

Modelos de linguagens de Programação

— Aula 05 —

Laboratório de
Programação Funcional

#2/3

Tópicos

- Revisão e complemento
- Formas funcionais
- Exercícios

Revisão

O que há de errado abaixo?

```
fun lstvezesx nil x = nil  
|   lstvezesx lst x = (hd(lst)*x)::(lstvezesx tl(lst) x);
```

```
fun lstvezesx nil x = nil  
|   lstvezesx lst x = (hd(lst)*x)::(lstvezesx (tl(lst)) x);
```

Por que?

- ❑ Lembrar que funções são elementos de 1ª ordem!
- ❑ A aplicação tem maior precedência do que a abstração
- ❑ A associação se dá da esquerda para a direita
- ❑ Problema de legibilidade: $f \ (x) == f(x)$

Revisão

Para a definição:

- ❑ `fun foo x = x + x;`

As aplicações abaixo são equivalentes:

- ❑ `foo 10;`

- ❑ `foo(10);`

- ❑ `foo (10);`

Revisão

Para as definições:

- ❑ `fun h f g x = f (g (x)) ;`
- ❑ `fun f x = x+x ;`
- ❑ `fun g x = x*2 ;`

As aplicações abaixo são válidas:

- ❑ `h f g 2 ;`
- ❑ `h f g (2) ;` (* cuidar para não ler/interpretar errado *)

Passa dois elementos de primeira ordem
e não um só como em linguagens imperativas!

Revisão

Definição de funções (duas formas):

- **fun** (funções nomeadas)

- `fun inc x = x + 1;`

- **fn** (funções não nomeadas)

- `fn x => x + 1;`

Complemento

- if ... then ... else ...

```
fun fatorial n = if n > 1 then n * fatorial (n-1)
                else 1;
```

Complemento

- Recursividade dupla:

```
fun fib 0 = 1 |  
    fib 1 = 1 |  
    fib n = fib(n-1) + fib(n-2);
```


Complemento

- Pattern matching (casamento de padrões)
 - tem a ver com a **sobrecarga** de funções
 - é feito para identificar a expressão mais adequada ao parâmetro passado
 - exemplo:

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


Complemento

■ Pattern matching (casamento de padrões)

□ exemplo:

```
fun past "run" = "ran"  
|   past "swim" = "swam"  
|   past x = x ^ "ed";
```

(Cumming, 1995)



Operador de
concatenação

Complemento

■ Pattern matching (casamento de padrões)

- Padrões sintáticos podem ser usados:

```
fun tamanhoLst nil = 0
|   tamanhoLst (h::t) = 1+(tamanhoLst t);
```

- Máscara (head não usado na função):

```
fun tamanhoLst nil = 0
|   tamanhoLst (_::t) = 1+(tamanhoLst t);
```

Complemento

■ Pattern matching (casamento de padrões)

□ exemplo com fn:

```
val dia = fn 1 => "Segunda"
          | 2 => "Terça"
          | 3 => "Quarta"
          | 4 => "Quinta"
          | 5 => "Sexta"
          | 6 => "Sábado"
          | 7 => "Domingo"
          | _ => "não existe";
```

Complemento

■ Polimorfismo

- É possível definir **funções polimórficas** assumindo que os **argumentos terão tipos não conhecidos**
- O tipo deles não precisa ser completamente especificado, mas o uso de objetos deve ser consistente

(* Compara 1º argumento com os demais, indicando se ele é igual ao primeiro, segundo, ambos ou nenhum *)

```
fun compare (a, b ,c) =  
  if a = b then if a = c then "todos iguais"  
                 else "somente 'a' e 'b'"  
  else if a = c then "somente 'a' e 'c'"  
        else "todos diferentes";
```

Complemento

■ Mais exemplos de manipulação de listas:

- `fun cabeca nil = nil`
 | `cabeca lst = hd(lst);`
- `fun rabo nil = nil`
 | `rabo lst = tail (lst);`
- `fun append nil lst = lst`
 | `append (h::t) lst = h::(append t lst);`

Complemento

■ Qual a diferença entre as funções seguintes?

