Projeto Sokoban

II FASE

Francisco Oliveira Vitor Peixoto

20 Dezembro 2015

Laboratórios de Informática I

Mestrado Integrado em Engenharia Informática

Conteúdo

Capítulo 1

Introdução

Este relatório foi realizado no âmbito da 2ª fase do projeto proposta na unidade curricular de Laboratórios de Informática I do Mestrado Integrado em Engenharia Informática. Foi-nos então proposto criar o jogo *Sokoban*, que já havia sido trabalhado na 1ª fase deste mesmo projeto, na linguagem *Haskell* em modo gráfico 2D usando como ferramenta a biblioteca *Gloss*.

Dada a sua utilidade para esta fase usamos funções de tarefas da 1ª fase, embora que ligeiramente modificadas e criamos outras funções de raiz, todas devidamente documentadas através da plataforma *Haddock*.

Para chegarmos a resultados concretos adotamos uma postura que se baseia na aprendizagem através do erros. Tudo o que foi desenvolvido nas três tarefas que são explicadas neste relatório foi alcançado através de diversos erros e falhas que nos fizeram aprender o funcionamento da linguagem e as capacidades da biblioteca *Gloss*.

Este relatório está orientado por quatro secções, onde será explorado os problemas a resolver, a nossa abordagem para as suas resoluções, os testes efetuados e por fim as conclusões que tiramos dos nossos erros e do resultado final em si.

Capítulo 2

Descrição do Problema

O ponto fulcral desta 2ª Fase do projeto está relacionado com o desenvolvimento do jogo *Sokoban* de modo a podermos executar uma sequência de comandos até obtermos a vitória ou até esses comandos terem terminado, uma inicialização na biblioteca *Gloss* e por fim a adaptação do jogo à biblioteca, com novas funcionalidades interessantes (como *undo*, *restart* e a seleção de vários níveis de jogo).

Para resolvermos este problema contamos com algumas funções importadas da 1ª Fase do projeto que serão úteis na medida em que auxiliam novas funções, contamos com o auxílio do *GlossExtras.hs*, podemos ainda obter ajuda em algumas explicações que são dadas quanto à utilização da biblioteca *Gloss* em ficheiros disponíveis no *Blackboard* e por fim, temos o sítio sokoban.info para servir de guião ao visual do nosso jogo. Com o auxílio destas ferramentas podemos agora passar à resolução dos problemas propostos para estas três tarefas.

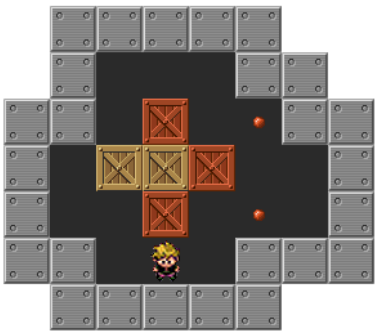

Figura 2.1: *Visual do jogo no sítio* sokoban.info

Capítulo 3

Concepção da Solução

3.1 Tarefa D

Esta tarefa será a responsável pela organização e sequencialização dos comandos. Assim, a biblioteca *Gloss* ainda não será usada nesta tarefa computacional. Com efeito, como dito no enunciado do projeto, esta tarefa não será mais do que uma sequência de comandos efetuados em um só comando.

```
tarefa3' :: [String] -> [String]
tarefa3' tabuleiro = tarefa3Aux (reverse (funcaoPlantaMelhor
    tabuleiro)) (head (sepTabCOORDpares tabuleiro)) (last (init
    tabuleiro)) where
    tarefa3Aux ((x:y):z) (a,b) direcao
        | direcao=="U" = movORnot ((x:y):z) (a,b) (0,1)
        | direcao=="D" = movORnot ((x:y):z) (a,b) (0,-1)
        | direcao=="L" = movORnot ((x:y):z) (a,b) (-1,0)
        | direcao=="R" = movORnot ((x:y):z) (a,b) (1,0)
        | otherwise = error "Comando nao aceite"
```

Listing 3.1: *Função* `tarefa3'` *que dado o tabuleiro, as coordenadas das caixas, do Sokoban e um movimento (U,D,L,R) devolve um par de inteiros com as novas coordenadas do Sokoban.*

A função supradescrita será extremamente útil pois, usando como auxiliar nesta nova tarefa, será capaz de dar a nova coordenada do *Sokoban* após um movimento. Ou seja, uma repetição desta função irá levar o *Sokoban* a fazer a sequência de comandos inserida até chegar à vitória ou até os comandos terem terminado.

A função que executa o conjunto de comandos e devolve a resposta é a função `tarefa4`. Nesta função é introduzido um tabuleiro com as coordenadas das caixas, do *Sokoban* e as sequências de movimentos e devolve `["FIM`

`<tick_count>"]` caso as caixas estejam todas colocadas nos pontos, ou seja, o jogo esteja terminado, onde `<tick_count>` é o número de movimentos funcionais (aqueles em que o *Sokoban* realmente moveu-se) efetuados, ou devolve `["INCOMPLETO <tick_count>"]` caso as caixas não estejam todas colocadas nos pontos, ou seja, o jogo não esteja ainda acabado.

```
1 tarefa4 :: [String] -> [String]
2 tarefa4 tabuleiro = rinceAndRepeat (tabuleiro) (last (init
     tabuleiro))
```

