Programação Funcional – Laboratório 2

(Funções, recursividade, polimorfismo e sobrecarga)

Parte I - Composição de Funções

A composição de funções, base do modelo de programação funcional, é representada por **f • g**, sendo **f** e **g** funções.

Ela possui o seguinte comportamento: é aplicada a função **g** sobre seu argumento e, após, a função **f** é aplicada sobre seu resultado.

- Teste o exemplo a seguir e depois faça uma função composta utilizando quaisquer das funções já definidas

```
fun mult(constante)(x)= constante * x;

(* fazer testes diversos tais como *)
mult (inc 2 ) 10;
mult (mult 2 3) 10;
mult 3 5 ;

(* Funções que recebem funções como argumentos *)
fun dobra f x = f ( f (x));
```

- Fazer os testes abaixo (investigue o tipo de cada função antes de fazer os testes):

```
val tri = mult 3;
tri 5;
quadr 5
dobra quadr 5
val pot 4 = dobra quadr;
pot4 4;
```

Parte II - Funções recursivas

Exemplo de função fatorial:

```
fun fat n = if n=0 then 1 else n* fat(n-1)

(* Outro exemplo: maximo divisor comum *)
fun gcd(m,n):int = if m=n then n
                   else if m>n then gcd(m-n,n)
                        else gcd(m,n-m);
```

Testes:

```
gcd(12,30);
gcd(126,2357);
```

Experimente outras funções recursivas:

1. fibonacci: calcula o n- esimo termo da serie 0 1 1 2 3 ...
   fib(n) = se n =1 then return 0 else if n = 2 then return 1 else return (fib(n-1) + fib(n-2))
2. Máximo divisor comum: outro algoritmo (Euclides)
   mdc(m,n) = se m>n entao mdc(n, m); se n=0 entao m; se n> 0 entao mdc(n, m mod n)

Parte III - Pattern Matching

Uma forma mais elegante de se definir funções recursivas é usando *pattern matching*. Em ML, pattern matching são usados para consistência de tipos quando tuplas estão envolvidas, mas também nos construtores de tipos compostos definidos pelo usuário.

```
(* pattern matching para tuplas *)
(* Seja a função compare definida anteriormente *)

fun compare (x, p ,q) =
            if x = p then
                        if x = q then “both”
                        else “first”
            else
                        if x = q then “second”
                        else “neither”;

(* qualquer uma das chamadas abaixo é válida *)

compare (1,2,3);

let val t = (“modelos”, “linguagens”, “programação”) in
     compare(t)     (* pattern matching de t com (x,p,q) *)
end;

let val d = (2, 3)  in
     let val (a,b) = d in   (* pattern matching de (a,b) com (2,3) *)
          compare(1,a,b)
    end
end;
```

Da mesma forma, podemos usar *pattern matching* para explicitar os possíveis argumentos passados para uma função:

```
fun fat 0 = 1
|   fat 1 = 1
|   fat n =  n* fat(n-1);

fun append (nil, lista2) = lista2
|   append (h::t, lista2) = h :: append (t, lista2) ;

(*-------------------implemente e teste as seguintesfunções recursivas---
------------------*)

(* fibonacci: calcula o n- esimo termo da serie 0 1 1 2 3 ... *)
fib(n) = se n = 1 then return 0 else if n = 2 then return 1 else return
(fib(n-1) + fib(n-2)) *)
```

```
(* maximo divisor comum: outro algoritmo (Euclides)
mdc(m,n) = se m>n entao mdc(n, m); se n=0 entao m; se n> 0 entao mdc(n, m
mod n) *)
```

Parte IV - Funções sobre listas

Em ML, todos os elementos de uma lista têm o mesmo tipo. Porém, funções que operam sobre listas podem ser polimórficas.

```
fun append (lista1, lista2) =
            if lista1 = nil
            then lista2
            else hd(lista1) :: append ( tl(lista1), lista2);

fun member (a, lista) =
            if lista = nil then false
            else if a = hd (lista) then true
            else member (a, tl(lista));
```

Como listas em ML são tipos homogêneos (elementos do mesmo tipo) e listas podem ser representadas como A::B::C::nil (seguindo a idéia de que uma lista é uma seqüência de construtores), qual é o tipo de **nil**? Na verdade, para que o nil possa ser usado em qualquer lista, ele não é definido como um objeto da linguagem (objetos têm tipos), mas como uma função polimórfica do tipo unit → ´a list. O tipo pré-definido **unit** é um “comando vazio”, análogo ao void do C. Assim, uma função em ML que não tem argumentos é uma função que recebe um unit:

fun teste () = “não faz nada, mas retorna um valor”;

