



Segundo trabalho prático de Linguagens de Programação

Jogo Nim na linguagem Haskell

Sob orientação do Professor Leonardo Vieira dos Santos Reis

Departamento de Ciência da Computação
Universidade Federal de Juiz de Fora
Minas Gerais - Brasil
ERE 2021.3

Sumário

1	Especificação técnica do trabalho - Jogo Nim	1
1.1	Execução do programa	1
1.1.1	Pré-requisitos	1
1.1.2	Como instalar o módulo <i>random</i>	1
1.1.3	Como carregar o programa	2
1.2	Funções, módulos e bibliotecas do interpretador	2
1.2.1	Função <i>getStdRandom</i>	2
1.2.2	Função <i>randomR</i>	2
1.3	Decisões de projeto	2
1.4	Dificuldades enfrentadas	3
2	Funções auxiliares implementadas no programa	4
2.1	Função <i>board</i>	4
2.2	Função <i>getRandomInt</i>	4
2.3	Função <i>dec2bin</i>	4
2.4	Função <i>bin2int</i>	5
2.5	Função <i>getEachDigit</i>	5
2.6	Função <i>checkAllEven</i>	5
2.7	Função <i>dec2binlen3</i>	5
2.8	Função <i>printWhoIsPlaying</i>	6
2.9	Função <i>printGameInfo</i>	6
2.10	Função <i>askForNextPage</i>	7
3	Detalhamento do código implementado	8
3.1	Funções de menu	8
3.1.1	Função <i>selectDifficulty</i>	8
3.1.2	Função <i>gameMenu</i>	8
3.2	Funções da máquina	9
3.2.1	Função <i>computerTurn</i>	9
3.2.2	Função <i>randomComputerTurn</i>	9
3.2.3	Função <i>perfectComputerTurn</i>	10
3.3	Funções do jogo	10
3.3.1	Função <i>gameLoop</i>	10
3.3.2	Função <i>checkWin</i>	11
3.4	Funções do tabuleiro	12
3.4.1	Função <i>getLineValue</i>	12
3.4.2	Função <i>setLineValue</i>	12
3.4.3	Função <i>isPerfectBoard</i>	12
3.4.4	Função <i>printBoard</i>	13
4	Exemplos de funcionamento	14
4.1	Iniciar o jogo	14
4.2	Exemplo de funcionamento do jogo	15
5	Referências	16

1 Especificação técnica do trabalho - Jogo Nim

O presente documento tem por objetivo apoiar o projeto desenvolvido durante o segundo trabalho da disciplina DCC019 - Linguagens de Programação.

O programa foi implementado na linguagem *Haskell* e todo o código fonte foi inserido em um arquivo que simula o funcionamento de um tabuleiro com o jogo Nim[8]. A posição inicial do tabuleiro contendo 4 linhas é: 1 palito na primeira linha, 3 palitos na segunda, 5 e 7 palitos nas respectivas linhas restantes. Um exemplo da posição inicial do tabuleiro pode ser vista na figura (1) logo abaixo:

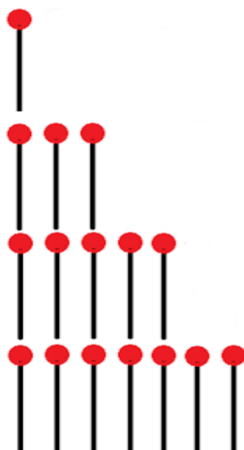


Figura 1: Posição inicial do tabuleiro de jogo

As citações dentro do texto estão organizadas de forma que as referências internas encontram-se entre parênteses e as externas entre colchetes. Todas elas são clicáveis de forma a encaminhar a navegação para o ponto exato do texto onde se encontram.

As caixas com código fonte foram feitas utilizando em suas representações a sintaxe descrita por Andrew Gibiansky[1] e com um padrão de colorização semelhante ao usado por Prajit Ramachandran[7].

1.1 Execução do programa

Abaixo seguem detalhes importantes para a correta execução do programa.

1.1.1 Pré-requisitos

Abaixo estão listados os pré-requisitos para o correto funcionamento do programa. Por favor, tenha certeza de que os atende para a correta execução do programa.

- Sistema operacional GNU/Linux ou Windows (10 ou superior)
- Compilador GHCi[5] (versão 9.0.1 ou superior)
- Módulo *random* (necessário para a geração de números aleatórios)

1.1.2 Como instalar o módulo *random*

Antes de instalar o módulo *random*, é necessário instalar o pacote *cabal*. Favor seguir as instruções disponíveis [clicando aqui](#) (é recomendado instalar o pacote dos repositórios de sua distribuição caso esteja usando Linux).

