Princípios de Programação

Projeto 2

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

2023/2024

O objectivo do segundo projeto é que os alunos consolidem o conhecimento sobre os seguintes conceitos em Haskell: recursão, funções de ordem superior, módulos, tipos e classes de tipos.
Pretende-se que após este projeto, os alunos sejam capazes de criar os seus próprios tipos e classes, definir funções sobre os tipos criados usando recursão ou funções de ordem superior, e criar um módulo que exporte os tipos e funções implementados.
Neste projecto vamos modelar o jogo de Blackjack🏹 entre um jogador e a casa.
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.
Existem muitas variantes do jogo Blackjack. Neste projeto vamos considerar as seguintes regras.

1. (**Um jogador**): o jogo decorre entre a casa e um jogador apenas, que começa com 100 créditos.

2. (**Vários baralhos**): existe um baralho de cartas que pode ser composto por vários baralhos clássicos de 52 cartas. É comum ser composto por 3 ou 6 baralhos clássicos, de forma a complicar a vida a quem conta cartas🏹. Isto implica que, a qualquer momento, o baralho pode ter cartas iguais.

3. (**Valores das cartas**): Uma mão tem um valor igual à soma dos valores das cartas. Cada carta numérica (dois a dez) tem o valor nela escrita. Cada figura (valete, dama, rei) vale 10 pontos. Cada ás pode valer 1 ou 11 pontos, conforme seja mais conveniente a qualquer momento.

4. (**Fim do jogo**): o jogo 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.

5. (**Início da ronda**): no início de cada ronda, o jogador deve decidir quantos créditos quer apostar. O valor da aposta deve ser um inteiro entre 1 (aposta mínima) e os créditos disponíveis.

6. (**Distribuição**): após a aposta feita, distribuímos duas cartas para o jogador, e seguidamente duas cartas para a casa. As cartas para a casa são visíveis pelo jogador.

7. (**Jogadas possíveis**): neste momento o jogador deve decidir uma de entre duas jogadas possíveis: *stand* ou *hit* (para simplificar, neste projeto não vamos implementar as jogadas *double down* ou *split*, normalmente existentes no Blackjack).

8. (**Stand**): com esta decisão o jogador para (com o valor que tiver na sua mão), e a ronda avança para a jogada da casa. O jogador é obrigado a parar se a sua mão tiver o valor igual a 21 pontos.

9. (**Hit**): com esta decisão o jogador recebe uma nova carta, atualizando o seu total. Caso o novo valor seja menor que 21, o jogador volta a ter de decidir entre *stand* e *hit*. Caso o novo valor seja igual a 21, o jogador é obrigado a parar. Caso o novo valor seja superior a 21, o jogador perde imediatamente os créditos apostados, todas as cartas são descartadas, e a ronda termina.

10. (**Jogada 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.

11. (**Comparação dos valores**): Neste momento, comparamos os valores das duas mãos. Se a casa terminar com um valor inferior ao do jogador, o jogador ganha. Se a casa terminar com um valor igual ao valor do jogador, o jogador empata. Se a casa terminar com um valor superior ao jogador, mas menor ou igual a 21 pontos, o jogador perde. Finalmente, se a casa terminar com um valor superior a 21 pontos, o jogador ganha.

12. (**Fim da ronda**): Em caso de derrota, o jogador perde os créditos apostados. Em caso de empate, o jogador recupera os créditos apostados. Em caso de vitória, o jogador recebe duas vezes o valor da aposta. Em qualquer caso, todas as cartas são descartadas e a ronda chega ao fim.

Para este projeto terão de submeter um único módulo, chamado `Blackjack.hs`. O vosso ficheiro deverá começar com um comentário com os números de aluno dos autores, separados por uma vírgula, seguindo-se a declaração do módulo. Usem o formato

```
-- fcXXXXX,fcXXXXX
module Blackjack (.....) where
```

em que no lugar de `XXXXX` deverão colocar os números de aluno e no lugar de `.....` deverão exportar os tipos de dados e as funções elencadas abaixo.

A. Baralho Devem importar o módulo `BaralhosExemplo` fornecido pelos docentes. Este módulo exporta um conjunto de baralhos, para os quais os resultados das simulações são disponibilizados abaixo. Neste módulo, cada carta é representada como uma **`String`** de dois caracteres (valor e naipe), e cada baralho é representado como uma lista de strings, de acordo com o formato usado no Projeto 1, sendo que as cartas de um baralho são distribuídas percorrendo a lista do início para o fim.
Podem (e devem!) utilizar os baralhos fornecidos nos vossos testes. No entanto, o vosso programa deve funcionar para qualquer baralho.
Definam um tipo (ou sinónimo de tipos) `Baralho` que represente um baralho de cartas. Podem usar a mesma representação do Projeto 1, ou usar uma representação diferente.

