Princípios de Programação

Trabalho 4

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

2022/2023

O objetivo deste trabalho é desenvolver um *programa* para jogar aos *labirintos*. Vamos começar por relembrar o que já foi dito nos trabalhos 1, 2 e 3. Podemos representar um labirinto como uma lista de listas de caracteres, ou equivalentemente, uma lista de strings. Cada caracter denota uma das posições possíveis:

- `'S'` : início
- `'F'` : fim
- `' '` : espaço
- `'*'` : parede
- `'a'`, `'b'` ou `'c'` : chaves
- `'A'`, `'B'` ou `'C'` : portas
- `'@'` : portais

Nem todas as combinações de strings correspondem a labirintos interessantes. Para este trabalho, apenas estamos interessados em labirintos *válidos*, obedecendo a três restrições:

1. A orla exterior do labirinto é composta exclusivamente por paredes (`'*'`).
2. O labirinto tem exatamente uma posição de início (`'S'`) e uma posição de fim (`'F'`).
3. O labirinto tem exatamente zero ou dois portais (`'@'`).

Estas são as únicas restrições que devem considerar. Em particular, podem existir chaves sem portas associadas, portas sem chaves associadas, chaves repetidas, portas repetidas, e labirintos sem solução.

A. Aplicação de jogo Devem produzir um ficheiro `Main.hs` que deverá conter uma função `main` de modo a poder ser executado pela linha de comandos, através de um comando do tipo

```
> stack ghc Main.hs
> ./Main [ficheiro]
```

onde `ficheiro` é um argumento opcional que corresponde a um ficheiro com toda a informação de um estado de jogo (abaixo descreveremos a estrutura destes ficheiros). O comando acima lê o ficheiro passado como argumento, imprime o estado do jogo no **`stdout`**, e aguarda instruções no **`stdin`**. Exemplo de execução:

```
> ./Main default.map
*******
*  @  *
* ***B*
*a*P*F*
*** ***
*bA  @*
*******
chaves:
                        <-- [A aguardar instruções]
```

Notem que o ficheiro é um argumento opcional. Se este não estiver presente, a vossa aplicação deve carregar o ficheiro `default.map` por omissão.
Depois de carregado o estado do jogo, a vossa aplicação deve estar preparada para ler uma linha do **`stdin`** e seguir uma das seguintes instruções:

A1. Movimentos A instrução `move movimentos` deve executar a sequência `movimentos` passada como string. Considerem quatro movimentos possíveis: uma posição para cima (representado por `'u'`, 'up'), uma posição para a esquerda (representado por `'l'`, 'left'), uma posição para a direita (representado por `'r'`, 'right') e uma posição para baixo (representado por `'d'`, 'down'). A implementação deverá obedecer às mesmas regras da função
`move :: EstadoJogo -> String -> EstadoJogo` do trabalho 3.
Voltem a reler o enunciado desse trabalho para relembrar o comportamento desejado.
Após executar os movimentos, o vosso programa deve imprimir o estado do jogo no **`stdout`**, e aguardar a próxima instrução no **`stdin`**. Exemplo:

```
> ./Main default.map
*******
*  @  *
* ***B*
*a*P*F*
*** ***
```

```
*bA  @*
*******
chaves:
move ddrr
*******
*  P  *
* ***B*
*a*S*F*
*** ***
*bA  @*
*******
chaves:
                    <-- [A aguardar instruções]
```

A2. Carregar um jogo A instrução `load ficheiro` deve ler o ficheiro passado como string, imprimir o estado do jogo no **`stdout`**, e aguardar a próxima instrução no **`stdin`**. A partir deste momento, passamos a jogar no jogo que foi carregado. No exemplo abaixo, fazemos uma sequência de movimentos e voltamos a carregar o labirinto original.

```
> ./Main default.map
*******
*  @  *
* ***B*
*a*P*F*
*** ***
*bA  @*
*******
chaves:
move ddrr
*******
*  P  *
* ***B*
*a*S*F*
*** ***
*bA  @*
*******
chaves:
load default.map
*******
*  @  *
* ***B*
*a*P*F*
*** ***
*bA  @*
*******
```

```
chaves:
                        <-- [A aguardar instruções]
```