Então, num terminal atualizar a base de módulos usando:

```
cabal update
```

E por fim instalar o módulo *random* por meio do *cabal*:

```
cabal install random
```

Agradecimentos aos usuários *palik* e *danidiaz* pelas instruções mostradas em [6].

1.1.3 Como carregar o programa

O processo de carregamento do programa no sistema é bem simples: basta obter uma cópia do arquivo *main.hs* e então abri-lo usando o ambiente interativo, como por exemplo no comando abaixo:

```
ghci main.hs
```

Nota: Atenção às instruções da subseção [1.1.2](#).

Com o programa carregado, para iniciar sua execução, basta executar o comando abaixo:

```
ghci> main
```

Dessa forma o menu principal será automaticamente carregado, como pode ser visto na figura (2).

1.2 Funções, módulos e bibliotecas do interpretador

Esta seção descreve, brevemente, alguns dos predicados provenientes dos módulos *Haskell built-in* do interpretador GHCi[4] que foram utilizados no programa.

1.2.1 Função *getStdRandom*

Esta função recebe uma função e usa ela como semente para obter um valor do gerador global de números aleatórios, então, ela atualiza o gerador global com o novo gerador retornado pela função recebida como parâmetro.

Assinatura:

```
1 getStdRandom :: MonadIO m => (StdGen -> (a, StdGen)) -> m a
```

Listing 1: Assinatura da função *getStdRandom*

Para mais informações, favor consultar o manual disponível em [2]

1.2.2 Função *randomR*

Esta função recebe um intervalo (min,max) e um gerador de números pseudo-aleatórios g, e retorna um número pseudo-aleatório uniformemente distribuído dentro do intervalo fechado [min,max] e um novo gerador.

Assinatura:

```
1 randomR :: RandomGen g => (a, a) -> g -> (a, g)
```

Listing 2: Assinatura da função *randomR*

Para mais informações, favor consultar o manual disponível em [3]

1.3 Decisões de projeto

Nesta seção estão descritas algumas das decisões de projeto tomadas durante a implementação do programa em *Haskell*.

Foi decidido representar o tabuleiro de jogo (*board*) por meio de uma lista contendo valores que representavam a quantidade de palitos (*sticks*) presentes. Cada posição da lista (iniciada em 0), representa uma linha do tabuleiro do jogo.

Cada jogada é feita subtraindo-se a quantidade de palitos de uma determinada linha. A nova configuração do tabuleiro é salva criando-se estados para a variável *board* em tempo de execução.

A máquina joga, como requisitado, de dois modos diferentes: no modo fácil, ela sorteia dois números aleatórios, um entre 0 e 3 e outro entre 1 e 7, para a linha a ser escolhida e para a quantidade de palitos a ser retirada daquela linha, respectivamente; Já no modo difícil, é verificado o estado atual do tabuleiro: caso o tabuleiro já esteja em um estado perfeito, a máquina faz uma jogada aleatória, se não, ela faz sempre uma jogada perfeita de acordo com a regra de estratégia perfeita do requisito 3.1.

Por fim, foi priorizada a modularização do código, exceto, por questão de simplificação, para o loop principal (função *gameLoop*[18](#)) onde se encontra quase toda a lógica por trás da execução do jogo.

1.4 Dificuldades enfrentadas

Durante o processo de desenvolvimento da solução, como era de se esperar, algumas dificuldades foram encontradas. As quatro maiores dificuldades enfrentadas neste trabalho, em ordem de tempo despendido para sua solução, foram:

1. implementação da função *randomComputerTurn* / a inteligência da máquina(16,3.2)
2. implementação da função *gameLoop*(18)
3. representação do tabuleiro
4. trabalhar com os tipos em *Haskell*

A implementação da função *randomComputerTurn*(16) não chegou a ser uma das mais complicadas, entretanto, descobrir uma boa forma de gerar um número (pseudo-)aleatório foi um tanto desafiador já que há a necessidade de se importar um módulo que não está incluído por padrão no compilador nem no pacote *master*. A implementação da inteligência da máquina foi o que mais exigiu descobrir e calcular métodos para que o funcionamento do jogo não fosse prejudicado, com destaque para a idealização da função *isPerfectBoard*(22), que possivelmente, foi a função que mais exigiu planejamento pré-implementação.

