Linguagem de Programação
Introdução à Linguagem Hugs98

TURMA A
PROF. MARCELO LADEIRA CIC/UNB

Paradigma Funcional

Uma função é definida como um conjunto de equações.

Cada equação é uma regra de reescrita (redução).

Exemplo:

```
filtro p (x:xs)
                | p x       = x:filtro p xs
                | otherwise = filtro p xs
filtro _ [] = []
Main> filtro even [1..50]
  [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,4
  4,46,48,50] :: [Int]
```

Casamento de Padrões

É o processo de avaliar um ou mais dos argumentos para
determinar qual a expressão do lado direito que se aplica.

É o processo de identificar a regra de reescrita que se aplica.

Reescrita (redução)
substituir a expressão corrente pela expressão do lado direito,
conforme os padrões casados com a instância de argumentos.

Guardas

Cada uma das equações na definição pode conter guardas.

Uma guarda é uma função predicativa.

Se a guarda for verdadeira, a expressão correspondente na função é executada.

Exemplo:

```
fat n | n == 0  || n==1  = 1
      | n > 1             = n* fat (n-1)
```

```
Main> fat 5
  120 :: Integer
```

Recursão

É a única estrutura de controle entre comandos em uma linguagem funcional pura.

O controle de seqüência intra-comando, em expressões, é definido pelas prioridades e associatividades das funções e operadores.

É o processo da função chamar a si mesma, direta ou indiretamente.

Exemplo:

```
fat 0 = 1
fat 1 = 1
fat n = n * fat (n-1)          -- e se n < 0 ?
```

Polimorfismo

Tipos polimórficos descrevem famílias de tipos:

[a]	família de listas para diferentes instâncias de tipos:	
a = Int,	lista de inteiro	[Int]
a = Char,	lista de caracter	[Char]
a = [Float],	lista de lista de float	[[Float]]

a - é uma variável de tipo

Nesse sentido a é mais geral que as instâncias de tipos Int, Char, [Float], etc.

Polimorfismo

Funções Polimórficas

Uma função que se aplica a qualquer tipo de parâmetro é uma função polimórfica.

Exemplos.:

```
size (x:xs) = 1+size xs
size [] = 0
Main> size [1,3,15]
3 :: Integer
Main> size ['a'..'z']
26 :: Integer
```

Polimorfismo

```
somamenor m xs
    | length xs <= 1 = []
    | z < m        = (x,y,z):somamenor m (y:t)
    | otherwise  = somamenor m (y:t) where (x:y:t) = xs; z = x+y
```

```
Main> somamenor 10 [1,2,3,4,5,6,7,8,9]
    [(1,2,3),(2,3,5),(3,4,7),(4,5,9)]
Main> somamenor 10 [1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5]
    [(1.5,2.5,4.0),(2.5,3.5,6.0),(3.5,4.5,8.0)]
```

Avaliação Preguiçosa

Uma expressão só é avaliada quando seu valor é requerido.

Uma expressão cotizada, aparecendo em vários lugares, é avaliada uma única vez

Exemplos:

```
zero x = 0
Main> zero 10
  0 :: Integer
Main> zero (1/0)
  0 :: Integer
Main> zero $! (1/0)  -- Força avaliação argumento mas não funcionou!
  Program execution error: {primDivDouble 1.0 0.0}
```

Avaliação Preguiçosa: qd x =x*x

Passagem de parâmetro por referência:

qd(qd (qd 2)) = (qd (qd 2)) * (qd (qd 2))
= (qd 2) * (qd 2) * (qd (qd 2))
= 2 * 2 * (qd 2) * (qd (qd 2))
= 4 * (qd 2) * (qd (qd 2))
= 4 * 4 * (qd (qd 2))
= 16 * (qd (qd 2))
= 16 * 16 = 256

Passagem de parâmetro por valor:

qd(qd(qd 2)) = qd(qd (2*2)) = qd(qd(4))=qd(4*4)
= qd(16) = 16*16

Ambas apresentam esforço próximo a 3 multiplicações

Objetos Infinitos

Graças a avaliação preguiçosa é possível lidar com lista infinitas.

Exemplos:

```
Main> let naturais n = take n [0..] in naturais 8
[0,1,2,3,4,5,6,7] :: [Integer]

Main> let impar n = take n [1,3..] in impar 10
[1,3,5,7,9,11,13,15,17,19] :: [Integer]
```

Avaliação Preguiçosa e Objetos Infinitos

```
fiblst =1:1:[x+y|(x,y) <- (zip fiblst (tail fiblst))]
```

```
Main> take 10 fiblst
    [1,1,2,3,5,8,13,21,34,55] :: [Integer]
```

		fiblst anterior	tail fiblst anterior	
1	[1]	[]	não utilizada	
2	[1,1]	[1]	a partir daqui calcule (x,y)	
3	[1,1,2]	[1,1]	[1]	(1,1)
4	[1,1,2,3]	[1,1,2]	[1,2]	(1,1), (1,2)
5	[1,1,2,3,5]	[1,1,2,3]	[1,2,3]	(1,1), (1,2), (2,3)
6	[1,1,2,3,5,8]	[1,1,2,3,5]	[1,2,3,5]	(1,1), (1,2), (2,3), (3,5)

Avaliação Preguiçosa e Objetos Infinitos

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
Main> take 10 primes
    [2,3,5,7,11,13,17,19,23,29] :: [Integer]
```