Trabalho II: Cérebro da Calculadora

Observação

Este material foi traduzido e adaptado do curso CS 193P - *iPhone Application Development* oferecido pela Universidade de Stanford através do iTunesU. Ele está protegido pela licença *Creative Commons Attribution-Noncommercial-Share*[1].

Objetivo

Você iniciará este trabalho melhorando o primeiro trabalho (Trabalho I: Calculadora) para incluir as mudanças realizadas na aula (i.e. CalculatorBrain, etx.). Este é o último trabalho em que você terá que replicar código da aula através de digitação.

Agora que nós adicionamos um Model de MVC à nossa calculadora, vamos aumentar as suas capacidades mais além. Você melhorará a sua calculadora para permitir a entrada de uma "variável" na pilha. Além disso, você fornecerá um meio para o usuário ver melhor o que foi digitado até o momento.

Materiais

- Você precisará ter completado o trabalho 1. Este trabalho inicia no final dele.
- Todas as mudanças realizadas na aula precisam ser feitas no código.

Lista de Tarefas Requeridas

1. Todas as mudanças feitas à calculadora em sala de aula devem ser aplicadas ao primeiro trabalho quando finalizado. Faça isso completamente antes de prosseguir para as outras tarefas desta lista. E, como no primeiro trabalho, digite as modificações, não as copie e cole.

1 http://creativecommons.org/licenses/by-nc-sa/3.0/us/

2. Não modifique qualquer API não privada na classe *CalculatorBrain* e continue a utilizar um *enum* como a estrutura de dados interna principal.

3. Sua IU deve sempre estar em sincronismo com o seu Model (a classe *CalculatorBrain*).

4. A tarefa com nota extra do trabalho 1 para transformar *displayValue* em um *Double?* (i.e. um *Optional* ao invés de um *Double*) agora é obrigatória. *displayValue* deve retornar *nil* quando o conteúdo do visor não puder ser interpretado como um *Double*. Atribuir *nil* à propriedade computada *displayValue* deve limpar o visor.

5. Adicione a capacidade à classe *CalculatorBrain* para permitir empilhar variáveis. Faça isso implementando a seguinte API em *CalculatorBrain*:

```
func pushOperand (symbol : String) -> Double?
var variableValues : Dictionary <String, Double>
```

Este método e variável de instância devem fazer exatamente o que você imaginaria que fariam: o primeiro empilha uma "variável" (por exemplo, *pushOperand ("x")* empilharia uma variável denominada x) e a segunda permite que usuários da classe *CalculatorBrain* atribua o valor para qualquer variável que eles desejem (por exemplo, *brain.variableValues ["x"] = 35.0*). *pushOperand* deve retornar o resultado de *evaluate ()* após ter empilhado a variável (exatamente como o outro *pushOperand* faz).

6. O método *evaluate ()* deve usar um valor de variável (do dicionário *variableValues*) quando uma variável é encontrada, ou retornar *nil* se ele encontrar uma variável sem valor correspondente.

7. Implemente uma nova propriedade computada somente de leitura (apenas *get*, sem *set*) para que a classe *CalculatorBrain* descreva o conteúdo de *brain* como uma *String*...

```
var description : String
```

a. Operações unárias devem ser exibidas usando a notação de função. Por exemplo, a entrada 10 cos mostraria em *description* cos (10).

b. Operações binárias devem ser exibidas usando a notação infixa. Por exemplo, a entrada 3 ↵ 5 - deve ser exibida como 3 - 5. Certifique-se de que a ordem está correta!

c. Todos os outros conteúdos da pilha (por exemplo, operandos, variáveis, constantes como pi, etc.) devem ser exibidos sem modificações. Por exemplo, 23.5 ⇒ 23.5, pi ⇒ π (não 3.1415), a variável x ⇒ x (não seu valor), etc.

d. Qualquer combinação de elementos na pilha deve ser apresentado corretamente. Exemplos:

10 √ 3 + ⇒ √ (10) + 3

3 ↵ 5 + √ ⇒ √(3 + 5)

3 ↵ 5 ↵ 4 + + ⇒ 3 + (5 + 4) ou (por nota extra) 3 + 5 + 4

3 ↵ 5 √ + √ 6 ÷ ⇒ √(3 + √ (5)) ÷ 6

e. Se há qualquer operando faltando, substitua-o por um ponto de interrogação. Por exemplo: 3 ↵ + ⇒ ? + 3.

f. Se há múltiplas expressões completas na pilha, separe-as por vírgulas: por exemplo, 3 ↵ 5 + √ π cos ⇒ √ (3 + 5), cos (π). As expressões devem estar em ordem histórica com a mais antiga no início da string e a mais recente empilhada/realizada no final.

g. Sua descrição deve representar corretamente a expressão matemática. Por exemplo, 3 ↵ 5 ↵ 4 + * não deve ter como saída 3 * 5 + 4, deve ser 3 * (5 + 4). Em outras palavras, você precisará, algumas vezes, adicionar parênteses nas operações binárias. Tendo dito isso, tente minimizar o uso de parênteses (de forma que a saída esteja matematicamente correta). Veja a lista de tarefas com nota extra se você realmente deseja fazer isso bem.