Listing 3.2: *Função* `tarefa4` *que dado o tabuleiro, as coordenadas das caixas, do Sokoban e uma sequência de movimentos ("UDLR"e.g.) devolve o estado do jogo e o número de movimentos funcionais efetuados.*

Esta função `tarefa4` é auxiliada por muitas outras funções, sendo uma das mais relevantes a função `vitoria` que deteta se ainda há algum carater `H` no tabuleiro. Caso ainda haja algum carater `H`, significa que o jogo ainda não está terminado. Esta função devolve `"INCOMPLETO"` ou `"FIM"` caso o jogo ainda não tenha terminado ou já tenha acabado, respetivamente. Esta função é chamada repetidamente na função `rinceAndRepeat` (função semelhante à `tarefa4`) para testar se a cada movimento que o *Sokoban* faça ainda existe algum carater `H`. Por isso é essencial uma função auxiliar que verifique se todas as caixas estão em cima de pontos.

Estas e outras mais funções, que podem ser visualizadas no módulo *Haskell* "trabalhoD.hs" localizado no servidor *SVN*, são essenciais para a funcionalidade desta tarefa.

Testes desta tarefa podem ser visualizados no capítulo Testes deste relatório.

3.2 Tarefa E

A Tarefa E não tem como propósito efetuar desenvolvimentos no jogo, mas sim aperfeiçoar as nossas capacidades e habilidades com a biblioteca *Gloss* através do tipo *Picture*, um tipo que será fundamental na Tarefa F. O objetivo é determinar as dimensões do menor retângulo que envolve uma *Picture*.

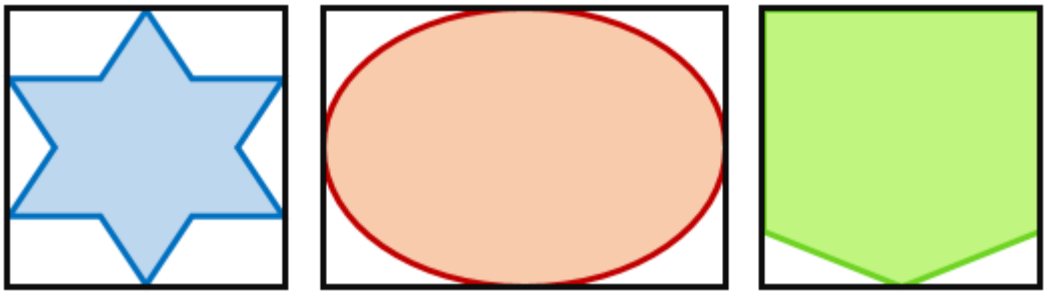

Figura 3.1: *Exemplo de retângulos de menores dimensões que envolve cada figura geométrica.*

Neste caso, uma *Picture* que ao ser criada ou ao sofrer modificações origine unicamente um píxel, deverá ser ignorada. As que originem realmente uma *Picture* serão "admitidas"para uma lista formada unicamente por essa *Picture*.

```
separadordePics :: Picture -> [Picture]

-- Pictures que originam um ponto
separadordePics (Circle 0) = []
separadordePics (Scale 0 _ p) = []
separadordePics (Line []) = []

-- Pictures que nao originam um ponto
separadordePics (Circle f) = (Circle f):[]
separadordePics (Scale f1 f2 p) = (Scale f1 f2 p):[]
separadordePics (Line p) = (Line p):[]

-- Separacao da lista de Pictures
separadordePics (Pictures (p:[])) = (separadordePics p)
separadordePics (Pictures (p:t)) = (separadordePics p)++(
   separadordePics (Pictures t))
```

Listing 3.3: *Alguns exemplos da ação da função* `separadordePics` *nas listas de Pictures.*

A função `separadordePics` separa uma *Picture* nos seus diversos componentes.

Após a *Picture* ser separada nas diversas *Pictures* que a formam, iremos usar a função que, recebendo as *Pictures*, descobre o retângulo de menores dimensões que envolve essa mesma *Picture* usando como suporte duas funções auxiliares, a função `quaisSaoCantos` e a função `kindaComplicated`.

A função `quaisSaoCantos` dada uma *[Point]* que define o Polígono ou Linha devolve o retângulo de menores dimensões que envolve esse mesmo Polígono ou Linha.

A função `kindaComplicated` é a função auxiliar da `listadeCaixasdasPics` que é chamada apenas quando queremos calcular o retângulo de menores dimensões que envolve uma *Picture* quando é sujeita a transformações, tais como `Rotate`, `Translate` ou `Scale`. Esta função foi executada com sucesso para as transformaçãoes `Translate` e `Scale`, no entanto, não conseguimos que o *output* no `Rotate` fosse o esperado.

Nos outros dois casos, `Circle` e `Bitmap` não há a necessidade de recorrer a qualquer função auxiliar, pois, no caso do `Circle` é facil determinar o retângulo que envolve o círculo através do raio.

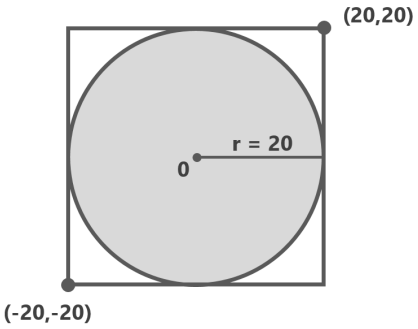

Figura 3.2: *O nosso método de cálculo das coordenadas do quadrado que envolve um círculo de raio 20.*

No caso de um `Bitmap`, como temos de introduzir a largura e comprimento que desejamos que esse `Bitmap` tenha, apenas temos de passar de *Int* para *Float* usando a função predefinida `fromIntegral` e definir as coordenadas do retângulo que envolve esse `Bitmap`, sendo elas ((-w,-h),(w,h)). Sendo o "h"a altura e o "w"o comprimento, já em *Float*.

