Programação Funcional
Aula 6 — Definições recursivas

Pedro Vasconcelos
DCC/FCUP

2021

Definições usando outras funções

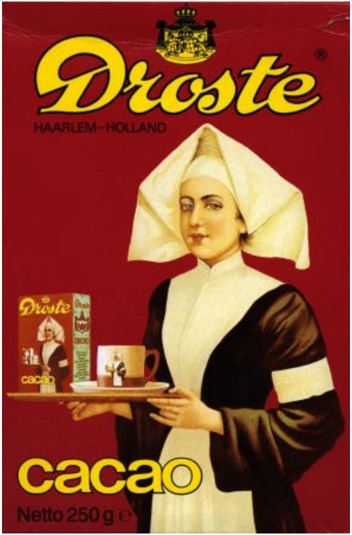
Podemos definir funções usando outras previamente definidas (por exemplo: do prelúdio-padrão).

Exemplo:

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

Definições recursivas

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



Definições recursivas (cont.)

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 está definido para inteiros não-negativos

  `factorial (-1)` *Não termina!*

- A ordem das equações é importante:

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

  A segunda equação nunca é usada, logo esta versão não termina para nenhum inteiro!

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?

- Não podemos usar ciclos numa linguagem puramente funcional porque não pudemos modificar variáveis
- A única forma funcional de exprimir repetição é usar recursão
- Mas qualquer algoritmo que pode escrito com ciclos também pode ser escrito com funções recursivas
- Mais tarde veremos que podemos demonstrar propriedades de programas recursivos 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*

O comprimento duma lista também pode ser definido por recursão.

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

A função *length* (cont.)

Exemplo de redução:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

A função *reverse*

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* (cont.)

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

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 (cont.)

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 (cont.)

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
```

Recursão mútua

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

Exemplo: testar se um natural é par ou impar.[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)
```

[1] De forma ineficiente.

Quicksort

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 (cont.)

Em Haskell:

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

Provavelmente a implementação mais concisa do algoritmo *Quicksort* em *qualquer* linguagem de programação!

Quicksort (cont.)

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]
```

Relação com compreensões

- Qualquer definição em compreensão também pode ser traduzida para funções recursivas
- O contrário nem sempre é verdade: as definições recursivas são mais gerais do que definições com listas em compreensão

Relação com compreensões (cont.)

Exemplo 1: listar todos os quadrados de 1 até *n*.

```
-- versão com lista em compreensão
listarQuadrados n = [i^2 | i<-[1..n]]

-- versão recursiva
listarQuadrados' n = quadrados 1
   where
     quadrados i
       | i<=n      = i^2 : quadrados (i+1)
       | otherwise = []
```

Relação com compreensões (cont.)

Ao transformar a definição em compreensão numa recursão podemos por vezes eliminar a lista.

Exemplo 2: somar todos os quadrados de 1 até *n*.

```
-- versão com lista em compreensão
somarQuadrados n = sum [i^2 | i<-[1..n]]

-- versão recursiva sem listas
somarQuadrados' n = quadrados 1
   where
     quadrados i
       | i<=n      = i^2 + quadrados (i+1)
       | otherwise = 0
```