8. Modifique o *UILabel* adicionado no primeiro trabalho para mostrar *description* da classe *CalculatorBrain* ao invés do que era exibido. Deve ser colocado um sinal de igual, =, no final dele (e posicionado estrategicamente para que *display* pareça ser o resultado daquele =). Este = estava na lista de tarefas com nota extra no primeiro trabalho, mas

dessa vez é uma tarefa requerida.

9. Adicione dois novos botões à calculadora: →M e M. Estes dois botões atribuirão e pegarão (respectivamente) uma variável na classe *CalculatorBrain* chamada *M*.

a. →M atribui o valor da variável M para o valor atual de *display* (se houver algum).

b. →M não deve realizar um ↵ automático (embora ele deva reiniciar *userIsInTheMiddleOfTypingANumber*)

c. O pressionamento de M deve empilhar a variável M (não o valor de M).

d. Pressionar qualquer um dos botões deve exibir a avaliação no visor (i.e. o resultado do método *evaluate ()*).

e. →M e M são componentes de *Controller*, não de *Model* (embora ambas usem as variáveis em *Model*).

f. Estes não são bons botões de memória para a calculadora, mas são bons para testar se nosso método para manipular variáveis implementado como uma tarefa requerida está funcionando corretamente. Exemplos:

7 M + √ ⇒ *description* é √(7 + M), o visor fica vazio porque M não contém um valor.

9 →M ⇒ o visor exibe 4 (a raiz quadrada de 16), *description* ainda é √(7 + M).

14 + ⇒ o visor exibe 18, *description* é √(7 + M) + 14

10. Certifique-se de que o botão C do trabalho 1 funciona corretamente neste trabalho.

11. Quando o botão C for pressionado, a variável M deve ser removida do dicionário *variablesValues* na classe *CalculatorBrain* (não deve ser atribuída a zero ou qualquer outro valor). Isso permitirá que você teste o caso de uma variável sem valor (porque isso fará *evaluate ()* retornar nil e assim o visor da calculadora estará vazio se M é usado sem →M).

12. Sua IU deve ficar boa em qualquer iPhone em qualquer orientação (por enquanto, não se preocupe com iPad). Isso significa que o *autolayout* deve estar bem configurado!

Dicas

1. Considere utilizar um encadeamento de *Optional* na implementação de *displayValue*.

2. Agora que você tem um *displayValue* melhor, certifique-se de que está sendo usado corretamente em todos os locais do *ViewController*.

3. Se você implementou π simplesmente empilhando *M_PI* no último trabalho, você provavelmente terá que melhorar a classe *CalculatorBrain* para que ela empilhe constantes de forma que *description* mostrará π ao invés de 3.1415926...

4. Você provavelmente não quer implementar π como uma variável com o nome π e o valor de 3.1415926... Por que não? Porque os programadores que usarem a sua classe *CalculatorBrain* poderiam limpar todos os valores de variáveis (para propósitos próprios) e ficariam surpresos em ver que π ainda tem um valor.

5. Se você ainda está um pouco confuso sobre o que fazer com π, não pense demais. É somente um outro *Op*. Sinta-se livre para melhorar *Op* como necessário para suportar constantes como π.

6. Quando limpar o visor, coloque um " " (espaço) lá, não use *nil* ou "" (string vazia), caso contrário seu *UILabel* encolherá verticalmente, bagunçando toda a sua IU.

7. Obviamente *evaluate ()* é bastante similar à nova funcionalidade de *description* que lhe foi pedida para implementar (por exemplo, ambos utilizam métodos auxiliares recursivos), então use-o como um guia.

8. Você provavelmente se dará melhor fazendo a tarefa requerida que tem uma parte de separar as expressões com vírgulas em *description* dentro do código da variável *description* (ao invés de fazer no método recursivo auxiliar).

9. Você pode remover um monte de código de coleta de histórico do trabalho 1 já que você adicionou uma nova API na classe *CalculatorBrain* que relata isso diretamente. A propósito, uma solução elegante provavelmente encontraria um único local no *ViewController* para atualizar este

UILabel (mas isso é só uma dica, não é uma tarefa requerida).

10. Você deveria ser capaz de implementar os métodos *@IBAction* para M e →M em duas ou três linhas de código cada uma deles. Isso não é uma tarefa requerida, é apenas uma dica!