No fim de calculadas as coordenadas que definem o retângulo somos agora obrigados a retribuir o output esperado para esta tarefa. Esse output será obtido através da função `tarefa5`, uma função que devolve as dimensões do retângulo de menores dimensões que envolve cada *Picture*. Para conseguirmos devolver as dimensões através das coordenadas das extremidades do

retângulo, precisamos de duas funções auxiliares, a `elCanto` e a `elSize`.

A função `elCanto` é a responsável por dizer qual são as coordenadas dos cantos que contém todas as outras coordenadas, ou seja, os pontos que se situam na coordenada mais distante da origem (0). Deste modo podemos obter o retângulo que envolve todas as *Pictures* inicialmente introduzidas.

A função `elSize` é aquela que simplesmente transforma as coordenadas dos cantos do maior retângulo que envolve uma *Picture* nas dimensões desse retângulo.

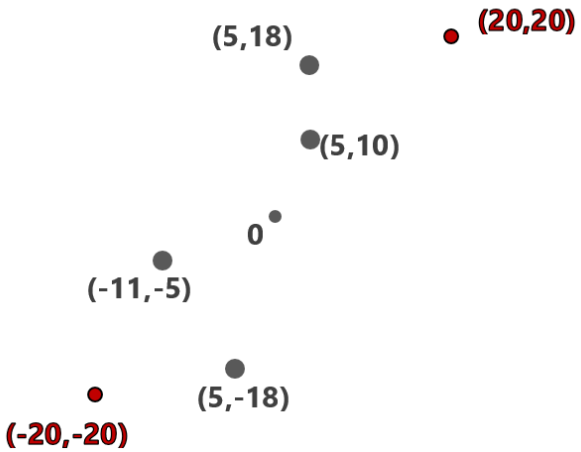

Figura 3.3: *Neste caso, as coordenadas dos cantos do maior retângulo que envolve todos os outros retângulos é ((-20,-20),(20,20)). A dimensão deste retângulo será de (40,40).*

A função `tarefa5` é nada menos do que a função que junta todos estas funções pela sua ordem sequencial correta:

```
1 tarefa5 :: Picture -> (Int, Int)
2 tarefa5 pic = elSize (elCanto (listadeCaixasdasPics (
    separadordePics pic)))
```

Listing 3.4: *Função* `tarefa5` *que dada uma Picture que pode, ou não ser um conjunto de Pictures devolve o retângulo de menores dimensões que envolve essa Picture.*

3.3 Tarefa F

Nesta tarefa iremos realmente criar o jogo *Sokoban* recorrendo às capacidades gráficas da biblioteca *Gloss*. Tendo em conta que o nosso ponto de partida seria inspirado no estilo do sítio sokoban.info criamos um tema baseado no famoso jogo *Super Mario Bros*. Para isso criamos diferentes *bitmaps* inspirados no visual gráfico do jogo, respeitando sempre a jogabilidade e coerência necessários para a execução do *Sokoban*. Estes *bitmaps* têm geralmente uma dimensão de 40x40 e foram criados usando a ferramenta de desenho *MSPaint*. Alguns exemplos podem ser vistos abaixo ou encontrados na sua totalidade na pasta *"bmp"* do servidor *SVN*.

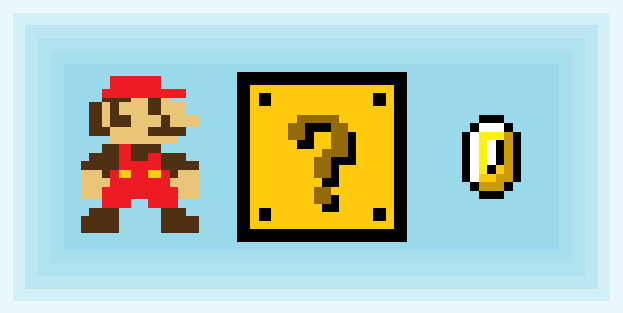

Figura 3.4: *Alguns exemplos dos bitmaps usados no jogo. O Mario representa o jogador. A caixa com o ponto de interrogação representa uma caixa que não está em cima do ponto. E a moeda é o ponto onde devemos colocar a caixa.*

Devemos então passar à inserção dos *bitmaps* no jogo, usando para isso o pacote *Graphics.Gloss.Data.Picture* que nos permite para além de inserir *bitmaps*, desenhar também linhas, círculos, polígonos, texto e efetuar modificações às *Pictures*.

```
1 type Estado = ([String],[Picture],String,Int,String)
2
3 main = do
4    boxoff <- loadBMP "BOXOFF.bmp"
5    boxon <- loadBMP "BOXON.bmp"
6    ponto <- loadBMP "PONTO.bmp"
7    (...)
8    l9 <- loadBMP "L9.bmp"
9    joga (tabuleiros,[boxoff,boxon,ponto,(...),l9],"",0,mapa0)
        desenhaJogo reageEvento
```

Listing 3.5: *Ponto de entrada* `main` *que carrega os diversos bitmaps e executa a função* `joga` *e o type Estado.*

```
joga :: Estado -> (Estado -> Picture) -> (Event -> Estado ->
   Estado) -> IO ()
joga tabuleiro desenha reage = play (FullScreen (1366,768))
(makeColorI 153 217 234 1) 90
tabuleiro desenha reage (\time estado -> estado)
```

Listing 3.6: *Função* `joga`

A função `joga` é a equivalente à função `display` definida no pacote *Graphics.Gloss* e tem como função atualizar e desenhar a janela. Dado um tabuleiro e a lista das *Pictures* [1] desenha o tabuleiro pedido (*mapa0*; *mapa1*; etc.) e atualiza o seu aspeto de acordo com os eventos introduzidos na função `reageEvento`.

