Universidade do Minho
Escola de Engenharia
Departamento de Informática

Projecto de Laboratórios de Informática I

Licenciatura em Engenharia Informática

1º Ano — 1º Semestre

Ano Lectivo 2021/2022

— Fase 1 de 2 —

Data de Lançamento: 30 de Outubro de 2021

Data Limite de Entrega: 27 de Novembro de 2021

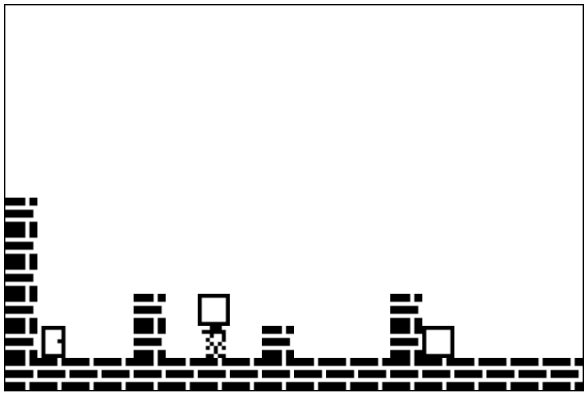
Outubro de 2021

1 Introdução

Neste enunciado apresentam-se as tarefas referentes à primeira fase do Projecto de Laboratórios de Informática I 2021/2022. O projecto consiste na implementação faseada de um pequeno jogo utilizando a linguagem *Haskell*. Por seu turno, cada fase consiste na implementação de um conjunto de *Tarefas*, definidas na secção seguinte.

1.1 Descrição do jogo

O jogo a implementar na presente edição é conhecido como *Block Dude*[1], onde o objectivo consiste em controlar um personagem até à porta de um mapa por intermédio de comandos muito simples.



Num mapa, para além do personagem e da porta, existem também blocos de pedra e caixas, sendo que é permitido ao personagem (*i.e.* jogador) mover as caixas para assim conseguir chegar à porta.

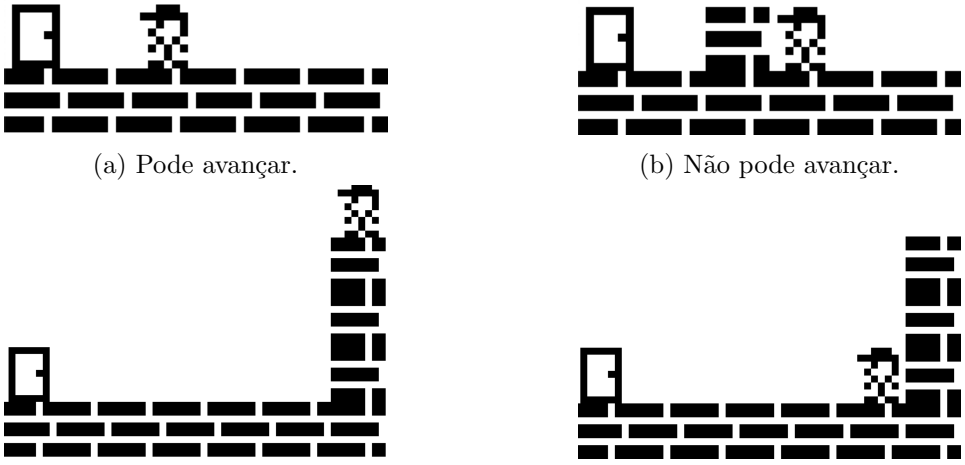
Os únicos movimentos permitidos ao personagem são carregar ou largar uma caixa, avançar nas direcções *Este/Oeste* e trepar um obstáculo (*i.e.* bloco de pedra ou caixa). Caracterizamos de seguida estes movimentos.

