Princípios de Programação

Projeto 3

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

2023/2024

O objectivo do terceiro projeto é que os alunos consolidem o conhecimento sobre os seguintes conceitos em Haskell: módulos, entrada e saída, teste de funções com `QuickCheck`.
Pretende-se que após este projeto, os alunos sejam capazes de construir um projeto em Haskell, dividido em vários módulos, interagir com a linha de comandos e com ficheiros, definir propriedades sobre as funções implementadas, e testar essas propriedades de forma automática.
Neste projeto vamos desenvolver um *programa* para jogar *Blackjack*.
Nota: os docentes da disciplina não promovem quaisquer apostas ou jogos a dinheiro. Lembrem-se que, na vida real, as apostas envolvem riscos e podem ter um impacto negativo em termos financeiros mas também em questões de saúde mental.
Vamos começar por relembrar o que já foi dito nos Projetos 1 e 2. Podemos representar uma carta como uma **`String`** de dois caracteres (valor e naipe), e representar um baralho como uma lista de strings, de acordo com o formato usado no Projeto 1. As cartas de um baralho são distribuídas percorrendo esta lista do início para o fim. Vamos utilizar as mesmas regras do Projeto 2. Releia-as antes de começar.

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]
```

No comando acima, `ficheiro` é um argumento opcional que corresponde a um ficheiro com o conteúdo de um baralho, ou seja, várias cartas (uma carta por linha). O comando acima inicia um jogo de Blackjack entre um jogador e a casa, lendo o conteúdo do ficheiro para um baralho.

A1. Baralho predefinido Notem que o ficheiro é um argumento opcional. Se este não estiver presente, a vossa aplicação deve carregar o ficheiro `default.bar` por omissão.

A2. Estrutura dos baralhos Os baralhos deverão ser lidos como ficheiros de texto, podendo utilizar as extensões `.txt` ou `.bar`. Nestes ficheiros estará guardada uma carta por linha. Por exemplo, o conteúdo do baralho `default.bar` começa por

```
8S
2D
6D
6H
QC
QC
...
```

A3. Baralho Aleatório Em alternativa, deve ser possível iniciar um jogo com um baralho de cartas aleatório, usando a flag `-n X`. Com esta opção devem produzir um baralho formado por `X` baralhos de cartas normais (de 52 cartas), baralhado de forma aleatória.

B. Implementação Após a inicialização (seja com a leitura de um ficheiro ou com a geração de um baralho aleatório), o programa deve imprimir no **`stdout`** a seguinte informação sobre o estado do jogo:

1. o número de cartas no baralho (inicialmente 24 para o baralho `default.bar`)
2. o número de créditos do jogador (inicialmente 100)

Dois exemplos de execução (comentários assinalados a verde):

```
> ./Main default.bar
cartas: 24
creditos: 100
...                 --[apostar ou sair]

> ./Main -n 6
cartas: 312
creditos: 100
...                 --[apostar ou sair]
```

Se o baralho tiver mais de 20 cartas, o programa inicia uma nova ronda de Blackjack (**B1** ou **B2**). Se o baralho tiver 20 ou menos cartas ou caso o jogador tenha 0 créditos, o programa deve terminar (**B9**).

B1. Apostar Um jogo de Blackjack é dividido em várias rondas. O jogo termina quando o jogador fica sem créditos ou quando o baralho tem 20 ou menos cartas. Caso contrário inicia-se uma nova ronda. No início de cada ronda, a instrução `apostar n` deve permitir ao jogador apostar `n` créditos. O valor da aposta deve ser um inteiro entre 1 (aposta mínima) e os créditos disponíveis.
Após realizar a aposta, o programa deve

1. distribuir duas cartas para o jogador
2. distribuir duas cartas para a casa
3. imprimir as mãos do jogador e da casa no **`stdout`**
4. avançar para a vez do jogador (**B3**)

Exemplo de execução (linhas de input em comentários assinalados a verde):

```
...
cartas: 24
creditos: 100
apostar 5            --[input]
jogador: 8S 2D
casa: 6D 6H
...                  --[vez do jogador]
```

B2. Sair No início de cada ronda, em vez de apostar, a instrução `sair` deve terminar o jogo. O programa deve

1. imprimir o saldo final do jogador
2. terminar a execução, regressando à linha de comandos

```
...
cartas: 24
creditos: 100
sair                  --[input]
saldo final: 100
>                     --[programa terminado]
```

B3. Vez do jogador Após a distribuição das cartas, é a vez do jogador procurar melhorar a sua mão. Caso a mão do jogador tenha o valor de 21 pontos (Blackjack), a ronda avança imediatamente para a vez da casa (**B6**).

