes a "variable" onto your brain's stack, and the second lets users of the Brain set the value for any variable they wish. `pushOperand` should return the result of `evaluate`() after having pushed the variable.

6. The `evaluate` function should use a variable's value (from the `variableValues` dictionary) whenever a variable is encountered or return `nil` if it encounters a variable with no corresponding value.
7. Implement a new read-only (get only, no set) `var` to `CalculatorBrain` to describe the contents of the brain as a `String`
   ```
   var description: String
   ```
   A. Unary operations should be shown using "function" notation. For example, the input `10 cos` should be displayed in the description as `cos(10)`
   B. Binary operations should be shown using "infix" notation. For example, the input 3 ↵ 5 - should be displayed as 3-5. Be sure your order is correct!

C. All other stack content should be displayed as is. E.g. 23.5 => 23.5, **π** => **π** (not 3.1415), the variable x => x (not its current value), etc.

D. Any combination of stack elements should be properly displayed. Examples:
10 √ 3 + => √(10)+3
3 ↵ 5 + √ => √(3+5)
3 ↵ 5 ↵ 4 + + => 3+(5+4) or (better) 3+5+4
3 ↵ 5 √ + √ 6 ÷ => √(3 + √(5)) ÷ 6

E. If there are any missing operands, substitute a ? for them, e.g. 3 ↵ + => ?+3

F. If there are multiple complete expressions on the stack, separate them by commas: E.g., 3 ↵ 5 + √ **π** cos => √ (3 + 5), cos(**π**). The expressions should be in historical order with the oldest at the beginning of the string and the most recently pushed/performed at the end.

G. Your description must properly convey the mathematical expression. For example, 3 ↵ 5 ↵ 4 + * must not output 3*5+4, it must be 3*(5+4). In other words, you will need to sometimes add parentheses around binary operations. Try to minimize this as much as possible.

8. Modify the `UILabel` you added in Assignment 1 to show your `CalculatorBrain’s` description instead. It should put an = on the end of it (and ideally should be positioned strategically so that the display looks like it is the result of that =).
9. Add two new buttons to your Calculator: →M and M. These two buttons will set and get (respectively) a variable in the `CalculatorBrain` called M.
   a. →M sets the value of the variable M in the brain to the current value of the display (if any).
   b. →M should not perform an automatic ↵ (though it should reset user is in the middle of typing a number).
   c. Touching M should push an M variable (not the value of M) onto the CalculatorBrain.
   d. Touching either button should show the evaluation of the brain.
   e. →M and M are Controller mechanics, not Model mechanics.
   f. This isn’t a real “memory” button on your calculator, it is a step towards graphing and it allows testing of whether your basic functionality works. Example tests:
   7 M + √ => description is √(7+M), display is blank because M is not set
   9 →M => display now shows 4 (the square root of 16), description doesn’t change
   14 + => display now shows 18, description is now √(7+M)+14
10. Make sure that your C button from Assignment 1 is updated to work with this assignment.
11. When you touch the C button, the M variable should be removed from the variableValues Dictionary in the CalculatorBrain (not set to zero or any other value). This will allow you to test the case of an “unset” variable (because it will make evaluate() return nil and thus your Calculator’s display will be empty if M is ever used without a →M).
12. Your Calculator should look good on any iPhone in both portrait and landscape. iPads will come in the next assignment.
13. The absolute bare minimum amount of testing should include all of the examples included in the assignment. Of course you should do much more.

Hints

1. Consider using optional chaining in your implementation of `displayValue`
2. Once you have a better `displayValue`, use it everywhere that makes sense in your ViewController

3. If you implemented **π** by just pushing M_PI on Assignment 1, you will probably have to make your brain accept the ability to push constants.
4. Don't implement **π** as a variable. Among other things, C might cause real problems for such a solution.
5. When clearing your display, put a " " in the display, not `nil` or "" (empty string), otherwise your UILabel may not behave itself (it could shrink down to nothing).
6. Your new `description` functionality is going to need to work an awful lot like `evaluate`(). Use that as a guide.
7. You are best off using the "separate expressions with a comma" part of the `description` task inside the `description` var's code (rather than in the recursive method).
8. You can remove a lot of history-collecting code from Assignment 1.
9. The UI needs to be in sync with the Model. Careful to cover all of your bases (including things like C, and the variable buttons).
10. If your Autolayout is messed up, consider starting over by removing all of the constraints in the scene. Then move the views to where they look good in the square layout (use the dashed blue lines). First get the `UILabels` set up. Next select the buttons and apply the constraints that you want. If you have real trouble consider going with a strict grid for the buttons.
11. Dashed blue lines are your friends.

More Opportunities

1. Make your description have as few parenthesis as possible. Note: this is conceptually pretty difficult, but once you figure it out the actual coding is not too hard.
2. Add Undo. This combines your backspace with real undo functionality.
3. Add a new method, `evaluateAndReportErrors()`. It should work like `evaluate`() except that if there is a problem of any kind evaluating the stack (not just unset variables or missing operands, but also divide by zero, square root of a negative number, etc.). Instead of returning `nil`, it will return a `String` with a description of the problem. Report this using your display.

Approaches to these additional challenges.

PARENTHESIS REMOVAL

1. This will require adding "operator precedence" to your Op.
2. Just like the `description`, a "precedence" var could return a default value for most Ops, but a specific, associated value for binary operations. The precedence for variables, constants, operands, and unary operations is all the same — the highest precedence possible.
3. It is fine to use Int to handle precedence. The highest precedence would be `Int.max`
4. Precedence only impacts the `description` of the op stack. It has no effect on evaluation. Evaluation is determined purely by stack order.

UNDO

1. Check out the section on backspace from Assignment 1 if you didn't try it.
2. Don't mess with setting the M variable.
3. You will need a new API function in your Brain. Said function will be extremely simple.

REPORTING ERRORS

1. This is a good chance for more practice with `Optionals`
2. Error reporting should be completely separate from your description.
3. It is likely that `evaluateAndReportErrors` will need to ask operations what error (if any) that could be created if certain operands were passed in.
4. One way to do this would be for the operations to have an associated function that analyzes potential arguments and returns an appropriate error `String` if performing that operation would generate an error (or `nil` otherwise). Keep in mind that operations already have one associated function, so this would just be one more.
5. Most operations won't have any possible errors. So you want this case to be easy. One solution is to pass `nil` in as a testing function. You can make a function be `Optional` by putting the function type in parentheses and then put the ? after. E.g. `((Double, Double) -> string?)?` is an `Optional` function which takes two `Doubles` and returns an `Optional` String.
6. Don't forget about Optional Chaining syntax. E.g. you could involve an `Optional` function called `errorTest` like this . . .

```
  if let failureDescription = errorTest?(argument) {
```

. . . and the if would fail if `errorTest` itself was `nil`, or it is not `nil`, but the function `errorTest` refers to returns `nil`.

7. In your VC you might find that implementing a new `var` called `displayResult` (which handles both values and errors) and then reimplementing `displayValue` in terms of that var will make your code cleaner.
8. Be sure to test on divide by zero, square root of a negative number, not enough operands, and variable x not set.
9. One solution to this whole business is to use "error codes" which are reported from the Model to the Controller. Then the Controller could use the code to update the View. You don't have to do this though, using Strings is fine.