A3. Guardar um jogo A instrução `save ficheiro` deve guardar o estado atual do jogo para o ficheiro passado como string, imprimir o estado do jogo no **`stdout`**, e aguardar a próxima instrução no **`stdin`**. Exemplo:

```
> ./Main default.map
*******
*  @  *
* ***B*
*a*P*F*
*** ***
*bA  @*
*******
chaves:
move ddrr
*******
*  P  *
* ***B*
*a*S*F*
*** ***
*bA  @*
*******
chaves:
save save001.map
*******
*  P  *
* ***B*
*a*S*F*
*** ***
*bA  @*
*******
chaves:
                        <-- [A aguardar instruções]
```

Após este exemplo, é criado um ficheiro `save001.map` com o estado atual do jogo.

A4. Sair do jogo A instrução `exit` deve terminar a execução do programa, regressando à linha de comandos.

```
> ./Main default.map
*******
*  @  *
* ***B*
```

```
*a*P*F*
*** ***
*bA  @*
*******
chaves:
exit
>                    <-- [Programa terminado]
```

A5. Estrutura dos ficheiros Os estados de jogo deverão ser guardados como ficheiros de texto, podendo utilizar as extensões `.txt` ou `.map`. Devem respeitar a seguinte estrutura para os ficheiros:

- a primeira linha é um par de coordenadas `(i,j)` com a posição do jogador.
- a segunda linha é uma string com as chaves já adquiridas.
- as restantes `n` linhas são a representação textual do labirinto.

Por exemplo, o conteúdo do ficheiro `default.map` é

```
(3,3)

*******
*  @  *
* ***B*
*a*S*F*
*** ***
*bA  @*
*******
```

Neste exemplo, a segunda linha é vazia pois ainda não foram adquiridas chaves. Um exemplo com chaves adquiridas é o ficheiro `05_savein.map`, cujo conteúdo é

```
(5,7)
bc
*********
*   *  a*
*** ** **
* A S   *
* * *****
* *     *
*B*******
*   C  F*
*********
```

A6. Bateria de testes Para vos ajudar a verificar a vossa implementação, disponibilizámos um modelo de submissão com cinco exemplos no moodle. Cada exemplo inclui: um ficheiro `XX_savein.map`, uma sequência de inputs `XX_input.txt`, um output esperado `XX_check.txt` e o labirinto resultante `XX_savecheck.map`. Podem usar o comando `diff` para verificar que o vosso programa produz o resultado esperado.

```
> ./Main 01_savein.map < 01_input.txt > 01_output.txt
> diff 01_output.txt 01_check.txt
- [Não produz output]
> diff 01_saveout.map 01_savecheck.map
- [Não produz output]
```

B. Testes

B1. Definição de propriedades A vossa aplicação deve incluir pelo menos *oito* testes QuickCheck relacionados com a função `move :: EstadoJogo -> String -> EstadoJogo` do trabalho 3.

- Um teste que verifique a propriedade: as dimensões de um labirinto não mudam após efetuar uma sequência de movimentos.
- Um teste que verifique a propriedade: o jogador não sai dos limites do mapa após efetuar uma sequência de movimentos.
- Um teste que verifique a propriedade: o número de chaves na posse do jogador não diminui após efetuar uma sequência de movimentos.
- Um teste que verifique a propriedade: o número de portas presentes num labirinto não aumenta após efetuar uma sequência de movimentos.
- Outros quatro testes da vossa autoria. Procuramos testes interessantes, capazes de apanhar erros subtis, e não testes triviais tais como: o número de linhas de um labirinto é maior ou igual a zero.