Esta janela vai estar em formato *FullScreen* com a dimensão de (1366x768) píxeis. E terá uma cor de fundo à nossa escolha, que foi definida com o auxílio da função `makeColorI` importada do pacote *Graphics.Gloss.Data.Color*. Sendo que o jogo não deverá sofrer alterações automaticamente com o passar do tempo, definimos a função anónima onde o tempo não é um fator influenciador do estado da janela: `(time estado -> estado)` . Os mapas dos jogos estão definidos em funções com o seu respetivo nome. Estas funções são agrupadas numa lista na função `tabuleiros`.

```
mapa0 :: String
mapa0 = outStr ["#######","# #","# .#","# .#","#######","1 1","2
    2","2 1",""]

tabuleiros :: [String]
tabuleiros = [mapa0,mapa1,(...),mapa9]
```

Listing 3.7: *Função que define o* `mapa0` *e função* `tabuleiros` *que agrupa os mapas numa lista.*

[1] Sempre que esteja escrita a lista de *Pictures* desta forma: `[boxoff,boxon,ponto,(...),l9]` as reticências representam todos as outras *Pictures* que foram ocultadas por questões de espaço. A lista completa seria: `[boxoff,boxon,ponto,chao,parede,mario,undo,restart,sokoban,victory,help,l1,l2,l3,l4,l5,l6,l7,l8,l9,l0]` e deverá sempre ser entendido desta forma.

Figura 3.5: *Seleção dos níveis (mapas) do jogo .*

Para este jogo foram definidos 10 níveis com um grau de dificuldade geralmente crescente de acordo com os respetivos níveis. Os níveis são selecionados através de um clique na *Picture* representativa do nível que queremos jogar ou através da tecla numérica correspondente ao número do nível.

O evento ocorrido após o clique do rato só é possível graças à função `reageEvento`. Esta função efetua modificações no jogo através da introdução de comandos pelo teclado e/ou rato usando capacidades de um pacote extremamente útil da biblioteca *Gloss*, *Graphics.Gloss.Interface.Pure.Game*.

```
1 -- Abre o Nivel 9
2 reageEvento :: Event -> Estado -> Estado
3 reageEvento (EventKey (MouseButton LeftButton ) (Down) (
    modifiers) (x,y)) (tabuleiro,[boxoff,boxon,ponto,(...),l9],
    moves,lvl,mapaatm)
4    | ((x>=(205))&&(x<=(245))&&(y>=(((fromIntegral (length (
         funcaoPlantaMelhor (init (inStr mapaatm)))*(20)))+10)))
         &&(y<=((fromIntegral (length (funcaoPlantaMelhor (init
         (inStr mapaatm)))*(20)))+50)) ) = (tabuleiro,[boxoff,
         boxon,ponto,chao,parede,mario,undo,restart,sokoban,
         victory,help,l1,l2,l3,l4,l5,l6,l7,l8,l9,l0],moves,9,
         mapa0)
```

Listing 3.8: *Excerto da função* `reageEvento`

Neste excerto da função `reageEvento` supradescrito, é chamado o construtor *EventKey*, onde um clique com o botão esquerdo do rato (`(MouseButton LeftButton)`) dentro da área da janela definida pelas coordenadas (x,y) provoca uma alteração em `lvl`, um valor inteiro, que passa a ter o valor de 9. Este valor está localizado no *type Estado*. A função `joga` irá ser executada novamente, sofrendo uma alteração:

```
1 -- Antes
2 joga (tabuleiros,[boxoff,boxon,(...),l9],"",0,mapa0) desenhaJogo
      reageEvento
3
4 -- Depois
5 joga (tabuleiros,[boxoff,boxon,(...),l9],"",9,mapa9) desenhaJogo
      reageEvento
```

Listing 3.9: *A função* `joga`*, que antes tinha o valor de 0, passa agora a ter o valor 9, fazendo com que apareça o mapa que escolhemos, o mapa9.*

Assim, o construtor *EventKey* será extremamente útil, pois vai ser através dele que iremos introduzir as alterações que queremos fazer no jogo. Os movimentos do *Sokoban* são introduzidos relacionando a tecla que se preme e introduzindo uma *String* com o carater responsável ao movimento do *Sokoban* como foi feito na `tarefa4` com "U", "D", "L" ou "R". Como demonstra o exemplo abaixo, ao carregar na tecla `KeyDown` irá inserir um "D"na lista de `moves`, levando o *Sokoban* a descer uma posição para baixo.

```
1 -- Move para baixo
2 reageEvento (EventKey (SpecialKey KeyDown ) (Down) (_) (pos)) (
    tabuleiro,[boxoff,boxon,ponto,(...),l9],moves,lvl,mapaatm) =
    (tabuleiro,[boxoff,boxon,ponto,(...),l9],moves++"D",lvl,
    mapaatm)
```

Listing 3.10: *Excerto da função* `reageEvento` *que desloca o Sokoban uma posição abaixo.*

Para além das ferramentas obrigatórias para o normal funcionamento deste jogo, foi também criada a possibilidade de fazer *undo* e *restart* ao jogo através do uso das teclas Z e X, respetivamente e também com clique numa *Picture*.

