Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Letivo de 2016/2017
Linguagens e Ambientes Programação – Solução Teste 1

12 de abril de 2017 às 16:00

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 do módulo `Lict`*.*
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	b	b

A) Quais das seguintes funções estão programadas num espírito funcional puro, evitando usar argumentos artificiais para simular estado?

```
let rec f l =
    match l with
      [] -> 0
    | _::xx -> 1 + f xx
;;
```

```
let rec g l n =
    match l with
      [] -> n
    | x::xx -> g xx (n+1)
;;
```

a) A primeira.

b) A segunda.

c) As duas.

d) Nenhuma das duas.

B) Qual destas afirmações é verdadeira?

a) Todos os problemas que se possam descrever em 5 linhas curtas são resolúveis em OCaml e C.

b) Todos os algoritmos que se possam escrever em 5 linhas curtas são implementáveis em OCaml e C.

c) Nenhum problema pode ser resolvido escrevendo apenas 5 linhas curtas em OCaml e C.

d) Todos os problemas podem ser resolvidos escrevendo 5 linhas curtas em OCaml ou C.

C) Sobre interpretadores e compiladores. Qual destas afirmações é falsa?

a) Na fase de desenvolvimento dos programas, normalmente é mais produtivo usar um interpretador do que um compilador.

b) É preciso que o compilador permaneça instalado na mesma máquina para executar um programa previamente compilado para código nativo.

c) Para executar um programa de forma interpretada, é preciso que o interpretador esteja disponível na mesma máquina.

d) Os interpretadores JIT usam internamente algumas técnicas típicas dos compiladores.

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

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

Solução:

```
('a -> 'a -> 'a) -> 'a -> ('a -> 'a)
```

3. O tipo **exp** define uma representação de expressões algébricas muito simples onde apenas podem ocorrer: operação binária de adição; constantes reais; variável **v**.

```
type exp =
        Add of exp*exp
      | Const of float
      | V
;;
```

Uma expressão é na realidade uma árvore cujas folhas são variáveis e constantes, e cujos nós internos representam adições.

Eis uma expressão que será usada nos exemplos dos problemas:

```
let example = Add(Add(Const 2.14,V), Add(V,Add(V,Const 5.0)));;
```

Eis uma função indutiva que processa uma expressão:

```
let rec count_consts t =  (* count the number of constants *)
      match t with
        Add (l, r) -> count_consts l + count_consts r
      | Const _ -> 1
      | _ -> 0
;;
```

a) [3 valores] Escreva uma função indutiva **mult** para multiplicar um valor inteiro positivo por uma expressão. Como o operador de multiplicação não está disponível nas nossas expressões, o resultado terá de envolver repetidas utilizações de **Add**. Neste problema, a forma de apresentar o resultado poderá variar.

```
mult: int -> exp -> exp
```

Exemplos:

```
mult 1 V = V
mult 5 V = Add(V, Add(V, Add(V, Add(V, V))))
mult 3 (Add(V, Const 0.1)) = Add(Add(V,Const 0.1), Add(Add(V,Const 0.1), Add(V,Const 0.1)))
```

Solução:

```
let rec mult n t =
    if n = 1 then t
    else Add(t, mult (n-1) t)
;;
```

b) [3 valores] Escreva uma função indutiva **belongs** para testar se a primeira expressão ocorre na segunda expressão.

```
belongs: exp -> exp -> bool
```

Exemplos:

```
belongs example example = true
belongs (Const 2.14) example = true
belongs (Const 4.4) example =  false
```

Solução:

```
let rec belongs t1 t2 =
    t1 = t2 ||
        match t2 with
            | Add (l, r) -> belongs t1 l || belongs t1 r
            | _ -> false
;;
```

c) [3 valores] Escreva uma função indutiva **`replace`** para produzir uma cópia da segunda expressão onde todas as ocorrências de **v** foram trocadas pela primeira expressão.

```
replace: exp -> exp -> exp
```

Exemplos:

```
replace (Const 2.14) V = Const 2.14
replace (Add(V,V)) example = Add(Add(Const 2.14, Add(V,V)), Add(Add(V,V), Add(Add(V,V), Const 5.0)))
```

Solução:

```
let rec replace t1 t2 =
    match t2 with
      Add (l,r) -> Add(replace t1 l, replace t1 r)
    | V -> t1
    | _ -> t2
;;
```

d) [3 valores] Escreva uma função indutiva **`right_desc`** para produzir uma expressão algebricamente equivalente que só cresça para a direita. Crescer para a direita significa que, nos nós aditivos, as subárvores da esquerda são sempre folhas. Neste problema, a forma do resultado poderá variar.

```
right_desc: exp -> exp
```

Exemplos:

```
right_desc example = Add(Const 2.14, Add(V, Add(V, Add(V, Const 5.0))))
```

Soluçao:

```
let rec right_join t1 t2 =
    match t1 with
      Add (l,r) -> Add(l, right_join r t2)
    | _ -> Add(t1, t2)
;;

let rec right_desc t =
    match t with
      Add (l,r) ->
        right_join (right_desc l) (right_desc r)
    | _ -> t
;;
```

e) [3 valores] Escreva uma função indutiva **`simpl_consts`** para consolidar todas as constantes duma expressão numa única constante. Esta aparecerá dentro do resultado no ponto em que você desejar. O resultado deverá ser algebricamente equivalente ao argumento. Se o argumento não contiver nenhuma constante, o resultado também não conterá nenhuma constante. Neste problema, a forma de apresentar o resultado poderá variar.

```
simpl_consts: exp -> exp
```

Exemplos:

```
simpl_consts (Const 6.7) = Const 6.7
simpl_consts V = V
simpl_consts (Add(Const 1.0, Add(Const 1.0, Const 1.0))) = Const 3.0
simpl_consts example = Add (Const 7.14, Add (V, Add  (V, V)))
```

Resolva nas costas

Solução:

```
let rec simpl_consts t =
    match t with
     Add (l, r) -> (
        match (simpl_consts l), (simpl_consts r) with
        | Add (Const a, r1), Add (Const b, r2) -> Add(Const (a+.b), Add(r1, r2))
        | Add (Const a, r1), Const b -> Add(Const (a+.b), r1)
        | Add (Const a, r1), t2 -> Add(Const a, Add(r1, t2))
        | Const a, Add (Const b, r2) -> Add(Const (a+.b), r2)
        | Const a, Const b -> Const (a+.b)
        | t1, Add (Const a, r2) -> Add(Const a, Add(t1, r2))
        | t1, Const b -> Add(Const b, t1)
        | t1, t2 -> Add(t1, t2)
        )
    | _ -> t
;;
```