11. Não esqueça de pensar que a classe *CalculatorBrain* deve ser a mais reutilizável possível: seria mais flexível (e não há motivo para não ser) para permitir a programadores usando a API pública limpar a pilha separadamente dos valores das variáveis (mesmo que o seu botão C limpe ambos).

12. Sua IU deve sempre estar sincronizada com o *Model* (a classe *CalculatorBrain*). Há meios sorrateiros para esquecer disso (por exemplo, M, →M, etc.). Tenha cuidado.

13. Se seu *autolayout* ficar todo bagunçado (com restrições conflitantes, etc.), considere reiniciá-lo removendo todas as restrições na cena (utilizando o pequeno botão no canto inferior direito do editor de *storyboard*), movendo as *views* para os locais em que elas parecem boas no layout (usando as linhas pontilhadas azuis), configurando as restrições dos *UILabel* (usando a tecla Ctrl e arrastando-os com o mouse para ligá-los com os lados e com o topo e usando o *Size Inspector* para editar qualquer restrição, se necessário), então selecionando todos os *UIButton* e (usando o pequeno botão denominado Pin no canto inferior direito do editor de *storyboard*) aplicando as restrições que você deseja para todos os botões de um só vez (tamanhos iguais, espaçados igualmente - *equal sizes, spaced evenly*). Finalmente, abra o *Document Outline* e veja se há qualquer avisos ou erros nas suas restrições e clique nos símbolos de avisos/erros para consertá-los.

14. *Autolayout* está diretamente relacionado com as linhas pontilhadas azuis. Se você não usá-las para posicionar os objetos gráficos, o Xcode lutará para entender o que você quer.

Aprendizado

Eis uma lista parcial de conceitos que este trabalho pretende fazer com que você ganhe prática ou demonstre o seu conhecimento:

1. Optionals
2. Closures
3. *enum*

4. *switch*
5. Dictionary
6. Tuplas
7. Autolayout
8. Recursividade

Avaliação

Em todos os trabalhos realizados com Swift, escrever código com qualidade que compila sem avisos ou erros, e então testar as aplicações resultantes até que elas funcionem corretamente é o objetivo.

Aqui estão as razões mais comuns para que trabalhos sejam negativados:

- o projeto não compila;
- o projeto não compila sem avisos;
- um ou mais itens da lista de tarefas não foi cumprido;
- um conceito fundamental não foi entendido;
- o código é visivelmente desorganizado e difícil de ler (por exemplo, a indentação não é consistente, etc.);
- a solução é difícil (ou impossível) para alguém que está lendo o código entender devido à falta de comentários, nomes ruins de variáveis ou métodos, estrutura de solução pobre, métodos longos, etc.

Com frequência os alunos perguntam “quanto comentário deve ser adicionado ao código?”. A resposta é “o suficiente para que o seu programa seja compreensível por qualquer pessoa que o leia”. Você pode assumir que o leitor conhece o SDK, mas não pode assumir que ele sabe a (ou uma) solução para o problema.

Lista de Tarefas com Nota Extra

A lista de tarefas que vale nota extra é uma oportunidade de expandir o que foi aprendido. É altamente recomendado que você tente fazer as tarefas desta lista para que possa aprender o máximo possível.

1. Faça com que *description* tenha o mínimo de parênteses possível para operações binárias.

2. Adicione um botão para desfazer na calculadora. Na lista de tarefas com nota extra do trabalho 1, você pode ter adicionado o botão de apagar. Aqui estamos falando em combinar ambos o botão de apagar e o desfazer em um só botão. Se o usuário está no meio da digitação de um número, este botão deve fazer o mesmo que apagar. Caso contrário, o usuário não está no meio da digitação de um número, ele deve desfazer a última coisa que foi feita na classe *CalculatorBrain*.

3. Adicione um novo método, *evaluateAndReportErrors ()*. Ele deve funcionar como *evaluate ()*, exceto no caso em que houver um problema de qualquer tipo na avaliação da pilha (não somente variáveis sem valores ou operandos faltando, mas também divisões por zero, raiz quadrada de números negativos, etc.), em vez de retornar *nil*, ele retornará uma *String* com a descrição do problema (se houver múltiplos problemas, você pode simplesmente retornar qualquer um deles). Apresente qualquer erro no visor da calculadora (em vez de somente deixá-lo em branco ou de apresentar algum valor estranho). Você ainda deve implementar *evaluate ()* como especifica na lista de tarefas requeridas acima, mas, se quiser, você pode fazer *evaluate ()* retornar *nil* se há erros (não somente os casos de variável sem valor ou operandos insuficientes). Os métodos *push* e *perform* ainda devem retornar *Double?* (que é uma forma de avaliação fraca, mas nós desejamos ser capazes de avaliar separadamente as tarefas com nota extra das tarefas requeridas).

