Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Letivo de 2015/2016
Linguagens e Ambientes Programação – Teste 1

11 de Abril de 2016 às 18:00

Teste com consulta com 1 hora e 30 minutos de duração + 15 minutos de tolerância

Nome:	Num:

Notas: *Este enunciado é constituído por 3 grupos de perguntas. Responda no próprio enunciado,* ***usando a frente e o verso****.*
Nos problemas em OCaml mostre que sabe usar o método indutivo e escreva, se possível, funções de categoria 1 ou 2.
Não use mecanismos ou raciocínios imperativos nem simule mecanismos ou raciocínios imperativos.
Pode definir funções auxiliares sempre que quiser e também pode chamar diretamente funções do módulo `List`*.*
Normalmente, respostas imperfeitas merecem alguma pontuação.
Fraude implica reprovação na cadeira.

1. [3 valores] Escolha múltipla. As respostas erradas não descontam. Indique as respostas aqui:

A	B	C

A) Depois dum programa OCaml ser compilado com sucesso, e quando se tenta correr o programa, qual das seguintes afirmações fica verdadeira?

a) Nunca ocorrem erros de execução da responsabilidade do código do programa.

b) Podem ocorrer erros de tipo, por exemplo ao tentar usar um inteiro como se fosse uma função, ou ao tentar somar um inteiro com uma string.

c) A sobrecarga (overloading) passa a poder ser tratada em tempo de execução.

d) A execução do programa pode não terminar, podendo assim não haver resultado.

B) Relativamente às linguagens OCaml e C, qual é a única afirmação verdadeira?

a) As duas linguagens possuem uma norma oficial ANSI.

b) Nenhuma das duas linguagens possui uma norma oficial ANSI.

c) Apenas o C possui uma norma oficial ANSI.

d) Apenas o OCaml possui uma norma oficial ANSI.

C) Qual das seguintes afirmações é a verdadeira?

a) Todos os paradigmas de programação incluem uma noção de estado.

b) Numa linguagem funcional pura não é possível usar estado, nem mesmo simulá-lo.

c) Sem usar o conceito de estado, não é possível raciocinar e escrever programas.

d) Estado é o conceito de base do paradigma imperativo. Se esse conceito também surgir numa linguagem orientada pelos objetos, então é porque essa linguagem junta dois paradigmas.

2. [3 valores] Diga qual o tipo OCaml da seguinte função:

```
let f x y = x :: (y x) ;;
```

3. Abaixo estão definidos dois tipos OCaml, no contexto do espaço cartesiano real a duas dimensões. O tipo **`point`** representa pontos 2D, e.g. **`(2.3,4.5)`**. O tipo **`vector`** representa vetores orientados do primeiro ponto para o segundo pontos, por exemplo, **`((2.3,4.5), (0.0,5.0))`**. Os dois pontos que definem um vetor chamam-se extremidades.

```
type point = double * double ;;
type vector = point * point ;;
```

As funções que se pedem abaixo devem ser escritas dentro do paradigma funcional, sem simular estado. Se as funções ficarem com tipo mais geral do que o pedido, isso é normal.

a) [3 valores] Escreva uma função booleana **`isValid`** que produza true apenas se todos os elementos duma lista de vetores forem não vazios. Basta ocorrer um vetor vazio (i.e. com extremidades sobrepostas) para que a resposta seja false.

```
isValid: vector list -> bool
```

Exemplos:

```
isValid [] = true
isValid [((2.3,4.5),(0.0,5.0)); ((1.5,4.5),(8.9,-123.1))] = true
isValid [((2.3,4.5),(0.0,5.0)); ((0.0,5.0),(2.3,4.5))] = true
isValid [((2.3,4.5),(0.0,5.0)); ((2.3,4.5),(2.3,4.5))] = false
```

Complete a função:

```
let rec isValid l =
   match l with
        [] ->
      |(p1,p2)::xs ->
   ;;
```

b) [3 valores] Escreva uma função para testar se uma sequência de vetores define no seu todo uma única **continuidade** (linha contínua) i.e.: **para cada par de vetores consecutivos**, a extremidade final do primeiro coincide com a extremidade inicial do segundo. Os diferentes vetores duma continuidade podem intersetar-se, que isso não é preocupação nossa.

```
isContinuity: vector list -> bool
```

Exemplos:

```
isContinuity [] = true
isContinuity [((2.3,4.5),(0.,5.))] = true
isContinuity [((2.3,4.5),(0.,5.)); ((0.,5.),(8.9,3.1)); ((8.9,3.1),(11.3,0.))] = true
isContinuity [((2.3,4.5),(0.0,5.0)); ((1.5,4.5),(1.5,9.9))] = false
```

Complete a função:

```
let rec isContinuity l =
   match l with
      [] ->
   |  [(p1,p2)] ->
   |  (p1,p2)::(q1,q2)::xs ->
;;
```

c) [3 valores] Escreva uma função para contar o número de **continuidades** (linhas contínuas de comprimento máximo) que se conseguem detetar numa lista de vetores. As diferentes continuidades podem intersetar-se, que isso não é preocupação nossa.

```
countContinuities: vector list -> int
```

Exemplos:

```
countContinuities [] = 0
countContinuities [((2.3,4.5),(0.,5.)); ((0.,5.),(8.9,3.1)); ((8.9,3.1),(11.3,0.))] = 1
countContinuities [((2.3,4.5),(0.0,5.0)); ((1.5,4.5),(1.5,4.5))] = 2
```

* Escreva a sua solução nas costas desta folha.

4. O tipo **`ctree`** permite representar árvores binárias de letras:

```
type ctree = Nil | Node of char * ctree * ctree ;;
```

a) [2.5 valores] Escreva uma função para converter uma árvore de letras numa representação linear alternativa com parêntesis e de vírgulas. Descreve-se a conversão usando um exemplo: estude bem como a árvore **`t1`** é convertida na lista **`l1`**. Para entender melhor o resultado, mostra-se também a essência da lista **`l1`** a negrito.

```
let t1 =
   Node('a',
       Node('b',
           Node('d', Nil, Nil),
           Node('e', Nil, Nil)
       ),
       Node('c',
           Nil,
           Node('f',
               Node('g', Nil, Nil),
               Nil
           )
       )
   );;
let l1 = ['a';'(';'b';'(';'d';',';'e';')';',';'c';'(';' ',';'f';'(';'g';',';')';')';')'];;
```

Essência de **`l1`**: **`a(b(d,e),c(,f(g,)))`**

Eis o tipo pretendido para a função:

```
treeToChars: ctree -> char list
```

Exemplos:

```
treeToChars Nil = []
treeToChars (Node('a',Nil,Nil))  = ['a']                                  (* a *)
treeToChars (Node('a',Node('b',Nil,Nil),Nil))  = ['a';'(';'b';',';')']  (* a(b,) *)
treeToChars t1  = l1
```

Complete a função:

```
let rec treeToChars t =
   match t with
        Nil ->
      |Node(x,Nil,Nil) ->
      |Node(x,l,r) ->
   ;;
```

b) [2.5 valores] Escreva uma função recursiva (sem simular estado) que faça a conversão inversa da alínea anterior.

```
charsToTree: char list -> ctree
```

Exemplos:

```
charsToTree [] = Nil
charsToTree ['a'] = Node('a',Nil,Nil)
charsToTree ['a';'(';'b';',';')'] = Node('a',Node('b',Nil,Nil),Nil)
charsToTree l1  = t1
```