A1. Conversão do baralho Definam uma função `converte :: [String] -> Baralho` que converte um baralho do módulo `BaralhosExemplo` para o tipo `Baralho` definido por vocês.

A2. Número de cartas Definam uma função `tamanho :: Baralho -> Int` que devolve o número de cartas de um baralho dado.

```
> tamanho (converte baralhoOrdenado)
312
> tamanho (converte baralhoInsuficiente)
20
> tamanho (converte baralhoSimples)
24
```

B. Estado do jogo Definam um tipo `EstadoJogo` que permita representar o estado de um jogo de Blackjack num dado momento. Deverá ser possível interagir com valores do tipo `EstadoJogo` da seguinte forma:

B1. Inicialização A função `inicializa :: Baralho -> EstadoJogo` recebe um baralho e devolve o estado inicial do jogo com esse baralho. No estado inicial, o jogador tem 100 créditos, nenhuma carta foi distribuída, e uma nova ronda está prestes a começar.

```
> ejOrdenado :: EstadoJogo
> ejOrdenado = inicializa (converte baralhoOrdenado)
> ejInsuficiente :: EstadoJogo
> ejInsuficiente = inicializa (converte
   baralhoInsuficiente)
> ejSimples :: EstadoJogo
> ejSimples = inicializa (converte baralhoSimples)
```

B2. Créditos do jogador A função
`creditos :: EstadoJogo -> Int`
recebe um estado de um jogo e devolve o valor atual dos créditos do jogador.

```
> creditos ejOrdenado
100
> creditos ejInsuficiente
100
> creditos ejSimples
100
```

B3. Baralho A função
`baralho :: EstadoJogo -> Baralho`
recebe um estado de um jogo e devolve o baralho atual (cartas por distribuir).

```
> tamanho (baralho ejOrdenado)
312
> tamanho (baralho ejInsuficiente)
20
> tamanho (baralho ejSimples)
24
```

B4. Terminação A função
`terminado :: EstadoJogo -> Bool`
recebe um estado de um jogo e indica se o jogo terminou. O jogo termina quando o jogador fica sem créditos ou quando o baralho tem 20 ou menos cartas.

```
> terminado ejOrdenado
False
> terminado ejInsuficiente
True
> terminado ejSimples
False
```

B5. Classe Show Tornem o tipo de dados `EstadoJogo` instância da classe **`Show`**. Para apresentar o estado de um jogo, deverão incluir *no mínimo* as cartas do jogador e da casa, bem como o valor atual de créditos do jogador. Podem incluir informação adicional que considerem relevante. Segue-se um possível exemplo.

```
> ejOrdenado
jogador:
casa:
creditos: 100
```

C. Estratégia Definam um tipo `Estrategia` que permita representar as funções que tomam decisões pelo jogador. O jogador decide quanto apostar (no início de uma ronda) e se deve fazer *stand* ou *hit* a cada momento. As decisões do jogador podem depender, por exemplo, dos créditos atuais, das suas cartas, das cartas da casa, ou de outros elementos que considerem relevantes. No entanto, as decisões *não devem* depender das cartas do baralho (por distribuir) - isso seria batota!

C1. Três estratégias Devem implementar três estratégias.
A estratégia `sempreStand :: Estrategia` deve representar a estratégia de apostar sempre 5 créditos e fazer sempre *stand*.
A estratégia `sempreHit :: Estrategia` deve representar a estratégia de apostar sempre 5 créditos e fazer *hit* sempre que possível (com esta estratégia, o jogador só para se atingir 21 pontos).
Devem implementar uma terceira estratégia à vossa escolha, e incluir num comentário a vossa explicação.

C2. Simulação de uma ronda A função
`simulaRonda :: Estrategia -> EstadoJogo -> EstadoJogo`
simula uma ronda do jogo Blackjack, utilizando a estratégia dada. A função devolve o estado de jogo no final da ronda (atualizando o valor dos créditos e do baralho).

C3. Simulação de um jogo A função principal deste projeto é a função `simulaJogo :: Estrategia -> Baralho -> Int` que, dada uma estratégia do jogador e um baralho, corre uma simulação de um jogo completo de Blackjack, com um saldo inicial de 100 créditos. A função devolve o número de créditos do jogador no final da simulação.

