



Laboratório #3

Linguagens Funcionais 3/3

Prof. Leandro Krug Wives



REVISÃO

Estruturas e conceitos do paradigma funcional com LP SML



Revisão: declaração de funções



- `fun soma (a,b) = a+b; (* recebe 1 tupla *)`
- `fun soma a b = a + b; (* curried: recebe um parametro por vez *)`
 - `val r = soma 10;`
 - `r 5;`
- `val soma = fn a=>fn b=>a+b; (* curried unnamed function *)`
- `infix mais; (* indica q função deve ser utilizada no meio dos operandos *)`
`fun x mais y = x+y;`



Revisão: *high order functions*



■ Função map

```
■ fun map f [] = []  
  |   map f (h::t) = (f h)::(map f t);
```

■ Função duplica

```
■ fun duplica lst = map (fn x => x*2) lst;
```

■ Somando elementos de uma lista de tuplas...

```
■ map (fn (x,y) => x+y) [(1,2), (3,4)];
```



Revisão: *high order functions*



■ foldl e foldr

- `foldl op :: [] [1,2,3,4,5];`
- `foldl (fn (x,y) => if x mod 2=0 then x+y else y) 0 [1,2,3,4];`

■ Exercício:

- Elabore uma função que some elementos de uma lista.
- Elabore uma função que receba uma lista de tuplas (`int * int`). Ela deve retornar a soma da multiplicação dos elementos da tupla.
E.g., `[(1,2),(3,4)] → (1*2)+(3*4) = 14`

+ Revisão: polimorfismo



- `fun identidade e = e; (* abstrata, polimórfica *)`

- `identidade 1;`

- `identidade "1";`

- `identidade [1,2,3];`

- `identidade (fn x => x*2);`

- `fun inverta lst = foldr op:: [] lst;`

- `val 'a inverta = fn 'a list -> 'a list`

→ “ ‘a ’ é dita uma “variável de tipo”



List comprehensions (Haskell)



- Definem listas utilizando expressões matemáticas:
 - `[x*2 | x <- [1..10]]`
 - `[x*y | x <- [2,5,10], y <- [8,10,11]]`
- Funcionam como filtros:
 - `[x*2 | x <- [1..10], x>5]`
 - `[x*2 | x <- [1..10], x*2 > 10]`
- Um exemplo interessante (retorna o tamanho da lista):
 - `Length' lst= sum [1 | _ <- lst]`



Revisão: conceitos importantes



- Funções são a base de Programação funcional (elas são elementos de primeira ordem)
- Funções de ordem maior manipulam outras funções (e.g., map, o, foldl)
- Funções com mais de 1 parâmetro são *curried*
- Recursão é a base das iterações, repetições
- A passagem dos parâmetros é por valor (em funções puras não é possível modificar os argumentos)
- Casamento de padrões e a construção de funções deve usado ao invés de construções imperativas e testes condicionais.
- Associações seguem o conceito de transparência referencial (e.g., val inc = soma 2; inc 3;)



DEFINIÇÃO DE TIPOS

Estruturas e conceitos do paradigma funcional com LP SML

Tipos definidos pelo usuário

- Definição de **tipos personalizados** e operações que funcionem sobre eles
- Há as seguintes opções em ML:

- type
- datatype
 - com constantes construtoras
 - com funções construtoras

- struct e signature
- abstype

Nosso foco

Tipos definidos pelo usuário

- Uma **declaração de tipos** em ML é usada para **permitir a definição de funções que recebem e retornam valores especiais** (e.g., tipos compostos)
- Uso do comando type:

```
type ncomplexo = real * real; (* define o tipo *)
```

Tipos definidos pelo usuário

- Dado que:

```
type ncomplexo = real * real; (* define o tipo *)
```

- Podemos então usar ncomplexo para definir funções:

```
fun soma ((xre, xim):ncomplexo, (yre, yim):ncomplexo)  
    = (xre+yre, xim+yim);
```

- Qual a resposta do interpretador para:

```
soma ((1.0, 2.0), (2.0, 2.0));
```

Mesmo especificando o tipo de retorno,
a resposta é a mesma:
real * real

Tipos definidos pelo usuário

- Podemos criar tipos mais complexos (e.g., registros):

```
type rei = { nome: string, nascimento: int,  
            coroacao: int, morte: int };  
fun tmpvida1 (k:rei) = #morte k - #nascimento k;  
fun tmpvida2 ({nascimento, morte,...}:rei) = morte -  
nascimento;
```

Por que o seguinte não funciona???

```
fun lifetime({nascimento, coroacao,...}:rei )  
    = morte - nascimento;
```

Tipos definidos pelo usuário

- Type é semelhante a typedef da linguagem C
- **Type não especifica um construtor do tipo** (não indica ao processador da linguagem como construir um dado daquele tipo)
- **Permite a verificação de tipos na definição de uma função** (no exemplo do “rei”, foi preciso nomear cada campo)
- Uma alternativa consiste em definir um novo tipo e **indicar, além de sua estrutura, um conjunto de funções associadas** (incluindo a função de construção)

→ datatype

Datatype

- Duas possibilidades:
 - com constantes construtoras
 - com funções construtoras

Datatype com constantes

- Equivale a uma enumeração em outras linguagens
- Define construtores de tipos
- Exemplo de uso de *datatype* para definir um **tipo simples** que possui 3 estados: verdadeiro, falso e indefinido:
 - **datatype** LOGIC03 = True | False | Undef;

Datatype com constantes

- datatype LOGIC03 = True | False | Undef;

> New type names := LOGIC03

Novo tipo definido!

