Programação Funcional

1º Ano

Maria João Frade - Dep. Informática, UM

O que é a programação funcional ?

- É um estilo de programação em que o mecanismo básico de computação é a aplicação de funções a argumentos.
- Uma linguagem de programação que suporte e encoraje esta forma de programação diz-se funcional.
- É um estilo de programação declarativo: um programa é um conjunto de declarações de funções (que descrevem a relação entre input e output).

Exemplo da função factorial

0! = 1
n! = n . (n-1)!

Haskell (uma linguagem declarativa)

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

As equações que são usadas na definição de `fact` são equações matemáticas. Elas indicam que o lado esquerdo e o lado direito do `=` têm o mesmo valor. O valor dos identificadores é **imutável**.

C (uma linguagem imperativa)

```
int factorial(int n)
{ int i, r;
  i = 1;
  r = 1;
  while (i<=n) {
     r = r*i;
     i = i+1; }
  return r;
}
```

Isto é muito diferente do uso do `=` nas linguagens imperativas como o C. Neste caso, o valor dos identificadores é mutável. Por exemplo, a instrução `i = i+1` representa uma atribuição (o valor anterior de `i` é destruído, e o novo valor passa a ser o anterior mais 1). Portanto, o valor de `i` é alterado.

Haskell

Enjoy long-term maintainable software you can rely on

Programa resumido

Esta UC corresponde a uma introdução ao paradigma funcional de programação, tendo por base a linguagem programação **Haskell** (uma linguagem puramente funcional).

1. Aspectos básicos da linguagem Haskell: Valores, expressões e tipos. O mecanismo de avaliação. Inferência de tipos. Definições multi-clausais de funções. Polimorfismo.
2. Listas. Funções recursivas sobre listas. Modelação de problemas usando listas.
3. Algoritmos de ordenação de listas: *insertion sort*, *quick sort* e *merge sort*.
4. Ordem superior: Padrões de computação. Programação com funções de ordem superior.
5. Tipos algébricos: Definição de novos tipos e sua utilização na modelação de problemas.
6. Árvores: Árvores binárias, árvores de procura, árvores irregulares e algoritmos associados.
7. Classes: O mecanismo de classes no tratamento do polimorfismo e da sobrecarga de funções.
8. IO: O tratamento puramente funcional do input/output. O monade IO.

Resultados de aprendizagem

- Resolver problemas de programação decompondo-os em problemas mais pequenos.
- Desenvolver e implementar algoritmos recursivos sobre listas e sobre árvores.
- Desenvolver programas tirando partido da utilização das funções de ordem superior.
- Definir tipos algébricos enquadrá-los na hierarquia de classes e programar com esses tipos.
- Escrever programas interativos.

Método de avaliação

1º teste: **4 valores** (questões de escolha múltipla)

2º teste: **4 valores** (questões de escolha múltipla) + **12 valores** (questões de desenvolvimento)

- As questões de desenvolvimento só serão corrigidas se o aluno tiver, pelo menos, 4 valores nas questões de escolha múltipla.
- Qualquer aluno pode ser chamado para uma prova oral para defender a nota.

Datas previstas para as avaliações

1º teste: 25 de Outubro
2º teste: 6 de Janeiro
Exame de recurso: 26 de Janeiro

Bibliografia

- Fundamentos da Computação. Livro II: Programação Funcional. José Manuel Valença e José Bernardo Barros. Universidade Aberta.
- Programming in Haskell. Graham Hutton. Cambridge University Press, 2016.
- Haskell: the craft of functional programming. Simon Thompson. Addison-Wesley.
- www.haskell.org/documentation
- Slides das aulas teóricas e fichas práticas: elearning.uminho.pt

Características das linguagens funcionais

- O mecanismo básico de programação é a definição e aplicação de funções.
- Funções são entidades de 1ª classe, isto é, podem ser usadas como qualquer outro objecto: passadas como parâmetro, devolvidas como resultado, ou mesmo armazenadas em estruturas de dados.
- Grande flexibilidade, capacidade de abstração e modularização do processamento de dados.
- Os programas são concisos, fáceis de manter e rápidos de desenvolver.