Para o *undo* funcionar basta retroceder um movimento na lista de `moves`, para isso temos de recorrer à função predefinida `init` que devolve todos os elementos da lista de `moves` exceto o último, ou seja, remove o último movimento. Isto não se aplica caso ainda não tenha havido nenhum movimento, nesse caso, carregar na tecla 'Z' não irá provocar nenhum acontecimento.

No caso do *restart* apenas temos de dizer que a lista de `moves` ainda não recebeu nenhum movimento, ou seja, uma lista vazia.

```
1 -- Reinicia o jogo
2 reageEvento (EventKey (Char 'x')(Down)(_)(pos)) (tabuleiro,[
    boxoff,boxon,ponto,(...),l9],moves,lvl,mapaatm) = (tabuleiro
    ,[boxoff,boxon,ponto,(...),l9],[],lvl,mapaatm)
3
4 -- Retrocede um movimento
5 reageEvento (EventKey (Char 'z')(Down)(_)(pos)) (tabuleiro,[
    boxoff,boxon,ponto,(...),l9],moves,lvl,mapaatm) = if (moves
    /=[])
6 then (tabuleiro,[boxoff,boxon,ponto,(...),l9],init moves,lvl,
    mapaatm)
7 else (tabuleiro,[boxoff,boxon,ponto,(...),l9],moves,lvl,mapaatm)
```

Listing 3.11: *Excerto da função* `reageEvento` *que é responsável por reiniciar a partida ou retroceder um movimento, usando as teclas X e Z.*

O *undo* e *restart* é também executado, usando o clique do rato numa exata área. Essa área é definida da mesma forma que foi definida para a seleção de níveis e depois basta introduzir uma maneira de definir o *output* exatamente igual à usada em cima.

Assim sendo temos duas maneiras distintas de fazer *undo* ou *restart*.

Figura 3.6: *Duas maneiras existentes de reiniciar a partida ou retroceder um movimento.*

A função responsável por desenhar o mapa e outras *Pictures* foi denominada de `desenhaJogo`.

```
1  desenhaJogo :: Estado -> Picture
2  desenhaJogo (tabuleiro,[boxoff,boxon,ponto,(...),l9],moves,lvl,
       mapaatm) =
3     Pictures [
4        -- Insere o mapa
5        ( Translate (fromIntegral (length (head (init (inStr
             mapaatm)))*(-20))) (fromIntegral (length (
             funcaoPlantaMelhor (init (inStr mapaatm)))*(-20))) (
             Pictures (desenhaJogoAux (reverse ( funcaoRemoveCard(
             tarefa4 ( (init (init (inStr mapaatm)))++[moves]++[""]
             ) )),[boxoff,boxon,ponto,(...),l9]) (0,0) ) ) ),
6        -- Insere o bitmap "undo"
7        (Translate (100) ((fromIntegral (length (
             funcaoPlantaMelhor (init (inStr mapaatm)))*(-20)))-40)
             undo),
8        -- Insere o bitmap "restart"
9        (Translate (-40) ((fromIntegral (length (
             funcaoPlantaMelhor (init (inStr mapaatm)))*(-20)))-40)
             restart),
10       -- Insere o bitmap "sokoban"
11       (Scale 1.1 1.1 (Translate (0) ((fromIntegral (length (
             funcaoPlantaMelhor (init (inStr mapaatm)))*(20)))+92)
             sokoban)),
12       (...)
```

Listing 3.12: *Excerto da função* `desenhaJogo` *que é responsável por desenhar as Pictures presentes na janela.*

Esta função irá desenhar cada *Picture* que queremos que apareça na janela. As *Pictures* que ficam estáticas no jogo são de fácil inserção. Apenas temos de recorrer ao construtor `Translate`, definir as coordenas X e Y da sua localização [2] e invocar a *Picture* que queremos colocar naquela posição. Este processo pode ser visto no excerto supradescrito da mesma função.

Por outro lado, para inserir as *Pictures* que formam o mapa na janela, o processo torna-se mais complicado. É também necessário recorrer ao construtor `Translate`, mas neste caso as *Pictures* que formam o mapa é composto por várias *Pictures* e estão em constante mudança de acordo com os movimentos introduzidos na função `reageEvento`. Assim, será necessário recorrer a uma função auxiliar, que recebe a `tarefa4` com o mapa e os seus movimentos a serem dados na lista `moves` e por fim com uma *String* vazia concatenada ao mapa. É também dado à função auxiliar a lista de *Pictures* que irão ser necessárias à construção do tabuleiro de jogo e um par de inteiros que indica que o mapa é desenhado a partir da coordenada (0,0), ponto a partir do qual todas as *Pictures* que formam o mapa são desenhadas.

