Assignment II: Programmable Calculator

Objective

The goal of this assignment is to extend the `CalculatorBrain` from last lecture's demo to allow inputting variables into the calculator's `program` and to show the user the steps they have entered to get the result showing in the display (i.e. showing the `CalculatorBrain`'s "`program`"). This involves using the `id` type, introspection, Foundation classes, enumeration, and understanding properties: all things talked about in class this week.

You will not be "walked through" this assignment line by line.

Next week's assignment will build on this week's so don't fall behind!

This assignment must be submitted using the submit script (see the class website for details) by the end of the day next Wednesday. You may submit it multiple times if you wish. Only the last submission will be counted.

Be sure to check out the Hints section below!

Also, check out the latest additions to the Evaluation section to make sure you understand what you are going to be evaluated on with this (and future) assignments.

Materials

- You should begin by taking your assignment from last week and adding the interface and implementation of the `CalculatorBrain` API demonstrated in the last lecture (`program` property, `descriptionOfProgram:` and `runProgram:`). This code is downloadable from the class website, but it does not include the additions you were required to make to last week's walkthrough, so you'll have to merge them in yourself.
- It is recommended that you make a copy of last week's assignment before you start modifying it for this week's.

Required Tasks

Your solution to this assignment must include properly functioning implementations for the three public methods added to the `CalculatorBrain` in lecture. You may change the internal implementations as necessary (though you almost certainly do not want to change the implementation for the `@property`), but do not change the public API for these (including whether a method is a class method or not):

```
@property (readonly) id program;
+ (double)runProgram:(id)program;
+ (NSString *)descriptionOfProgram:(id)program;
```

Also, do not break any existing `CalculatorBrain` functionality with your new features in this assignment.

1. Add the capability to your `CalculatorBrain` to accept variables as operands (in addition to still accepting `double`s as operands). You will need new public API in your `CalculatorBrain` to support this.

   A variable will be specified as an `NSString` object. To simplify your implementation, you can ignore attempts to push a variable whose name is the same as an operation (e.g. you can ignore an attempt to push a variable named "`sqrt`").

   The values of the variables will only be supplied when the "program" is "run." You must add a new version of the `runProgram:` class method with the following signature ...

```
+ (double)runProgram:(id)program
  usingVariableValues:(NSDictionary *)variableValues;
```

   The keys in the passed `variableValues` dictionary are `NSString` objects corresponding to the names of variables used in the passed `program`, and the values in the dictionary are `NSNumber` objects specifying the value of the corresponding variable (for this assignment we will supply "test" values, see Required Task #3).

   If there are variables in the specified `program` for which there are no values in the specified `NSDictionary`, use a value of 0 for those variables when you run the program. This should be the case if someone calls the original `runProgram:` method (the one shown in the demo in class).

   In addition, create another class method to get all the names of the variables used in a given `program` (returned as an `NSSet` of `NSString` objects) ...

```
+ (NSSet *)variablesUsedInProgram:(id)program;
```

   If the program has no variables return `nil` from this method (not an empty set).

2. Re-implement the `descriptionOfProgram:` method from the last lecture to display the passed `program` in a more user-friendly manner. Specifically ...

   a. It should display all single-operand operations using "function" notation. For example, `10 sqrt` should display as `sqrt(10)`.

   b. It should display all multi-operand operations using "infix" notation if appropriate, else function notation. For example, `3 Enter 5 +` should display as `3 + 5`.

   c. Any no-operand operations, like π, should appear unadorned. For example, π.

   d. Variables (Required Task #1) should also appear unadorned. For example, `x`.

   Any combination of operations, operands and variables should display properly. Examples (E means "Enter key") ...

   a. `3 E 5 E 6 E 7 + * -` should display as `3 - (5 * (6 + 7))` or an even cleaner output would be `3 - 5 * (6 + 7)`.

   b. `3 E 5 + sqrt` should display as `sqrt(3 + 5)`.

   c. `3 sqrt sqrt` should display as `sqrt(sqrt(3))`.

   d. `3 E 5 sqrt +` should display as `3 + sqrt(5)`.

   e. `π r r * *` should display as `π * (r * r)` or, even better, `π * r * r`.

   f. `a a * b b * + sqrt` would be, at best, `sqrt(a * a + b * b)`.

   As you can see, you will have to use parentheses in your output to correctly display the program. For example, `3 E 5 + 6 *` is not `3 + 5 * 6`, it is `(3 + 5) * 6`. Try to keep extraneous parentheses to a minimum though (see Hints).

   It might be that there are multiple things on the stack. If so, separate them by commas in the output with the top of the stack first, for example `3 E 5 E` would display as "`5, 3`". `3 E 5 + 6 E 7 * 9 sqrt` would be "`sqrt(9), 6 * 7, 3 + 5`".

