

Mestrado Integrado em Engenharia Informática (FCT/UNL)

Ano Letivo de 2017/2018

Linguagens e Ambientes Programação – Solução Teste 1

17 de abril de 2018 às 18:00

Teste com consulta com 1 hora e 30 minutos de duração

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 dos módulos List e String. Normalmente, respostas imperfeitas merecem alguma pontuação. Fraude implica reprovação na cadeira.*

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

A	B	C
d	b	d

A) Para conseguir executar um programa...

- a) É suficiente usar um compilador.
- b) É suficiente usar um compilador juntamente com um conjunto de bibliotecas.
- c) Não é possível executar programas; apenas se executam algoritmos.
- d) Mesmo que se use um compilador, precisamos sempre dum interpretador no final, quer ele seja implementado em software ou em hardware.

```
let a = 1;;
let b = 2;;
let g x = x * a + b ;;
let f x = let b = 4 in g x;;
let z = let a = 5 in f(1);;
```

B) No programa anterior, qual o valor da constante z, sabendo que o OCaml usa escopo estático? E qual seria o valor de z se o OCaml usasse escopo dinâmico?

- a) 3 3
- b) 3 9
- c) 9 3
- d) 9 9

C) Nas implementações habituais de Java, em que momento é determinado: (1) qual o endereço duma variável local a um método e (2) qual o método invocado num envio de mensagem:

- a) Completamente em tempo de compilação.
- b) Completamente em tempo de execução.
- c) Parcialmente em tempo de compilação e parcialmente em tempo de carregamento.
- d) Parcialmente em tempo de compilação e parcialmente em tempo de execução.

2. [3 valores] Diga qual o tipo OCaml da seguinte função. A função chama-se **f** e tem três argumentos.

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

Solução:

```
('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

3. Listas. Estamos interessados em determinar o comprimento duma sequência contígua de ocorrências da letra 'z' dentro duma lista de caracteres.

a) [2.5 valores] Escreva uma função indutiva `countAtStart` para contar o número de 'z's no início duma lista de caracteres. Para além do comprimento, a função também retorna a parte final da lista que não chegou a ser usada na contagem. Para perceber como se organiza o resultado, veja os exemplos.

```
countAtStart: char list -> int * char list
```

Exemplos:

```
countAtStart ['z';'z';'a';'z';'z';'z';'a';'x'] = (2, ['a';'z';'z';'z';'a';'x'])
countAtStart ['x';'z';'a'] = (0, ['x';'z';'a'])
```

[Se a função produzir apenas o valor inteiro, em vez do par pedido, a cotação máxima será 70%.]

Solução:

```
let rec countAtStart l =
  match l with
  | [] -> (0, [])
  | x::xs ->
    let (c, r) = countAtStart xs in
    if x = 'z' then (c+1, r)
    else (0, l)
;;
```

outra maneira:

```
let rec countAtStart l =
  match l with
  | [] -> (0, [])
  | 'z'::xs ->
    let (c, r) = countAtStart xs in
    (c+1, r)
  | x::xs -> (0, l)
;;
```

b) [2.5 valores] Escreva uma função indutiva `countFirst` para determinar o comprimento da primeira sequência contígua de 'z's numa lista de caracteres. A função retorna zero apenas no caso de não ocorrer qualquer 'z' na lista. Para além do comprimento, a função também retorna a parte inicial da lista que não chegou a ser usada na contagem e a parte final da lista que não chegou a ser usada na contagem. Para perceber como se organiza o resultado, veja os exemplos.

```
countFirst: char list -> char list * int * char list
```

Exemplos:

```
countFirst ['q';'q';'z';'z';'z';'a';'z';'z';'a';'x'] = (['q'; 'q'], 3, ['a'; 'z'; 'z'; 'a'; 'x'])
countFirst ['x';'a';'a'] = ([], 0, ['x';'a';'a'])
```

[Se a função produzir apenas o valor inteiro, em vez do triplo pedido, a cotação máxima será 70%.]

Solução:

```
let rec countFirst l =
  match l with
  [] -> ([], 0, [])
  | x::xs ->
    let (c,r) = countAtStart l in
    if c > 0 then ([], c, r)
    else
      let (p,c,r) = countFirst xs in
      if c > 0 then (x::p, c, r)
      else ([], 0, l)
;;
```

outra maneira:

```
let rec countFirst l =
  match l with
  [] -> ([], 0, [])
  | 'z'::xs ->
    let (c,r) = countAtStart l in
    ([], c, r)
  | x::xs ->
    let (p,c,r) = countFirst xs in
    if c > 0 then (x::p, c, r)
    else ([], 0, l)
;;
```

4. Árvores. Uma base de dados implementada em OCaml armazena **inteiros, strings, e registos**. Cada registo associa valores a nomes usando uma lista de pares ordenada crescentemente por nome. (Existe um exemplo mais abaixo).

Quando se programou a base de dados, era impossível prever quais os tipos dos valores que cada utilizador iria usar e por isso foram introduzidos tipos dinâmicos na base de dados.

Eis a representação em OCaml dos valores da base de dados:

```
type dbValue =
  | VInt of int
  | VStr of xstring
  | VRec of (xstring * dbValue) list
;;
```

e a representação em OCaml dos tipos dinâmicos da base de dados:

```
type dbType =
  | TInt
  | TStr
  | TRec of (xstring * dbType) list
;;
```

Para perceber bem o que está em causa, estude este exemplo de valor da base de dados:

```
let joao =
  VRec [("altura", VInt 180);
        ("nome", VStr "joao");
        ("telefone", VRec [("numero", VInt 123456789); ("prefixo", VInt 351)])]
;;
```

e o respetivo tipo dinâmico:

```
let joaoType =
  TRec [("altura", TInt);
        ("nome", TStr);
        ("telefone", TRec [("numero", TInt); ("prefixo", TInt)])]
;;
```

O tipo `joaoType` indica que a altura do `joao` é um inteiro, o seu nome é uma string e o seu número de telefone é um registo com duas partes inteiras.

A base de dados pode conter valores muito diversos, alguns tão simples como o valor `joao`, outros muito mais complicados.

Podemos processar os valores e os tipos dinâmicos usando diversas técnicas da programação funcional, por exemplo a técnica que se mostra a seguir, cujo uso se recomenda. Esta função soma todos os valores inteiros que ocorrem num valor:

```
let rec xum v =
  match v with
  | VInt i -> i
  | VStr x -> 0
  | VRec [] -> 0
  | VRec ((x,v)::xx) -> xum v + xum (VRec xx)
;;
```

Nos exercícios que se seguem, aproveite e chame as seguintes funções `cons` e `tcons` já definidas. Respetivamente, acrescentam um par (com valor) à cabeça dum registo e um par (com tipo) à cabeça dum tipo-registo.

```
let cons (x,v) (VRec l) =
  VRec ((x,v)::l);;

let tcons (x,t) (TRec l) =
  TRec ((x,t)::l);;
```

[Não escreva nada nesta folha nem nas suas costas. Esta folha, que não tem alíneas. Será ignorada na correção.]

a) [3 valores] Escreva uma função indutiva **getType** para obter o tipo dum valor.

```
getType: dbValue -> dbType
```

Exemplo: `getType joao = joaoType`

Solução:

Também podia ser programada usando o esquema para árvores n-árias da teórica 4. Mas fica um pouco mais fácil usar o esquema sugerido no enunciado.

```
let rec getType v =
  match v with
  | VInt _ -> TInt
  | VStr _ -> TStr
  | VRec [] -> TRec []
  | VRec ((n,v)::xs) -> tcons (n,getType v) (getType (VRec xs))
;;
```

b) [3 valores] Escreva usando o método indutivo uma função **makeZero** para criar um valor dum dado tipo, no qual todos os inteiros que ocorrem são zero e todas as strings que ocorrem são vazias.

```
makeZero: dbType -> dbValue
```

Exemplo: `makeZero joaoType = VRec(["altura", VInt 0]; ("nome", VStr "")); ("telefone", VRec(["numero", VInt 0]; ("prefixo", VInt 0)))]`

Solução:

Também podia ser programada usando o esquema para árvores n-árias da teórica 4. Mas fica um pouco mais fácil usar o esquema sugerido no enunciado.

```
let rec makeZero t =
  match t with
  | TInt -> VInt 0
  | TStr -> VStr ""
  | TRec [] -> VRec []
  | TRec ((n,t)::ts) -> cons (n, makeZero t) (makeZero (TRec ts))
;;
```

c) [3 valores] Escreva usando o método indutivo uma função **similar** que compara dois valores e produz true apenas no caso deles serem praticamente iguais: permite-se que difiram apenas no valor dum único inteiro ou duma única string.

```
ximilar: dbValue -> dbValue -> bool
```

Exemplos: `ximilar joao joao = true`

```
ximilar (VInt 3) (VInt 3) = true
```

```
ximilar (VInt 3) (VInt 2) = true
```

```
ximilar (VRec[("a",VInt 0);("b",VInt 0)]) (VRec[("a",VInt 1);("b",VInt 1)]) = false
```

[Resolva a alínea c) nas costas desta mesma folha.]

Solução:

Este problema não é indutivo e tem de ser programado usando uma função de categoria 4 (ver final da teórica 8).

```
let rec diffs v1 v2 = (* conta as diferencas *)
  match v1, v2 with
  | VInt a, VInt b -> if a = b then 0 else 1
  | VStr a, VStr b -> if a = b then 0 else 1
  | VRec [], VRec [] -> 0
  | VRec ((n1,v1)::xs), VRec ((n2,v2)::ys) ->
    if n1 <> n2 then 999
    else
      (diffs v1 v2) + diffs (VRec xs) (VRec ys)
  | _, _ -> 999
;;

let rec similar v1 v2 = (* o numero de diferencas tem de ser <= 1 *)
  diffs v1 v2 <= 1
;;
```