Então, a implementação da função *gameLoop*(18) foi um pouco mais custosa que as demais, muito provavelmente porque ela contém diversos condicionais (necessários para implementar as regras e a mecânica do jogo). Talvez tivesse sido melhor refatorar o código de modo a aplicar uma melhor modularização neste, o que favoreceria a sua legibilidade, entretanto, isso aumentaria a complexidade do código (principalmente considerando a característica de forte tipagem da linguagem *Haskell*) e ainda sim o deixaria fortemente acoplado.

Já decidir sobre como implementar a representação do tabuleiro do jogo Nim foi desafiador, pois o documento de requisitos, provavelmente propositalmente, deixou essa questão em aberto e, devido à seção 3 dele, dá-se a impressão de que a melhor forma de representar o tabuleiro seria por meio de uma matriz, entretanto, apesar de visualmente mais agradável, essa representação poderia levar a uma dificuldade extra durante a implementação das operações necessárias para a correta simulação do jogo.

Por fim, como foi a minha primeira vez implementando um programa completo do zero em *Haskell*, além da necessidade em me acostumar com o novo paradigma (que agora era o funcional), trabalhar com uma linguagem fortemente tipada foi um desafio considerável, e muitas vezes, a curva de aprendizado somada com o tempo disponível para o processo de aprendizado tornaram a fase inicial extremamente lenta e até certo ponto, complicada.

2 Funções auxiliares implementadas no programa

Nesta seção estão descritas todas as funções, na ordem em que aparecem no código fonte, presentes na implementação do jogo e que são necessárias para o programa funcionar corretamente.

2.1 Função *board*

Esta função é a responsável por criar o estado inicial do tabuleiro do jogo, logo, ela gera uma lista contendo os números 1, 3, 5 e 7.

Parâmetros de entrada da função:

(Não há, funcionamento semelhante ao de uma variável do paradigma imperativo)

Código em *Haskell*:

```
1 board :: [Int]
2 board = [1, 3, 5, 7]
```

Listing 3: Código da função *board*

2.2 Função *getRandomInt*

Esta função lê dois números inteiros. Então, os torna os limites inferior e superior, respectivamente, das funções que gerarão um número inteiro de forma (pseudo-)randômica.

Parâmetros de entrada da função:

- x: Representa o limite inferior para a função de números (pseudo-)randômicos
- y: Representa o limite superior para a função de números (pseudo-)randômicos

Código em *Haskell*:

```
1 getRandomInt :: Int -> Int -> IO Int
2 getRandomInt x y = getStdRandom (randomR (x, y))
```

Listing 4: Código da função *getRandomInt*

2.3 Função *dec2bin*

Esta função lê um número inteiro e então o transforma em binário, retornando uma lista de dígitos binários na ordem correta da representação binária (da direita - menos significativa - para esquerda - mais significativa).

Parâmetros de entrada da função:

- n: Representa o número decimal a ser convertido

Código em *Haskell*:

```
1 dec2bin :: Int -> [Int]
2 dec2bin 0 = [0]
3 dec2bin n = reverse (dec2binAux n) --reverse to avoid adding 0s at the end and to get the
   right order
4
5 dec2binAux 0 = []
6 dec2binAux n
7   | n `mod` 2 == 1 = 1 : dec2binAux (n `div` 2)
8   | n `mod` 2 == 0 = 0 : dec2binAux (n `div` 2)
```

Listing 5: Código da função *dec2bin*

2.4 Função *bin2int*

Esta função lê uma lista de dígitos binários e então os transforma em um número inteiro.

Parâmetros de entrada da função:

- *monad* x: Representa o número binário a ser convertido

Código em *Haskell*:

```
1 bin2int :: [Int] -> Int
2 bin2int = foldl1 (\x y -> x * 10 + y) 0
```

Listing 6: Código da função *bin2int*

2.5 Função *getEachDigit*

Esta função lê um número inteiro e então o separa em uma lista de dígitos inteiros.

Parâmetros de entrada da função:

- n: Representa o número inteiro a ser particionado

Código em *Haskell*:

```
1 getEachDigit :: Int -> [Int]
2 getEachDigit 0 = []
3 getEachDigit n = getEachDigitAux n : getEachDigit (n `div` 10)
4
5 getEachDigitAux 0 = 0
6 getEachDigitAux n = n `mod` 10
```

Listing 7: Código da função *getEachDigit*

2.6 Função *checkAllEven*

Esta função lê uma lista de números inteiros e então os testa para saber se todos são pares.