1.1.1 Como e quando pode o personagem avançar

O personagem apenas pode avançar uma unidade na direcção a que está voltado desde que esse espaço esteja livre de obstáculos. Na Figura 1a o personagem pode avançar e ficar assim mais próximo da porta. Já na Figura 1b o personagem não pode avançar uma vez que existe um obstáculo

[1] http://azich.org/blockdude/

(bloco de pedra) à sua frente. Em caso de queda iminente (Figura 1c), o personagem avança directamente para a última posição da queda (Figura 1d). Na secção 2.4 explicamos como pode o personagem trocar a direcção a que está voltado.



(a) Pode avançar.

(b) Não pode avançar.

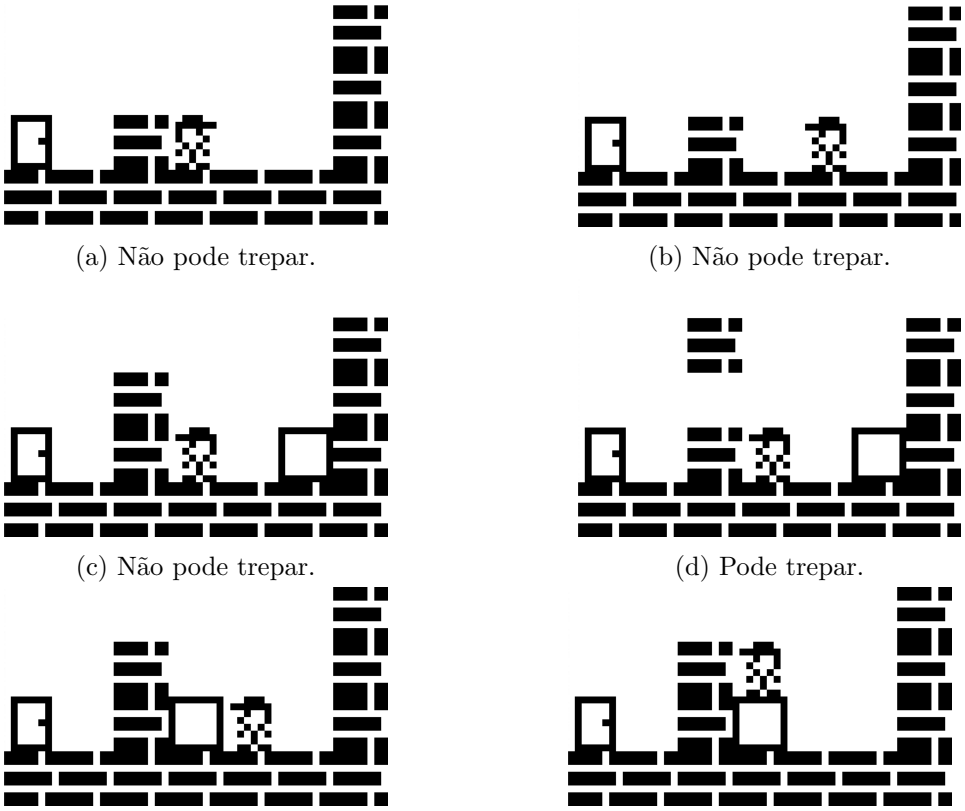
(c) Queda: antes de avançar.

(d) Queda: depois de avançar.

Figura 1: Exemplo de situações nas quais o personagem pode ou não avançar.

1.1.2 Como e quando pode o personagem trepar um obstáculo

Trepar um obstáculo consiste simplesmente em avançar para cima deste. Assim, o personagem apenas pode trepar um obstáculo que se encontre imediatamente à sua frente e sem outro obstáculo por cima. Na Figura 2a o personagem não pode trepar o bloco de pedra porque este se encontra atrás do personagem. Na Figura 2b o personagem não pode trepar porque não há nenhum obstáculo para trepar no espaço imediatamente à sua frente. Na Figura 2c o personagem não pode trepar o bloco de pedra porque outro bloco de pedra (acima) obstrui o avanço do personagem; *ou seja*, o personagem não pode trepar um obstáculo que não esteja à sua altura. Já na Figura 2d o personagem pode trepar o bloco de pedra. Na Figura 2e o personagem pode trepar a caixa, avançando para a posição da Figura 2f, donde pode também trepar o bloco de pedra.



