LightBot - O jogo para programadores
Laboratórios de Informática I - LEI

Carlos Pereira - 61887 Rafael Braga - 61799

30 de Novembro de 2014

Resumo

Neste relatório apresenta-se uma implementação do jogo *LightBot*, realizada em *Haskell*. Este jogo consiste em acender as lâmpadas existentes num tabuleiro. Para isso, no início dá-se instruções ao robô que irá segui-las. O jogo termina quando todas as lâmpadas estão acesas ou quando não existem mais instruções para o robô.

Capítulo 1

Introdução

O presente trabalho consiste na descrição e explicação de três programas feitos em *Haskell*. Estes programas (ou tarefas) pretendem simular diferentes etapas do jogo *LightBot*. O objetivo do jogo é instruir o robô (a partir de uma série de comandos) de forma a que ele acenda todas as luzes do tabuleiro.

A primeira tarefa começa por ler um ficheiro de texto. Este ficheiro de texto consiste na descrição do tabuleiro do jogo, das coordenadas iniciais do robô e as instrução para o robô. O que o programa tem de fazer é verificar se cada linha de texto do ficheiro cumpre as regras do jogo. Por exemplo: não podem existir na linha de instruções, comandos que já não se encontram previamente definidos. Se o ficheiro for válido, o programa apresenta uma mensagem a indicá-lo. Caso contrário, apresenta uma mensagem de erro.

Na segunda tarefa supõe-se que o ficheiro de entrada é válido, sendo pedido que mandemos o robô executar a primeira ordem da linha de comandos, apresentando no ecrã (caso seja possível) as novas coordenadas do robô. Caso não seja possível executá-la, deve aparecer no ecrã uma mensagem de erro.

Depois, na terceira tarefa começa-se por assumir que o ficheiro de entrada é válido. Aqui, é-nos pedido que mandemos o robô executar todas as instruções na linha de comandos até que todas as lâmpadas estejam acesas ou até não existirem mais instruções. Sempre que o robô acender ou apagar uma luz, aparecerá no ecrã uma linha a indicar as coordenadas dessa lâmpada. Quando o robô acende todas as lâmpadas, é indicado no ecrã o número de comandos válidos que o robô executou.

A quarta tarefa, que é considerada por nós a mais desafiante, consiste em determinar as instruções a fornecer ao robô, dado um tabuleiro e uma posição inicial, de maneira a que ele acenda todas as lâmpadas existentes.

Por fim, a quinta tarefa, consiste no mesmo que a tarefa 3, com a excepção de que se tem de produzir um código *HTML* que apresente. Isto para que se possa visualizar o *Lightbot* numa página *Web*.

Durante este trabalho, não se realizou apenas programas em *Haskell*. Utilizámos a plataforma *Mooshak*, de maneira a verificar se cada uma das nossas tarefas funcionava da maneira pretendida. Recorremos também a um sistema de controlo de versões - *SVN*. Também criámos um makefile para este projeto. Por fim a realização deste relatório em *LaTex*.

1.1 Versão Online

O *LightBot* começa com um robô num tabuleiro, onde o utilizador fornece as instruções ao robô. As instruções podem fazer o robô andar, saltar, rodar para a esquerda, rodar para a direita ou acender uma lâmpada. Contudo, existem limitações para alguns comandos:

- só se acende uma luz quando o robô se encontra sob uma lâmpada apagada, caso contrário irá desligar uma lâmpada ligada.
- o robô só anda quando a plataforma para que se desloca se encontra ao mesmo nível da atual.
- o robô apenas salta para uma plataforma imediatamente acima da atual, contudo consegue saltar para qualquer plataforma de nível inferior.

Também é possível criar funções com instruções diferentes, sendo estas chamadas na função principal. Pode-se utilizar isto quando é preciso utilizar-se mais que uma vez o mesmo conjunto de instruções.

Um jogo pode terminar quando todas as lâmpadas estão acesas ou quando não existem mais instruções para o robô. No último, é pedido ao utilizador que adicione ou altere instruções. O utilizador apenas avança de nível quando acende todas as lâmpadas.

1.2 Versão Haskell

Na versão em *Haskell* a interface do jogo segue o modelo seguinte:

```
aaa
bbA
Baa
1 0 S
LAEDSAS
```

- neste caso, as três primeiras linhas correspondem ao tabuleiro propriamente dito, a penúltima linha à posição inicial do robô e a última aos comandos a serem executados pelo robô.
- ao contrário da versão online as linhas do tabuleiro nesta versão devem ter todas o mesmo tamanho.
- as letras maiúsculas simbolizam uma lâmpada nesse local do tabuleiro.
- diferentes letras correspondem a diferentes níveis do tabuleiro, sendo 'a' o nível 0 (ex: 'b' é um nível superior a 'a' e 'c' é um nível superior a 'b').
- a posição do robô segue o seguinte modelo (x, y, o). Em que 'x' e 'y' correspondem respetivamente à coluna e à linha às quais o robô se encontra. A linha 0 corresponde à última linha do tabuleiro. 'o' corresponde à orientação do robô ('N' - Norte, 'E' - Este, 'O' - Oeste e 'S' - Sul).
- os comandos a serem executados pelo robô são os seguintes:
  - 'A' - avançar o robô uma posição.
  - 'E' - rodar o robô à esquerda.
  - 'D' - rodar o robô à direita.
  - 'S' - fazer com que o robô salte para um nível superior ou vários níveis inferiores.
  - 'L' - acender ou apagar uma lâmpada.

Capítulo 2

Análise do Problema

2.1 Tarefa 1

Esta tarefa consite em analisar cada linha do ficheiro de texto recebido e verificar se este segue as seguintes regras:

- não existirem linhas em branco.
- o tabuleiro ser apenas constituído por letras.
- as linhas do tabuleiro terem todas o mesmo tamanho.
- a orientação ser Norte, Sul, Este ou Oeste.
- as coordenas 'x' e 'y', bem como a orientação 'o' devem ser separadas por um único espaço.
- as coordenas 'x' e 'y' devem pertencer ao tabuleiro.
- os comandos a serem executados serem 'L', 'A', 'E', 'D', ou 'S'.

Caso todas estas condições se verifiquem é devolvida a mensagem *OK*. Caso contrário, é devolvida uma mensagem com o valor da linha do ficheiro onde ocorreu o primeiro erro.

2.2 Tarefa 2

Na Tarefa 2, é-nos pedido que o robô execute a primeira instrução da linha dos comandos, considerando que as linhas lidas são válidas. Caso seja possível executá-la, o programa deverá apresentar a próxima posição do robô: coordenada x, coordenada y e orientação. Se não for possível a execução do

comando (por este não ser aplicável à actual posição do robô), deverá aparecer no ecrã a mensagem "ERRO", sendo estes os casos:

- quando o comando é 'A' e o robô tem de se deslocar para um nível diferente daquele em que se encontra.
- quando o comando é 'S' e o robô tem de saltar para um nível igual ao seu ou dois níveis superiores.
- quando o comando é 'L' e o robô localiza-se num sítio onde não existe uma lâmpada.
- quando o robô tenta deslocar-se para fora do tabuleiro.

2.3 Tarefa 3

Esta tarefa consiste em executar todos os comandos fornecidos pela linha de comandos. Os comandos devem ser executados em sequência.

Caso todas as lâmpadas estejam acesas, o programa deverá terminar mesmo que ainda existam comandos por executar. Sempre que um comando não é válido (ex: avançar para fora do tabuleiro) o estado do robô mantém-se inalterado e é executado o comando seguinte (caso este exista). É possível um robô apagar uma lâmpada que já esteja acesa. Sempre que o robô acende, ou apaga uma lâmpada, é devolvida uma mensagem com as suas coordenas 'x' e 'y'.

Quando todas as lâmpadas estiverem acesas é devolvida a mensagem "FIM"juntamente com o número de comandos válidos executados. Se, no final de todos os comandos serem executados, ainda existirem lâmpadas apagadas, é devolvida a mensagem "INCOMPLETO".

