

# Mestrado Integrado em Engenharia Informática (FCT/UNL)

Ano Lectivo 2013/2014

## Linguagens e Ambientes Programação – Teste 1

04 de Abril de 2014 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 escrever, quando possível, funções de tipo 1 ou 2. Pode definir funções auxiliares sempre que quiser e também pode usar funções do módulo `List` do OCaml. 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) Sobre a linguagem OCaml, qual é a afirmação verdadeira?

- a) É uma linguagem funcional pura.
- b) É uma linguagem multiparadigma, mas em LAP estamos essencialmente a usar apenas a sua parte funcional.
- c) Não é possível escrever funções realmente funcionais por causa dos efeitos laterais, que são inevitáveis.
- d) A linguagem possui uma norma oficial, publicada por um organismo oficial de normalização.

B) Interpretador versus compilador para código nativo. Qual destas afirmações é falsa?

- a) Uma das principais vantagens de se usar um compilador, é a portabilidade do código executável.
- b) A execução dum programa já compilado é normalmente mais rápida do que a de um programa que está a ser interpretado. Uma das razões é o primeiro não ter de se preocupar com o processamento do código fonte.
- c) Na implementação dos interpretadores mais sofisticados, usam-se internamente algumas das técnicas típicas dos compiladores.
- d) Durante a fase de desenvolvimento dos programas, normalmente é mais produtivo usar um interpretador do que um compilador.

C) O que faz a seguinte função OCaml?

```
let rec f l =  
  match l with  
  [] -> []  
  | (a,b)::xs -> (a b)::f xs ;;
```

- a) A lista `l` contém pares ordenados  $(a, b)$ . A função `f` cria uma lista com a segunda componente desses pares.
- b) Trata-se duma forma equivalente de escrever a conhecida função `map`, que faz parte da biblioteca do OCaml e que foi estudada nas aulas.
- c) Só produz resultado se `l` for a lista vazia. No outro caso, entra em ciclo.
- d) A lista `l` contém pares (função, argumento). A função `f` aplica todas essas funções aos respetivos argumentos e produz a lista dos resultados.

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

```
let rec f l a =  
  match l with  
  [] -> 0  
  | x::xs -> (x a) + f xs a ;;
```

3. [14 valores] Neste problema, chamamos **árvore de classificação** a uma árvore binária que classifica grupos de indivíduos de acordo com um conjunto de propriedades. Os nós internos da árvore chamam-se **nós de seleção** e têm três componentes: uma string **p** que denota uma propriedade; uma subárvore esquerda **y** que representa todos os indivíduos que verificam essa propriedade; uma subárvore direita **n** que representa todos os indivíduos para os quais essa propriedade é falsa. As folhas chamam-se **classes** e representam conjuntos de indivíduos.

O nosso tipo OCaml para árvores de classificação é o seguinte:

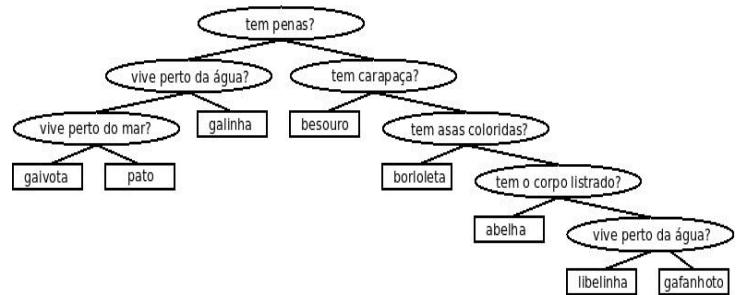
```
type ctree = Class of string | Selection of string * ctree * ctree ;;
```

Eis um exemplo minimal duma árvore de classificação sobre mobiliário, que só tem uma folha:

```
let furniture = Class "mesa"
```

Agora um exemplo, mais substancial, duma árvore de classificação para animais que conseguem voar:

```
let flying =
  Selection("tem penas?",
    Selection("vive perto da água?",
      Selection("vive perto do mar?",
        Class("gaivota"),
        Class("pato")
      ),
      Class("galinha")
    ),
    Selection("tem carapaça?",
      Class("besouro"),
      Selection("tem asas coloridas?",
        Class("borboleta"),
        Selection("tem o corpo listrado?",
          Class("abelha"),
          Selection("vive perto da água?",
            Class("libelinha"),
            Class("gafanhoto")
          )
        )
      )
    )
  ) ;;
```



Confirme que percebeu a árvore anterior. Por exemplo, estude as classes dos patos e dos besouros. Confirme que um pato *tem penas*, *vive perto de água*, mas **não** *vive perto do mar*. Um besouro **não** *tem penas* mas *tem carapaça*.

Eis um exemplo duma função que testa se uma dada classe está presente numa árvore.

```
let rec belongs ca t =
  match t with
  | Class c -> c = ca
  | Selection(p,y,n) -> belongs ca y || belongs ca n
;;
```

Programe as seguintes funções em OCaml usando o método indutivo:

a) [3 valores] Função para obter o nome de todas as classes contidas numa árvore.

```
allClasses: ctree -> string list
exemplo: allClasses flying =
  ["gaivota"; "pato"; "galinha"; "besouro"; "borboleta"; "abelha"; "libelinha"; "gafanhoto"]

let rec allClasses t =
  match t with
  | Class c ->
    | Selection(p,y,n) ->

  ;;
```

b) [3 valores] Função para testar se é possível afirmar que uma dada classe verifica uma certa propriedade. [Sugestão: durante o raciocínio indutivo, use a função belongs se lhe der jeito.]

```
check: string -> string -> ctree -> bool
exemplos: check "gaivota" "vive perto da água?" flying = true
          check "gaivota" "tem o corpo listrado?" flying = false
          check "gaivota" "tem bico?" flying = false
          check "cadeira" "tem pernas?" furniture = false

let rec check ca pa t =
  match t with
  | Class c ->
    | Selection(p,y,n) ->

  ;;
```

c) [3 valores] Função para obter o nome de todas as classes que verificam uma dada propriedade. [Sugestão: durante o raciocínio indutivo, se lhe der jeito, use a função `allClasses`.]

```
allYes: string -> ctree -> string list
exemplo: allYes "vive perto da água?" flying = ["gaivota"; "pato"; "libelinha"]
          allYes "tem pernas?" furniture = []

let rec allYes pa t =
```

d) [3 valores] Função para obter o nome de todas as classes que não se sabe se verificam ou não uma dada propriedade.

```
allUnknown: string -> ctree -> string list
exemplo: allUnknown "vive perto da água?" flying = ["besouro"; "borboleta"; "abelha"]
          allUnknown "tem pernas?" furniture = ["mesa"]

let rec allUnknown pa t =
```

e) [2 valores] Função para determinar a distância mínima entre duas ocorrências numa árvore dada uma dada propriedade **pa**. Trata-se do número mínimo de arcos da árvore que é preciso atravessar para ir de um nó até ao outro. Assume-se que a árvore nunca tem mais do que um milhão de arcos. Se a propriedade **pa** não ocorrer pelo menos duas vezes na árvore, o valor a retornar é um milhão mais um (1000001) que intuitivamente representa "*mais infinito*". Esta convenção simplifica a programação da função [Sugestão: provavelmente, precisará de definir uma função auxiliar.]

```
minDist: string -> ctree -> int
exemplo: minDist "vive perto da água?" flying = 5
          minDist "tem penas?" flying = 1000001
          minDist "tem pernas?" furniture = 1000001

let rec minDist pa t =
```