Dicas da Lista de Tarefas com Nota Extra

Aqui estão algumas dicas para abordar os itens da lista de tarefas com nota extra.

1. Remoção de parênteses

   1.1. Isso requer a adição do conceito de precedência de operadores a *Op* na classe *CalculatorBrain*.

   1.2. Como em *description* de *Op*, uma variável com precedência poderia retornar um valor padrão para a maioria dos *Op*, mas então retornar um valor específico associado para operações binárias. A precedência de uma variável, constante, operação unária ou operando unário é feita da mesma forma, i.e., a mais alta precedência possível.

1.3. É provavelmente bom representar a precedência como um *Int* (valor mais alto significando a maior precedência). Neste caso, a mais alta precedência possível seria *Int.max*.

1.4. A precedência somente afeta a descrição da pilha de *op* (ou seja, é uma informação extra que *description* precisa ter para que saiba como otimizar sua descrição da pilha). Não tem efeito na maneira como *evaluate ()* age. A precedência de avaliação é determinada pela ordem de coisas na pilha.

2. Desfazer

2.1. Verifique a tarefa com nota extra do trabalho 1 para apagar e pegue algumas dicas em como fazer a parte de apagar desta tarefa se você não fez.

2.2. Apagar/desfazer não devem desfazer a atribuição da variável M. Isso permite que os usuários que utilizem a calculadora computem o valor de M que eles queiram, então desfaçam para ter a expressão com a qual estavam trabalhando que usa M.

2.3. Provavelmente você desejará adicionar alguma API, não privada à classe *CalculatorBrain* (embora a implementação dela seja de apenas uma ou duas linhas).

2.4. Se você implementou um código limpo até agora, esta tarefa pode provavelmente ser feita com um único método na sua classe *ViewController* com meia dúzia de linhas de código ou menos. Se está maior do que isso, tente entender o porquê e considere reorganizar o resto do código antes.

3. Relatar erros

3.1. Que tipo de estrutura de dados é para situações como esta?

3.2. O relato de erros não dever ter efeito em description.

3.3. É provável que você precise ter um meio para *evaluateAndReportErrors ()* pedir a *BinaryOperation* e *UnaryOperation* que erro (se houve) seria gerado se certos operandos fossem passados.

3.4. Uma maneira de fazer isso é ter um valor associado para *BinaryOperations* e

UnaryOperations que é uma função que analisa argumentos potenciais e retorna uma *String* de erro apropriada se a realização de tal operação gera um erro (ou nil, caso contrário). Há várias outras maneiras de fazer isso, mas esta é uma seção de dicas...

3.5. A maioria das operações não pode relatar qualquer erro, então você desejará facilitar a criação de uma operação que relata nenhum erro. Se você utilizar as dicas acima, então passar *nil* como a função de teste de erro seria uma maneira simples de fazer isso. Você pode fazer um tipo inteiro de função ser um *Optional* colocando o tipo da função em parênteses e então inserindo o ponto de interrogação depois. Por exemplo, *((Double, Double) -> String?)?* é uma função *Optional* que recebe dois parâmetros *Double* e retorna uma *String Optional*.

3.6. Não esqueça da sintaxe e encadeamento de *Optional*. Por exemplo, você poderia chamar uma função *Optional* de *errorTest* assim:

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

O *if* falharia se errorTest em si fosse *nil* ou se ela não é *nil*, mas a função *errorTest* referencia retornos que são *nil*. Isso é conveniente.

3.7. Na sua classe *ViewController*, você pode achar que implementar uma nova variável chamada *displayResult* (que manipula ambos, valores e erros), e então re-implementar *displayValue* em termos dessa nova variável, fará o seu código ficar mais limpo. Ou talvez não. Isso é apenas uma dica e você pode decidir tomar outra decisão.

3.8. Quatro grandes casos de teste são: divisão por zero, raiz quadrada de um número negativo, operandos insuficientes e a variável não tem valor atribuído.

3.9. Se voê pensar nas ramificações de MVC para relatar um erro, há um bom argumento para relatá-los como um código de erro do *Model* (a parte M de MVC) para o *Controller* (a parte C). O *Controller* então seria responsável por comunicar o erro para o usuário através da *View* (a parte V). Por exemplo, o *Controller* poderia comunicar o erro na linguagem nativa do usuário. Entretanto, para fazer com que esta atividade para nota extra fique mais simples, você pode apenas retornar erros como *Strings* do seu *Model* e apresentar estas *Strings* ao usuário diretamente. A propósito, alguém pode discutir que enquanto as *Strings* de erros não forem *private* na classe *CalculatorBrain*, elas não podem ser consideradas códigos de erros (e poderiam ser traduzidas ou manipuladas de qualquer forma pelo Controller).