Programação Funcional

Vander Alves

O que é Programação Funcional?

- Programação com alto nível de abstração
- Soluções elegantes, concisas e poderosas
- Funções: computam um resultado que depende apenas dos valores das entradas
- Forte fundamentação teórica: mais facilmente provas de propriedades sobre os programas

Por que um curso de programação funcional?

- Dá uma visão clara de conceitos fundamentais da computação moderna:
  - abstração (em uma função)
  - abstração de dados (tipos abstratos de dados)
  - genericidade, polimorfismo, overloading

Conceitos importantes

- Sistema de tipos fortes
- Uso extensivo de recursão
- Polimorfismo e funções de alta ordem
- Tipos de dados algébricos

Objetivos da programação funcional

- Programação com um alto nível de abstração, possibilitando:
  - alta produtividade
  - programas mais concisos
  - programas mais fáceis de entender
  - menos erros
  - provas de propriedades sobre programas

Uso prático

- Programas com milhares de linhas de código: compiladores, provadores de teoremas etc.
- Ericsson utiliza Erlang, uma linguagem funcional concorrente, para programação de switches de redes/telecomunicações, com excelentes resultados.
- Possibilidade de integração com partes de programas escritos em outras linguagens.

Ambiente a ser utilizado no curso

- Linguagem de programação: Haskell
- Plataforma Haskell
  - Interpretador (GHCi)
  - Compilador (GHC)
- Disponível em
  `https://www.haskell.org/platform/`

Notação: Programação baseada em definições

```
answer :: Int
answer = 42

greater :: Bool
greater = (answer > 71)

yes :: Bool
yes = True
```

Definição de Funções

```
square :: Int -> Int
square x = x * x

allEqual :: Int -> Int -> Int -> Bool
allEqual n m p = (n == m) && (m == p)

maxi :: Int -> Int -> Int
maxi n m | n >= m    = n
         | otherwise = m
```

Avaliando Expressões

- Encontrar o valor de uma expressão
- Usar definições das funções

```
addD :: Int -> Int -> Int
addD a b = 2 * (a+b)

addD 2 (addD 3 4)
= 2 * (2 + (addD 3 4))
= 2 * (2 + 2 * (3 + 4))
= 32
```

Prova de propriedades

- Exemplo:

```
addD a b = 2 * (a+b)
         = 2 * (b+a) = addD b a
```

- Válida para quaisquer argumentos a e b
- Não seria válida em linguagens imperativas, com variáveis globais.

Em uma linguagem imperativa...

```
int b;
...
int f (int x) {
    b = x;
    return (5)
}

addD (f 3) b == addD b (f 3) ?
```

Exercícios

- Defina as seguintes funções:
  - fatorial`fat :: Int -> Int`
  - compara se quatro números são iguais

```
all4Equal :: Int -> Int -> Int -> Int -> Bool
```

  - `all4Equal` usando `allEqual`
  - retorna quantos parâmetros são iguais

```
howManyEqual :: Int -> Int -> Int -> Int
```

Tipos Básicos

- Inteiros
  - `1,2,3,...:: Int`
  - `+, *, -, div, mod :: Int -> Int -> Int`
  - `>, >=, ==, /=, <=, < :: Int -> Int -> Bool`
  - `Int:` tamamnho fixo por implementação (8/16 bytes)
  - `Integer:` tamanho arbitrário
- Booleanos
  - `True, False :: Bool`
  - `&&, || :: Bool -> Bool -> Bool`
  - `not :: Bool -> Bool`

Exemplo

- suponha vendas semanais dadas pela função
  ```
  sales :: Int -> Int
  ```
- total de vendas da semana 0 à semana n?
  ```
  totalSales :: Int -> Int
  ```
- sales 0 + sales 1 + ... + sales (n-1) + sales n
  ```
  totalSales n
    | n == 0    = sales 0
    | otherwise = totalSales (n-1) + sales n
  ```

Recursão

- Definir **caso base**, i.e. valor para `fun 0`
- Definir o valor para `fun n`
  usando o valor de `fun (n-1)`
  Este é o **caso recursivo**.

```
maxSales :: Int -> Int
maxSales n
  | n == 0     = sales 0
  | otherwise = maxi (maxSales (n-1))
                     (sales n)
```

Casamento de Padrões

- Permite usar padrões no lugar de variáveis, na definição de funções:

```
maxSales :: Int -> Int
maxSales 0 = sales 0
maxSales n = maxi (maxSales (n-1))(sales n)

totalSales :: Int -> Int
totalSales 0 = sales 0
totalSales n = totalSales (n-1) + sales n
```

Casamento de Padrões

```
myNot :: Bool -> Bool
myNot True  = False
myNot False = True

myOr :: Bool -> Bool -> Bool
myOr True  x = True
myOr False x = x

myAnd :: Bool -> Bool -> Bool
myAnd False x = False
myAnd True  x = x
```

Exercícios

- Defina uma função que dado um valor inteiro `s` e um número de semanas `n` retorna quantas semanas de `0` a `n` tiveram venda igual a `s`.