Parâmetros de entrada da função:

- x: Representa a cabeça da lista de números inteiros a serem testados
- xs: Representa a cauda da lista de números inteiros a serem testados

Código em *Haskell*:

```
1 checkAllEven :: [Int] -> Bool
2 checkAllEven [] = True
3 checkAllEven (x : xs)
4   | even x = checkAllEven xs
5   | otherwise = False
```

Listing 8: Código da função *checkAllEven*

2.7 Função *dec2binlen3*

Esta função lê um número inteiro, o converte em binário e então o separa em uma lista de dígitos binários, sendo que, caso eles não estejam no formato de comprimento igual a 3, adiciona zeros até que se enquadrem nele.

Parâmetros de entrada da função:

- n: Representa o número inteiro a ser convertido em binário

Código em *Haskell*:

```

1 dec2binlen3 :: Int -> [Int]
2 dec2binlen3 n = do
3   let bin = dec2bin n
4   if length bin < 3
5     then do
6       let bin' = replicate (3 - length bin) 0 ++ bin --adds zeros to the left if necessary
7       bin'
8     else do
9       bin

```

Listing 9: Código da função *dec2binlen3*

2.8 Função *printWhoIsPlaying*

Esta função lê um booleano, que no caso determina se o player é humano ou não e então imprime uma mensagem representando o jogador atual.

Parâmetros de entrada da função:

- p: Representa o estado do jogador atual

Código em *Haskell*:

```

1 printWhoIsPlaying p = do
2   if p
3     then putStrLn "-> Human Player is playing"
4     else putStrLn "-> Computer is playing"

```

Listing 10: Código da função *printWhoIsPlaying*

2.9 Função *printGameInfo*

Esta função imprime as informações do jogo, quando solicitadas pelo jogador.

Parâmetros de entrada da função:

- n: Representa o número da página a ser mostrada

Código em *Haskell*:

```

1 printGameInfo :: Int -> IO ()
2 printGameInfo n
3   | n == 0 = do
4     putStrLn "\n|-----|"
5     putStrLn "|           Game information           |"
6     putStrLn "|-----|"
7     putStrLn "- The game is a two-player game."
8     putStrLn "- The game is played on a board with 4 piles of stickers on each one."
9     putStrLn "- The goal is to remove all the stickers from the board."
10    putStrLn "\n=> Rules:\n"
11    putStrLn "- The game is played in rounds, one player plays per turn."
12    putStrLn "- The player who takes the last sticker from the board wins the game.\n"
13    nextPage <- askForNextPage n
14    printGameInfo nextPage
15  | n == 1 = do
16    putStrLn "\n=> Back-end:\n"
17    putStrLn "- The board is represented by a list of 4 integers, each representing the number
18      of stickers on each pile."
19    putStrLn "- The line number is the index of the list, starting from 0."
20    putStrLn "- The player is represented by a boolean, True for the human player and False
21      for the machine."
22    putStrLn "- There are two levels available: easy and hard."
23    putStrLn "--Easy mode: the human player starts the game."
24    putStrLn "--Hard mode: the machine starts the game and then, if possible, only makes
25      perfect moves.\n"

```



```

23     nextPage <- askForNextPage n
24     printGameInfo nextPage
25 | n == 2 = do
26     putStrLn "\n=> Board:\n"
27     putStrLn "- The board is displayed containing:"
28     putStrLn "--The number of the line of the board;"
29     putStrLn "--The number of stickers on each pile; and"
30     putStrLn "--The simulated physical representation of each sticker."
31     putStrLn "\n=> Example:\n"
32     putStrLn "- If the board is the list [1, 3, 5, 7], it will be represented as below:\n"
33     putStrLn "0 ( 1 ): |"
34     putStrLn "1 ( 3 ): |||"
35     putStrLn "2 ( 5 ): |||||"
36     putStrLn "3 ( 7 ): |||||||"
37     putStrLn "\nGood luck!\n"
38     nextPage <- askForNextPage n
39     printGameInfo nextPage
40 | otherwise = gameMenu --goes back to the main menu when finished

```

Listing 11: Código da função *printGameInfo*

2.10 Função *askForNextPage*

Esta função recebe um número inteiro, aguarda por um "ENTER" e depois retorna o sucessor desse número.