2.4 Tarefa 4

Esta tarefa consiste em resolver um tabuleiro recebido, quer seja este um tabuleiro fácil ou difícil. Um tabuleiro é fácil quando os seus níveis não diferem em mais de uma unidade.

O programa termina quando todas as lâmpadas estiverem acesas ou caso não existam lâmpadas no tabuleiro. Dentro dos tabuleiros complicados existem ainda casos em que é necessário acender certas lâmpadas antes de outras e casos em que é necessário acender certas lâmpadas no final.

O programa devolve como mensagem os comandos executados pelo robô de modo a resolver o tabuleiro.

2.5 Tarefa 5

Esta tarefa consiste em imprimir um código *HTML*, com recurso ao formato *X3dom*, que permita visualizar o jogo *Lightbot* num *browser web*.

É fornecido ao programa o mesmo *input* da Tarefa 2 e da Tarefa 3. O *lightbot* executa as instruções pela ordem que elas aparecem na linha de instruções. Termina quando todas as lâmpadas estão acesas ou quando não existem mais comandos para executar.

Capítulo 3

Implementação

3.1 Tarefa 1

Esta tarefa começa com o processamento das linhas de texto do ficheiro lido. De seguida deve-se verificar se todo o input fornecido é válido. Para isso são executadas as seguintes instruções por ordem (caso alguma falhe o programa termina):

- separa-se o tabuleiro da posição inicial do robô e instruções. Obtém-se também a valor da largura de cada linha do tabuleiro e a altura do tabuleiro.
- verifica-se se a primeira linha do tabuleiro não é vazia.
- certifica-se que todas as linhas do tabuleiro têm a mesma largura.
- averigua-se se a linha das coordenadas não é vazia.
- averigua-se se as coordenadas estão corretas.
- verifica-se se a linha das instruções não é vazia.
- verifica-se se as instruções estão corretas.
- verifica-se se não existem mais linhas.

3.1.1 Funções *getBoard* e *isBoard*

```
getBoard :: [String] -> ([Int], Int, [String], [String])
getBoard []    = ([], 0, [], [])
getBoard (h:t) =
```

```
let (width, height, tab, r) = getBoard t
    w = isBoard h 0
in if (w /= 0)
   then ([w] ++ width, height + 1, [h] ++ tab, r)
   else ([], 0, [], (h:t))

isBoard :: String -> Int -> Int
isBoard [] _                  = 0
isBoard (h:t) c | isAlpha h = 1 + (isBoard t (c - 1))
                | otherwise = c
```

Estas duas funções são responsáveis pela separação do tabuleiro das coordenas do robô e instruções. Devolvem também as valores das larguras de cada linha do tabuleiro e a altura do tabuleiro. Para uma maior eficiência do programa fez-se apenas uma travessia na lista de *String* recebida. A função *getBoard* recebe as linhas do ficheiro de texto lido. Passando depois ao seu processamento:

Se "w"for maior que 0 isto quer dizer que a linha lida pertence ao tabuleiro (só contém letras) e que tem "w"colunas. Essa linha é então adicionada ao tabuleiro (*tab*) e "valor"é adicionado à lista de valores das colunas do tabuleiro. Sempre que se verifica que uma linha pertence ao tabuleiro é, também, incrementada a sua altura (*height*). Quando esta condição não se verifica significa que a linha lida não pertence ao tabuleiro mas sim às coordenadas e instruções, terminando a função.

A função *isBoard* recebe uma linha do ficheiro lido e um contador inicializado com o valor 0. A linha é processada caracter a caracter. Sempre que o caracter lido for uma letra o contador é incrementado (mais uma coluna). Caso isto não se verifique é devolvido o valor 0 (esta linha não pertence ao tabuleiro).

3.2 Tarefa 2

Esta tarefa começa com o processamento das linhas do ficheiro de texto. De seguida são armazenadas todas as informações necessárias para se testar o primeiro comando da sequência de instruções (tabuleiro, coordenadas iniciais do robô, tamanho do tabuleiro e caracter do tabuleiro coorespondente à posição inicial do robô). Posto isto, verifica-se a validade do comando a ser executado dadas as seguintes regras:

- 'L' - a posição onde o robô se encontra contém uma lâmpada (letra maiúscula).
- 'D' - é sempre válido.

- 'E' - é sempre válido.
- 'A' -
  - a posição para onde o robô irá avançar pertence ao tabuleiro.
  - a posição para onde o robô irá avançar pertence ao mesmo nível onde o robô se encontra.
- 'S' -
  - a posição para onde o robô irá saltar pertence ao tabuleiro.
  - a posição para onde o robô irá saltar é um nível acima ou qualquer nível abaixo ao nível onde o robô se encontra.

Caso alguma destas regras não se verifique (dependendo do comando a ser executado) o programa termina. Caso, contrário é alterada a posição do robô.

3.2.1 Função *process*

```
process :: [String] -> String
process str =
let (tab, height, coord, commands) = separateTab str
    co     = getCoords (words coord)
    com    = head (commands)
    ch     = locate tab co height
    width = length (head tab)
in if (not (isValidCommand (tab, co, ch, (width, height)) com))
   then "ERRO"
   else changePosition co com
```

Esta função recebe uma lista de *Strings* (linhas de texto do ficheiro lido) e devolve uma *String* que pode ser a mensagem "ERRO"ou as novas coordenadas do robô.

Começa por invocar uma função que irá separar as linhas do tabuleiro, a altura do tabuleiro, a linha com as coordenadas e a linha com os comandos (*separateTab str*).

Depois disto, faz a conversão da linha com as coordenadas para um tuplo de dois *Int* e um *Char* (*getCoords (words coord)*) e guarda numa variável o comando a ser executado - primeiro caracter na linha de comandos (*head (commands)*).

Com as coordenadas do robô, esta função chama uma outra função que procura (e devolve) do tabuleiro o caracter que corresponde à posição do robô, guardando numa variável (*locate tab co height*).

Para a função saber a largura do tabuleiro, apenas é preciso saber o comprimento da primeira linha do tabuleiro (*length (head tab)*).

Caso a função que valida os comandos (retorna um *Bool*) devolver *False*, significa que o comando a executar não é possível para a posição atual do robô e retorna "ERRO". Se devolver *True*, a função *process* devolve as novas coordenadas do robô.

3.2.2 Função *separateTab*

```
separateTab :: [String] -> ([String], Int, String, String)
separateTab [x]    = ([], 0, [], x)
separateTab (h:t) =
let (tb, height, cd, co) = separateTab t
in if(isAlpha(head h))
    then ([h] ++ tb, 1 + height, cd, co)
    else (tb, height, h ++ cd, co)
```

Esta função recebe uma lista de *Strings*: as linhas de texto do ficheiro lido. Passando depois ao seu processamento.

As primeiras n linhas em que a *head* é caracter alfabético, são as linhas do tabuleiro e os seus respetivos caracteres. Sendo que o programa guarda cada linha, à medida que a encontra, numa variável (*[h] ++ tb*). À medida que a função percorre esta linhas, incrementa em 1 o contador das linhas do tabuleiro (*1 + height*).

Após isto, a função encontra uma linha que já não começa por uma letra, mas sim por um número (já que o *input* está correto). A função guarda esta linha na variável das coordenadas do robô (*h ++ cd*).

Quando sobra apenas um elemento na lista de entrada, sabemos que esta é a linha de instruções, guardando-a na variável respetiva (*separateTab [x] = ([], 0, [], x*).

Desta maneira, a função consegue separar os dados pretendidos com uma só travessia da lista.

3.3 Tarefa 3

A fase inicial desta tarefa é idêntica à da tarefa 2. A única diferença consiste no armazenamento das coordenadas das lâmpadas apagadas do tabuleiro numa lista. De seguida testa-se se cada comando recebido é válido. Para isto, são usadas as mesmas funções da tarefa 2. Caso não seja válido a posição do robô mantém-se. Caso contrário a posição do robô é alterada

conforme a instrução recebida. Sempre que o robô acende ou apaga uma lâmpada, é acrescentada a sua posição à mensagem a ser devolvida no final do programa.

