Universidade da Beira Interior
Departamento de Informática
Paradigmas de Programação

Frequência 2
Duração: 01h45m
Tolerância: 15 minutos

INSTRUÇÕES

Todas as questões devem ser respondidas usando o ficheiro Freq2.hs que está disponível no Moodle para download. A estrutura do ficheiro Freq2.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 docente de que a sua submissão foi recebida. Terá também de participar numa entrevista pelo Zoom (ver detalhes no Moodle).

O ficheiro Freq2.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
A frequência tem a duração de 1h45m com 15 minutos adicionais de tolerância para a sua submissão no Moodle.

Todas as funções têm de ser acompanhadas do seu tipo. Para obter nota máxima 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. Caso também não consiga terminar algum outro exercício, deve deixar no ficheiro a solução parcial.

Chama-se a atenção para o **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 6 PÁGINAS E 5 GRUPOS DE QUESTÕES.

Universidade da Beira Interior
Departamento de Informática **Frequência 2**
Paradigmas de Programação Duração: 01h45m
Tolerância: 15 minutos

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

   (a) Dada a função:

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

   Das opções seguintes indique todas as que têm o mesmo comportamento que a função `f` (i.e. que produzem o mesmo output). **(0.75 pontos)**

   i) g s >>= h >>= j >>= h

   ii) Just s >>= g >>= h >>= j >>= h >>= return

   iii) g s >>= h >>= h

   iv) g s >>= h >>= j

   v) g s >>= h >>= return >>= j >>= h

   (b) Indique qual a semântica formal que dá significado aos programas através das propriedades lógicas que os descrevem. **(0.75 pontos)**

   i) Operacional

   ii) Axiomática

   iii) Denotacional

Universidade da Beira Interior
Departamento de Informática
Paradigmas de Programação

Frequência 2
Duração: 01h45m
Tolerância: 15 minutos

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

   (a) Usando apenas listas, o operador `<*>` e a função `pure`, crie uma função que, dadas duas listas, devolve uma lista com o resultado de adicionar todas as combinações possíveis dos elementos das duas listas e de multiplicar esse resultado por 2.

   Por exemplo, para as listas $[1, 2]$ e $[3, 4]$ a função deve devolver a lista $[8, 10, 10, 12]$ que corresponde aos resultados das seguintes operações:

   $[2 * (1 + 3), 2 * (1 + 4), 2 * (2 + 3), 2 * (2 + 4)]$. **(1.5 pontos)**

   (b) Usando apenas o operador `<$>` defina uma função que recebe um argumento do tipo [[Maybe Int]] e incrementa por dois todos os inteiros desse input.

   Por exemplo, para a lista

   [[Just 1, Just 2],[Just 3]]

   a função devolve a lista

   [[Just 3, Just 4],[Just 5]]. **(1.5 pontos)**

   (c) Considere uma função `g` que recebe como input um inteiro `x` e devolve `Nothing` caso esse inteiro seja igual a zero e devolve `Just (x-1)` caso contrário. Indique qual o resultado das seguintes computações: **(1.5 pontos)**

   i. `g 3 >>= g >>= g >>= (\x -> Just (x + 1)) >>= g`

   ii. `g 2 >>= g >>= g >>= g`

   (d) Considere a função:

   ```
   f x = [x * x]
   ```

   Resolva o erro na seguinte computação: **(0.75 pontos)**

   ```
   pure f <$> [1,2,3]
   ```

   que pretende aplicar `f` a todos os elementos da lista para obter o resultado $[[1], [4], [9]]$.

Universidade da Beira Interior
Departamento de Informática
Paradigmas de Programação

Frequência 2
Duração: 01h45m
Tolerância: 15 minutos

3. 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 `Sqr Expr`, que multiplica uma expressão por si mesma (i.e. calcula o quadrado da expressão; exemplo: `Sqr (Val 2)` corresponde a $2 * 2$). Adapte todo o código acima para suportar este novo constructor. **(1.5 pontos)**

(b) Nas aulas foi demonstado que este compilador estava correcto quando a linguagem continha apenas os constructores `Val` e `Add`. Prove que o compilador continua correcto com a adição do novo constructor `Sqr` (escreva apenas a prova para `Sqr`). Formule claramente a equação que exprime a correcção do compilador. **(2 pontos)**
Assuma a regra da distributividade: `exec (c ++ d) s = exec d (exec c s)`

