Programação Funcional
Aula 8 — Listas infinitas

Pedro Vasconcelos
DCC/FCUP

2021

Listas infinitas

Podemos usar listas para *sequências finitas*, por ex.:

```
[1,2,3,4] = 1:2:3:4:[]
```

Nesta aula vamos ver que podemos também usar listas para representar *sequências infinitas*, e.g.

```
[1..] =  1:2:3:4:5:...
```

Não podemos descrever uma lista infinita em extensão; usamos listas em compreensão ou definições recursivas.

Exemplos

```haskell
-- todos os números naturais
nats :: [Integer]
nats = [0..]

-- todos os números pares não-negativos
pares :: [Integer]
pares = [0,2..]

-- a lista infinita 1, 1, 1,...
uns :: [Integer]
uns = 1 : uns

-- todos os inteiros a partir de um número
intsFrom :: Integer -> [Integer]
intsFrom n = n : intsFrom (n+1)
```

Processamento de listas infinitas

Por causa da *lazy evaluation* as listas são calculadas à medida da necessidade e apenas até onde for necessário.

```
  head uns
=
  head (1:uns)
=
  1
```

Processamento de listas infinitas (cont.)

Uma computação que necessite de percorrer toda a lista infinita não termina.

```
  length uns
=
  length (1:uns)
=
  1 + length uns
=
  1 + length (1:uns)
=
  1 + (1 + length uns)
=
  ⋮
```

não termina

Produzir listas infinitas

Muitas funções do prelúdio-padrão produzem listas infinitas quando os argumentos são listas infinitas:

```
> map (2*) [1..]
[2, 4, 6, 8, 10, ...

> filter (\x->x`mod`2/=0) [1..]
[1, 3, 5, 7, 9, ...
```

Também podemos usar notação em compreensão:

```
> [2*x | x<-[1..]]
[2, 4, 6, 8, 10 ...

> [x | x<-[1..], x`mod`2/=0]
[1, 3, 5, 7, 9 ...
```

Produzir listas infinitas (cont.)

Algumas funções do prelúdio-padrão produzem especificamente listas infinitas:

```
repeat :: a -> [a]
-- repeat x = x:x:x:...

cycle :: [a] -> [a]
-- cycle xs = xs++xs++xs++...

iterate :: (a -> a) -> a -> [a]
-- iterate f x = x: f x: f(f x): f(f(f x)): ...
```

(Note que *iterate* é de ordem-superior porque o primeiro argumento é uma função.)

Produzir listas infinitas (cont.)

Podemos testar no interpretador pedido prefixos finitos:

```
> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]

> take 10 (repeat ’a’)
"aaaaaaaaaa"

> take 10 (cycle [1,-1])
[1,-1,1,-1,1,1,-1,1,-1,1]

> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512]
```

Produzir listas infinitas (cont.)

As funções *repeat*, *cycle* e *iterate* estão definidas no prelúdio-padrão usando recursão:

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs

cycle :: [a] -> [a]
cycle [] = error "empty list"
cycle xs = xs' where xs' = xs++xs'

iterate :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Porquê usar listas infinitas?

- Permite simplificar o processamento de listas finitas combinando-as com listas infinitas
- Permite separar a geração e o consumo de sequências
- Permite maior modularidade na decomposição dos programas

Exemplo 1: Preenchimento de texto

Escrever uma função

```
preencher :: Int -> String -> String
```

que preenche uma cadeia com espaços de forma a perfazer *n* caracteres.

Se a cadeia já tiver comprimento *n* ou maior, deve ser truncada a *n* caracteres.

Exemplo 1: Preenchimento de texto (cont.)

Exemplos

```
> preencher 10 "Haskell"
"Haskell   "

> preencher 10 "Haskell B. Curry"
"Haskell B."
```

Exemplo 1: Preenchimento de texto (cont.)

- Uma solução que calcula e acrescenta o número correto de espaços testando uma condição:

```
preencher n xs
    | k < n     = xs ++ replicate (n-k) ' '
    | otherwise = take n xs
    where k = length xs
```

- Mas há uma solução mais simples usando *take* e uma lista infinita:

```
preencher n xs = take n (xs++repeat ' ')
```

Exemplo 2: Aproximação da raiz quadrada

Calcular uma aproximação de $\sqrt{q}$ pelo *método babilónico*:

1. Começamos com $x_0 = q$
2. Em cada passo, melhoramos a aproximação tomando

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{q}{x_n}\right)$$

3. Critérios de paragem:

número de iterações parar ao fim de um certo número de iterações

erro absoluto parar quando a distância entre aproximações é inferior a ϵ:

$$|x_{n+1} - x_n| < \epsilon$$

Exemplo 2: Aproximação da raiz quadrada (cont.)

```
-- sucessão infinita de aproximações à raiz quadrada
aproximações :: Double -> [Double]
aproximações q = iterate (\x->0.5*(x+q/x)) q

-- critério de paragem por erro absoluto
erroAbsoluto :: [Double] -> Double -> Double
erroAbsoluto xs eps
  = head [x' | (x,x')<-zip xs (tail xs), abs(x-x')<eps]
```

Exemplo 2: Aproximação da raiz quadrada (cont.)

Exemplos para calcular $\sqrt{2}$

```
> aproximações 2.0
[2.0,1.5,1.4166667,1.4142157, 1.4142135, 1.4142135, ...

> aproximações 2.0 !! 5
1.4142135

> (aproximações 2.0) `erroAbsoluto` 0.01
1.4166667

> (aproximações 2.0) `erroAbsoluto` 0.001
1.4142135
```

Exemplo 3: A sucessão de Fibonacci

A sucessão de Fibonacci:

- começa com 0, 1;
- cada valor seguinte é a *soma dos dois anteriores*.

$$0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : \ldots : a : b : a{+}b : \ldots$$

Exemplo 3: A sucessão de Fibonacci (cont.)

Solução em Haskell: uma lista infinita definida recursivamente.

```
fibs :: [Integer]
fibs = 0 : 1 : [a+b | (a,b)<-zip fibs (tail fibs)]
```

Alternativa usando *zipWith* em vez de lista em compreensão (ver folha de exercícios):

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Exemplo 3: A sucessão de Fibonacci (cont.)

Os primeiros dez números de Fibonacci:

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

O nono número Fibonacci (índices começam em zero):

```
> fibs!!8
21
```

O primeiro Fibonacci superior a 100:

```
> head (dropWhile (<=100) fibs)
144
```

Exemplo 4: O crivo de Eratóstenes

Gerar *todos* os números primos usando o *crivo de Eratóstenes*.

1. Começar com a lista $[2, 3, 4, \ldots]$;
2. Marcar o primeiro número p na lista como primo;
3. Remover da lista p e todos os seus múltiplos;
4. Repetir o passo 2.

Observar que o passo 3 envolve processar uma lista infinita.

Exemplo 4: O crivo de Eratóstenes (cont.)

Em Haskell

```
primos :: [Integer]
primos = crivo [2..]

crivo :: [Integer] -> [Integer]
crivo (p:xs) = p : crivo [x | x<-xs, x`mod`p/=0]
```

Exemplo 4: O crivo de Eratóstenes (cont.)

Os primeiros 10 primos:

```
> take 10 primos
[2,3,5,7,11,13,17,19,23,29]
```

Quantos primos são inferiores a 100?

```
> length (takeWhile (<100) primos)
25
```