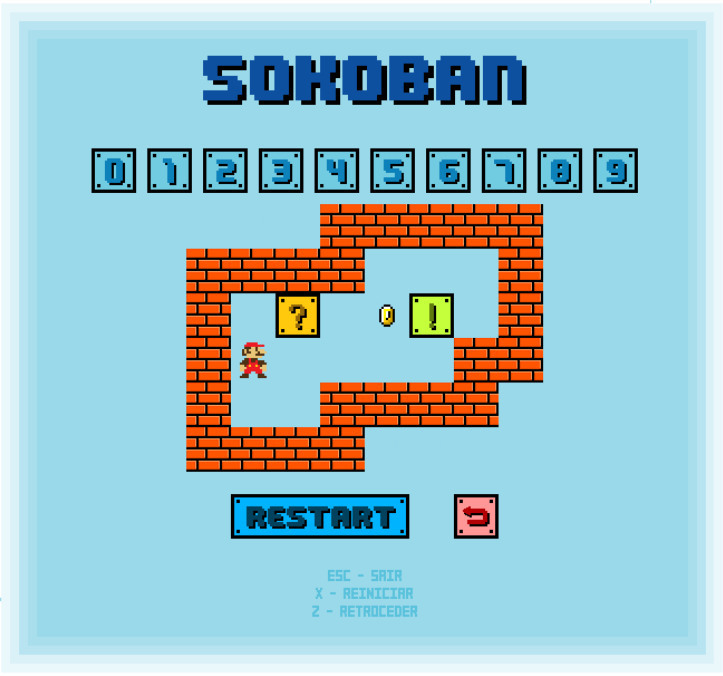

Figura 3.7: *Resultado final do jogo.*

[2] A localização das *Pictures*, em alguns casos, vai ser relativa ao tabuleiro, por isso calculamos o `length` do mapa para podermos definir a localização (Y) da *Picture* através da altura do mapa

Capítulo 4

Testes

4.1 Tarefa D

Nesta secção são executados diversos testes à Tarefa D. Estes testes estão também disponíveis no servidor *SVN* do nosso grupo de trabalho no formato ".txt"para serem executados, e provar a sua veracidade se necessário, na linha de comandos.

Teste D.1:

```
*Main> tarefa4 ["#####","#.  .#","#   #","#   #","#####","2
1","1 3","3 3","RU"," "]
["FIM 0"]
```

Teste D.2:

```
*Main> tarefa4 ["#######","#     #","#    .#","#    .#","####
###","3 1","4 1","5 2","R",]
["FIM 1"]
```

Teste D.3:

```
*Main> tarefa4 ["#####","#.  .#","#   #","#   #","#####","2
1","1 2","3 2","RUDLLU"," "]
["FIM 6"]
```

Teste D.4:

```
*Main> tarefa4 ["#####","#.  .#","#   #","#   #","#####","2
1","1 2","3 2","DDUDDDRUDLLU"," "]
["FIM 8"]
```

Teste D.5:

```
*Main> tarefa4 ["#######","#     #","#    .#","#    .#","####
###","1 1","2 2","2 1","RRRUUULLLDRRR"," "]
["FIM 12"]
```

Teste D.6:

```
*Main> tarefa4 ["#####","#.  .#","#   #","#   #","#####","2
1","1 2","3 2","UURLLRDDRUDLLU"," "]
["FIM 14"]
```

Teste D.7:

```
*Main> tarefa4 ["########","####   #","#   .. #","#     ##",
"#  #####","#### ####","4 4","3 3","4
3","DLDLLURRRURRDLDLLURLLLDDRULURR"," "]
["FIM 30"]
```

Teste D.8:

```
*Main> tarefa4 ["######","#  ###","#  ###","#  ###","#  ###",
"#.   #","#  . #","#  ###","######","4 3","2 3","3 3",
"DLLUUUDDDRRUULLDLURUULUURDDDDLDRUUULD"," "]
["FIM 36"]
```

Teste D.9:

```
*Main> tarefa4 ["######","#  ###","#  ###","#  ###","#  ###",
"#.   #","#  . #","#  ###","######","4 3","2 3","3 3",
"DDDUUULLLRRRRRRLLLUUUDDD"," "]
["INCOMPLETO 3"]
```

Teste D.10:

```
*Main> tarefa4 ["########","####   #","#   .. #","#     ##",
"#  #####","#### ####","4 4","3 3","4 3","UUDLDR"," "]
["INCOMPLETO 4"]
```

Teste D.11:

```
*Main> tarefa4 ["#####","#.  .#","#   #","#   #","#####","2
1","1 2","3 2","RUDLL"," "]
["INCOMPLETO 5"]
```

Teste D.12:

```
*Main> tarefa4 ["########","####   #","#   .. #","#     ##",
"#  #####","#### ####","4 4","3 3","4 3","DLLUDRR"," "]
["INCOMPLETO 6"]
```

Teste D.13:

```
*Main> tarefa4 ["######","#  ###","#  ###","#  ###","#  ###",
"#.   #","#  . #","#  ###","######","4 3","2 3","3 3",
"DLLUUUU"," "]
["INCOMPLETO 7"]
```

Teste D.14:

```
*Main> tarefa4 ["#######","#     #","#    .#","#    .#","####
###","1 1","2 2","2 1","RRUULLDRR",]
["INCOMPLETO 9"]
```

4.2 Tarefa E

Nesta secção são apresentados diversos testes à Tarefa E. Estes testes estão também disponíveis no servidor *SVN* do nosso grupo de trabalho no formato ".txt"para serem executados, e provar a sua veracidade se necessário, na linha de comandos. No caso do Teste E.7 o resultado apresentado não é o correspondente ao que deveria ser na realidade. Devia dar (10,20) no entanto, como não conseguimos chegar a bom porto no que toca a este construtor, o resultado apresentado (0,0) não é o correto.

