

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
DCC019 - Linguagem de Programação

CriptoGame em Haskell

Grupo

André Dias Nunes — MAT: 201665570C

Gabriel Di iorio Silva — MAT: 201765551AC

Professor

Leonardo Vieira dos Santos Reis

Juiz de Fora

Julho de 2022

Sumário

| | | |
|----------|---------------------------------|----------|
| 1 | Introdução | 2 |
| 2 | Interface e Arquitetura | 2 |
| 3 | Segredo | 3 |
| 4 | Validação | 3 |
| 5 | Turnos | 4 |
| 6 | Contabilização de pontos | 4 |
| 6.1 | Acertos Completos | 5 |
| 6.2 | Acertos Parciais | 5 |
| 7 | Testes | 6 |
| 8 | Instruções | 6 |

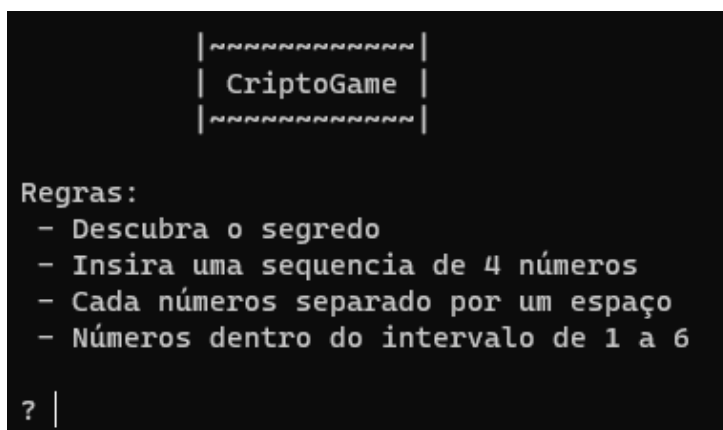
1 Introdução

O objetivo deste trabalho é implementar, em *Haskell*, o jogo Cripto Game. Ao iniciar o programa, o usuário irá digitar os chutes para o segredo. Após cada chute, o programa irá exibir na tela a quantidade de acertos completos e parciais e, seguirá neste processo até que a sequência correta seja encontrada. Sua implementação deverá fazer uma validação da entrada digitada pelo usuário, verificando se há valores fora do intervalo digitado ou quantidade incorreta de números.

Os capítulos a seguir representaram as etapas do desenvolvimento.

2 Interface e Arquitetura

Por se tratar de um jogo, decidimos criar uma apresentação ao iniciar, para melhorar a interação com o usuário, contendo o nome e as regras do jogo. Representado na figura 1.



```
      | ~~~~~~ |  
      | CriptoGame |  
      | ~~~~~~ |  
  
Regras:  
- Descubra o segredo  
- Insira uma sequencia de 4 números  
- Cada números separado por um espaço  
- Números dentro do intervalo de 1 a 6  
  
? |
```

Figura 1: Apresentação inicial do jogo

Quanto a arquitetura do projeto, optamos por modularizar em dois arquivos:

- `Main.hs`: responsável pela *interface* e criação do segredo randomizado.
- `Game.hs`: responsável pelo jogo, sendo o *loop* de turnos, validação do input recebido pelo usuário, contabilização de acertos (completos) e parciais, até que por fim, declarar a vitória.

3 Segredo

O objetivo do jogo é que o jogador descubra o segredo gerado, este segredo precisa ser randomizado a cada novo jogo. Para facilitar a comparação com as respostas do jogador, optamos que este segredo fosse uma *string*, para isso utilizamos dois passos:

- Uma variável recebe uma lista de inteiros da função `randomList` da figura 2. Esta função recebe um tamanho e gera uma lista de inteiros no intervalo de [1..6]. Esta função utiliza a biblioteca `System.Random`.
- A seguir, outra variável recebe a conversão da lista anterior para string, com seus elementos intercalados por espaço. Função `listToString` da figura 2

```
-- Gera lista de números aleatórios
randomList :: Int -> IO([Int])
randomList 0 = return []
randomList n = do
    r <- randomRIO (1,6)
    rs <- randomList (n-1)
    return (r:rs)

-- Transforma lista para string
listToString :: (Show a) => [a] -> String
listToString [] = ""
listToString [x] = (show x)
listToString (x:xs) = (show x) ++ " " ++ listToString xs
```

Figura 2: Funções para criação do segredo

4 Validação

A resposta do usuário (*input*) precisa obedecer às regras do jogo, dessa forma a cada interação é necessário verificar este *input*. Para isso cria-se uma variável que recebe um *bool* da função `validate` tendo o input como parâmetro. Nesta função, primeiro é verificado se o tamanho do *input* difere de 7, já que este é o tamanho esperado. Caso esteja correto, então é verificado cada *char* da *string*, onde os *chars* das posições que deveriam ser números é verificado se está no intervalo de '1' a '6' e os *chars* das posições que deveriam ser espaços se são espaços.