Parâmetros de entrada da função:

- pageNumber: Representa um número de página

Código em *Haskell*:

```

1 askForNextPage :: Int -> IO Int
2 askForNextPage pageNumber = do
3     putStrLn "Press enter to continue..."
4     getLine
5     return (pageNumber + 1)

```

Listing 12: Código da função *askForNextPage*

3 Detalhamento do código implementado

Nesta seção estão descritas as funções alvo do processo de avaliação dentro do programa e que foram usadas para implementação dos objetivos presentes no documento de requisitos do trabalho.

3.1 Funções de menu

Essa seção contém os detalhes de implementação das funções relacionadas à configuração de alguns parâmetros que influenciam no funcionamento do jogo.

3.1.1 Função *selectDifficulty*

Esta função lê a entrada dada pelo jogador para ajustar a dificuldade das ações da máquina de acordo com a opção selecionada.

Parâmetros de entrada da função:

(Não há, ocorre somente leitura da entrada padrão)

Código em *Haskell*:

```
1 selectDifficulty :: IO Int
2 selectDifficulty = do
3   putStrLn "Select difficulty: "
4   putStrLn "1. Easy"
5   putStrLn "2. Hard"
6   putStrLn "0. Exit"
7   putStr "> "
8   difficulty <- getLine
9   let d = (read difficulty :: Int)
10  if d == 0 -- quits the game
11    then return 99
12    else return d
```

Listing 13: Código da função *selectDifficulty*

3.1.2 Função *gameMenu*

Esta função lê a entrada dada pelo jogador e, caso autorizado por ele, chama a função (13) que serve para ajustar a dificuldade das ações da máquina. Ou ainda é possível que seja chamada a função (11) que exibe algumas informações sobre o jogo.

Parâmetros de entrada da função:

(Não há, ocorre somente leitura da entrada padrão)

Código em *Haskell*:

```
1 gameMenu :: IO ()
2 gameMenu = do
3   putStrLn "Welcome to Nim!\n"
4   putStrLn "Are you ready?"
5   putStrLn "1. Yes, let's play!"
6   putStrLn "9. Hold on, give me some information about the game first"
7   putStrLn "0. Not yet, exit"
8   putStr "> "
9   option <- getLine
10  let selectedOption = (read option :: Int)
11  if selectedOption == 1
12    then do
13      --evaluate difficulty levels
```

```

14     level <- selectDifficulty
15     if level == 1 --easy
16     then gameLoop board True False --human starts, machine is in easy mode
17     else
18         if level == 2 --hard
19         then gameLoop board False True --the human does not start, machine is in hard mode
20         else
21             if level == 99 -- quits the game
22             then putStrLn "Bye!"
23             else putStrLn "Invalid option"
24     else
25         if selectedOption == 9
26         then do
27             printGameInfo 0 --prints the game information, starting from the first page
28             else putStrLn "Bye!"

```

Listing 14: Código da função *gameMenu*

3.2 Funções da máquina

Essa seção contém os detalhes de implementação das funções relacionadas ao controle das ações da máquina e da sua "inteligência".

3.2.1 Função *computerTurn*

Esta função lê um tabuleiro e uma *flag*. Então, a depender do valor da *flag*, ela chama uma (16) ou outra (17) sequência de funções que funcionam como auxiliares para as decisões de jogada da máquina.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) onde deverá ocorrer a jogada
- godMode: Representa a *flag* de controle do *hard mode*

Código em *Haskell*:

```

1 computerTurn :: [Int] -> Bool -> IO [Int]
2 computerTurn board godMode = do
3     if godMode --checks if it is in the hard mode
4     then do
5         if isPerfectBoard board
6         then randomComputerTurn board
7         else perfectComputerTurn board
8     else randomComputerTurn board --makes a move taking a random number of stickers at a time
        on a random line

```

Listing 15: Código da função *computerTurn*

3.2.2 Função *randomComputerTurn*

Esta função lê um tabuleiro, sorteia números pseudo-randômicos e checa se a jogada é possível ou não. Caso seja, retorna o novo estado do tabuleiro, caso não, tenta outra jogada.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) onde deverá ocorrer a jogada

Código em *Haskell*:

```

1 randomComputerTurn :: [Int] -> IO [Int]
2 randomComputerTurn board = do
3     line <- getRandomInt 0 3
4     quantityToRemove <- getRandomInt 1 7
5     let lineOldValue = getLineValue board line