(a) Não pode trepar.

(b) Não pode trepar.

(c) Não pode trepar.

(d) Pode trepar.

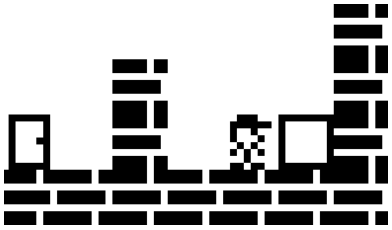
(e) Pode trepar: antes.

(f) Pode trepar: depois.

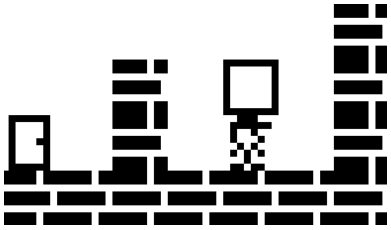
Figura 2: Exemplo de situações nas quais o personagem pode ou não trepar um obstáculo.

1.1.3 Como e quando pode o personagem carregar/largar uma caixa

O personagem pode levantar e transportar consigo uma (e no máximo uma) caixa. Para carregar uma caixa é necessário que esta esteja na posição imediatamente à frente do personagem e que não haja nenhum outro obstáculo acima quer do personagem quer da caixa em questão. Ver Figuras 3a–3e.

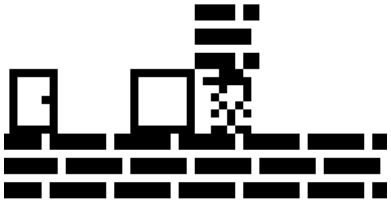


(a) Pode carregar: antes

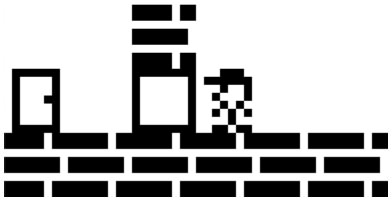


(b) Pode carregar: depois

(c) Não pode carregar porque a caixa encontra-se atrás do personagem.



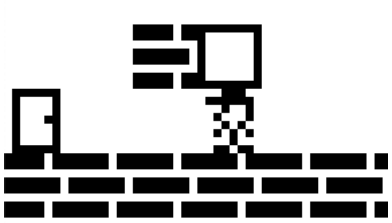
(d) Não pode carregar porque há um obstáculo acima do personagem.



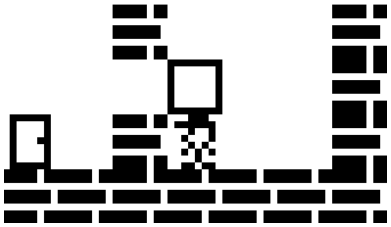
(e) Não pode carregar porque há um obstáculo acima da caixa.

Figura 3: Exemplo de situações nas quais o personagem pode ou não carregar uma caixa.

Enquanto carrega uma caixa o espaço necessário para o personagem avançar e trepar alteram-se, na medida em que é necessário espaço também para a caixa a ser transportada se mover. Ver Figuras 4a e 4b.



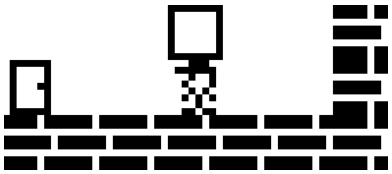
(a) Não pode avançar porque não há espaço para a caixa.



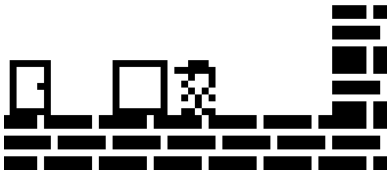
(b) Não pode trepar porque não há espaço para a caixa.