Cronologia das linguagens funcionais

1930's - *Lambda calculus*: uma teoria matemática the funções. *(Alonso Church, Haskell Curry)*

1950's - Lisp: a 1ª ling. prog. funcional, sem tipos, impura. *(John McCarthy)*

1960's - ISWIN: a 1ª ling. prog. funcional pura. *(Peter Landin)*

1970's - FP: ênfase nas funções de ordem superior e no raciocínio sobre programas. *(John Backus)*
ML: a 1ª ling. funcional moderna, com polimorfismo e inferência de tipos. *(Robin Milner)*

1980's - Miranda: ling. funcional com *lazy evaluation,* polimorfismo e inferência de tipos. *(David Turner)*

1990's - Haskell: ling. funcional pura, *lazy*, com um sistema de tipos extremamente evoluído, criada por um comité de académicos.

2000's - Publicação do Haskell Report: a 1ª versão estável da linguagem, actualizada em 2010.

2010's - Haskell Platform: distribuição standard do GHC (Glasgow Haskell Compiler), que inclui bibliotecas e ferramentas de desenvolvimento (actualmente chamada de **GHCup**)

Haskell

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic IO system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages.

(The Haskell 2010 Report)

www.haskell.org

Haskell

An advanced, purely functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

GHCup

GHCup is the main installer for the general purpose language Haskell.

Contém o compilador de Haskell **GHC**, que vamos usar.

Glasgow Haskell Compiler

- Principal compilador de Haskell da actualidade.
- Usado na indústria.
- Inclui um compilador e um interpretador de Haskell:
  - **GHC** - o compilador que a partir do programa Haskell cria código executável.
  - **GHCi** - o interpretador que actua como uma "máquina de calcular". Tem uma natureza interactiva adequada ao desenvolvimento passo a passo de um programa. É o que usaremos nas aulas.

O ciclo de funcionamento do interpretador é o seguinte:

***lê** uma expressão, **calcula** o seu valor e **apresenta** o resultado*

O interpretador GHCi

O interpretador arranca, a partir de um terminal, com o comando **ghci**

```
$ ghci
GHCi, version … : http://www.haskell.org/ghc/. :? for help
Prelude>
```

(Dependendo da versão de GHCi, em vez de **Prelude>** pode aparece apenas **ghci>**)

- O *prompt* **>** indica que o GHCi está pronto para avaliar.
- **Prelude** é o nome da biblioteca que é carregada, por omissão, no arranque do GHCi e que disponibiliza uma vasta lista de funções.

```
Prelude> 5+3*2
11
Prelude> sqrt 9
3.0
```


A biblioteca `Prelude`

O `Prelude` é a biblioteca Haskell que contém as declarações de tipos, funções e classes que constituem o núcleo central da linguagem Haskell. É sempre carregada por omissão.

Por exemplo, tem muitas funções sobre listas:

```
> length [4,2,6,3,1]
5
> head [4,2,6,3,1]
4
> tail [4,2,6,3,1]
[2,6,3,1]
> reverse [4,2,6,3,1]
[1,3,6,2,4]
> last [1..5]
5
> sum [4,2,6,3,1]
16
```

```
> product [1..5]
120
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> head (tail [1..5])
2
> take 2 [3,4,7,1,8]
[3,4]
> drop 2 [3,4,7,1,8]
[7,1,8]
> length [4,2,1] + head [7,5]
10
```


Aplicação de funções

A notação usada em Haskell para a aplicação de funções difere da notação matemática tradicional.

- Na matemática, a aplicação de funções é denotada usando parêntesis e multiplicação denotada por um espaço.
- Em Haskell, a aplicação de funções é denotada por um espaço e multiplicação denotada por *.

Notação matemática	Notação Haskell
f(a,b) + c d	`f a b + c*d`

Em Haskell a aplicações de funções tem prioridade máxima sobre todos os outros operadores.

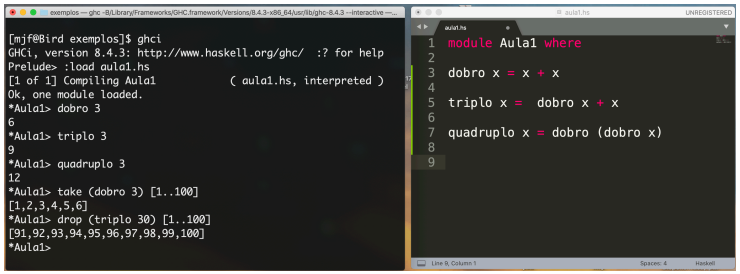
`f a + b` significa `(f a) + b`

Aplicação de funções

Notação matemática	Notação Haskell
f(x)	`f x`
f(x,y)	`f x y`
f(g(x))	`f (g x)`
f(x,g(x))	`f x (g x)`
f(a) + b	`f a + b`