Para controlar as lâmpadas acesas e apagadas, sempre que um comando 'L' é executado sobre uma lâmpada testa-se se esta posição existe na lista de lâmpadas apagadas. Caso não exista é sinal que esta lâmpada estava acesa e é então acrescentada à lista de lâmpadas apagadas. Caso contrário é sinal que esta lâmpada estava apagada e é então removida da lista de lâmpadas apagadas.

O programa termina caso a lista de lâmpadas apagadas seja vazia (todas as lâmpadas estão acesas) ou caso as instruções terminem. Para o primeiro caso é acrescentada à mensagem a devolver a *String* "FIM"seguida do número de comandos válidos executados. Para o segundo, é acrescentada a mensagem "INCOMPLETO".

3.3.1 Função *separateTab*

```
separateTab :: [String] -> TabInfo
separateTab [x]    = ([], 0, [], x, [])
separateTab (h:t) =
let (tb, height, cd, co, l) = separateTab t
in if(not (isAlpha(head h)))
    then (tb, height, h ++ cd, co, l)
    else let lOff = map (\x -> (x, height)) (getLC h 0)
in ([h] ++ tb, 1 + height, cd, co, lOff ++ l)
```

Esta função é igual à função utilizada na Tarefa 2. Recebe o mesmo parâmetro: lista de *String*. A única diferença é a devolução de mais um parâmetro: as coordenadas no tabuleiro das lâmpadas por acender.

De maneira a devolver as coordenadas das lâmpadas, foi preciso alterar a ordem das comparações das cabeças de cada *String*. Isto é, primeiro verifica-se se a linha começa por um algarismo e caso seja verdade, o processo é igual ao da Tarefa 2. Enquanto esta condição não for verdadeira, antes de adicionar a linha à variável das linhas do tabuleiro, percorre-se a linha à procura de caracteres maiúsculos:

```
map (\ x -> (x, height)) (getLC h 0)}
```

Devolve uma lista de duplos (coluna do caracter maiúsculo e linha atual do tabuleiro).

3.3.2 Função *exCommands*

```
exCommands :: CommandInfo -> String
exCommands (_, _, _, _, [], tK)          =
        "FIM" ++ "␣" ++ (show tK)
exCommands (_, [], _, _, _, _)           =
        "INCOMPLETO"
exCommands (tab, (h:t), s, (x, y, o), lg, tK) =
let ch = locate tab (x, y, o) (snd s)
in if (not (isValidCommand (tab, (x, y, o), ch, s) h))
   then exCommands (tab, t, s, (x, y, o), lg, tK)
   else if (h /= 'L')
        then let (newTab, (x1, y1, o1)) =
                    changePosition (x,y,o) h tab (snd s)
             in exCommands (newTab, t, s,
                            (x1, y1, o1), lg, (tK + 1))
        else if (elem (x, y) lg)
             then ((show x) ++ "␣" ++
                   (show y) ++ ['\n']) ++
                   (exCommands (tab, t, s, (x, y, o),
                   (rmL lg (x, y)), (tK + 1)))
             else ((show x) ++ "␣" ++ (show y) ++
                   ['\n']) ++
                   (exCommands (tab, t, s, (x, y, o),
                   (lg ++ (x, y) : []), (tK + 1)))
```

A função *exCommands* processa a linha de comandos recebida. Recebe como argumento o tabuleiro do jogo, as dimensões do tabuleiro, a lista de lâmpadas apagadas, a linha de comandos e um contador para os comandos válidos executados. A primeira condição de paragem serve para o caso de todas as lâmpadas estarem acesas. A segunda condição serve para o caso de todos os comandos terem sido testados e executados, mas ainda existirem lâmpadas apagadas.

Para um caso geral, esta função começa por obter o valor do caracter onde o robô se encontra e testa se o comando é válido. Sendo este comando não válido, é processado o comando seguinte. Caso contrário, se este comando não foi acender ou apagar uma lâmpada, a posição do robô é alterada e processado o comando seguinte. Se o comando executado foi acender ou apagar uma lâmpada, é retirada ou acrescentada à lista de lâmpadas apagadas a posição do robô e processado o comando seguinte.

3.4 Tarefa4

A fase inicial desta tarefa é idêntica às tarefas anteriores. A única diferença é que nesta tarefa não se recebe os comandos a serem executados pelo robô. Os comandos serão determinados. De seguida determina-se se o tabuleiro é ou não complicado. Caso seja complicado a lista de lâmpadas apagadas é organizada de acordo com a prioridade de cada lâmpada. Finalmente procede-se à resolução do tabuleiro. A resolução termina quando a lista de lâmpadas apagadas for vazia. A cada elemento da lista faz-se o deslocamento da posição onde o robô se encontra até à posição da lâmpada seguindo a solução simples ou complicada conforme o tipo do tabuleiro. Quando se acende uma lâmpada a posição onde o robô se encontra é atualizada (passa a ser igual à posição da lâmpada acesa). Para reduzir o número de comandos executados pelo robô fez-se o seguinte:

- no caminho até uma dada lâmpada, se o robô passar por cima de outra lâmpada ele acende-a (caso esta esteja apagada).
- quando o robô precisa de rodar é determinado o menor número de rotações que este pode fazer.

Para a resolução do tabuleiro foram usadas duas matrizes auxiliares, cada uma com o mesmo tamanho do tabuleiro. As duas matrizes só são usadas nos tabuleiros complicados, nos tabuleiros fáceis só é necessário utilizar uma matriz. Os valores da matriz de soluções são os seguintes:

- 'N' - posição do tabuleiro por onde o robô não passou ou que não é possível ele passar.
- 'T' - terminal. Posição para onde se quer chegar (lâmpada).
- 'U' - Up. Mostra que o robô desloca-se para norte.
- 'D' - Down. Mostra que o robô desloca-se para sul.
- 'L' - Left. Mostra que o robô desloca-se para oeste.
- 'R' - Right. Mostra que o robô desloca-se para este.

A matriz auxiliar serve para testar se o robô já visitou ou não uma certa posição, evitando deste modo que o robô ande em quadrado. A matriz tem os seguintes valores:

- 'N' - posição desconhecida. O robô ainda não esteve nesta posição.
- 'A' - posição já visitada pelo robô.

3.4.1 Função *easyPath*

```
easyPath :: PCondition -> [[Char]]
easyPath (b, (x, y), (xF, yF), (w, h), sol) =
if ((x, y) == (xF, yF))
then updateMatrix sol (x, h - y - 1) 'T'
else if (y < yF)
     then easyPath (b, (x, y + 1),
          (xF, yF), (w, h),
          updateMatrix sol (x, h - 1 - y) 'U')
     else if (y > yF)
          then easyPath (b, (x, y - 1), (xF, yF),
          (w, h), updateMatrix sol (x, h - 1 - y) 'D')
          else if (x < xF)
               then easyPath (b, (x + 1, y), (xF, yF),
               (w, h), updateMatrix sol (x, h - 1 - y) 'R')
               else easyPath (b, (x - 1, y), (xF, yF), (w, h),
               updateMatrix sol (x, h - 1 - y) 'L')
```

A função *easyPath* é a função responsável pela determinação de soluções de tabuleiros fáceis. Neste tipo de tabuleiros não é necessário preocupar-se com os casos em que o robô sai fora do tabuleiro ou que este não se pode mover por causa de estar preso num local em que não consiga saltar. Para este tipo de solução basta mover o robô para as coordenadas x e y da lâmpada destino e assinalar as direções que este percorreu.

