

Programação Funcional

5ª Aula — Definições recursivas

Sandra Alves
DCC/FCUP

2019/20

Definições usando outras funções

Podemos definir funções usando outras previamente definidas (por exemplo: do prelúdio-padrão).

Exemplo:

```
factorial :: Int -> Int
factorial n = product [1..n]
```

Definições recursivas I

Também podemos definir uma função por **recorrência**, i.e. usando a própria função que estamos a definir; tais definições dizem-se **recursivas**.



Definições recursivas II

Exemplo: factorial definido recursivamente.

```
factorial :: Int -> Int
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n-1)
```

Exemplo de uma redução

factorial 3

=

3 * factorial 2

=

3 * (2 * factorial 1)

=

3 * (2 * (1 * factorial 0))

=

3 * (2 * (1 * 1))

=

6

Observações

- A primeira equação define o factorial de zero.
- A segunda equação define o factorial de n usando factorial de $n - 1$.
- Logo: o factorial fica definido apenas para inteiros não-negativos.

```
> factorial (-1)
```

Não termina!

```
^C
```

```
Interrupted
```

Alternativas

Duas equações sem guardas:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Uma equação com guardas:

```
factorial n | n==0      = 1
            | otherwise = n*factorial (n-1)
```

Uma equação com uma condição:

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

Porquê recursão?

- **Modelo universal de computação**: qualquer algoritmo pode ser escrito usando funções recursivas.
- Vamos ver que podemos **demonstrar propriedades** de funções recursivas usando indução matemática.

Recursão sobre listas

Também podemos definir funções recursivas sobre listas.

Exemplo: a função que calcula o produto de uma lista de números (do prelúdio-padrão).

```
product []      = 1  
product (x:xs) = x*product xs
```

Exemplo de redução

```
product [2,3,4]
=
2 * product [3,4]
=
2 * (3 * product [4])
=
2 * (3 * (4 * product []))
=
2 * (3 * (4 * 1))
=
24
```

A função *length* I

O comprimento duma lista também pode ser definido por recursão.

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

A função *length* II

Exemplo de redução:

$$\begin{aligned} & \text{length } [1,2,3] \\ = & \\ & 1 + \text{length } [2,3] \\ = & \\ & 1 + (1 + \text{length } [3]) \\ = & \\ & 1 + (1 + (1 + \text{length } [])) \\ = & \\ & 1 + (1 + (1 + 0)) \\ = & \\ & 3 \end{aligned}$$

A função *reverse* I

A função *reverse* (que inverte a ordem dos elementos numa lista) também pode ser definida recursivamente.

```
reverse :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

A função *reverse* II

Exemplo de redução:

```
reverse [1,2,3]
=
reverse [2,3] ++ [1]
=
(reverse [3] ++ [2]) ++ [1]
=
((reverse [] ++ [3]) ++ [2]) ++ [1]
=
(([] ++ [3]) ++ [2]) ++ [1]
=
[3,2,1]
```

Funções com múltiplos argumentos I

Também podemos definir recursivamente funções com múltiplos argumentos.

Por exemplo: a concatenação de listas.

```
(++) :: [a] -> [a] -> [a]  
[]      ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

Funções com múltiplos argumentos II

A função *zip* que constroi a lista dos pares de elementos de duas listas.

```
zip :: [a] -> [b] -> [(a,b)]  
zip []      _      = []  
zip _      []      = []  
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```


Funções com múltiplos argumentos III

A função *drop* que remove um prefixo de uma lista.

```
drop :: Int -> [a] -> [a]
drop 0 xs          = xs
drop n []          = []
drop n (x:xs) | n>0 = drop (n-1) xs
```

A implementação seguinte segue a versão do prelúdio que permite inteiros negativos como primeiro argumento:

```
drop :: Int -> [a] -> [a]
drop n [] = []
drop n (x:xs) | n > 0 = drop (n-1) xs
               | otherwise = (x:xs)
```

Recursão mútua

Podemos também definir duas ou mais funções que dependem mutuamente umas das outras.

Exemplo: testar se um natural é par ou ímpar.¹

```
par :: Int -> Bool
par 0      = True
par n | n>0 = impar (n-1)
```

```
impar :: Int -> Bool
impar 0      = False
impar n | n>0 = par (n-1)
```

¹De forma ineficiente.

Quicksort I

O algoritmo *Quicksort* para ordenação de uma lista pode ser especificado de forma recursiva:

se a lista é vazia então já está ordenada;

se a lista não é vazia seja x o primeiro valor e xs os restantes:

1. recursivamente ordenamos os valores de xs que são **menores ou iguais** a x ;
2. recursivamente ordenamos os valores de xs que são **maiores** do que x ;
3. concatenamos os resultados com x no meio.

Quicksort II

Em Haskell:

```
qsort :: [Int] -> [Int]
qsort []      = []
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
    where menores = [x' | x'<-xs, x'<=x]
          maiores = [x' | x'<-xs, x'>x]
```

Esta é provavelmente a implementação mais concisa do algoritmo *Quicksort* em *qualquer* linguagem de programação!

Quicksort III

Exemplo de execução (abreviando qsort para qs):

```
qs [3,2,4,1,5]
=
qs [2,1] ++ [3] ++ qs [4,5]
=
(qs [1]++[2]++qs []) ++ [3] ++ (qs []++[4]++qs [5])
=
([1]++[2]++) ++ [3] ++ ([4]++[5])
=
[1,2,3,4,5]
```

Como escrever definições recursivas

1. Definir o tipo da função
2. Enumerar os casos a considerar usando equações com padrões
3. Definir o valor nos casos simples
4. Definir o valor nos outros casos assumindo que a função está definida para valores de tamanho inferior
5. Generalizar e simplificar

Exemplo I

Escrever uma definição recursiva da função *init* que remove o último elemento duma lista.

```
> init [1,2,3,4,5]  
[1,2,3,4]
```

```
> init [1]  
[]
```

```
> init []
```

```
*** Exception: Prelude.init: empty list
```

Exemplo II

Passo 1: o tipo da função é

```
init :: [a] -> [a]
```


Exemplo III

Passo 2: enumerar os casos.

```
init :: [a] -> [a]  
init (x:xs) =
```

Notar que *init* não está definido para a lista vazia.

Exemplo IV

Passo 3: definir o caso simples.

```
init :: [a] -> [a]
init (x:xs) | null xs    = []
             | otherwise =
```

Notar que se xs é a lista vazia então a lista $(x : xs)$ tem um só elemento.

Exemplo V

Passo 4: definir o caso recursivo.

```
init :: [a] -> [a]
init (x:xs) | null xs    = []
              | otherwise = x : init xs
```

Notar que xs é uma sub-lista de $(x : xs)$, logo tem comprimento menor.

Exemplo VI

Passo 5: simplificação.

```
init :: [a] -> [a]
init [x]      = []
init (x:xs) = x : init xs
```

Podemos separar o caso da lista com um só elemento numa equação e assim eliminar as guardas.

Exercícios:

- Usando o hugs experimente as funções sum, maximum, minimum, notElem, take, unzip, zip3, unzip3.
- Defina as funções indicadas na alínea anterior.