Haskell scripts

- Um programa Haskell é constituído por um, ou mais, ficheiros de texto que contêm as definições das novas funções, tipos e classes usados na resolução de um dado problema.
- A esses ficheiros Haskell costumam-se chamar *scripts*, pelo que o nome dos ficheiros Haskell termina normalmente com **.hs** (de Haskell script).
- No desenvolvimento de um programa Haskell é útil manter duas janelas abertas: uma com o editor de texto onde se vai desenvolvendo o programa, e outra com o GHCi para ir testando as funções que se vão definindo.



Haskell scripts

Mantendo o GHCi aberto podemos acrescentar mais definições ao ficheiro aula1.hs e depois recarrega-lo no GHCi para as testar.

Por exemplo, podemos acrescentar ao ficheiro a definição da função factorial

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

E depois recarrega-lo no GHCi

```
*Aula1> :reload
[1 of 1] Compiling Aula1     (aula1.ha. interpreted)
Ok, one module loaded
*Aula1> fact 5
120
*Aula1>
```

Repare na mudança de nome do prompt para *Aula1> que é o nome do módulo que está no ficheiro aula1.hs. (Este aspecto pode depender da versão do GHCi que se está a usar).
Neste momento tem disponíveis no interpretador todas as funções do Prelude e do módulo Aula1.

Alguns comandos do GHCi

Comando	
`:?`	Mostra todos os comandos disponíveis
`:load` *nome*	Carrega no GHCi o ficheiro *nome*
`:reload`	Carrega de novo o ficheiro corrente
`:type` *expressão*	Indica o tipo de uma expressão
`:quit`	Sai do GHCi

Pode usar apenas a primeira letra do comando. Por exemplo, `:l aula1.hs`

Valores, expressões e tipos

- Os valores são as entidades básicas da linguagem Haskell. São os elementos atómicos.
- Uma expressão ou é um valor ou resulta de aplicar funções a expressões.
- O interpretador atua como uma calculadora: *lê uma expressão, calcula o seu valor e apresenta o resultado.*

```
> 5.3 + 7.2 * 0.1
6.02
> 2 < length [4,2,5,1]
True
> not True
False
```

- Um tipo é um nome que denota uma coleção de valores.
- Se da avaliação de uma expressão `e` resultar um valor do tipo `T`, então dizemos que a *expressão `e` tem tipo `T`*, e escrevemos

```
e :: T
```

Por exemplo,

```
> :type not True
not True :: Bool
```

Tipos

- Um tipo é um nome que denota uma coleção de valores. Por exemplo,
  - O tipo **`Bool`** contém os dois valores lógicos: **`True`** e **`False`**
  - O tipo **`Integer`** contém todos os números inteiros: **`…, -2, -1, 0, 1, 2, 3, …`**
- As funções só podem ser aplicadas a argumentos de tipo adequado. Por exemplo,

```
>  2 + True
error: …
```

Porque + deve ser aplicada a números e `True` não é um número.

- Se não houver concordância entre o tipo das funções e os seus argumentos, o programa é rejeitado pelo compilador.

Tipos

- Toda a expressão Haskell bem formada tem um tipo que é automaticamente calculado em tempo de compilação por um mecanismo chamado inferência de tipos.
- Por isso se diz que a linguagem Haskell é *"statically typed".*
- Todos os erros de tipo são encontrados em tempo de compilação, o que torna os programas mais robustos.
- Os tipos permitem assim programar de forma mais produtiva, com menos erros.
- Num programa Haskell não é obrigatório escrever os tipos, o compilador infere-os, mas é boa prática escreve-los pois é uma forma de documentar o código.