3.4.2 Função *complicatedPath*

```
complicatedPath :: PCondition -> [[Char]] -> ([[Char]], [[Char]])
complicatedPath (b, (x, y), pF, (w, h), sol) p =
if ((x, y) == pF)
then (updateMatrix sol (x, h - 1 - y) 'T', p)
else if (((p !! (h - y - 1)) !! x) == 'A')
     then (sol, p)
     else let actual    =
     updateMatrix p (x, h - 1 - y) 'A'
               (sUp, up) =
               cCondition (b, (x, y), pF, (w, h), sol)
               actual (y, (h - 1), 1, 0)
          in if (isValid sUp)
             then (updateMatrix sUp (x, h - 1 - y) 'U', up)
```

```
                else let (sR, ri) = cCondition
                (b, (x, y), pF, (w, h), sol) up (x, (w - 1), 0, 1)
                     in if (isValid sR)
                        then
                        (updateMatrix sR (x, h - y - 1) 'R', ri)
                        else let (sD, dw) =
                        cCondition (b, (x, y), pF,
                                    (w, h), sol) ri (y, 0, -1, 0)
                             in if (isValid sD)
                                then (updateMatrix sD
                                     (x, h - y - 1) 'D', dw)
                                else let (sL, le) =
                                cCondition (b, (x, y), pF,
                                (w, h), sol) dw (x, 0, 0, -1)
                                     in if (isValid sL)
                                        then
                                        (updateMatrix sL
                                        (x, h - y - 1) 'L', le)
                                        else (sol, le)

cCondition :: PCondition -> [[Char]] -> PCompare ->
([[Char]], [[Char]])
cCondition (b, (x, y), pF, (w, h), sol) p (k, z, i, j) =
if (k /= z &&
((ord (toLower ((b !! (h - 1 - (y + i))) !! (x + j)))) -
(ord (toLower ((b !! (h - 1 - y)) !! x)))) < 2)
then complicatedPath (b, (x + j, y + i), pF, (w, h), sol) p
else (sol, p)
```

A função *complicatedPath* é responsável pela resolução de tabuleiros complicados. A função percorre o tabuleiro de uma maneira recursiva até encontrar a solução. No pior caso possível a função percorre completamente o tabuleiro. Em cada posição em que se encontra o robô é testado se essa posição coincide com a posição onde se quer chegar, e se assim for, é encontrada a solução e a função termina. Sempre que se muda de posição marca-se na lista auxiliar que o robô ja esteve nessa posição. A recursividade desta função segue a seguinte ordem: testa-se os caminhos para cima, para a direita, baixo e finalmente esquerda. O robô volta para trás se um caminho que ele percorreu foi inválido, isto é:

- caso esteja a sair do tabuleiro.
- caso ja tenha estado numa certa posição e a volte a percorrer.

- caso seja uma posição à qual ele não consiga aceder.

A função *cCondition* serve para auxiliar a função *complicatedPath*. Esta função recebe adicionalmente como parâmetros a coordenada x ou y do robô conforme ele se desloque para os lados ou para cima ou para baixo no tabuleiro. Recebe também a largura ou altura conforme tenha recebido x ou y, para testar se este se encontra nos limites dos tabuleiro e não possa avançar. Recebe também um valor j a ser somado à coordena x e um valor i a ser somado à coordenada y. Estes valores servem para se obter a coordenada a testar conforme a direção do robô. Basicamente, esta função testa se o robô pode ou não mover-se para a posição seguinte, conforme a sua direção.

3.4.3 Função *getFinalL*

```
getFinalL :: Status -> [LightsOff]
getFinalL (_, _, _, [], _, _, _)          = []
getFinalL (b, c, s, (h:t), sol, p, tp)  =
let lI =
filter (\x ->
(isValid (fst (complicatedPath (b, x, h, s, sol) p)))) t
    lF =
filter (\x ->
(not (isValid (fst (complicatedPath (b, x, h, s, sol) p))))) t
in (getFinalL (b, c, s, lI, sol, p, tp)) ++
    [h] ++
    (getFinalL (b, c, s, lF, sol, p, tp))
```

A função *getFinalL* só é executada se o tabuleiro recebido for complicado. Esta função trata tabuleiros especiais em que é necessário aceder a certas lâmpadas antes de outras e deixar certas lâmpadas para o fim. Um exemplo deste tipo de tabuleiros será um tabuleiro com um poço. A função usa o algoritmo *Quicksort* para ordenar a lista de lâmpadas apagadas. Para a sua ordenação separa-se a lista em duas: as lâmpadas prioritárias e as lâmpadas não prioritárias. Uma lâmpada é prioritária se da posição dela consiguir aceder-se a qualquer outra lâmpada da lista. Uma lâmpada não é prioritária se dela não se conseguir aceder a uma lâmpada da lista, devendo esta ser deixada para o fim.

Estas funções consistem na nossa estratégia para a resolução da tarefa 4. Embora estas funções sejam rápidas mesmo para tabuleiros de grandes dimensões, elas não oferecem o caminho mais curto. Numa fase inicial nós

optamos por outra estratégia que devolvia sempre o caminho mais curto. No entanto, vimo-nos obrigados a abandonar essa solução pelo facto de ela ser muito pesada para tabuleiros de grandes dimensões. As funções desta estratégia inicial foram as seguintes:

```
data Path = Invalid
          | Destination
          | Unknown
          | Node Path Path Path Path
          deriving (Eq)

getAllP :: Path -> PCondition -> Path
getAllP Unknown _       = Unknown
getAllP Invalid _       = Invalid
getAllP Destination _ = Destination
getAllP (Node u r d l) (brd, (x, y), pF, (w, h), lP) =
let up    =
pathCondition (brd, (x, y), pF, (w, h), lP) (1, 0, y, h - 1)
    right =
pathCondition (brd, (x, y), pF, (w, h), lP) (0, 1, x, w - 1)
    down  =
pathCondition (brd, (x, y), pF, (w, h), lP) (-1, 0, y, 0)
    left  =
 pathCondition (brd, (x, y), pF, (w, h), lP) (0, -1, x, 0)
in (Node up right down left)

pathCondition :: PCondition -> PChange -> Path
pathCondition (brd, (x, y), pF, (w, h), lP) (i, j, k, z) =
          if (k == z ||
            (ord (toLower ((brd !!
            (h - 1 - (y + i))) !! (x + j)))) -
            (ord (toLower
            ((brd !! (h - 1 - y)) !! x))) > 1 ||
            (elem (x + j, y + i) lP))
          then Invalid
          else if ((x + j, y + i) == pF)
               then Destination
               else (getAllP addNode
                    (brd, (x + j, y + i), pF, (w, h),
                    (lP ++ [(x, y)])))
```

```
addNode :: Path
addNode = (Node Unknown Unknown Unknown Unknown)

solve :: SolveP -> String
solve (Destination, _, _, _, _)
  = "L"
solve (Invalid, _, _, _, _)
   = []
solve ((Node u r d l), brd,  (x, y, o), (w, h), lF) =
       let up    =
getCommands (u, brd, (x, y, o), (w, h), lF) (1, 0) N
           right =
getCommands (r, brd, (x, y, o), (w, h), lF) (0, 1) E
          down   =
getCommands (d, brd, (x, y, o), (w, h), lF) (-1, 0) S
           left  =
getCommands (l, brd, (x, y, o), (w, h), lF) (0, -1) O
       in shortestPath [right, down, left] (up, length up)

shortestPath :: [String] -> (String, Int) -> String
shortestPath [] (x, y)    = x
shortestPath (h:t) (x, y) =
        let l = length h
       in if (y == 0 && l == 0)
          then shortestPath t (x, y)
          else if (y == 0 && l /= 0)
          then shortestPath t (h, l)
          else if (y /= 0 && l < y && l /= 0)
          then shortestPath t (h, l)
          else shortestPath t (x, y)

getCommands :: SolveP -> (Int, Int) -> Orientation -> String
getCommands (p, b, (x, y, o), (w, h), l) (i, j) d =
       if (isInvalid p)
       then []
       else let auxR = rotation o d
                auxL =
                checkLight (b, (x, y), (w, h), l)
            in if
            (ord (toLower ((b !!
```

```
                (h - 1 - (y + i))) !! (x + j))) -
                (ord (toLower
                ((b !! (h - 1 - y)) !! x))) /= 0)
                   then auxL ++
                        auxR ++ "S" ++
                        (solve
                        (p, b, (x + j, y + i, d), (w, h), l))
                   else auxL ++ auxR ++ "A" ++
                        (solve
                        (p, b, (x + j, y + i, d), (w, h), l))

isInvalid :: Path -> Bool
isInvalid Invalid       = True
isInvalid Destination   = False
isInvalid (Node u r d l) =
        isInvalid u &&
        isInvalid r &&
        isInvalid d &&
        isInvalid l
```

Nesta estratégia inicial optou-se por criar uma árvore com 4 ramos possíveis: Up, Right, Down e Left. Para além disso haviam três campos adicionais na estrutura de dados: Unknown, Invalid e Destination. O campo Unknown traduzia uma posiçao à qual o robô ainda não visitou. O campo Invalid traduzia que uma certa direção não era válida (fora do tabuleiro ou um nível muito alto à qual o robô não conseguiria aceder). O campo destination traduzia a posição que se queria chegar (a lâmpada apagada).