Figura 4: Exemplo de situações nas quais o personagem não pode avançar nem trepar carregando uma caixa.

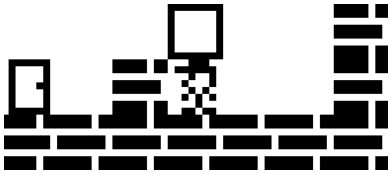
Largar uma caixa consiste em colocar a caixa que o personagem carrega imediatamente à sua frente, *cf.* Figuras 5a e 5b. Para tal é necessário que não haja nenhum obstáculo na posição imediatamente à frente da caixa, *cf.* Figura 5g. Se à frente do personagem existir um outro obstáculo, a caixa será largada por cima deste, desde que o obstáculo esteja à altura do jogador, *cf.* Figuras 5c e 5d. Em caso de queda iminente, a caixa será largada imediatamente para a última posição da queda, *cf.* Figuras 5e e 5f.



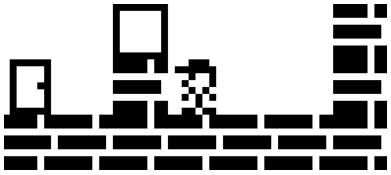
(a) Pode largar: antes.



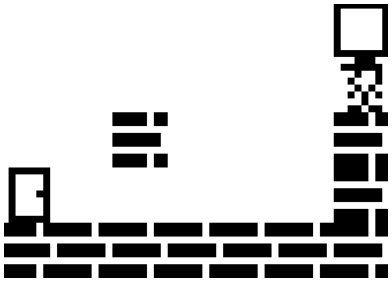
(b) Pode largar: depois.



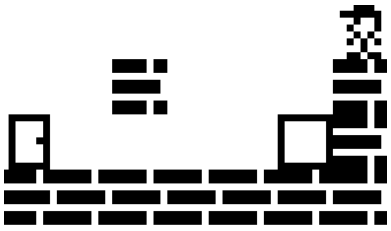
(c) Pode largar: antes.



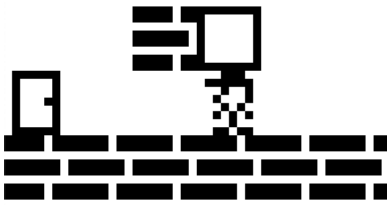
(d) Pode largar: depois.



(e) Queda: antes.



(f) Queda: depois.



(g) Não pode largar porque há um obstáculo na posição imediatamente à frente da caixa.

Figura 5: Exemplo de situações nas quais o personagem pode ou não largar uma caixa.

1.2 Tipos de dados

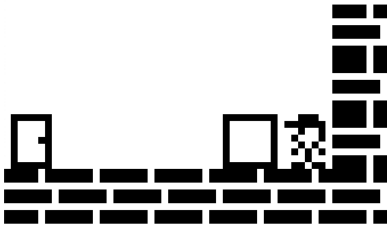
Apresentamos de seguida o modelo de jogo que deverá ter em conta na realização das tarefas propostas na secção seguinte. Será fornecido um módulo *Haskell* comum contendo estas definições preliminares, pelo que não necessita de as copiar.

1.2.1 Mapa

O *mapa* do jogo será representado através de uma grelha rectangular onde cada célula comporta uma *peça*, podendo esta ser *vazia* (valor por omissão), *bloco de pedra*, *porta*, ou *caixa*. Nesta grelha, a coordenada $(0, 0)$ refere-se à célula no canto superior esquerdo. A título de exemplo, a Figura 6 ilustra simultaneamente a representação interna e gráfica de um mapa. Note que o personagem não faz parte desta abstracção.

						Bloco
						Bloco
Porta				Caixa		Bloco
Bloco	Bloco	Bloco	Bloco	Bloco	Bloco	Bloco

(a) Representação interna.



(b) Representação gráfica.

Figura 6: Exemplo de um mapa e sua representação interna.