3. Update the user-interface of your Calculator to test all of the above.

   a. Your UI should already have a `UILabel` which shows what has been sent to the brain. Change it to now always show the latest description of the program currently in the `CalculatorBrain` using your `descriptionOfProgram:` method. It should show the description without substituting variable values (obviously, since `descriptionOfProgram:` does not take a variable value dictionary as an argument).

   b. Add a few variable buttons (e.g, `x`, `y`, `foo`). These are buttons that push a variable into the `CalculatorBrain`.

   c. Change your calculator to update its `display` (as needed) by calling your new `runProgram:usingVariableValues:` method. The variable values dictionary it passes to this method should be a property in your Controller (let's call it "`testVariableValues`").

   d. Add a `UILabel` to your UI whose contents are determined by iterating through all the `variablesUsedInProgram:` and displaying each with its current value from `testVariableValues`. Example display: "`x = 5  y = 4.8  foo = 0`".

   e. Add a few different "test" buttons which set `testVariableValues` to some preset testing values. One of them should set `testVariableValues` to `nil`. Don't forget to update the rest of your UI when you change `testVariableValues` by pressing one of these test buttons. Make sure that your preset values are good edge-cases for testing (we are intentionally not telling you what to use since part of good programming is figuring out how to thoroughly test your application).

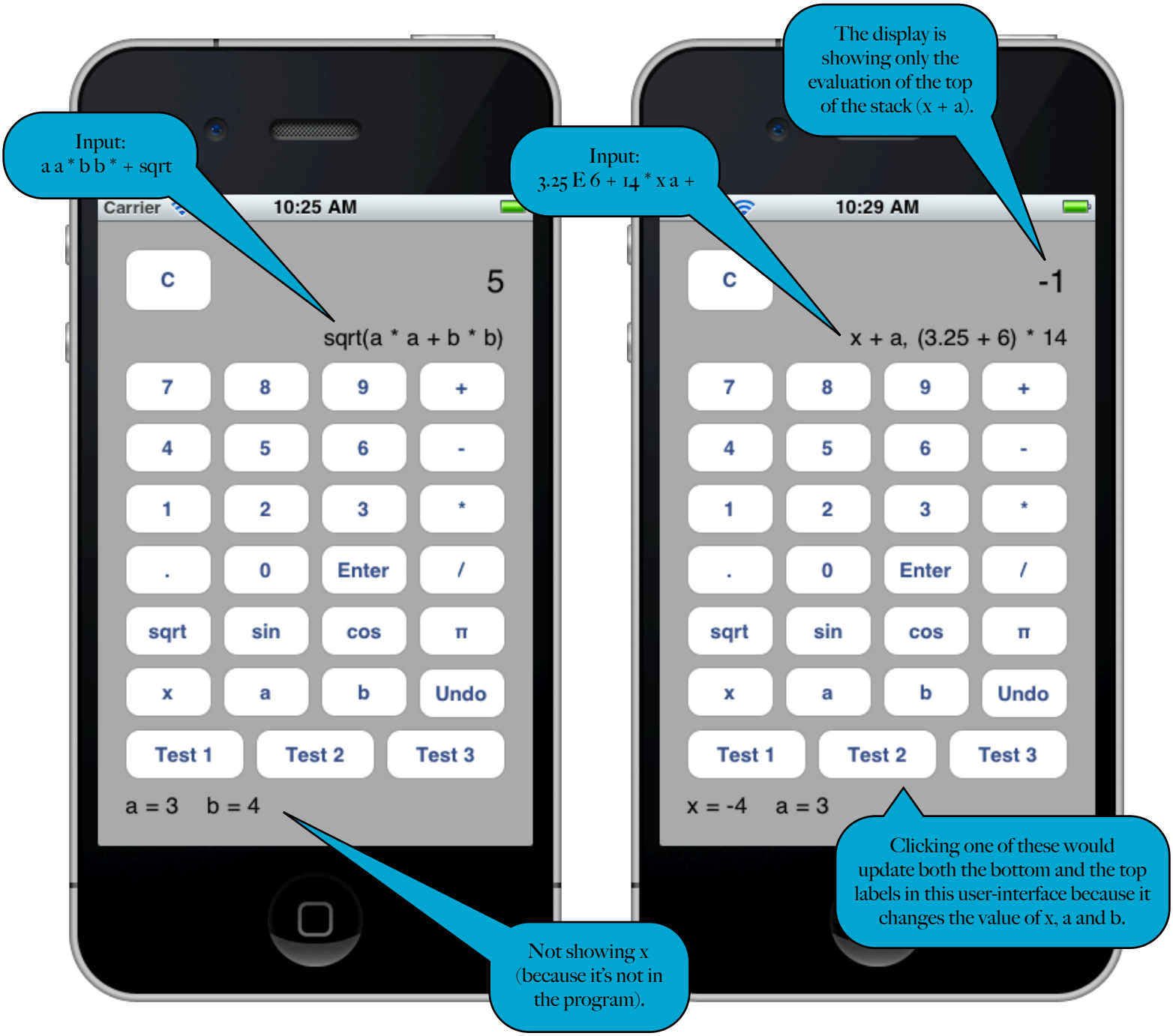
4. Add an Undo button to your calculator. Hitting Undo when the user is in the middle of typing should take back the last digit or decimal point pressed until doing so would clear the `display` entirely at which point it should show the result of running the `brain`'s current `program` in the `display` (and now the user is clearly not in the middle of typing, so take care of that). Hitting Undo when the user is not in the middle of typing should remove the top item from the program stack in the `brain` and update the user-interface.