Cada teste deverá vir acompanhado de um breve comentário que explica a propriedade a ser testada. Os testes são exercitados quando se usa a flag `-t` na linha de comandos.

```
> ./Main -t
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

B2. Geração de dados de teste Para poder utilizar a ferramenta QuickCheck, devem tornar o tipo de dados `EstadoJogo` instância da classe `Arbitrary`. A vossa implementação da função `arbitrary` deve gerar apenas instâncias válidas de labirintos, obedecendo às três restrições indicadas no início do enunciado.
Para além disso, se se limitarem a gerar uma **`String`** aleatória, é pouco provável que a **`String`** corresponda a uma sequência válida de movimentos (só com `'u','l','r','d'`). Sugerimos que definam um tipo para sequências de movimentos válidas, do estilo

```
newtype Movimentos = Movimentos String
```

e que tornem o novo tipo instância da classe `Arbitrary`. Este tipo serve apenas para gerar input válido para testes, não devendo extravazar o módulo dos testes (não deve aparecer na lista de exportação do módulo).

C. Erros na interação A vossa aplicação não se deve atrapalhar com argumentos inválidos na linha de comandos. Em vez disso deverá imprimir uma pequena explicação sobre a utilização da aplicação (usage). Por exemplo:

```
> ./Main olá como vais?
Utilização:
  ./Main [ficheiro] -- carrega um ficheiro para jogar
  ./Main            -- carrega o ficheiro default.map
  ./Main -t         -- corre os testes
```

Como utilizações inválidas, considerem chamar a aplicação com mais do que um argumento; ou chamar a aplicação com um ficheiro que não existe (espreitem a função `doesFileExist` do módulo **`System`**`.Directory`).

D. Organização em módulos A vossa aplicação deverá estar organizada em vários módulos. O módulo `Main` deverá apenas conter a parte do código que faz interação com o mundo exterior (as funções **`IO`**). Sugerimos um módulo separado para os testes e um ou mais para as funções sobre labirintos. A declaração de cada módulo deve listar explicitamente os tipos de dados e funções exportados (exportando apenas o que fizer sentido).
Na página do moodle fornecemos um modelo de submissão, que podem utilizar. Este modelo consiste numa única pasta com

- um ficheiro `Main.hs`,
- alguns labirintos `default.map`, `01_savein.map`, `01_savecheck.map`, ..., `05_savein.map`, `01_savecheck.map`,
- ficheiros de texto para simular input/output `01_input.txt`, `01_check.txt`, ..., `05_input.txt`, `05_check.txt`.

Ao submeter o vosso código podem apagar os ficheiros `.map` e `.txt`, deixando *apenas* os ficheiros `.hs`.

Pontos de atenção

1. O vosso trabalho deverá ser constituído por um ficheiro zip de nome `t4_XXXXX_YYYYY.zip`, onde `XXXXX`, `YYYYY` são os vossos números de aluno (por ordem crescente). O ficheiro zip deverá conter no mínimo um ficheiro `Main.hs`, bem como outros módulos adicionais que julguem relevantes.

2. Para simplificar a sua implementação, podem assumir que todos os ficheiros passados como argumento existem e constituem labirintos válidos.

3. Notem que um labirinto válido pode ter dimensões arbitrárias (não restrinjam o código a labirintos `5x5` ou `7x7`).

4. Os trabalhos serão avaliados semi-automaticamente. Respeitem a sintaxe das instruções para correr o executável `./Main`.

5. Cada função (ou expressão) que escreverem deverá vir sempre acompanhada de uma assinatura. Isto é válido para as funções ou expressões enunciadas acima bem como para outras funções ou expressões ajudantes que decidirem implementar.

6. Lembrem-se que as boas práticas de programação Haskell apontam para a utilização de várias funções simples em lugar de uma função única mas complicada.

7. Iremos considerar os seguintes pontos para avaliar o vosso trabalho: percentagem de testes (da bateria preparada pelos docentes) passados automaticamente; legibilidade, organização e qualidade do código; qualidade das propriedades QuickCheck implementadas.

Entrega. Este é um trabalho de resolução em pares. Os trabalhos devem ser entregues no Moodle até às 23:55 do dia 12 de dezembro de 2022.

Plágio. Os trabalhos de todos os alunos serão comparados por uma aplicação computacional. Relembramos aqui um excerto da sinopse: "Alunos detetados em situação de fraude ou plágio, plagiadores e plagiados, ficam reprovados à disciplina (sem prejuízo de ser acionado processo disciplinar concomitante)".