```

```

6  if (lineOldValue /= 0) && (quantityToRemove <= lineOldValue) -- checks if the line still has
    stickers and if the quantity to remove is less than the line's value
7  then do
8      -- if yes, removes the random number of stickers
9      let lineNewValue = lineOldValue - quantityToRemove
10     let newBoard = setLineValue board line lineNewValue
11     return newBoard
12 else do
13     -- if no, tries again with other values
14     randomComputerTurn board

```

Listing 16: Código da função *randomComputerTurn*

3.2.3 Função *perfectComputerTurn*

Esta função lê um tabuleiro, chama a função que gera uma jogada randômica (16) e checa se a jogada faz parte da estratégia perfeita ou não. Caso seja, retorna o novo estado do tabuleiro, caso não, tenta outra jogada.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) onde deverá ocorrer a jogada

Código em *Haskell*:

```

1 perfectComputerTurn :: [Int] -> IO [Int]
2 perfectComputerTurn board = do
3     newBoard <- randomComputerTurn board
4     if isPerfectBoard newBoard
5     then return newBoard
6     else perfectComputerTurn board

```

Listing 17: Código da função *perfectComputerTurn*

3.3 Funções do jogo

Essa seção contém os detalhes de implementação das principais funções relacionadas à mecânica de execução do jogo e suas regras de funcionamento.

3.3.1 Função *gameLoop*

Esta função lê um tabuleiro, que será aquele utilizado durante a execução do jogo, e mais dois booleanos que funcionam como *flags* de controle de estado. É a maior e principal função do programa e implementa as regras e mecânicas base do jogo.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) do jogo
- player: Representa quem é o jogador de determinada rodada
- machineGodMode: Representa o nível de dificuldade da máquina

Código em *Haskell*:

```

1 gameLoop :: [Int] -> Bool -> Bool -> IO ()
2 gameLoop board player machineGodMode = do
3     putStrLn "\nBoard:"
4     printBoard board
5     putStrLn "" --displays a new line for better readability
6     printWhoIsPlaying player
7     if (player == False) --computer's turn
8     then do
9         board <- computerTurn board machineGodMode
10        if checkWin board

```

```

11     then do
12         putStrLn "Board:\n"
13         printBoard board
14         putStrLn "\n\n0h no!...\n\n---The Computer wins this time---\n"
15         putStrLn "Going back to the main menu...\n"
16         gameMenu
17     else gameLoop board (not player) machineGodMode
18 else do
19     -- player's turn
20     putStrLn "Enter the line number you choose:"
21     putStr "> "
22     number <- getLine
23     let lineNumber = read number :: Int
24     putStrLn "Enter a how many sticks to take:"
25     putStr "> "
26     quantity2 <- getLine
27     let quantity = read quantity2 :: Int
28     if quantity <= 0 --checks if the quantity is valid
29     then do
30         putStrLn "Invalid quantity, please select at least 1 stick on each turn"
31         gameLoop board player machineGodMode
32     else do
33         let lineVal = getLineValue board lineNumber
34         if lineNumber >= 0 && lineNumber < 4 --checks if the line exists
35         then do
36             if lineVal >= quantity --checks if the stickers exist
37             then do
38                 let newLineValue = subtract quantity lineVal
39                 let newBoard = setLineValue board lineNumber newLineValue
40                 if checkWin newBoard
41                 then do
42                     putStrLn "Board:\n"
43                     printBoard newBoard
44                     putStrLn "\n\n0h yeah!\n\n!!!You win!!!\n"
45                     putStrLn "Going back to the main menu...\n"
46                     gameMenu
47                 else do
48                     gameLoop newBoard (not player) machineGodMode
49             else do
50                 putStrLn "Invalid move, quantity entered is greater than the number of
sticks in this line"
51                 gameLoop board player machineGodMode
52             else do
53                 putStrLn "Invalid line number, please select a number between 0 and 3"
54                 gameLoop board player machineGodMode

```

Listing 18: Código da função *gameLoop*

3.3.2 Função *checkWin*

Esta função lê um tabuleiro e testa se este representa um estado final do jogo, retornando o resultado da avaliação.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) que deve ser testado

Código em *Haskell*:

```

1 checkWin :: [Int] -> Bool
2 checkWin board = all (== 0) board --if the sticks are all gone, the game is over

```

Listing 19: Código da função *checkWin*

3.4 Funções do tabuleiro

Essa seção contém os detalhes de implementação das funções estritamente relacionadas ao tabuleiro do jogo.