As funções *getAllP*, *pathCondition* e *addNode* são as responsáveis pela construção da árvore. Estas funções constroem todos os caminhos possíveis que o robô pode tomar para chegar à lâmpada que tem de acender. Estas funções são de componente recursiva. Para cada posição são testadas as quatro direções possíveis que o robô pode tomar, seguindo a seguinte ordem: Up, Right, Down e Left. Para cada uma destas direções é marcada numa lista as posições onde o robô esteve evitando deste modo que ele volte a testar tudo o que andou para trás e que ele se mova em quadrado, entrando em ciclo infinito. Para cada uma destas direções testa-se também se é possível o robô avançar ou se este chegou à posição destino. Se isto acontecer a árvore toma o valor Invalid ou Destination conforme a posição seja inválida ou coincidente à posição da lâmpada apagada e a recursividade termina para esta sub-árvore. Caso contrário a função *addNode* é invocada, adicionando um novo nó a esse nó, havendo deste modo mais quantro direções a testar.

Segue-se uma exemplo de um tabuleiro simples e a respetiva árvore criada para chegar à solução do tabuleiro.

Tabuleiro:

abc

aaD

0 0 N

Árvore:

(Node (Node Invalid (Node Invalid (Node Invalid Invalid Destination Invalid) (Node Invalid Invalid Invalid Invalid) Invalid) Invalid Invalid) (Node (Node Invalid (Node Invalid Invalid Destination Invalid) Invalid (Node Invalid Invalid Invalid Invalid)) Invalid Invalid Invalid) Invalid Invalid)

Tendo a àrvore construída as restantes funções encarregam-se de encontrar o caminho mais curto. Caso hajam dois ou mais caminhos com o mesmo peso, é escolhido aquele que permite ao robô efectuar o menor número de rotações. Para isto são armazenados numa *string* os comandos executados pelo robô em cada sub-árvore e no final escolhe-se aquela que tem menor comprimento. Só se faz a procura de comandos executados pelo robô em sub-árvore válidas (aquelas que chegam a uma solução). Para isso, para cada sub-árvore usa-se a função *isInvalid* para testar se nela existe alguma sub-árvore válida e se assim for são armazenados os comandos que o robô executa para esta solução.

Estas funções funcionam muito bem para tabuleiros de pequenas a médias dimensões. No entanto, para tabuleiros de dimensões superiores a construção da árvore é muito demorada, tendo sido este o motivo para termos abandonado esta solução.

3.5 Tarefa 5

Esta tarefa imprime código *HTML* que permite a visualização do jogo *Lightbot*.

A fase inicial desta tarefa é idêntica à da tarefa 2 e à da tarefa 3. No início fazem-se os cálculos necessários dos valores a fornecer às funções que imprimem o código *HTML*.

Começa-se por separar as diferentes componentes do ficheiro de texto (tabuleiro, coordenadas iniciais e linha de instruções) e faz-se a transformação da *String* com as coordenadas para um tuplo com três elementos.

Feito isto, calculam-se as várias posições que o robô percorre enquanto executa as instruções. Isto irá servir para depois se poder animar o movimento do robô, visto que são necessárias todas as coordenadas x, y e z percorridas. Com a lista das posições, determina-se o intervalo de tempo entre cada uma destas, sendo necessário para a animação do movimento.

Depois, determina-se os valores das cores (em formato *RGB*) de cada uma das lâmpadas e da antena do robô, para que ambas estejam sincronizadas - quando o robõ acende ou apaga uma lâmpada, a antena fica iluminada por um período de tempo. Irá servir para animar o robô a acender ou apagar uma lâmpada.

Por fim, calcula-se as orientações que as pernas do robô têm de ter para que seja possível simular o andar dele.

Com tudo isto calculado, imprime-se o código *HTML* com estes valores.

3.5.1 Função *process*

```
process :: [String] -> String
process s =
let
(board, hght, c, com, lights) = separateBoard s
(x, y, o) = getCoord (words (c)) 0
width = length (head board)
i = 0.25 * (fromIntegral width)
j = 0.25 * (fromIntegral hght)
positions =
[(x, y, convert (locate board (x, y) hght), NOT)] ++
getP (board, (x, y, o), com, (width, hght), lights)
lgt = length positions
cycle = if (lgt == 1)
        then 1
        else (1 /(fromIntegral (lgt - 1)))
timeInterval = if(cycle == 1 && lgt == 1)
               then "1"
               else htmlKey 0 cycle (lgt - 1)
botPos = convertL positions (getRadius o) (i, j) hght
htmlL  = convertLights lights (i, j) hght
botWalk = if (com /= "")
          then if (head com == 'A')
               then ("1 0 0 -0.35  1 0 0 0.35",
                     "1 0 0 0.35  1 0 0 -0.35"):
                    convert2Walk (tail botPos)
               else convert2Walk botPos
          else convert2Walk (botPos)
lgtLEGS = lgt * 2
```

```
cycleLEGS = (1/(fromIntegral (lgtLEGS - 1)))
legsInterval = htmlKey 0 (cycleLEGS) (lgtLEGS - 1)
in
htmlHead ++
htmlBodyShapes ++
(htmlBoard board 0 i j) ++
htmlBot ++
"<timeSensor def=\"clock\" cycleInterval=\"" ++
show (0.75 * (fromIntegral lgt)) ++
"\" loop=\"true\"></timeSensor>\n" ++
"<positionInterpolator def=\"moveRobot\" key=\""
++ timeInterval ++
"\" keyValue=\"" ++ (htmlAnimatronics botPos) ++ "\">" ++
"</positionInterpolator>\n" ++
htmlRoutePosition ++ "\n\n\n" ++

"<orientationInterpolator def=\"rotate\" key=\""
++ timeInterval ++
"\" keyValue=\"" ++ (htmlRotation botPos) ++ "\">" ++
"</orientationInterpolator>\n" ++
"htmlRouteOrientation ++ "\n\n\n" ++

"<orientationInterpolator def=\"rotateLeftLeg\"
key=\"" ++ legsInterval ++
"\" keyValue=\"" ++ (htmlRotationLegs 'l' botWalk) ++ "\">" ++
"</orientationInterpolator>\n" ++
(htmlRouteOrientationLegs "rotateLeftLeg" "leftLeg") ++

"<orientationInterpolator def=\"rotateRightLeg\"
key=\"" ++ legsInterval ++
"\" keyValue=\"" ++ (htmlRotationLegs 'r' botWalk)
++ "\">" ++
"</orientationInterpolator>\n" ++
(htmlRouteOrientationLegs "rotateRightLeg" "rightLeg")
 ++ "\n\n\n" ++

"<colorInterpolator def=\"colorA\" key=\"" ++ timeInterval ++
"\" keyValue=\"" ++ (htmlColor botPos) ++ "\">" ++
"</colorInterpolator>\n" ++
"<route fromNode=\"clock\" toNode=\"colorA\"" ++
"fromField=\"fraction_changed\" toField=\"set_fraction\">
```

```
</route>\n" ++
"<route fromNode=\"colorA\" toNode=\"antennaColor\"" ++
"fromField=\"value_changed\" toField=\"diffuseColor\">
</route>\n" ++
(htmlLightsK htmlL botPos timeInterval) ++
htmlEnding
```

Pode-se dizer que a função *process* é a função principal desta tarefa. Isto porque é a função que chama as funções necessárias para efetuar os cálculos referidos acima, para além de juntar tudo no final, estruturando assim o código *HTML*. De seguida explica-se cada uma das linhas do *let*.