Tipos básicos

`Bool`	Booleanos	`True, False`
`Char`	Caracteres	`'a', 'b', 'A', '3', '\n', …`
`Int`	Inteiros de tamanho limitado	`5, 7, 154243, -3452, …`
`Integer`	Inteiros de tamanho ilimitado	`26456118068672433368303400 43, …`
`Float`	Números de vírgula flutuante	`55.3, 23E5, 743.2e12, …`
`Double`	Números de vírgula flutuante de dupla precisão	`55.3, 23.5E5, …`
`()`	*Unit*	`()`

Tipos compostos

- **Tuplos** - sequências de tamanho fixo de elementos de diferentes tipos

```
(5, True) :: (Int, Bool)
(8, 3.5, 'b') :: (Int, Float, Char)
```

- **Listas** - sequências de tamanho variável de elementos de um mesmo tipo

```
[1,2,3,4,5] :: [Int]
[True, True, False] :: [Bool]
```

- **Funções** - mapeamento de valores de um tipo (o domínio da função) em valores de outro tipo (o contra-domínio da função)

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

Tuplos

Um tuplo é uma sequência de tamanho fixo de elementos que podem ser de diferentes tipos.

$(T_1, T_2, \ldots, T_n)$ é o tipo dos tuplos de tamanho n, cujo 1º elemento é de tipo T_1, o 2º elemento é de tipo T_2, …, e na posição n tem um elemento de tipo T_n.

Exemplos:

```
('C', 2, 'A') :: (Char, Int, Char)

(True, 1, False, 0) :: (Bool, Int, Bool, Int)

(3.5, ('a',True), 20) :: (Float, (Char,Bool), Int)
```


Listas

As listas são sequências de tamanho variável de elementos do mesmo tipo.

[T] é o tipo das listas de elementos do tipo T.

Exemplos:

```
[10,20,30] :: [Int]

[10, 20, 6, 19, 27, 30] :: [Int]

[True,True,False,True] :: [Bool]

[('a',True), ('b',False)] ::[(Char,Bool)]

[[3,2,1], [4,7,9,2], [5]] :: [[Int]]
```


Funções

Uma função é um mapeamento de valores de um tipo (o domínio da função) em valores de outro tipo (o contra-domínio da função)

T_1 -> T_2 é o tipo das funções que *recebem* valores do tipo T_1 e *devolvem* valores do tipo T_2.

Exemplos:

```
even :: Int -> Bool

odd :: Int -> Bool

not :: Bool -> Bool
```


Funções com vários argumentos

Uma função com vários argumentos pode ser codificada de duas formas.

- Usando tuplos:

```
soma :: (Int,Int) -> Int
soma (x,y) = x + y
```

`soma` recebe um par de inteiros `(x,y)` e devolve o resultado inteiro `x+y`.

- Retornando funções como resultado:

```
add :: Int -> (Int -> Int)
add x y = x + y
```

`add` recebe um inteiro `x` e devolve uma função `(add x)`.
Depois esta função recebe o inteiro `y` e devolve o resultado `x+y`.

Curried functions

```
soma :: (Int,Int) -> Int

add :: Int -> (Int -> Int)
```

soma e add produzem o mesmo resultado final, mas soma recebe os dois argumentos ao mesmo tempo (num par), enquanto add recebe um argumento de cada vez.

- As funções que recebem os seus argumentos um de cada vez dizem-se *"curried"* em honra do matemático Haskell Curry que as estudou.
- Funções que recebem mais do que dois argumentos podem ser *curried* retornando funções aninhadas.

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

mult recebe um inteiro x e devolve uma função (mult x), que por sua vez recebe o inteiro y e devolve a função (mult x y), que finalmente recebe o inteiro z e devolve o resultado x*y*z.

Porque é que as funções *curried* são úteis?

- As funções *curried* são mais flexíveis porque é possível gerar novas funções, aplicando parcialmente uma função *curried.*

```
add 1 :: Int -> Int
```

- As funções Haskell são normalmente definidas na forma *curried*.

```
take :: Int -> [Int] -> [Int]

take 5 :: [Int] -> [Int]
```


Convenções

Para evitar o uso excessivo de parêntesis quando se usam funções *curried* são adotadas as seguintes convenções:

- A seta -> associa à direita

```
Int -> Int -> Int -> Int
```

Significa `Int -> (Int -> (Int -> Int))`

- A aplicação de funções associa à esquerda

```
mult x y z
```

Significa `((mult x) y) z`

Funções polimórficas

Há funções às quais é possível associar mais do que um tipo concreto. Por exemplo, a função `length` (que calcula o comprimento de uma lista) pode ser aplicada a quaisquer listas independentemente do tipo dos seus elementos.

```
> length [3,2,2,1,4]
5
> length [True,False]
2
```

Um função diz-se polimórfica se o seu tipo contém variáveis de tipo (representadas por letras minúsculas).

```
length :: [a] -> Int
```

Para qualquer tipo a, a função length recebe uma lista de valores do tipo a e devolve um inteiro.