Caso o `validate` retorne *False*, para melhorar a interação com o jogador, a mensagem da figura 3 é exibida para informar que a resposta foi inválida. Optamos por não contabilizar

respostas inválidas como tentativas. Caso retorne *True*, então é passa para a etapa de análise se a resposta foi correta.

```
? 5 6
Resposta inválida.
Sendo "n" um número entre 1 e 6. Insira no formato: ? n n n n

? 5   5 6
Resposta inválida.
Sendo "n" um número entre 1 e 6. Insira no formato: ? n n n n
```

Figura 3: Output para respostas inválidas

5 Turnos

Em *haskell*, variáveis são imutáveis, então como contabilizar a quantidade de turnos, acertos, parciais ou alterar *booleans*? Para isso utilizamos a biblioteca *Data.IORef* que, basicamente, cria referências mutáveis *IO monad*.

Assim, inicializamos um novo *IORef*, com a função *newIORef*, antes de iniciar o jogo, responsável pela contagem de turnos, onde a cada resposta válida é incrementado nele o total de tentativas, através da função *modifyIORef'*. Para exibi-lo no console, convertemos seu valor para um inteiro utilizando a função *readIORef'*.

6 Contabilização de pontos

Caso a resposta do jogador seja válida, é verificado se essa resposta é igual ao segredo. Se sim, então o jogo é encerrado sendo exibido o número de tentativas, se não, a função intermediária *check* é chamada para iniciar o processo de contabilização de acertos completos e parciais. Para isso é criado 4 *IORef* com valor *bool True*, denominados *open*, para representar a condição de cada número do segredo, onde *open = True* significa que a posição está aberta, ou seja, não é um acerto completo. Em seguida, é criado outros 4 *IORef* com valor *bool False*, denominados *marked*, também para representar a condição de cada número do segredo, mas agora *marked = True* significa que aquela posição foi marcada como parcial para outro número.

Esses estados serão enviados como parâmetros para as funções a seguir, para ser calculado o total de acertos completos e parciais.

6.1 Acertos Completos

A função `hitPosition` recebe então a posição do número na *string*, seu estado `open`, o *input*, o segredo e o contador de acertos completos. Se os números na mesma posição de ambas as *strings* forem iguais, então o estado `open` é reescrito como *False* (fechado), ou seja, é um acerto completo. Em seguida é incrementado o contador de acertos completos. A figura 4 apresenta a função.

```
-- Função para encontrar acertos
hitPosition :: Int -> String -> String -> IORef Integer -> IORef Bool -> IO ()
hitPosition pos secret input hits open = do
    if(input !! pos == secret !! pos)           -- Se input[pos] == secret[pos]
    then do
        writeIORef open False                  -- Muda estado da posição para fechado
        modifyIORef hits (+1)                  -- Incrementa o hits
    else return ()
```

Figura 4: Função para encontrar acertos completos

6.2 Acertos Parciais

A função `partialPosition` é um pouco mais extensa. Ela recebe todos os estados `open` e `marked`, a posição que esta sendo analisada, o *input*, o segredo e o contador de acertos parciais. A lógica empregada segue os seguintes passos:

- Para facilitar a visualização é criado variáveis que recebem cada caractere das *strings* de *input* e segredo. A seguir, é criado variáveis que recebem o estado *bool* de `open` e `marked` para poderem ser utilizadas nas condicionais.
- Então é reconhecido qual posição está sendo analisado e se ele já não foi definido como acerto completo, para compreensão chamarei essa posição de amostra.
- Se a amostra está aberto, então para cada uma das demais posições é verificado se a esta outra posição também está aberta, não marcada e se o número da amostra é igual ao número desta outra posição.
- Se a condição acima for verdadeira, então o *bool* referente a marcação desta outra posição é alterado para *True* e incrementado o contador de parciais.

7 Testes

Para exemplificar os testes, usaremos os exemplos fornecidos na especificação do trabalho e obtemos os mesmos resultados apresentados. As figuras 5 e 6 mostram os resultados.

```
? 1 1 2 2
0 Completo, 0 Parcial
? 3 3 4 4
1 Completo, 1 Parcial
? 3 5 3 6
2 Completo, 0 Parcial
? 3 4 6 6
Parabéns, você acertou após 4 tentativas
```

Figura 5: Teste para segredo "3 4 6 6"

```
? 1 2 3 4
1 Completo, 1 Parcial
? 1 3 5 6
0 Completo, 2 Parcial
? 5 2 1 5
2 Completo, 0 Parcial
? 5 1 1 4
Parabéns, você acertou após 4 tentativas
```

Figura 6: Teste para segredo "5 1 1 4"

8 Instruções

O programa foi testado utilizando o *WSL* com *Ubuntu* 20.04 e o compilador/interpretador utilizado foi *The Glorious Glasgow Haskell Compilation System, version 8.6.5*

Para compilar, basta utilizar o comando `ghc Main.hs` e para executar `ghci Main.hs`. A seguir para iniciar o jogo, basta utilizar o comando `main`.