- (board, hght, c, com, lights) -¿ tal como nas tarefa 2 e tarefa 3, esta função devolve as informações que se podem obter a partir do ficheiro de texto (tabuleiro, altura do tabuleira, linha com as coordenadas, linha com as instruções e a lista com as coordenadas das lâmpadas).

- (x, y, o) -¿ converte a *String* com as coordenadas e orientação iniciais, num tuplo com dois inteiros e uma *Orientation*.

- width - largura do tabuleiro.

- i - factor que irá permitir centrar o tabuleiro, segundo o eixo x, na página *Web*.

- j - factor que irá permitir centrar o tabuleiro, segundo o eixo y, na página *Web*.

- positions -¿ variável que armazena todas as posições que o robô percorre.

- lgt - número de posições que o robô percorre.

- cycle - intervalo de tempo entre cada posição. Caso não haja instruções para o robô, ele mantém a posição inicial.

- timeInterval - *String* com cada instância de tempo para cada posição do robô. É impressa na construção de cada *colorInterpolator*, *positionInterpolator* e *orientationInterpolator* do código *HTML* (excepto no da orientação das pernas).

- botPos - lista com as posições do robô, prontas a serem impressas no código *HTML*.

- htmlL - lista com as coordenadas das lâmpadas, convertidas para o código *HTML*.

- botWalk - lista de duplos de *String*, que representa as orientações das pernas do robô. De forma a simular o movimento das pernas enquanto ele anda.
- lgtLEGS - visto que entre cada posição as pernas alteram a sua orientação duas vezes, esta variável é o dobro do número de posições que o robô percorre.
- cycleLEGS - intervalo de tempo entre cada mudança de orientação das pernas.
- legsInterval - *String* com cada instância de tempo para as orientações das pernas do robô. É impressa na construção de cada *orientationInterpolator* de cada perna.

No final do *let*, temos o *in* que estrutura o código *HTML* com os valores calculados previamente.

3.5.2 Função *getP*

```
getP :: PositionP -> [Position]
getP (_, _, _, _, [])                    = []
getP (_, _, [], _, _)                    = []
getP (b, (x, y, o), (h:t), (w, hgt), l) =
       case h of
            'D' ->
            (x, y, convert ((b !! (hgt - 1 - y)) !! x), R)
             : (getP (b, (x, y, right o), t, (w, hgt), l))
            'E' ->
            (x, y, convert ((b !! (hgt - 1 - y)) !! x), L)
             : (getP (b, (x, y, left o), t, (w, hgt), l))
            'L' ->
            let (lC, lF) = getLightC (x, y) (w, hgt) l b
            in (x, y,
            convert ((b !! (hgt - 1 - y)) !! x), lC) :
            (getP (b, (x, y, o), t, (w, hgt), lF))
             _   -> let cmdL      =
             getWJC b (x, y, o) (w, hgt) h
                        (xF, yF) = getXY (last cmdL)
                    in cmdL ++ (getP
                    (b, (xF, yF, o), t, (w, hgt), l))
```

A função *getP* devolve uma lista com todas as posições (x, y e z) que o robô tem enquanto percorre o tabuleiro e o comando que ele está a executar nessa posição. Para isso, verifica cada um dos comandos na linha de instruções, invocando a função apropriada para o comando lido. Só não chama uma quando a instrução é de rodar para a esquerda ou para a direita, visto que o que altera nesses casos é a orientação do robô e não as suas coordenadas.

Quando o comando é acender (ou apagar) uma luz, é invocada a função *getLightC*.

Caso não seja nenhuma instrução das anteriores, significa que o comando a ser executado é saltar ou andar, invocando-se a função *WJC*.

3.5.3 Função *getLightC*

```
getLightC :: (Int, Int) -> Size ->
[LightsOff] -> Board -> (Command, [LightsOff])
getLightC (x, y) (w, h) l b
| isLower ((b !! (h - 1- y)) !! x) = (IL, l)
| elem (x, y) l                    =
(LON, rmL l (x, y))
| otherwise = (LOFF, l ++ [(x, y)])
```

Esta função começa por verificar se no tabuleiro existe uma lâmpada na posição correspondente à do robô. Caso não se verifique isto, está-se tentar acender uma luz onde não existe lâmpada, o que é inválido.

Caso exista uma lâmpada nesse sítio, o programa verifica se aquelas coordenadas existem na lista de lâmpadas apagadas. Se existirem, é devolvido o comando *LON* (significa que uma lâmpada ficou acesa) e a lista de lâmpadas apagadas com este par de coordenadas a menos.

Se estas coordenadas não existirem na lista das lâmpadas apagadas, significa que a luz está acesa. Sendo assim, devolve-se o comando *LOFF* (significa que uma lâmpada ficou apagada) e a lista de lâmpadas apagadas em conjunto com as coordenadas atuais.

3.5.4 Função *getWJC*

```
getWJC :: Board -> Coord ->
Size -> Char -> [Position]
getWJC b (x, y, o) s cmd =
let (i, j) = getNextP o
in if (isInvalid (x + i, y + j) s)
then if (cmd == 'A')
```

```
then [(x, y, convert (locate b (x, y) (snd s)), W)]
else [(x, y,
(convert (locate b (x, y) (snd s))) + 0.25, J),
(x, y, (convert (locate b (x, y) (snd s))), J)]
else case cmd of
'A' -> getWC b (x, y) (i, j) s
'S' -> getJC b (x, y) (i, j) s
```

getWJC devolve as posições que o robô percorre quando a instrução é saltar ou andar.

Começa por calcular, para a posição e orientação atual, os factores de transformação das coordenadas caso ele salte ou ande. Depois, verifica se as novas coordenadas são possíveis dentro do tabuleiro.

Caso não seja, ele devolve as coordenadas (inválidas) por onde ele se movimenta. Quando não é possível ao robô saltar, ele salta na mesma, mas apenas na vertical. Quando não pode andar, ele mantém-se no sítio.

Caso as coordenadas sejam válidas, são invocadas as funções *getJC* quando é para saltar e *getWC* quando é para andar.

3.5.5 Função *getJC*

```
getJC :: Board -> (Int, Int) ->
(Int, Int) -> Size -> [Position]
getJC b (x, y) (i, j) (w, h)
        | (ord (toLower (locate b (x + i, y + j) h)))
        == (ord (toLower (locate b (x, y) h)))
        = [(x, y,
        (convert (locate b (x, y) h)) + 0.25, J),
        (x, y, (convert (locate b (x, y) h)), J)]
        | (ord (toLower (locate b (x + i, y + j) h))) -
        (ord (toLower (locate b (x, y) h))) > 1 =
        [(x, y, (convert (locate b (x, y) h)) + 0.25, J),
        (x, y, (convert (locate b (x, y) h)), J)]
        | (ord (toLower (locate b (x + i, y + j) h))) -
        (ord (toLower (locate b (x, y) h))) < 0 =
        [(x + i, y + j, convert (locate b (x, y) h), JD),
        (x + i, y + j, convert (locate b (x + i, y + j) h), JD)]
        | otherwise
        = [(x, y, (convert (locate b (x, y) h)) + 0.25, JU),
        (x + i, y + j, convert (locate b (x + i, y + j) h), JU)]
```

A função *getJC* devolve as posições do robô quando ele salta.

Verifica se a posição para onde o robô se irá deslocar se encontra a um nível superior ou a vários níveis inferiores. Caso não seja nenhuma das anteriores, são devolvidas as coordenadas de salto inválido.

Caso o robô salte para um nível imediatamente superior, são devolvidas as coordenadas dele a saltar na vertical e as coordenadas dele a avançar um bloco para a frente, mas num nível superior.

Caso o robô salte para um nível inferior, são devolvidas as coordenadas dele a andar um bloco para a frente e as coordenadas dele a descer os níveis necessários para chegar ao pretendido.