C4. Classe Show Tornem o tipo de dados `Estrategia` instância da classe **`Show`**. A apresentação de uma estratégia no terminal consiste numa pequena descrição textual da mesma. Segue-se um possível exemplo.

```
> sempreStand
Estrategia: apostar sempre 5 créditos, fazer sempre stand
```

D. Exemplo Passamos a explicar um exemplo em detalhe. Considere o `baralhoSimples`, com vinte e quatro cartas. As primeiras quatro cartas são `8S`, `2D`, `6D`, `6H`, após o que se seguem vinte cópias da carta `QC` (rainha de copas).
Usando a estratégia `sempreStand`, inicia-se uma ronda em que o jogador aposta 5 créditos, ficando com 95 créditos. Distribuem-se duas cartas ao jogador e à casa.

```
jogador: 8S 2D
casa: 6D 6H
creditos: 95
```

Seguindo a estratégia `sempreStand`, o jogador decide parar (com um valor de 10 pontos), passando à vez da casa. A mão da casa vale 12 pontos (inferior a 17), pelo que escolhe receber uma carta (`QC`).

```
jogador: 8S 2D
casa: 6D 6H QC
creditos: 99
```

Neste momento a casa rebenta com 22 pontos, o que dá a vitória ao jogador. O jogador recebe duas vezes o valor da aposta, ficando com 105 créditos. Todas as cartas são descartadas.

```
jogador:
casa:
creditos: 105
```

Neste momento, o jogo termina pois há apenas 19 cartas no baralho.
Em alternativa, consideremos a estratégia `sempreHit`. Apostando 5 créditos e distribuindo duas cartas ao jogador e à casa, obtemos as mesmas mãos iniciais.

```
jogador: 8S 2D
casa: 6D 6H
creditos: 95
```

Desta vez, o jogador decide receber uma carta (`QC`).

```
jogador: 8S 2D QC
casa: 6D 6H
creditos: 95
```

Como a mão do jogador vale 20 pontos, a estratégia `sempreHit` determina que se recebe outra carta (`QC`).

```
jogador: 8S 2D QC QC
casa: 6D 6H
creditos: 95
```

Neste momento é o jogador que rebenta com 30 pontos, e perde imediatamente a ronda. Todas as cartas são descartadas.

```
jogador:
casa:
creditos: 95
```

Neste momento, o jogo termina pois há apenas 18 cartas no baralho.

E. Testes Para testarem o vosso programa, podem (e devem!) usar os vários baralhos definidos no ficheiro `BaralhosExemplo.hs`. Em seguida, indicamos os resultados da simulação com vários exemplos, para testarem o vosso código.
Simulações de uma ronda:

```
> ej1 = simulaRonda sempreStand ejOrdenado
> creditos ej1
95
> tamanho (baralho ej1)
304
> terminado ej1
False
> ej2 = simulaRonda sempreHit ejOrdenado
> creditos ej2
95
> tamanho (baralho ej2)
298
> terminado ej2
False
> ej3 = simulaRonda sempreStand ejSimples
> creditos ej3
105
> tamanho (baralho ej3)
19
> terminado ej3
True
> ej4 = simulaRonda sempreHit ejSimples
> creditos ej4
95
> tamanho (baralho ej4)
18
> terminado ej4
True
```

Simulações de um jogo completo:

```
> simulaJogo sempreStand (converte baralhoOrdenado)
20
```

```
> simulaJogo sempreHit (converte baralhoOrdenado)
0
> simulaJogo sempreStand (converte baralhoInsuficiente)
100
> simulaJogo sempreHit (converte baralhoInsuficiente)
100
> simulaJogo sempreStand (converte baralhoSimples)
105
> simulaJogo sempreHit (converte baralhoSimples)
95
> simulaJogo sempreStand (converte baralhoEmpate)
100
> simulaJogo sempreHit (converte baralhoEmpate)
100
> simulaJogo sempreStand (converte baralhoPerdido)
95
> simulaJogo sempreHit (converte baralhoPerdido)
90
> simulaJogo sempreStand (converte baralhoGanho)
110
> simulaJogo sempreHit (converte baralhoGanho)
105
```

Notas

1. Devem submeter um único ficheiro com o nome `Blackjack.hs`.
2. Como referido anteriormente, o vosso ficheiro deverá incluir a declaração de módulo

   ```
   module Blackjack (.....) where
   ```

   em que no lugar de `.....` devem exportar os tipos de dados e as funções elencadas acima.
3. Podem definir outros tipos ou sinónimos de tipos que acharem adequados para a resolução do projeto.
4. Os trabalhos serão avaliados automaticamente. Respeitem os nomes e os tipos das funções e listas enunciadas acima, bem como o comentário com o nome dos alunos. Em particular, **Blackjack** escreve-se *sempre* com 'j' minúsculo.
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. Para resolver estes problemas podem usar qualquer função constante no `Prelude` bem como nos módulos vistos nas aulas: `Data.Char`, `Data.List`, `Data.Map`, `Data.Set`.

7. Para obter a cotação máxima, devem utilizar *pelo menos uma* função de ordem superior (`map`, `filter`, `foldl`, `foldr`, etc.), *pelo menos uma* expressão lambda e devem implementar *pelo menos uma* função recursivamente.

8. 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.

Entrega. Este é um projetos de resolução em pares. Os projetos devem ser entregues no Moodle até às 23:55 do dia 20 de novembro 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 submetidos 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.