**Universidade da Beira Interior**
**Departamento de Informática**
**Paradigmas de Programação**

**Frequência Modelo 2**
Duração: 01h30m

## ATENÇÃO

Esta é uma frequência modelo que é disponibilizada para ajudar os alunos a prepararem-se para a frequência. A frequência real será diferente e poderá ter mais ou menos questões e também questões sobre tópicos que não são cobertos nesta frequência modelo.

## INSTRUÇÕES

Todas as questões devem ser respondidas usando o ficheiro Freq2Modelo.hs que está disponível no Moodle para download. A estrutura do ficheiro Freq2Modelo.hs deve ser seguida, caso contrário poderá perder pontos. Para que a frequência seja considerada terá de estar presente na reunião Zoom com a câmara e som ligados até receber confirmação da regente de que a sua submissão foi recebida. Terá também de participar numa entrevista pelo Zoom (ver detalhes no Moodle).

**O ficheiro Freq2Modelo.hs tem que ser submetido no Moodle até ao final da frequência para ser avaliado e de seguida deve ser também enviado para o email amendes@di.ubi.pt**

**Todas as funções têm de ser acompanhadas do seu tipo. Para obter nota máxima nas questões terá de usar uma abordagem funcional e será valorizada a elegância e concisão das soluções apresentadas. Todas a provas têm que ser apresentadas no formato utilizado nas aulas mostrando para todos os passos da prova a sua justificação.**

Caso alguma função esteja a dar erro e não o consiga resolver, comente a função mas deixe-a no ficheiro indicando que dá erro. Assim, essa função poderá ser também considerada.

Chama-se a atenção para os documentos do **código de integridade** e **regulamento disciplinar dos Estudantes da Universidade da Beira Interior** (links para estes documentos encontram-se disponíveis no Moodle).

**ESTE TESTE TEM 5 PÁGINAS E 6 GRUPOS DE PERGUNTAS.**

**Universidade da Beira Interior**
**Departamento de Informática**
**Paradigmas de Programação**

**Frequência Modelo 2**
Duração: 01h30m

1. Escolha a(s) resposta(s) correctas para cada uma das alíneas seguintes:

   (a) As funções em Haskell são puras. Uma função pura é uma função que:
   i) Não tem estado.
   ii) Devolve exactamente o mesmo resultado sempre que é chamada com os mesmos argumentos.
   iii) Pode devolver resultados diferentes quando é chamada com os mesmos argumentos.
   iv) Tem efeitos secundários.

   (b) Dada a função:

```
f :: Int -> Maybe Int
f s = do m <- g s >>= h
         n <- h m
         j n
```

   Das opções seguintes indique todas as que têm o mesmo comportamento que a função `f`.
   i) g s >>= h >>= j
   ii) Just s >>= g >>= h >>= h >>= j
   iii) Just s >>= g >>= h >>= h >>= j >>= return
   iv) g s >>= h >>= h >>= j
   v) g s >>= h >>= return >>= h >>= j

   (c) Indique qual a semântica formal que dá significado aos programas através da descrição directa do efeito que estes têm nos estados do programa.
   i) Operacional
   ii) Axiomática
   iii) Denotacional

2. Considere as seguintes funções:

```
compL :: [a] -> Int
compL [] = 0
compL (x:xs) = 1 + compL xs

replica :: Int -> a -> [a]
replica 0 _ = []
replica n x = x : (replica (n-1) x)
```

   prove por indução em $n$, com $n \geq 0$, que:

```
compL (replica n x) = n
```

**Universidade da Beira Interior**
**Departamento de Informática** **Frequência Modelo 2**
**Paradigmas de Programação** Duração: 01h30m

3. Este grupo é sobre o uso dos operadores `>>=`, `>>`, `<*>` e `<$>` e das funções `pure`, `return` e `sequence`. Para todas as questões é esperada a solução mais simples possível.

   (a) Utilizando apenas os operadores monádicos `>>=` e `>>` para encadear as computações necessárias, escreva um programa que pede ao utilizador para escrever uma frase, lê essa frase, pede uma segunda frase, lê também essa frase e devolve uma frase com as duas frases concatenadas mas com um espaço em branco entre elas.
   Por exemplo, se o utilizador introduzir primeiro "Paradigmas" e depois "de Programação", o output é "Paradigmas de Programação".

   (b) Usando apenas listas e o operador `<*>` crie uma função que dadas duas listas, devolve uma lista com o resultado de aplicar as operações de multiplicação, adição e subtração a todas as combinações possíveis dos elementos das duas listas.
   Por exemplo, para as lista $[1, 2]$ e $[3, 4]$ devolve a lista

   $[3, 4, 6, 8, 4, 5, 5, 6, -2, -3, -1, -2]$

   que corresponde aos resultados das seguintes operações

   $[1 * 3, 1 * 4, 2 * 3, 2 * 4, 1 + 3, 1 + 4, 2 + 3, 2 + 4, 1 - 3, 1 - 4, 2 - 3, 2 - 1]$.