Definimos então os seguintes tipos de dados:

```
type Coordenadas = (Int, Int)
data Peca = Bloco | Porta | Caixa | Vazio
  deriving (Show, Eq)
type Mapa = [[Peca]]
```

1.2.2 Personagem

A abstracção para o *personagem* (ou *jogador*) do jogo deverá ter em conta a sua *posição* no mapa, *direcção*, e se *carrega* ou não uma caixa. Definimos então os seguintes tipos de dados:

```
data Direcao = Este | Oeste
data Jogador = Jogador Coordenadas Direcao Bool
```

em que o parâmetro `Bool` no construtor `Jogador` indica se o personagem carrega (`True`) ou não (`False`) uma caixa.

1.2.3 Jogo

Finalmente, o *jogo* é simplesmente composto pela informação do mapa e do personagem:

```
data Jogo = Jogo Mapa Jogador
```

2 Tarefas

2.1 Tarefa 1 – Validação de um potencial mapa

O objectivo desta tarefa é implementar a função

```
validaPotencialMapa :: [(Peca, Coordenadas)] -> Bool
```

que dada uma lista de peças e respectivas coordenadas testa se estas definem correctamente um mapa. Para tal, a lista deverá satisfazer os seguintes critérios:

1. Não haver mais do que uma declaração de peça para a mesma posição.
2. Declarar exactamente uma porta.
3. Todas as caixas devem estar posicionadas em cima de outra caixa ou bloco, *i.e.* não podem haver caixas a "flutuar".
4. Devem existir espaços vazios (no mínimo um), *i.e.* o mapa não pode estar totalmente preenchido por caixas, blocos e porta.
5. A base do mapa deve ser composta por blocos, *i.e.* deve existir um chão ao longo do mapa.

Note que a peça vazia poderá ser declarada por omissão. Por exemplo, a lista

```
[(Porta, (0, 2)),
 (Bloco, (0, 3)), (Bloco, (1, 3)), (Bloco, (2, 3))]
```

descreve um mapa 4×3 onde as peças omitidas (8 no total) correspondem a peças vazias.

2.2 Tarefa 2 – Construção/Desconstrução do mapa

O objectivo desta tarefa é implementar o par de funções

```
constroiMapa    :: [(Peca, Coordenadas)] -> Mapa
desconstroiMapa :: Mapa -> [(Peca, Coordenadas)]
```

que dada uma lista válida (*cf. Tarefa 1*) de peças e respectivas coordenadas constrói um mapa (a grelha propriamente dita), e a sua "inversa"[2] que toma um mapa e devolve a listagem das suas peças e respectivas coordenadas.

A título de exemplo, o mapa representado na Figura 6 poderia ser construído através da expressão:

```
constroiMapa [(Porta, (0,2)), (Bloco, (0,3)), (Bloco, (1,3))
             ,(Bloco, (2,3)), (Bloco, (3,3)), (Caixa, (4,2))
             ,(Bloco, (4,3)), (Bloco, (5,3)), (Bloco, (6,0))
             ,(Bloco, (6,1)), (Bloco, (6,2)), (Bloco, (6,3))]
```

donde se obteria o seguinte valor do tipo `Mapa`:

```
[[Vazio, Vazio, Vazio, Vazio, Vazio, Vazio, Bloco]
,[Vazio, Vazio, Vazio, Vazio, Vazio, Vazio, Bloco]
,[Porta, Vazio, Vazio, Vazio, Caixa, Vazio, Bloco]
,[Bloco, Bloco, Bloco, Bloco, Bloco, Bloco, Bloco]]
```

E, por seu turno, invocando-se `descontroiMapa` com o `Mapa` acabado de apresentar poder-se-ia obter a seguinte lista:

```
[(Bloco, (6,0)), (Bloco, (6,1)), (Porta, (0,2)), (Caixa, (4,2))
,(Bloco, (6,2)), (Bloco, (0,3)), (Bloco, (1,3)), (Bloco, (2,3))
,(Bloco, (3,3)), (Bloco, (4,3)), (Bloco, (5,3)), (Bloco, (6,3))]
```

[2] Inversa a menos da ordem dos elementos da lista e células definidas por omissão (*i.e.* vazias).