Teste E.1:

```
*Main> tarefa5 (Line [(0,0),(20,0),(30,10),(20,20),(-30,0)])
(60,20)
```

Teste E.2:

```
*Main> tarefa5 (Pictures [ Translate (-10) 0 (Circle 20),
Translate 10 0 (Circle 15)])
(55,40)
```

Teste E.3:

```
*Main> tarefa5 (Scale 2 3 (Line [(0,0),(20,0),(10,10),(0,0)]))
(40,30)
```

Teste E.4:

```
*Main> tarefa5 (Translate 10 10 (Circle 15))
(30,30)
```

Teste E.5:

```
*Main> tarefa5 (Circle 10)
(20,20)
```

Teste E.6:

```
*Main> tarefa5 (Line [(0,0),(20,0),(10,10),(0,0)])
(20,10)
```

Teste E.7:

```
*Main> tarefa5 (Rotate 90 (Line [(0,0),(20,0),(10,10),(0,0)]))
(0,0)
```

Teste E.8:

```
*Main> tarefa5 (Translate 30 30 Blank)
(0,0)
```

Teste E.9:

```
*Main> tarefa5 (Pictures [(Translate 10 10 (Circle 15)),
(Scale 10 10 Blank),(Polygon [(10,20),(10,40),(30,40)])])
(35,45)
```

Teste E.10:

```
*Main> tarefa5 (Scale 0 10 (Polygon [(10,20),(10,40),
(30,40)]))
(0,0)
```

4.3 Tarefa F

Nesta secção são apresentados diversos mapas criados para a Tarefa F. Estes testes estão também disponíveis no servidor *SVN* do nosso grupo de trabalho no formato ".txt"para serem apenas visualizados. A sua execução é apenas possível através do módulo *Haskell* "trabalhoF.hs"ou no executável "Sokoban.exe".