```
> ./Main perfect.bar
cartas: 26
creditos: 100
apostar 5            --[input]
```

```
jogador: TS AD
casa: 2S 5S
...                    --[vez da casa]
```

Caso a mão do jogador tenha um valor menor que 21 pontos, o jogador pode escolher uma de entre duas jogadas possíveis: *stand* ou *hit*.

B4. Stand Durante a vez do jogador, a instrução `stand` deve terminar a vez do jogador e avançar para a vez da casa (**B6**).

```
...
jogador: 8S 2D
casa: 6D 6H
stand                  --[input]
...                    --[vez da casa]
```

B5. Hit Durante a vez do jogador, a instrução `hit` deve distribuir uma carta ao jogador. O programa deve imprimir no **`stdout`** a nova mão do jogador e a mão da casa.

```
...
jogador: 8S 2D
casa: 6D 6H
hit                    --[input]
jogador: 8S 2D QC
casa: 6D 6H
...                    --[vez do jogador]
```

Caso o novo valor seja menor que 21, o jogador volta a ter de decidir entre `stand` e `hit` (**B4** ou **B5**). Caso o novo valor seja igual a 21, o jogador é obrigado a parar, passando à vez da casa (**B6**). Caso o novo valor seja superior a 21, o programa deve

1. imprimir no **`stdout`** a palavra `Derrota`
2. atualizar os créditos do jogador de forma correspondente
3. avançar para o fim da ronda (**B8**), não passando pela vez da casa

```
...
jogador: 8S 2D QC
casa: 6D 6H
hit                    --[input]
jogador: 8S 2D QC QC
casa: 6D 6H
Derrota
...                    --[fim da ronda]
```

B6. Vez da casa Se o jogador tiver parado (com um valor menor ou igual a 21 pontos), é a vez da casa jogar. A estratégia da casa é de receber cartas novas enquanto o valor for inferior a 17 pontos e parar assim que tiver um valor igual ou superior a 17 pontos. O programa deve

1. imprimir no **`stdout`** a mão do jogador e a mão final da casa (não imprimir as mãos intermédias da casa)
2. avançar para a comparação das mãos (**B7**).

```
...
jogador: 8S 2D
casa: 6D 6H
stand                --[input]
jogador: 8S 2D
casa: 6D 6H QC
...                  --[comparacao]
```

B7. Comparação dos valores Neste momento, comparamos os valores das duas mãos. O programa deve

1. imprimir no **`stdout`** a palavra `Vitoria` (sem acento), `Derrota` ou `Empate`, consoante o jogador tenha ganho, perdido ou empatado
2. atualizar os créditos do jogador de forma correspondente
3. avançar para o fim da ronda (**B8**)

```
...
jogador: 8S 2D
casa: 6D 6H
stand                --[input]
jogador: 8S 2D
casa: 6D 6H QC
Vitoria
...                  --[fim da ronda]
```

B8. Fim da ronda Ao chegar ao fim da ronda, o programa deve

1. descartar as mãos do jogador e da casa
2. imprimir no **`stdout`** o número de cartas no baralho e o (novo) número de créditos do jogador

```
...
jogador: 8S 2D
casa: 6D 6H QC
Vitoria
cartas: 19
creditos: 105
...                    --[fim do jogo]
```

Se o baralho tiver mais de 20 cartas e o jogador ainda tiver créditos, o jogador pode escolher iniciar uma nova ronda ou sair (**B1** ou **B2**). Caso contrário o jogo termina (**B9**).

B9. Fim do jogo O jogo termina quando o jogador fica sem créditos, quando o baralho tem 20 ou menos cartas, ou com a instrução `sair` durante o início de uma ronda. Em qualquer dos casos o programa deve

1. imprimir o saldo final do jogador
2. terminar a execução, regressando à linha de comandos

```
...
jogador: 8S 2D
casa: 6D 6H QC
Vitoria
cartas: 19
creditos: 105
saldo final: 105
>                      --[programa terminado]
```

B10. Exemplos Apresentamos quatro exemplos completos de execução para o ficheiro `default.bar`.