(c) Defina o tipo `Expr` como instância da class `Show`. Elementos do tipo `Expr` devem ser mostrados da seguinte forma:

- com parênteses à volta dos argumentos de cada operação;
- com o operador `Add` substituído pelo símbolo da adição (+) e mostrado de forma infixa (i.e. `Add (Val 2) (Val 3)` é mostrado no ecrã como "$(2 + 3)$");
- com o operador `Sqr` substituído por ^2 (i.e. `Sqr (Val 3)` é mostrado no ecrã como "(3^2)").

Por exemplo, a expressão `Add (Sqr (Val 4)) (Val 5)` deve ser mostrada da seguinte forma: "((4^2) + 5)" **(1.5 pontos)**

Nota: Comente o `deriving Show` no tipo `Expr` para não ter conflitos de definições.

(d) Considere agora o seguinte tipo de dados de expressões polimórficas:

```
data ExprP a = ValP a | AddP (ExprP a) (ExprP a) deriving Show
```

Dadas as seguintes definições de `ExprP` como instância das classes `Functor` e `Applicative`:

```
instance Functor ExprP where
    fmap f (ValP n) = ValP (f n)
    fmap f (AddP a b) =  AddP (fmap f a) (fmap f b)

instance Applicative ExprP where
    pure = ValP
    ValP f <*> x = fmap f x
    AddP f g <*> s = AddP (f <*> s) (g <*> s)
```

Defina `ExprP` como instância da classe `Monad`. **(1.25 pontos)**

(e) Prove que a sua definição de `ExprP` como instância da classe `Monad` obedece às 3 leis monádicas. **(2.5 pontos)**

4. Dado o seguinte tipo que representa o saldo de duas contas bancárias:

```
data EstadoContas = EJ {c1::Int, c2::Int} deriving Show
```

Use um `monad transformer` para implementar um programa que pretende simular uma versão muito simplificada de um sistema bancário online para um utilizador com duas contas.

- *c1* representa a conta 1;
- *c2* representa a conta 2;

O programa pede ao utilizador que introduza o número da conta (1 ou 2) para indicar qual a conta em que pretende fazer um depósito ou débito e pede de seguida o montante a depositar/debitar (valor inteiro, positivo para crédito e negativo para débito).

O programa procede até que um número de conta igual a zero seja introduzido – só termina neste caso, em todos os outros continua a executar. Caso o número de conta introduzido seja diferente dos valores 0, 1 e 2, o programa deve indicar que o número de conta está errado e voltar a pedir novo número. Caso um débito exceda o saldo actual da conta, o programa deve recusar a transação e mostrar uma mensagem apropriada ao utilizador. **(2.5 pontos)**

Universidade da Beira Interior
Departamento de Informática
Paradigmas de Programação

Frequência 2
Duração: 01h45m
Tolerância: 15 minutos

5. Considere o seguinte tipo de dados:

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

Considere as seguintes definições de funções:

```
tamanho :: Arv a -> Int
tamanho (Folha a)  = 1
tamanho (Nodo a b) = tamanho a + tamanho b

balanceada :: Arv a -> Bool
balanceada (Folha x)   = True
balanceada (Nodo e d) = tamanho e == tamanho d
                        && balanceada e
                        && balanceada d

espelhar :: Arv a -> Arv a
espelhar (Folha a) = Folha a
espelhar (Nodo a b) = Nodo (espelhar b) (espelhar a)
```

- A função `tamanho` devolve o tamanho da árvore, definido como o número de folhas contidas na árvore;
- A função `balanceada` indica se uma árvore é balanceada, i.e. se todos os seus nodos têm a propriedade de que a sub-árvore esquerda e a sub-árvore direita têm o mesmo tamanho. As folhas são trivialmente balanceadas.
- A função `espelhar` recebe uma árvore como input e devolve uma outra árvore que corresponde à original espelhada, i.e. na qual as folhas aparecem na ordem inversa da original.

Usando estas definições e assumindo a propriedade:

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

prove que

```
balanceada (espelhar t) = balanceada t
```

usando indução sobre a árvore `t`. **(2 pontos)**