Da mesma forma, uma função que é executada apenas pelos seus efeitos colaterais (ML não é puramente funcional), retorna um tipo unit:

```
load “Io”; (* io = input/output *)
TextIO.output(TextIO.stdOut,"fiz, mas não retornei!");
```

- Defina e verifique o tipo de cada uma das seguintes funções:

```
fun fone(x:int) = [x,x,x];

fun ftwo(x) = (x,x,x);

fun fthree(x,y) = [x ^ "b", y];

fun ffour(x,y,z) = (x+(size y),z);
```

- Descubra o tipo de cada uma das funções pré-definidas: explode, rev, hd and tl. Execute cada função em uma lista ou uma string e verifique:

  - O tipo de explode e o nome de sua inversa.
  - O tipo de rev e o nome de sua inversa.
  - O tipo de hd e seu significado
  - O tipo de tl e seu significado.

Qual o resultado das avaliações abaixo?

- `hd(explode "south");`
- `hd(tl(explode "north"));`
- `hd(rev(explode "east"));`
- `hd(tl(rev(explode "west")));`

- Podemos usar o operador o para criar funções de funções (outra forma de definir funções de mais alta ordem):

```
val first = hd o explode;

val second = hd o tl o explode;
```

Crie as funções third, fourth e last usando o operador de composição. Veja que essas funções extraem um caractere de um string. Note que em uma cadeia de funções compostas, a última função é aplicada primeiro.

- Use as funções first, second ... para criar as funções abaixo:

```
fun roll s = implode [fourth s , first s , second s , third s];

fun exch s = implode [second s , first s , third s , fourth s];
```

Teste essas funções em algum string de 4 caracteres ("ache" e "vile", por exemplo).

Aplicação geral (apply-to-all):

Forma funcional que usa uma única função como parâmetro e fornece uma lista de valores obtidos pela aplicação da função dada a cada elemento da lista de parâmetros.

Notação: α

Exemplo:

Para h (x) ≡ x * x * x, α (h, (3, 2, 4)) resulta em (27, 8, 64)

Em ML, a função **map** implementa esta composição: veja abaixo junto com os exemplos de operações sobre listas.

```
val words = ["ache", "vile", "amid", "evil", "ogre"];

map roll words;

map exch words;
```

Usando apenas composição das funções roll e exch, defina as funções que executam as seguintes transformações (uma função para cada):

- `fb "seat"  -> "eats"`
- `fc "silt"  -> "slit"`
- `fd "more"  -> "rome"`

Parte V – Polimorfismo e Sobrecarga

Em ML, o uso de objetos deve ser consistente (coerente), mas o tipo dos objetos não precisam ser completamente especificados na definição da função.

Analise a seguinte definição de função que compara três objetos quaisquer, observando a assinatura de função identificada pelo interpretador:

```
fun compare (x, p ,q) = if x = p then if x = q then “both”
                                      else “first”
                        else if x = q then “second”
                                      else “neither”;
```

Teste com:

- `compare(1,0,0);`
- `compare(1,1,0);`
- `compare(1,1,1);`
- `compare(1,1,1);`
- `compare(#"A",#"A",#"B");`

O token ´a é chamado de tipo variável e representa qualquer tipo. O operador “=” é uma função polimórfica pré-definida da linguagem e possui a seguinte assinatura: ´a * ´a -> bool, ou seja, recebe um par de argumentos de mesmo tipo e produz um valor lógico. Veja que na avaliação da função, ´a tem um tipo definido e é o mesmo para todos os parâmetros. Porém, em diferentes chamadas, ´a pode representar diferentes tipos. A partir da inferência sobre o tipo da função “=”, o compilador infere o tipo da função “compare” que é, portanto, polimórfica também. Ela não depende do tipo dos parâmetros x,p e q desde que eles tenham o mesmo tipo.

Veja que é possível definir funções polimórficas assumindo que os argumentos terão tipos não conhecidos e diferentes:

```
fun qq(a,b)=(a,b);
```

Neste caso, o compilador não tem nenhuma informação para inferir os tipos de a e b e a assinatura da função será:

```
> val ('a, 'b) qq = fn : 'a * 'b -> 'a * 'b
```

Ao contrário do operador de igualdade “=”, os operadores aritméticos e relacionais não são polimórficos, pois funcionam e operam de forma diferente para cada tipo. Assim, em ML estes operadores são definidos como um conjunto de funções pré-definidas mas de mesmo nome (sobrecarregadas). Assim, cada função é definida para um tipo distinto válido.

Não esqueça de salvar seus arquivos !