```
> ./Main default.bar
cartas: 24
creditos: 100
sair                   --[input]
saldo final: 100
>                      --[programa terminado]

> ./Main default.bar
cartas: 24
creditos: 100
apostar 5              --[input]
jogador: 8S 2D
casa: 6D 6H
stand                  --[input]
```

```
jogador: 8S 2D
casa: 6D 6H QC
Vitoria
cartas: 19
creditos: 105
saldo final: 105
>                    --[programa terminado]

> ./Main default.bar
cartas: 24
creditos: 100
apostar 10           --[input]
jogador: 8S 2D
casa: 6D 6H
hit                  --[input]
jogador: 8S 2D QC
casa: 6D 6H
stand                --[input]
jogador: 8S 2D QC
casa: 6D 6H QC
Vitoria
cartas: 18
creditos: 110
saldo final: 110
>                    --[programa terminado]

> ./Main default.bar
cartas: 24
creditos: 100
apostar 60           --[input]
jogador: 8S 2D
casa: 6D 6H
hit                  --[input]
jogador: 8S 2D QC
casa: 6D 6H
hit                  --[input]
jogador: 8S 2D QC QC
casa: 6D 6H
Derrota
cartas: 18
creditos: 40
saldo final: 40
>                    --[programa terminado]
```

B11. 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 baralho `XX_baralho.bar`, uma sequência de inputs `XX_input.txt` e o output esperado `XX_check.txt`. Podem usar o comando `diff` para verificar que o vosso programa produz o resultado esperado.

```
> ./Main 01_baralho.bar < 01_input.txt > 01_output.txt
> diff 01_output.txt 01_check.txt
[não produz output]
```

C. Testes

C1. Definição de propriedades A vossa aplicação deve incluir pelo menos *seis* testes QuickCheck relacionados com a funções que implementarem.

- Um teste que verifique a propriedade: uma mão inicial (de duas cartas) vale sempre no máximo 21 pontos.
- Um teste que verifique a propriedade: o número de créditos do jogador após uma ronda é um de três valores possíveis `n-a`, `n`, `n+a`, onde `n` é o número de créditos no início da ronda e `a` é o valor da aposta.
- Um teste que verifique a propriedade: no final da vez da casa, a mão da casa tem sempre pelo menos 17 pontos.
- Outros três testes da vossa autoria, sobre as funções que entenderem. Procuramos testes interessantes, capazes de apanhar erros subtis, e não testes triviais tais como: o número de cartas de um baralho 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.
```

C2. 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 estados de jogo.
Para além disso, se utilizarem o tipo de dados **`String`** para representar uma carta e se se limitarem a gerar uma **`String`** aleatória, é pouco provável que a

`String` corresponda a uma carta válida (uma de entre `"AS"`,`"2S"`,... `"QC"`,`"KC"`). Nesse caso, sugerimos que definam um tipo para cartas válidas, do estilo

```
newtype CartaValida = CV 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). As mesmas considerações aplicam-se à geração de baralhos e mãos do jogador ou da casa.

D. 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 baralho para jogar
     Blackjack
  ./Main          -- carrega o baralho default.bar
  ./Main -n X     -- carrega um baralho aleatório formado
      por X baralhos normais de cartas
  ./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`).

E. 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 Blackjack. 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 baralhos `default.bar`, `perfect.bar`, `01_baralho.bar`, ..., `05_baralho.bar`,
- 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 `.bar` e `.txt`, deixando *apenas* os ficheiros `.hs`.

Pontos de atenção

1. O vosso trabalho deverá ser constituído por um ficheiro zip de nome `p3_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 inputs durante a execução do programa fazem sentido (por exemplo, podem assumir que na vez do jogador as únicas instruções que irão aparecer são `stand` ou `hit`).

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

4. 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.

5. 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.

6. 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 11 de dezembro de 2023.

Plágio. A nível académico, alunos detetados em situação de fraude ou plágio (plagiadores e plagiados) em alguma prova ficam reprovados à disciplina. Serão ainda alvo de processo disciplinar, ficando registado no processo de aluno, podendo conduzir à suspensão letiva, expulsão da universidade e/ou denúncia no Ministério Público.
Qualquer situação em que um aluno submete material que não é da sua autoria é considerada fraude ou plágio. Isto inclui material da autoria de colegas, de terceiros, de fontes online não identificadas ou por inteligência artificial generativa (ex: ChatGPT). Tais ferramentas são portanto proibidas na realização dos projetos.
Todos os trabalhos são submetidos a uma ferramenta de verificação de semelhanças de software. Aqueles em que o sistema assinalar um elevado grau de semelhança são posteriormente analisados manualmente pelos docentes.