1. `fun soma1 (x,y) = x + y;`
`> val soma1 = fn : int * int -> int`
2. `fun soma3 x y = x + y;`
`> val soma3 = fn : int -> int -> int`

Complemento

■ *Currying*

- *Schönfinkelisation*
- homenagem a Haskell Curry (técnica inventada por Moses Schönfinkel e Gottlob Frege)
- técnica de transformação de uma função que recebe múltiplos argumentos em uma que recebe um único argumento

“[...] ela nos permite visualizar a função como se ela pegasse no máximo um único parâmetro. Currying pode ser visto como uma maneira de gerar funções intermediárias que aceitam parâmetros adicionais para completar um cálculo” (Normak, 2007)

Complemento

■ *Currying*


- no cálculo lambda as funções só podem receber um único argumento e retornar um único valor
 - funções que recebem diversos argumentos podem então ser simuladas com *currying*
- usado em funções de ordem superior para que os parâmetros sejam consumidos tardiamente (onde fornecer um argumento resulta em avaliação parcial da função)

Complemento

■ *Currying*

- ❑ em ML, funções também só trabalham com um único argumento
- ❑ perceba que funções que não realizam *currying* aceitam diversos parâmetros utilizando a técnica de **passagem de objetos compostos**
- ❑ exemplo:

```
- fun add (x, y) = x + y;  
> val add = fn : int x int -> int
```



tupla (objeto composto) que contém dois argumentos

Complemento

■ *Currying*

□ vantagens reais:

- permite definir versões especializadas de uma função existente (funções onde somente um ou mais, mas não todos, argumentos são informados)
- permite fornecer uma função que é parâmetro de outra

□ exemplo:

- `fun sum x y = x + y; (* fun sum x y = fn y => x + y; *)`
- `sum 1;`
- `it 2;`
- `val inc = sum 1;`
- `val add2 = sum 2;`

Complemento

- Escopo usual: uma sessão
- Delimitação de escopo (associações locais):

- Sintaxe:

```
let <decl> in <expr> end;
```

- Exemplo:

- val m = 5.6;

- val m = 5.6 : real

- let val m = 3 in
 m * 2

- end;

- > val it = 6 : int

- m;

- > val it = 5.6 : real

Complemento

- Delimitação de escopo (funções locais):
 - Muito utilizado para definir funções auxiliares!

- Sintaxe:

`local <decl> in <expr> end;`

- Exemplo:

```
local
  fun helper (0,r:int) = r |
    helper (n:int,r:int) = helper (n-1,n*r)
in
  fun factorial (n:int) = helper (n,1)
end;
```

Tópicos

- Revisão e complemento
- Formas funcionais
- Exercícios

Formas funcionais

■ Funções de ordem superior

□ dado que:

■ $f(x) \equiv x + 2$

■ $g(x) \equiv x * x$

■ $h(x) \equiv 3 * x$

□ Composição: $h \equiv f \bullet g \quad \rightarrow \quad h(x) \equiv f(g(x))$

□ Construção: $[f, g, h](4) \quad \rightarrow \quad (6, 8, 12)$

□ *Apply-to-all*: $\alpha(f, (2, 3, 4)) \rightarrow (4, 5, 6)$

Formas funcionais

■ Dadas as seguintes funções:

```
fun f(x) = x * x;  
fun g(x) = x + 3;
```

■ Composição:

```
fun compor f1 f2 = f1 o f2;  
compor f g 5;      (* resulta em f(g(5)) => 64 *)  
compor g g 5;  
compor g f 5;
```

OBS: a avaliação é tardia (substituição feita no momento do uso)!
Mude a função f para 'f=x+2;' e teste: '**compor f g 5;**' Continua igual?

■ Apply to all:

```
map f [2, 3, 6];      (* resulta em [4, 6, 36] *)
```


Formas funcionais

- Dadas as seguintes funções:

```
fun f(x) = x * x;
```

```
fun g(x) = x + 3;
```

- Como fazer construção?

```
contruir [f, g] 2 = [4, 5];
```

- Exercício:

Elabore uma função que simule o operador de construção.



Bibliografia

- Normak, Kurt. (2007). *Currying*. In: **Notas de aula da disciplina de programação funcional** (Functional Programming in Scheme: With Web Programming Examples). Department of Computer Science, Aalborg University, Denmark. Acesso em: Julho, 2007. Disponível em: http://www.cs.auc.dk/~normark/prog3-03/html/notes/higher-order-fu_themes-curry-section.html
- **Currying** (Wikipedia). <http://en.wikipedia.org/wiki/Currying>
- Paulson, Lawrence C. **ML for the working programmer**. Cambridge: Cambridge University Press, c1991. 429 p.
- Harper, Robert. **Programming in standard ML** (class notes). <http://www.cs.cmu.edu/~rwh/introsm1/>
- Cumming, A. **A gentle introduction to ML**. 1995. <http://ftp.utcluj.ro/pub/docs/diverse/ml/gentle-intro-ML/>