   (c) Usando apenas a função *sequence* e o operador `<$>` defina uma função que incrementa por um todos os elementos de uma lista com elementos do tipo Maybe Int (i.e. o input é do tipo [Maybe Int]) e que devolve uma lista do tipo Maybe [Int] com todos os resultados.
   Por exemplo, para a lista [Just 1, Just 2, Just 3] devolve a lista Just [2,3,4] .

   (d) Considere uma função $g$ que recebe como input um inteiro $x$ e devolve `Nothing` caso esse inteiro seja zero e `Just (x-1)` caso contrário. Indique qual o resultado da seguinte computação:

   ```
   return 2 >>= g >>= g >>= g >>= g
   ```

   (e) Considere a função:

   ```
   f x = Just (x*2)
   ```

   Resolva o erro na seguinte computação:

   ```
   f <*> [1,2,3]
   ```

   que pretende aplicar $f$ a todos os elementos da lista para obter o resultado [Just 2, Just 4, Just 6].

**Universidade da Beira Interior**
**Departamento de Informática**
**Paradigmas de Programação**

**Frequência Modelo 2**
Duração: 01h30m

4. Considere os seguintes tipos e funções:

```
data Expr = Val Int | Add Expr Expr deriving Show
type Stack = [Int]
type Code = [Op]
data Op = PUSH Int | ADD deriving Show

eval :: Expr -> Int
eval (Val n) = n
eval (Add n m) = eval n + eval m

exec :: Code -> Stack -> Stack
exec [] s = s
exec (PUSH n:rc) s = exec rc (n:s)
exec (ADD: c) (m:n:s) = exec c (n+m:s)

comp :: Expr -> Code
comp (Val n) = [PUSH n]
comp (Add n m) = comp n ++ comp m ++ [ADD]
```

   (a) Adicione a esta linguagem o constructor `Mult Expr Expr`, que multiplica duas expressões, e adapte todo o código para suportar este novo constructor.

   (b) Nas aulas foi demonstado que o compilador estava correcto quando o linguagem continha apenas os constructores `Val` e `Add`. Prove que o compilador continua correcto com a adição do novo constructor `Mult` (basta escrever a prova para `Mult`). Formule claramente a equação que exprime a correção do compilador.
   Assuma a regra da distributividade:

```
exec (c ++ d) s = exec d (exec c s)
```

5. Dado o seguinte tipo:

```
data EstadoJogo = EJ {j1::Int, j2::Int, jogador::Bool} deriving Show
```

   Use um monad transformer para implementar um programa que pede alternadamente ao jogador 1 ou jogador 2 para introduzir um valor. Esse valor é adicionado ao total do respectivo jogador. O jogo deve proceder até que um número menor ou igual a zero seja introduzido por um dos jogadores. No final deve imprimir uma mensagem que indica que o jogo terminou.

**Universidade da Beira Interior**
**Departamento de Informática**
**Paradigmas de Programação**

**Frequência Modelo 2**
Duração: 01h30m

6. Considere o seguinte tipo de dados:

```
data Arv a = Folha a | Nodo (Arv a) (Arv a)
```

   (a) Defina as funções:

```
folhas :: Arv a -> [a]
tamanho :: Arv a -> Int
```

   que, respectivamente, devolvem a lista dos valores contidos nas folhas da árvore e o tamanho da árvore (definido como o número de folhas contidas na árvore).

   (b) Define uma função:

```
espelhar :: Arv a -> Arv a
```

   que dada uma árvore, devolve uma outra árvore que corresponde à original espelhada, na qual as folhas aparecem na ordem inversa da original (i.e. `folhas (espelhar t) = reverse (folhas t)` para qualquer árvore `t`).

   (c) Usando a sua definição de `tamanho` e `espelhar` prove a propriedade

```
tamanho (espelhar t) = tamanho t
```

   Use o formato de prova utilizado nas aulas e justifique todos os passos.

   (d) Defina o tipo *Arv* como instância da class *Show*. Elementos do tipo *Arv* devem ser mostrados com a string "–*–" para cada nodo, tendo do lado direito a sub-árvore direita e do lado esquerdo a sub-árvore esquerda com parênteses à volta de cada nodo.
   Por exemplo, a árvore

```
Nodo (Nodo (Folha 1) (Nodo (Folha 2) (Folha 4))) (Nodo (Folha 3) (Folha 5))
```

   deve ser mostrada da seguinte forma:

```
((1--*--(2--*--4))--*--(3--*--5))
```

   (e) Dadas as seguintes definições de *Arv* como instância das classes `Functor` e `Applicative`:

```
instance Functor Arv where
  fmap f (Folha a)    = Folha $ f a
  fmap f (Nodo t1 t2) = Nodo (fmap f t1) (fmap f t2)

instance Applicative Arv where
  pure = Folha
  Folha f       <*> t       = f <$> t
  Nodo t1 t2 <*> Folha a    = Nodo (fmap ($ a) t1) (fmap ($ a) t2)
  Nodo t1 t2 <*> Nodo t3 t4 = Nodo (t1 <*> t3) (t2 <*> t4)
```

   defina *Arv* como instância da classe *Monad*. Mostre um exemplo da utilização do operador `>>=` com o tipo *Arv*.