3.5.6 Função *getWC*

```
getWC :: Board -> (Int, Int) ->
(Int, Int) -> Size -> [Position]
getWC b (x, y) (i, j) (w, h)
| (ord (toLower ((locate b (x + i, y + j) h))))
/= (ord (toLower (locate b (x, y) h))) =
[(x, y, convert (locate b (x, y) h), W)]
        | otherwise
= [(x + i, y + j, convert (locate b (x + i, y + j) h), W)]
```

Esta função devolve a nova posição do robô, quando ele anda. Contudo, pode ser um andar válido ou um andar inválido.

É inválido, quando o bloco para onde se está a mover não se encontra ao mesmo nível do bloco onde está localizado. Sendo que, as suas coordenadas mantêm-se. Caso contrário, é um andar válido e dependendo da orientação, a sua coordenada x ou y irá ser alterada.

3.5.7 Função *getRadians*

```
getRadians :: Orientation -> Float
getRadians N = 0
getRadians E = 1.5 * pi
getRadians S = pi
getRadians O = 0.5 * pi
```

getRadians transforma a orientação do robô no seu equivalente em radianos. Isto é importante para animar os comandos de rotação que o robô poderá ter que executar.

3.5.8 Função *convertL*

```
convertL :: [Position] -> Float ->
(Float, Float) -> Int -> [Bot]
convertL [] _ _ _                    =
[]
convertL ((x, z, y, c):t) r (i, j) h =
let (xF, yF, zF) =
((fromIntegral x) * 0.525 - i, y,
(fromIntegral (h - 1 - z)) * 0.525 - j)
in case c of
R -> (xF, yF, zF, r - (0.5 * pi), c) :
(convertL t (r - (0.5 * pi)) (i, j) h)
L -> (xF, yF, zF, r + (0.5 * pi), c) :
(convertL t (r + (0.5 * pi)) (i, j) h)
_ -> (xF, yF, zF, r, c) :
(convertL t r (i, j) h)
```

Esta função faz a conversão da lista com as posições do robô para um formato que esteja pronto a ser impresso para o código *HTML*. Isto é, transforma os valores inteiros da lista inicial em *Float*, multiplicando ao mesmo tempo por um factor que faz com que na janela do *X3dom*, o tabuleiro apareça centrado na página *Web*.

3.5.9 Função *convertLights*

```
convertLights :: [LightsOff] ->
(Float, Float) -> Int -> [(Float, Float)]
convertLights [] _ _                 = []
convertLights ((x, y):t) (i, j) h =
((fromIntegral x) * 0.525 - i,
(fromIntegral (h - 1 - y)) * 0.525 - j) :
(convertLights t (i, j) h)
```

convertLights faz a conversão das coordenadas inteiras das lâmpadas para *Float*, multiplicando ao mesmo tempo pelo factor que faz com que sejam as coordenadas em *HTML*.

3.5.10 Função *auxRound*

```
auxRound :: Float -> Float
auxRound n =
(fromIntegral (round (n * 1000))) / 1000
```

A função *auxRound* é uma função auxiliar que faz o arredondamento de números à milésima. Foi necessária a sua criação porque existiam tabuleiros onde nas lâmpadas, no código *HTML*, o nomes dado ao tipo do material (para mais tarde se conseguir alterar a cor) não correspondia ao nome no *colorInterpolator*. Já que o nome era dado baseado nas coordenadas da lâmpada, o que acontecia por vezes era que as coordenadas eram arredondadas na parte do *colorInterpolator* e não eram arredondadas na declaração do material da lâmpada. Decidindo-se assim arredondar todas as coordenadas, de todas as lâmpadas, até à casa da milésima.

3.5.11 Função *htmlLightsK*

```
htmlLightsK :: [(Float, Float)] ->
[Bot] -> String -> String
htmlLightsK [] _ _              = []
htmlLightsK ((x, y):t) bot tI =
"\t\t\t\t<colorInterpolator def=\"color" ++
(show (auxRound x)) ++ "_" ++
(show (auxRound y)) ++
"\" key=\"" ++ tI ++
"\" keyValue=\"" ++
(htmlLightsC (x, y) "0.6 0.86 0.97" bot) ++
"\">" ++ "</colorInterpolator>\n" ++
"\t\t\t\t<route fromNode=\"clock\"
toNode=\"color" ++
(show (auxRound x)) ++ "_" ++
(show (auxRound y)) ++ "\"" ++
" fromField=\"fraction_changed\"
toField=\"set_fraction\"></route>\n" ++
"\t\t\t\t<route fromNode=\"color" ++
(show (auxRound x)) ++ "_"
++ (show (auxRound y)) ++
"\" toNode=\"" ++ (show (auxRound x)) ++
"_" ++ (show (auxRound y)) ++ "\"" ++
" fromField=\"value_changed\"
toField=\"diffuseColor\"></route>\n" ++
htmlLightsK t bot tI
```

htmlLightsK devolve uma *String* (para cada lâmpada existente) que, em *HTML*, faz com que as lâmpadas estejam acesas ou apagadas. Ou seja, faz com que uma lâmpada esteja acesa até que o comando *LOFF* seja aplicado

nela ou faz com que uma lâmpada esteja apagada até que o comando *LON* seja aplicado nela.

Capítulo 4

Testes

À medida que fomos concretizando cada tarefa, houve a necessidade de testar cada uma delas, antes de se submeter na plataforma *Mooshak*. Para isso, criámos para cada tarefa um conjunto de testes, onde o resultado do programa teria de ser igual ao esperado por nós. Alguns destes testes serão apresentados neste relatório, sendo que os restantes encontram-se na pasta *TestsA, TestsB, TestsC, TestsD e TestsE* da pasta *Tests*.

4.0.12 Tarefa 1

A Tarefa 1 deve indicar, a partir da leitura de um ficheiro de texto, se existem erros ou não na maneira como os dados se encontram.

Por exemplo, se a entrada fosse algo deste género:

- *"a aa", "acaA", "aa ", "0 0 S", "ASDESA"*

O que a Tarefa 1 teria de fazer era indicar que existe um erro na primeira linha do ficheiro: existe um caracter ' ' entre duas letras. A programa teria de indicar esta linha como sendo o erro, apesar de existirem outras linhas que contêm erros, visto que é o que nos foi pedido no enunciado.

Um outro teste que também tivemos em conta foi:

- *"aaa", "Aaa", "aaaA", "0 0 S", "ASDESA"*

Como se pode ver, todas as linhas que representam o tabuleiro (as três primeiras) contêm caracteres que são aceitáveis para a sua construção. Contudo, o programa apresentará como *output* "3". Isto porque é nesta linha onde o número de colunas no tabuleiro não se mantém: nas duas primeiras linhas é três e na terceira linha passa para quatro.

Para além dos erros que poderíamos ter nas linhas do tabuleiro, também fizemos testes onde as coordenadas iniciais do robô encontram-se fora do tabuleiro, o que não é possível. Isto verifica-se no exemplo seguinte:

- *"aaa", "Aaa", "aaa", "0 10 S", "ASDESA"*

Onde o número de colunas do tabuleiro é 3 e o robô terá início na coluna 11.

Por fim, é necessário verificar a linha de instruções. Esta linha só pode conter caracteres que representem comandos válidos: 'E', 'D', 'S', 'A' e 'L'. Como se pode ver a seguir:

- *"aaa", "Aaa", "aaa", "0 0 S", "ASDTESA"*

Para este exemplo, o programa iria imprimir no ecrã 5. Visto que o caracter 'T' não representa nenhuma instrução.

Caso todas as linhas do ficheiro de texto estivessem corretas, o programa imprimia *String OK*.

4.0.13 Tarefa 2

Para a Tarefa 2 era necessário considerar que todas as linhas de texto do ficheiro de entrada estavam corretas.

Nesta tarefa era preciso executar apenas o primeiro comando da linha de instruções, devolvendo as novas coordenadas do robô ou uma mensagem de erro.

Para isso consideramos os seguintes testes.

- *"aaa", "daa", "aaA", "0 0 N", "SAA"*

Neste caso, a mensagem devolvida seria *ERRO*. Isto porque o comando a ser executado é 'S' (saltar) e a plataforma para onde o robô pretende saltar encontra-se 3 níveis acima da atual, o que vai contra as regras do jogo.