3.4.1 Função *getLineValue*

Esta função lê e retorna o valor de uma linha de um tabuleiro de jogo.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) que deve ser lido
- n: Representa o número da linha que foi anteriormente escolhida em *board*

Código em *Haskell*:

```
1 getLineValue :: [Int] -> Int -> Int
2 getLineValue board n = board !! n
```

Listing 20: Código da função *getLineValue*

3.4.2 Função *setLineValue*

Esta função altera a linha de um tabuleiro para que esta represente o novo estado do jogo após uma jogada ter sido efetuada.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) que deve ser alterado
- n: Representa o número da linha que foi anteriormente escolhida em *board*
- val: Representa a nova quantidade de palitos (*sticks*) da linha *n*

Código em *Haskell*:

```
1 setLineValue :: [Int] -> Int -> Int -> [Int]
2 setLineValue board n val =
3   let (x, _ : y) = splitAt n board
4   in x ++ [val] ++ y
```

Listing 21: Código da função *setLineValue*

3.4.3 Função *isPerfectBoard*

Esta função testa um tabuleiro para saber se este já se encontra num estado dentro da estratégia perfeita.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) que deve ser testado

Código em *Haskell*:

```
1 isPerfectBoard :: [Int] -> Bool
2 isPerfectBoard board = do
3   --split the board in 4 lines
4   let item1 = head board
5   let item2 = board !! 1
6   let item3 = board !! 2
7   let item4 = board !! 3
8   --checks how much sticks are in each line and converts them to binary
9   let binItem1 = dec2binlen3 item1
10  let binItem2 = dec2binlen3 item2
11  let binItem3 = dec2binlen3 item3
12  let binItem4 = dec2binlen3 item4
```

```

13  --get the digits together
14  let columnSum = bin2int binItem1 + bin2int binItem2 + bin2int binItem3 + bin2int binItem4
15  --puts all digits in a list
16  let columnSumDigits = getEachDigit columnSum
17  --checks if all digits are even numbers
18  checkAllEven columnSumDigits

```

Listing 22: Código da função *isPerfectBoard*

3.4.4 Função *printBoard*

Esta função retorna a situação atual do tabuleiro, imprimindo-o na tela.

Parâmetros de entrada da função:

- board: Representa o tabuleiro (*board*) que deve ser impresso

Código em *Haskell*:

```

1 printBoard :: [Int] -> IO ()
2 printBoard board = putStr $ unlines [show rowNumber ++ " " ++ "( " ++ show stickers ++ " )" ++
    ": " ++ replicate stickers '|' | (stickers, rowNumber) <- zip board [0 .. length board]]

```

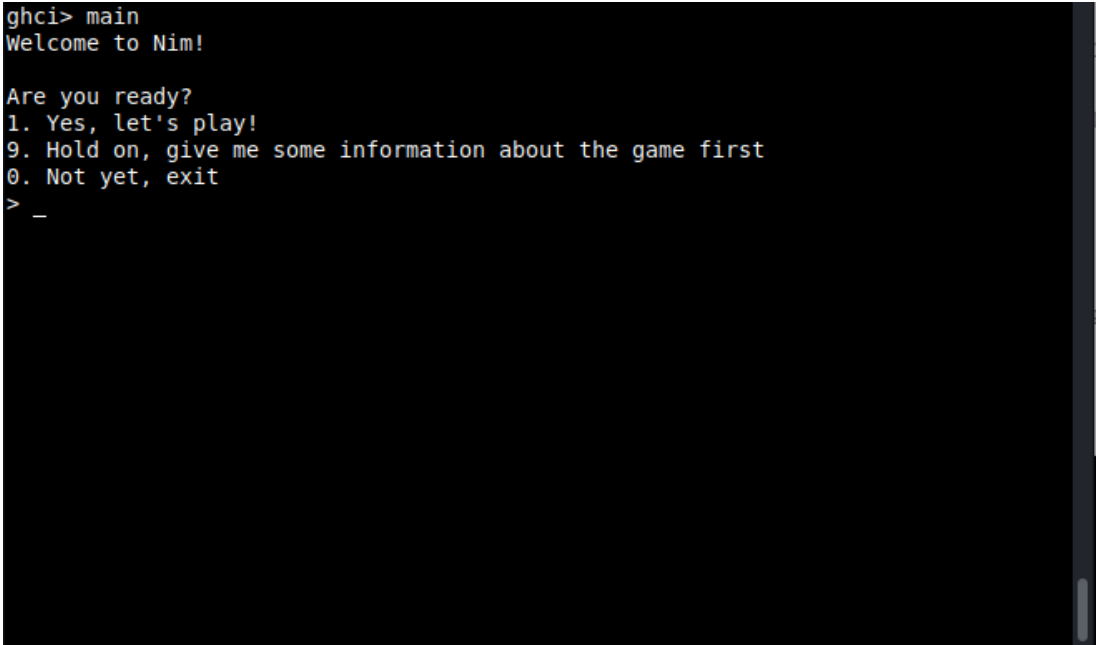
Listing 23: Código da função *printBoard*