Resposta do
interpretador {
datatype LOGIC03 = (LOGIC03, {con False : LOGIC03,
con True : LOGIC03, con Undef : LOGIC03})
con False = False : LOGIC03
con True = True : LOGIC03
con Undef = Undef : LOGIC03

'con' vem de "constructor" – o construtor do tipo!

Datatype com constantes

- Na definição anterior:

```
- datatype LOGIC3 = True | False | Undef;
```

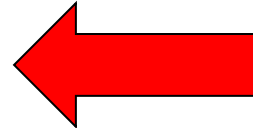
- Os três valores enumerados definem os construtores possíveis para objetos desse novo tipo, ou seja, as formas de criação de um objeto LOGIC3 (no caso, os construtores são constantes)
- O tipo pode assumir, portanto, 3 estados: True (verdadeiro), False (falso) e Undef (indefinido)
- Perceba que True é diferente de true (que já existe em ML)!

Datatype com constantes

- Para criar associações deste tipo, basta seguir os exemplos:

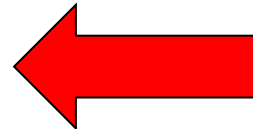
- `val a = True;`

- > `val a = True : LOGICO3`



- `val b = Undef;`

- > `val b = Undef: LOGICO3`



- Perceba que ML **infere o tipo correto** (**LOGICO3**)!

Datatype com constantes

- Ao definir um novo tipo, devemos especificar operações válidas para ele:

- `fun not3 True = False`
 - | `not3 False = True`
 - | `not3 Undef = Undef;`
- `fun and3 (True,True) = True`
 - | `and3 (False,_) = False`
 - | `and3 (_,False) = False`
 - | `and3 (_,_) = Undef;`

O símbolo `_` (sublinhado) é uma máscara (*wild_card* / coringa) que indica “qualquer coisa nesta posição”

OBS: coloque a máscara menos restritiva por último, pois a avaliação se dá pela ordem

Datatype com constantes

■ Com base no que já definimos:

```
- datatype LOGICO3 = True | False | Undef;  
- fun not3 True = False  
  | not3 False = True  
  | not3 Undef = Undef;  
- fun and3 (True,True) = True  
  | and3 (False,_) = False  
  | and3 (_,False) = False  
  | and3 (_,_) = Undef;
```

■ Teste:

```
- val a = True;  
- val b = Undef;  
- and3 (not3 (a), b);
```

■ Crie o operador or3:

```
- or3 (a, c);
```

Datatype com funções

- Quando o domínio é mais complexo e **não depende de constantes, precisamos especificar sua estrutura**
- **Tipos complexos estruturados usam funções "construtoras"** que criam um objeto de determinado tipo levando em conta os valores passados como argumentos

- Exemplo:

Datatype RETANGULAR = Retangular **of** real*real;

Datatype POLAR = Polar **of** real*real;

Nome da função

A palavra **of** indica uma função construtora de estrutura (parâmetros) conforme o especificado

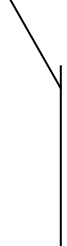
Datatype com funções

- A função construtora é especial, pois avalia os operandos e cria um objeto do tipo para o qual foi especificada:

datatype RETANGULO = Retangulo of real*real*real*real;



Função construtora que recebe uma tupla de 4 números reais e retorna um objeto do tipo RETANGULO



- Perceba a diferença entre o construtor e o tipo!
 - Construtor **só com a primeira letra maiúscula**
 - Nome do tipo TODO EM MAIÚSCULAS

Datatype com funções

- Levando em conta os datatypes definidos e a função topolar:

```
datatype RECT = Rect of real*real;  
datatype POLAR = Polar of real*real;  
fun toPolar(Rect(x,y))  
    = Polar(Math.sqrt(x*x+y*y), Math.atan(y/x));
```

- Avalie o resultado das seguintes expressões:

- val a = Rect(2.3, 7.04);
- Polar(1.2, 5.6);
- a = (2.3, 7.04);
- a = Rect(2.3, 7.04);
- val Rect(a,b) = Rect(2.3, 7.04);
- toPolar(Rect(3.2,4.5));

Definição de tipos: resumo

■ Type

- Define um nome específico, agregando semântica
- Melhora a escrita e a leitura do código
- Exemplos:
 - `type ncomplexo = real * real;`
 - `type rei = { nome: string, nascimento: int,
 coroacao: int, morte: int };`

■ Datatype

- Define constantes ou funções que definem ou constroem elementos de um determinado tipo (simples ou estruturado)
- Exemplos:
 - Simples: `datatype LOGICO3 = True | False | Undef;`
 - Estruturado: `datatype PONTO = Ponto of real*real;`