Para além deste teste, também nos lembrámos de um outro que achamos pertinente:

- *"aaa", "baa", "aaA", "0 0 N", "SAA"*

Este exemplo é igual ao anterior (o comando a executar é o mesmo), excepto que o tabuleiro é diferente: a plataforma para onde o robô pretende saltar passou para um nível inferior. Para este, o *output* dado pelo programa seria *"0 1 N"*, sendo estas as novas coordenadas do robô. Demonstrando que o comando 'S' era aplicável para este tabuleiro e para as coordenadas do robô.

4.0.14 Tarefa 3

Tal como na tarefa anterior, para a Tarefa 3 era necessário considerar que todas as linhas de texto do ficheiro de entrada estavam corretas.

Neste tarefa era preciso executar a linha de comandos até que todas as lâmpadas estivessem acesas ou até não existirem mais instruções para executar.

Para isso considerámos os seguintes testes.

- *"aaa", "baa", "aaA", "0 0 N", "SAA"*

Neste caso, a mensagem devolvida pelo programa seria *INCOMPLETO*. Isto porque a linha de instruções terminou sem todas as lâmpadas estarem acesas.

Também fizemos testes sobre luzes que eram acesas e depois eram apagadas:

- *"Aaa", "baA", "AaA", "0 0 N", "LLDAALEAL"*

Com este tabuleiro e esta linha de instruções, o programa iria devolver no fim *INCOMPLETO*, visto que ficou uma lâmpada por acender quando o programa terminou de executar a linha de comandos. Isto aconteceu porque a primeira lâmpada acesa foi apagada imediatemente a seguir, procedendo-se depois ao acendimento das restantes. No fim, isto resultou a uma lâmpada ficar por acender.

Finalmente, para testar se o programa conseguia calcular as novas coordanas do robô, utilizámos o teste seguinte:

- *"aaa", "AaA", "baa", "0 0 N", "AASLDAALSD"*

Para este teste o programa apresenta no ecrâ as seguintes mensagens: "0 1", "2 1"e "FIM 6". As duas primeiras são as coordenadas das lâmpadas acesas (tal como pedia o enunciado). No fim, o programa indicou que todas as lâmpadas se encontram acesas e o número de comandos válidos. O resultado obtido era o previsto, já que sabíamos que todos os comandos que não válidos (ex: tentar andar ('A') de um nível superior para um inferior) não são contados.

4.0.15 Tarefa 4

Na tarefa 4 era recebido um tabuleiro, bem como as coordendas iniciais do robô. No final, o programa devolve os comandos a serem executados pelo robô de modo a que este resolva o tabuleiro. Criámos vários testes para esta tarefa. Dá-mos ênfase aos seguintes:

Testes de tabuleiros fáceis

- *"aaaa", "aaaa", "aaaa", "0 0 S"*

Neste tabuleiro em que não existem lâmpadas o program termina devolvendo a mensagem . Este é o caso mais simples possível.

- *"aaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "aaaaaaaaaaaaaaaaAbB", "aaaaBaaaaaaBaaaaaa", "4 10 N"*

Este tabuleiro foi criado com o intuido de testar a rapidez do programa. Este é uma tabuleiro simples pois so contém dois níveis (a e b), mas é um tabuleiro de grandes dimensões e com muitas lâmpadas por acender, o que exige muitas instruções a serem executadas.

Testes de tabuleiros difíceis

- *"aaaaaaa", "aaacccC", "Aaacaaa", "aaacaaa", "aaabaaA", "0 0 N"*

Tabuleiro complicado com duas lâmpadas:

- Uma de fácil acesso (A).
- Uma de acesso complicado (C). O robô tem que percorrer um caminho em "L"todo no nível c para chegar a esta lâmpada.

Dentro dos tabuleiros complicados este teste corresponde ao tipo de testes mais simples em que é irrelevante acender uma lâmpada antes de outra. No entanto existem casos mais complicados a serem testados em que a ordem das lâmpadas é relevante. Seguem-se dois testes que traduzem estes casos.

- *"aaaAaa", "dddddd", "aaadaa", "aaadaa", "dddddd", "Aaacaa", "aaabaa", "0 0 E"*

Este tabuleiro traduz um caso em que a ordem das lâmpadas é relevante. Este tabuleiro contém duas lâmpadas. Uma delas está localizada num poço, por isso, esta deve ser acesa em último lugar. Caso contrário, o robô não conseguiria voltar atrás para acender a outra lâmpada. Note-se também que no caminho de uma lâmpada para a outra existem mais dois poços que o robô deve evitar, pois se cair num deles ele não conseguirá sair de lá.

- *"aaaaaaaaaaaaaaaa", "aaaaaaaaaaaaaaaa", "aaaaaaaaaaaaaaaA", "aaaaaaZaaaaaaaaa", "aaaaaaUuuaaaaaaa", "aaaaaaaPSaaaaaaa", "aaaaaaMaaCaaaaa", "aaaaaaaIaafaaaaa", "aaaaaaaFfffaaaaa", "6 5 N"*

Este tipo de tabuleiro é o mais complicado. Foi possível resolver este tabuleiro devido à ordenação da lista de lâmpadas apagadas de acordo com a sua prioridade. Neste tabuleiro todas as lâmpadas devem ser acesas por uma ordem específica. De notar que para a resolução deste tabuleiro ser possível o robô deve obrigatóriamente estar posicionado inicialmente no nível mais elevado (neste caso o nível z). O robô deve descer do nível mais alto para o nível mais baixo. Cada nível é superior ou outro em duas ou mais unidades. Deste modo, saltando para um nível mais abaixo torna-se impossível voltar atrás para o nível mais elevado. Como este tabuleiro é relativamente grande, serve também para testar a rapidez do programa face a tabuleiros complicados.

4.0.16 Tarefa 5

Na tarefa 5 considera-se que o *input* é o mesmo que o da tarefa 3. Ou seja, é válido e o robô apenas tem que executar as tarefas fornecidas, tendo-se que apresentar numa página *HTML* o robô a percorrer o tabuleiro.

Pondo isto, considerámos para esta tarefa os mesmos testes considerados para a tarefa 3.

Capítulo 5

Conclusões

Cada tarefa deste trabalho cumpre com tudo aquilo que nos foi pedido. Durante a escrita do código, tivémos sempre em mente duas coisas: simplicidade e rapidez.

Quando falamos da simplicidade, estamos a falar do aspeto que o código tem. Optámos por criar várias funções de tamanho reduzido, em vez do contrário. Utilizámos funções de ordem superior quando vimos que, segundo a nossa experiência, era possível o seu uso. Isto para que o código fosse de fácil leitura e entendimento.

Quando falamos da rapidez, realçamos o facto de na Tarefa 1, 2, 3, 4 e 5, quando era preciso separar o tabuleiro e os restantes dados de entrada, optámos por fazer apenas uma travessia da lista - diminuindo o tempo de execução. Isto fez com que devolvessemos um tuplo com muitos elementos, aumentando a eficiência do programa.

Na Tarefa 4, optámos, numa primeira fase, tentar fazer com que o programa determine o caminho mais curto. Embora tenhamos chegado a uma solução, admitimos que a solução chegada não seja a mais indicada para este trabalho. Deve-se ao facto de, para tabuleiros de grandes dimensões, a nossa solução ser muito lenta ao resolver. Sendo assim, fomos obrigados a criar uma nova solução, por um outro método, chegando à solução atual.

Terminada a tarefa anterior, vimos que ainda tínhamos tempo para tentar executar a parte complicada da Tarefa 5. Sendo assim, tratámos de criar as animações do robô a percorrer o tabuleiro. Terminando isto, e como ainda sobrava tempo, decidimos continuar a aperfeiçoar esta tarefa. Fizemos animações para o robô quando ele movimenta as pernas e quando acende ou apaga uma lâmpada (a antena dele acende ao mesmo tempo). Em último lugar, decidimos centrar a janela onde aparecia o *Lightbot*, alterando também a cor de fundo. Para tudo isto, tivemos que efetuar pesquisas na *Internet* onde pudéssemos aprender sobre isto.