4 Exemplos de funcionamento

Esta seção contém exemplos de funcionamento do programa e de algumas de suas funções.

4.1 Iniciar o jogo

Depois de ser carregado, como pode ser visto na figura (2), assim que o jogo é iniciado, o menu principal é aberto e aguarda uma resposta do jogador.

A screenshot of a terminal window with a black background and white text. The text shows the execution of a Haskell program in the GHCi environment. The user enters 'main', and the program outputs 'Welcome to Nim!'. It then asks 'Are you ready?' and displays a menu with three options: '1. Yes, let's play!', '9. Hold on, give me some information about the game first', and '0. Not yet, exit'. The prompt '>' is shown, followed by a hyphen '-' which is the user's input.

```
ghci> main
Welcome to Nim!

Are you ready?
1. Yes, let's play!
9. Hold on, give me some information about the game first
0. Not yet, exit
> -
```

Figura 2: Programa aguardando a autorização do jogador humano para iniciar a execução

4.2 Exemplo de funcionamento do jogo

Abaixo, seguem as figuras (3) e (4) que mostram o início da dinâmica do jogo e o estado dele após uma primeira rodada ser concluída.

```
ghci> main
Welcome to Nim!

Are you ready?
1. Yes, let's play!
9. Hold on, give me some information about the game first
0. Not yet, exit
> 1
Select difficulty:
1. Easy
2. Hard
0. Exit
> 1

Board:
0 ( 1 ): |
1 ( 3 ): |||
2 ( 5 ): |||||
3 ( 7 ): |||||

-> Human Player is playing
Enter the line number you choose:
> -
```

Figura 3: Programa aguardando a primeira jogada do jogador humano

```
-> Human Player is playing
Enter the line number you choose:
> 3
Enter a how many sticks to take:
> 6

Board:
0 ( 1 ): |
1 ( 3 ): |||
2 ( 5 ): |||||
3 ( 1 ): |

-> Computer is playing

Board:
0 ( 1 ): |
1 ( 3 ): |||
2 ( 3 ): |||
3 ( 1 ): |

-> Human Player is playing
Enter the line number you choose:
> -
```

Figura 4: Estado do programa após a primeira rodada de jogo

5 Referências

- [1] Andrew Gibiansky. *Intro to Haskell Syntax*. 2014. URL: <https://andrew.gibiansky.com/blog/haskell/haskell-syntax/>.
- [2] The University of Glasgow. *System.Random*. Acesso em 10 de Janeiro de 2022. URL: <https://hackage.haskell.org/package/random-1.2.1/docs/System-Random.html#v:getStdRandom>.
- [3] The University of Glasgow. *System.Random*. Acesso em 10 de Janeiro de 2022. URL: <https://hackage.haskell.org/package/random-1.2.1/docs/System-Random.html#v:randomR>.
- [4] University of Glasgow et al. *The Glasgow Haskell Compiler*. Acesso em 09 de Janeiro de 2022. URL: <https://www.haskell.org/ghc/>.
- [5] Glasgow Haskell Compiler User's Guide. *Obtaining GHC*. Acesso em 09 de Janeiro de 2022. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/intro.html#obtaining-ghc.
- [6] hautor. *How to install modules*. Acesso em 10 de Janeiro de 2022. 2020. URL: <https://discourse.haskell.org/t/how-to-install-modules/1363/2>.
- [7] Prajit Ramachandran. *A Quick Tour of Haskell Syntax*. 2013. URL: <https://prajitr.github.io/quick-haskell-syntax/>.
- [8] Wikipedia. *Nim*. Acesso em 09 de Janeiro de 2022. URL: <https://en.wikipedia.org/wiki/Nim>.