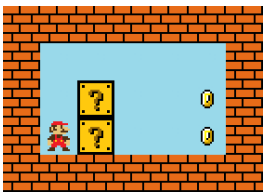

Figura 4.1: *Teste F.1*

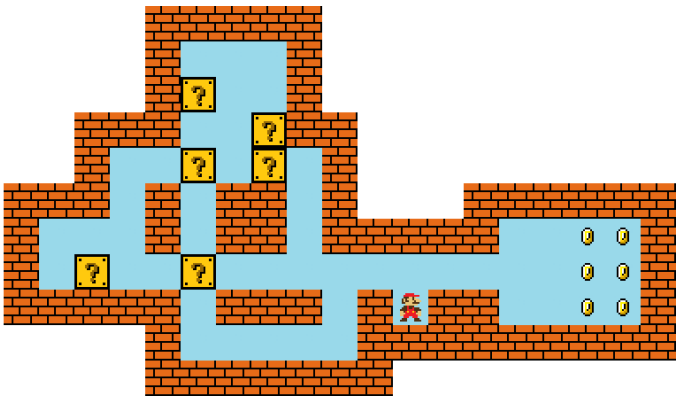

Figura 4.2: *Teste F.2*

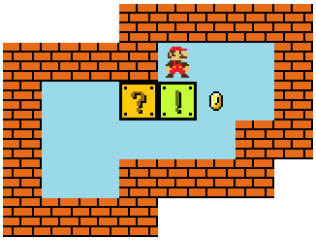

Figura 4.3: *Teste F.3*

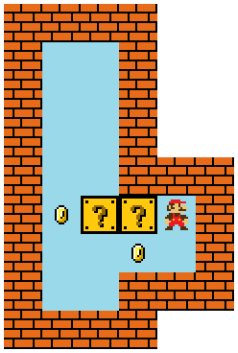

Figura 4.4: *Teste F.4*

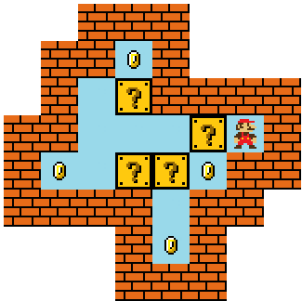

Figura 4.5: *Teste F.5*

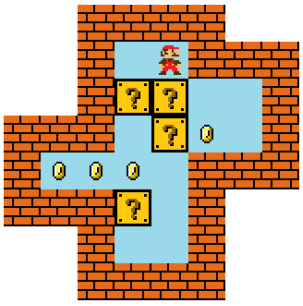

Figura 4.6: *Teste F.6*

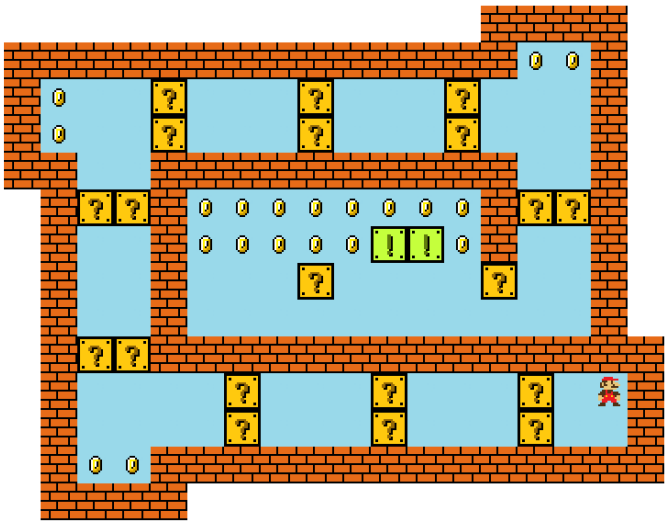

Figura 4.7: *Teste F.7*

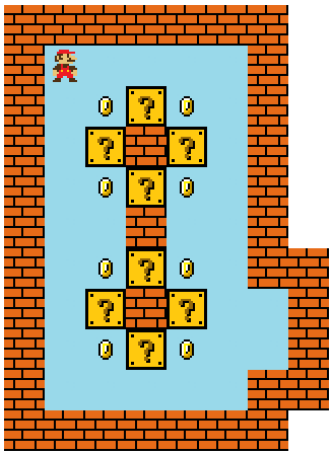

Figura 4.8: *Teste F.8*

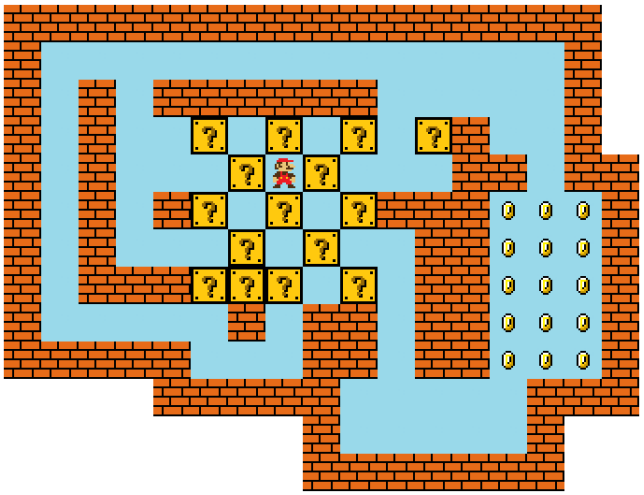

Figura 4.9: *Teste F.9*

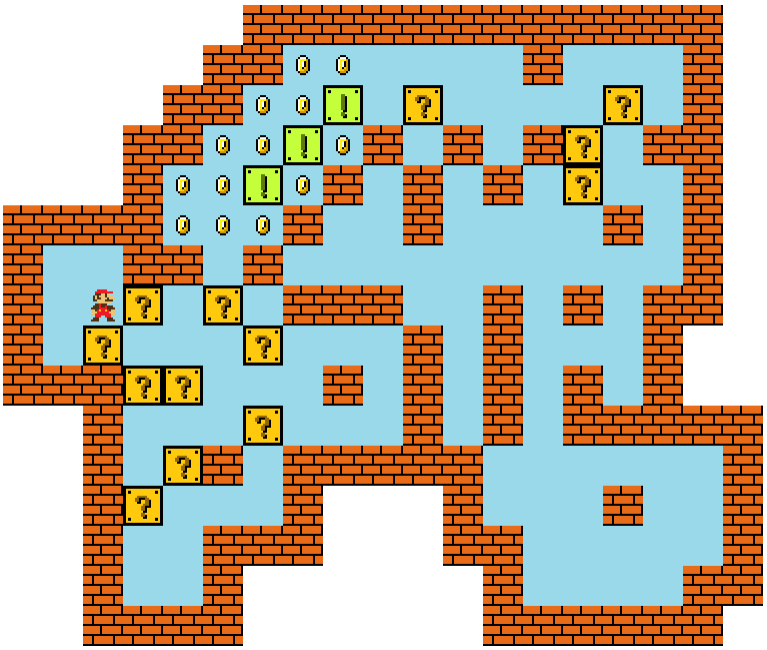

Figura 4.10: *Teste F.10*

Capítulo 5

Conclusões

A partir do problema inicial, procuramos de todas as formas encontrar uma solução viável para resolver cada uma das tarefas propostas. A partir daí foram surgindo outros problemas que após várias tentativas foram na sua maioria resolvidos, outros não infelizmente.

Assim podemos concluir que a linguagem de programação funcional *Haskell* é bastante útil e cria inúmeras possibilidades quando potenciada com a biblioteca *Gloss* e os pacotes que dela derivam. Apesar das potencialidades existentes da linguagem *Haskell*, é ainda assim uma linguagem difícil de trabalhar comparada a outras linguagens que os elementos deste grupo de trabalho já têm conhecimento e acima de tudo é uma linguagem que requer muita atenção ao mais ínfimo detalhe, pois é muito resumida.

As nossas dificuldades focaram-se principalmente na adaptação ao *Gloss*. Enquanto que na Tarefa F foi necessário fazer alguma pesquisa *online* em relação às capacidades da biblioteca e como saber aproveitar das mesmas, na Tarefa E tivemos mais dificuldades principalmente em relação a como criar uma função `rotate` que foi a única função que não conseguimos concluir.

A partir dos nossos erros e daqueles que conseguimos corrigir conseguimos chegar ao resultado final, concluindo assim que o *Haskell* é uma boa linguagem para criar jogos básicos em 2D quando auxiliados com o *Gloss* e será certamente uma escolha no nosso futuro escolar e profissional quando nos for proposto um desafio semelhante a este.

Lista de Funções

Lista de Figuras