Caracteres e Strings

- `'a','b',...:: Char`
- `'\t', '\n', '\\', '\'', '\"' :: Char`
- `"abc", "andre" :: String`
- `ord :: Char -> Int`
- `chr :: Int -> Char`
- **`ord e chr assumem Unicode`**
- `++ :: String -> String -> String`
  `"abc" ++ "def"`
- `' '`, `""` e `" "` são diferentes!

Caracteres e Strings

```
offset :: Int
offset = ord 'A' - ord 'a'

capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)

isDigit :: Char -> Bool
isDigit ch = (ch >= '0') && (ch <= '9')
```

isDigit, ord, chr: biblioteca `Data.Char`

(`import Data.Char`)

Exercícios

- Defina a função `makeSpaces :: Int -> String` que produz um string com uma quantidade `n` de espaços

- Defina `pushRight :: Int -> String -> String`, utilizando a definição de `makeSpaces`, para adicionar uma quantidade `n` de espaços a um dado string.

Ponto Flutuante

- `Float` e `Double`
- `22.3435 :: Float`
- `+,-,*,/ :: Float -> Float -> Float`
- `pi :: Float`
- `ceiling, floor, round :: Float -> Int`
- **`fromIntegral`** `:: Int -> Float`

Exercícios

- Defina a função `averageSales :: Int -> Float` que dado um número de semanas `n`, retorna a média de vendas das semanas de `0` a `n`.

Estruturas de dados - Tuplas

```
intP :: (Int, Int)
intP = (33,43)

(True, 'x') :: (Bool, Char)

(34, 22,'b') :: (Int, Int, Char)

addPair :: (Int,Int) -> Int
addPair (x,y) = x+y

shift :: ((Int,Int),Int) -> (Int,(Int,Int))
shift ((x,y),z) = (x,(y,z))
```

Sinônimos de Tipos

```
type Name = String
type Age = Int
type Phone = Int
type Person = (Name, Age, Phone)

name :: Person -> Name
name (n,a,p) = n
```

Definições Locais

- Estilo bottom-up ou top-down

```
sumSquares :: Int -> Int -> Int

sumSquares x y = sqX + sqY
    where sqX = x * x
          sqY = y * y

sumSquares x y = sq x + sq y
    where sq z = z * z

sumSquares x y = let sqX = x * x
                     sqY = y * y
                 in sqX + sqY
```

Definições Locais

```
maxThreeOccurs :: Int -> Int -> Int -> (Int,Int)
  maxThreeOccurs m n p = (mx, eqCount)
    where mx = maxiThree m n p
          eqCount = equalCount mx m n p
    ...
```

- `let` definições `in` expressão
- definições `where` definições

Exemplo

```
        Week    Sales
          0       12
          1       23
          2       17
   Total          52
Average           17.333
```

```
printTable :: Int -> String
printTable n = heading
               ++ printWeeks n
               ++ printTotal n
               ++ printAverage n
```

Notação

- Maiúsculas: Tipos e Construtores
- Minúsculas: funções e argumentos
- case sensitive
- comentários:

```
-- isto e' um comentario de uma linha
{- comentario de varias linhas... -}
```

Notação

```
f n + 1
f (n + 1)

2 + 3
(+) 3 2

maxi 2 4
2 `maxi` 4
```

Erros comuns

```
square x =          x
 *x

answer = 42; newline = '\n'

funny x = x +
  1
Error! Unexpected ';'

Funny x = x+1
Error! Undefined constructor 'Funny'
```

Exemplo: equações de segundo grau

- `a*X² + b*X + c = 0.0`
- duas raízes, se `b² > 4.0*a*c`
- uma raiz, se `b² == 4.0*a*c`
- não tem raízes, se `b² < 4.0*a*c`
- `The quadratic equation`
  `1.0*X^2 + 5.0*X + 6.0 = 0.0`
  `has two roots: -2.0 -3.0`
- `(-b ± sqrt(b²-4ac))/2a`

Resolução bottom-up

- Definir as funções auxiliares

```
oneRoot :: Float -> Float -> Float -> Float
oneRoot a b c = -b/(2.0*a)

twoRoots :: Float -> Float -> Float -> (Float, Float)
twoRoots a b c = (d-e, d+e)
  where
  d = -b/(2.0*a)
  e = sqrt(b^2-4.0*a*c)/(2.0*a)
```

Resolução bottom-up

- Definir a função principal

```
roots :: Float -> Float -> Float -> String
roots a b c
   | b^2 == 4.0*a*c = show (oneRoot a b c)
   | b^2 > 4.0*a*c = show f ++ " " ++show s
   | otherwise = "no roots"
     where (f,s) = twoRoots a b c
```

ou

```
f = fst(twoRoots a b c)
s = snd(twoRoots a b c)
```