2.3 Tarefa 3 – Representação textual do jogo

O objectivo desta tarefa é tornar o tipo de dados `Jogo` uma instância da class `Show` de acordo com a seguinte formatação:

1. Espaços vazios denotam-se por " " (um espaço em branco).
2. Blocos de pedra denotam-se por "`X`".
3. Caixas denotam-se por "`C`".
4. A porta denota-se por "`P`".
5. O personagem denota-se por "`<`" (se estiver voltado para esquerda) ou "`>`" (se estiver voltado para a direita).

A título de exemplo, o (estado do) jogo ilustrado na Figura 6 terá a seguinte representação textual:

```
      X
      X
P   C<X
XXXXXXX
```

2.4 Tarefa 4 – Movimentação do personagem

O objectivo desta tarefa é implementar a função

```
moveJogador :: Jogo -> Movimento -> Jogo
```

que aplica o efeito de um comando (*i.e.* `Movimento`) sobre o jogador, e a sua generalização

```
correrMovimentos :: Jogo -> [Movimento] -> Jogo
```

que aplica consecutivamente os comandos dados pela lista. Os movimentos possíveis são dados pelo seguinte tipo de dados:

```
data Movimento = AndarEsquerda | AndarDireita | Trepar | InterageCaixa
```

significando:

1. *AndarEsquerda/AndarDireita*: faz com que o jogador se volte para Oeste/Este e avance (se possível) uma unidade.
2. *Trepar*: faz com o jogador trepe (se possível) o obstáculo imediatamente à sua frente.

3. *InterageCaixa*: faz com que o jogador carregue/largue uma caixa.

A função `moveJogador` deve testar se o movimento é aplicável. Deve ter também em conta quedas do personagem ou de caixas, salientando-se que a queda de várias unidades deve ser instantânea.

3 Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é de **27 de Novembro de 2021** e a respectiva avaliação terá um peso de **55%** na nota final da UC. A submissão será feita automaticamente através do GitLab onde, nessa data, será feita uma cópia do repositório de cada grupo, sendo apenas consideradas para avaliação os programas e demais artefactos que se encontrem no repositório nesse momento. O conteúdo dos repositórios será processado por ferramentas de detecção de plágio e, na eventualidade de serem detectadas cópias, estas serão consideradas fraude dando-se-lhes tratamento consequente.

Para além dos programas *Haskell* relativos às tarefas será considerada parte integrante do projecto todo o material de suporte à sua realização armazenado no repositório do respectivo grupo (código, documentação, ficheiros de teste, *etc.*). A utilização das diferentes ferramentas abordadas no curso (como *Haddock* ou *git*) deve seguir as recomendações enunciadas nas respectivas sessões laboratoriais. A avaliação desta fase do projecto terá em linha de conta todo esse material, atribuindo-lhe os seguintes pesos relativos:

Componente	**Peso**
Avaliação automática da Tarefa 1	15%
Avaliação automática da Tarefa 2	15%
Avaliação automática da Tarefa 3	15%
Avaliação automática da Tarefa 4	15%
Avaliação qualitativa das tarefas	15%
Documentação do código	10%
Quantidade e qualidade dos testes	10%
Utilização do sistema de versões	5%

Os grupos de trabalho devem ser compostos por **dois elementos** pertencentes ao **mesmo turno PL**. A nota final será atribuída **independentemente** a cada membro do grupo em função da respectiva prestação. A avaliação automática será feita através de um conjunto de testes que não serão revelados aos grupos. A avaliação qualitativa incidirá sobre aspectos

da implementação não passíveis de serem avaliados automaticamente (como a estrutura do código ou elegância da solução implementada).