Funções polimórficas

As variáveis de tipo podem ser instanciadas por diferentes tipos consoante as circunstâncias.

```
length :: [a] -> Int
```

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

> length [True,False]
2
```

a = Int

a = Bool

- As variáveis de tipo começam por letras minúsculas (normalmente usam-se as letras `a`, `b`, `c`, etc).
- O nome dos tipos concretos começa sempre por uma letra maiúscula (ex: `Int`, `Bool`, `Float`, etc)

Funções polimórficas

A maioria das funções da biblioteca `Prelude` são polimórficas.

```
head :: [a] -> a

tail :: [a] -> [a]

take :: Int -> [a] -> [a]

fst :: (a,b) -> a

snd :: (a,b) -> b

id :: a -> a

reverse :: [a] -> [a]

zip :: [a] -> [b] -> [(a,b)]
```


Sobrecarga (*overloading*) de funções

Considere os seguintes exemplos:

```
> 3 + 2
5
> 10.5 + 1.7
12.2
```

Qual será o tipo da soma (+) ?

Note que `a -> a -> a` é um tipo demasiado permissivo para a função (+), pois não é possível somar elementos de qualquer tipo

```
> 'a' + 'b'
:error …
```

O Haskell resolve o problema com restrições de classes

```
(+) :: Num a => a -> a -> a
```

Para qualquer tipo numérico a, a função `(+)` recebe dois valores do tipo a e devolve um valor do tipo a.

Sobrecarga (*overloading*) de funções

Uma função polimórfica diz-se sobrecarregada se o seu tipo contém uma ou mais restrições de classes.

```
(+) :: Num a => a -> a -> a
```

```
> 3 + 2
5
> 10.5 + 1.7
12.2
> 'a' + 'b'
error: …
```

a = Int

a = Float

`Char` não é um tipo numérico

Mais exemplos:

```
sum :: Num a => [a] -> a

(*) :: Num a => a -> a -> a

product :: Num a => [a] -> a
```

Class constraints

O Haskell tem muitas classes mas, por agora, apenas precisamos de ter a noção de que existem as seguintes

Num - a classe dos tipos numéricos (tipos sobre os quais estejam definidas operações como a soma e a multiplicação).

Eq - a classe dos tipos que têm o teste de igualdade definido.

Ord - a classe dos tipos que têm uma relação de ordem definida sobre os seus elementos.

```
(+) :: Num a => a -> a -> a

(==) :: Eq a => a -> a -> Bool

(<) :: Ord a => a -> a -> Bool
```

Mais tarde estudaremos em mais detalhe o mecanismo de classes do Haskell.

!!! Aviso !!!

Na versão actual do GHCi, se perguntarmos o tipo de certas funções sobre listas temos uma surpresa!!

```
> :type sum
sum :: (Foldable t, Num a) => t a -> a
```

Este é um tipo mais geral do que `sum :: Num a => [a] -> a` mas como as listas pertencem à classe `Foldable`, quando aplicamos `sum` a uma lista de números, este é o tipo efetivo da função `sum`.

Ao longo das aulas iremos sempre apresentar o tipo destas funções usando **listas** em vez da classe `Foldable`, para simplificar a apresentação.

Sempre que virem a classe `Foldable` num tipo das funções do Prelude, podem entender isso como sendo o tipo das listas.

Class constraints

Qual será o tipo das funções `elem` e `max` ?

```
> elem 3 [1,2,3,4,5]
True
> elem 7 [1,2,3,4,5]
False
> max 35 28
35
> max 5.6 10.7
10.7
```

```
elem :: Eq a => a -> [a] -> Bool
```

Porque se usa a função (==) na sua implementação.

```
max :: Ord a => a -> a -> a
```

Porque se usa a função (<) na sua implementação.

Mais alguns operadores do `Prelude`

Lógicos: `&&` (conjunção), `||` (disjunção), `not` (negação)

Numéricos: `+`, `-`, `*`, `/` (divisão de reais), `^` (exponenciação com inteiros), `div` (divisão inteira), `mod` (resto da divisão inteira), `abs` (módulo) `**` (exponenciações com reais), `log`, `sin`, `cos`, `tan`, …

Relacionais: `==` (igualdade), `/=` (desigualdade), `<`, `<=`, `>`, `>=`

Condicional: `if ... then ... else ...`

`:: Bool` `:: a`