   This task can be implemented with 1 method in your Controller (of about 5-6 lines of code, assuming you've factored out the updating of your user-interface into a single method somewhere) and 1 method (with 1 line of code) in your Model. If it's taking much more than that, you might want to reconsider your approach.

5. Don't let anything crash your program in `CalculatorBrain`. It should defend against any value being passed through its API.

Screenshots

Your UI absolutely, positively does NOT have to look like this. In fact, we hesitate to put screen shots into homework assignments at all because we don't want to stifle your creativity in how you build your UI. However, in the past, students have complained that they can't visualize the Required Tasks by text description alone, so here is an example screenshot of how your application might look.



Hints

1. The `NSMutableArray` method `replaceObjectAtIndex:withObject:` might be useful to you in this assignment. Note that you cannot call this method inside a `for-in` type of enumeration (since you don't have the index in that case): you'd need a `for` loop that is iterating by index through the array.
2. It is possible to implement `runProgram:usingVariableValues:` without modifying the method `popOperandOffStack:` at all. You don't have to do it that way (this is only a hint, after all) but it is possible to do it that way.
3. You will almost certainly want to use recursion to implement `descriptionOfProgram:` (just like we did to implement `runProgram:`). You might find it useful to write yourself a `descriptionOfTopOfStack:` method too (just like we wrote ourselves a `popOperandOffStack:` method to help us implement `runProgram:`). If you find recursion a challenge, think "simpler," not "more complex." Your `descriptionOfTopOfStack:` method should be less than 20 lines of code and will be very similar to `popOperandOffStack:`. The next hint will also help.
4. One of the things your `descriptionOfProgram:` method is going to need to know is whether a given string on the stack is a two-operand operation, a single-operand operation, a no-operand operation or a variable (because it gives a different description for each of those kinds of things). Implementing some private helper method(s) to determine this is probably a good idea. You could implement such a method with a lot of `if-then`s, of course, but you might also think about whether `NSSet` (and its method `containsObject:`) might be helpful.
5. You might find the private helper methods mentioned in Hint #4 useful in distinguishing between variables and operations in your other methods as well. It's very likely that you're going to want a `+ (BOOL)isOperation:(NSString *)operation` method.
6. It might be a good idea not to worry about extraneous parentheses in your `descriptionOfProgram:` output at first, then, when you have it working, go back and figure out how to *suppress* them in certain cases where you know they are going to be redundant. As you're thinking about this, at least consider handling the case of the highest precedence operations in your `CalculatorBrain` where you clearly do not need parentheses. Also think about the need for parentheses inside parentheses when doing function notation (e.g. `sqrt((5 + 3))` is ugly).
7. The `NSDictionary` class method `dictionaryWithObjectsAndKeys:` is great for Required Task 3e.
8. As always, fewer lines of code is better than more lines of code. For example, the changes to `CalculatorBrain` in this assignment can be done with way fewer than 100 lines of code (in fact, it can be done with half that). Even fewer in your Controller.

Links

Most of the information you need is best found by searching in the documentation through Xcode (see the Help menu there), but here are a few links to Apple Documentation that you might find helpful. Get used to referring to this documentation to figure out how to do things. No one expects anyone to memorize every method in every object, so being facile with looking things up in the Apple Documentation is key to mastering the iPhone Development Environment.

- Objective-C Primer
- Introduction to Objective-C
- NSObject Reference
- NSString Reference
- NSNumber Reference
- NSArray Reference
- NSMutableArray Reference
- NSDictionary Reference
- NSSet Reference
- NSMutableSet Reference
- NSMutableDictionary Reference

Evaluation

In all of the assignments this quarter, writing quality code that builds without warnings or errors, and then testing the resulting application and iterating until it functions properly is the goal.

Here are the most common reasons assignments are marked down:

- Project does not build.
- Project does not build without warnings.
- One or more items in the Required Tasks section was not satisfied.
- A fundamental concept was not understood.
- Code is sloppy and hard to read (e.g. indentation is not consistent, etc.).
- Your solution is difficult (or impossible) for someone reading the code to understand due to lack of comments, poor variable/method names, poor solution structure, etc.
- Assignment was turned in late (you get 3 late days per quarter, so use them wisely).
- Code is too lightly or too heavily commented.
- Code crashes.

Extra Credit

1. Enhance your application to show the user error conditions like divide by zero, square root of a negative number, and insufficient operands. One way to do this (and to get more experience with using `id`) might be to have `runProgram:usingVariableValues:` in `CalculatorBrain` return an `id` (instead of a `double`) which is an `NSNumber` with the result or an `NSString` with the description of an error if it encounters one. Then update your Controller to use introspection on the return value and display the appropriate thing to the end-user.