Programação Funcional

Mestrado Integrado em Engenharia Informática

2015 / 2016

Maria João Frade (mjf@di.uminho.pt)

Departamento de Informática
Universidade do Minho

Programa Resumido

Nesta disciplina estuda-se o paradigma funcional de programação, tendo por base a linguagem de programação ***Haskell***.

- Programação funcional em Haskell.
  - *Conceitos básicos:* expressões, tipos, redução, funções e recursividade.
  - *Conceitos avançados:* funções de ordem superior, polimorfismo, tipos algébricos, classes, modularidade e monades.
- Estruturas de dados e algoritmos.

Objectivo

- **Saber escrever programas em Haskell para resolver problemas.**

Avaliação

Nota Final = (Nota do 1ª teste) * (Nota do 2ª teste)

- **1º teste**: uma questão seleccionada aleatoriamente de um conjunto de questões simples previamente divulgado. (Notas: 0 ou 1)
- **2º teste**: prova escrita sobre toda a matéria. (Notas: de 0-20)

1º teste: 21-Out-2015
2º teste: 11-Jan-2016

Exame de recurso: 1-Feb-2016

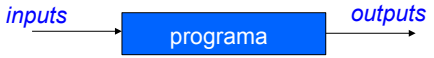
Bibliografia

- Fundamentos da Computação, Livro II: Programação Funcional.
  José Manuel Valença e José Bernardo Barros. Universidade Aberta, 1999.
- Programming in Haskell.
  Graham Hutton. Cambridge University Press, 2007.
  http://www.cs.nott.ac.uk/~pszgmh/book.html
- Haskell: the craft of functional programming.
  Simon Thompson. Addison-Wesley.
- Introduction to Functional Programming using Haskell.
  Richard Bird. Prentice-Hall, 1998.
- Real World Haskell.
  Bryan O'Sullivan, Don Stewart, and John Goerzen. O'Reilly, 2008.
  http://book.realworldhaskell.org
- Learn You a Haskell for Great Good.
  Miriam Lipovaca. http://learnyouahaskell.com
- Apontamentos da aulas teóricas e fichas práticas. http://elearning.uminho.pt
- https://www.haskell.org/documentation

Haskell

- Foi criado no final dos anos 80 por um comité de académicos.
- É uma linguagem:
  - "**Purely functional**"
  - "**Statically typed**"
  - "**Lazy**"
- É cada vez mais usada na indústria: Google, Facebook, AT&T, NASA, ...
- Permite programar de forma clara, concisa e com alto nível de abstracção.

Um **programa** pode ser visto como algo que transforma informação



Existem 2 grandes classes de linguagens de programação:

Imperativas - um programa é uma sequência de instruções (ou seja de "ordens"). (ex: C, Java, ...)

-- difícil estabelecer uma relação precisa entre o input e o output e de raciocinar sobre os programas; ...
\+ normalmente mais eficientes; ...

Declarativas - um programa é um conjunto de declarações que descrevem a relação entre o input e o output. (ex: ML, Haskell, ...)

\+ facíl de estabelecer uma relação precisa entre o input e o output e de raciocinar sobre os programas; ...
-- normalmente menos eficientes (mas cada vez mais); ...

Exemplo: A função factorial é descrita matematicamente por

$0! = 1$
$n! = n * (n-1)!$, se $n>0$

Dois programas que fazem o cálculo do factorial de um número, implementados em:

C

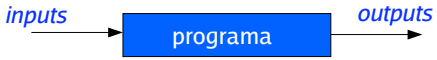
```
int factorial(int n)
{ int i, r;

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

Haskell

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

Qual é mais facil de entender ?



Na programação (funcional) faremos uma distinção clara entre três grandes grupos de conceitos:

Dados – Que tipo de informação é recebida e como ela se pode organizar por forma a ser processada de forma eficiente.

Operações – Os mecanismos para manipular os dados. As operações básicas e como construir novas operações a partir de outras já existentes.

Cálculo – A forma como o processo de cálculo decorre.

A linguagem **Haskell** fornece uma forma rigorosa e precisa de descrever tudo isto.

O Paradigma Funcional de Programação

Haskell

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

As equações que são usadas na definição da função `fact` são **equações matemáticas**. Elas indicam que o lado esquerdo e direito têm o mesmo valor.

C

```
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.

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 valor anterior mais 1). Portanto `i` é redefinido.

No paradigma funcional não existe a noção de atribuição!

Porque = em Haskell significa ***"é, por definição, igual a"***, e não é possível redefinir, o que fazemos é raciocinar sobre equações matemáticas.

Um pouco de história ...

1960s **Lisp** *(untyped, not pure)*

1970s **ML** *(strongly typed, type inference, polymorphism)*

1980s **Miranda** *(strongly typed, type inference, polymorphism, lazy evaluation)*

1990s **Haskell** *(strongly typed, type inference, polymorphism, lazy evaluation, ad-hoc polymorphism, monadic IO)*

Haskell

- O Haskell é uma linguagem puramente funcional, fortemente tipada, e com um sistema de tipos extremamente evoluido.
- A linguagem usada neste curso é o **Haskell 98**. (Novo: **Haskell 2010**)
- Exemplos de interpretadores e um compilador para a linguagem Haskell 98:
  - **Hugs** *Haskell User's Gofer System*
  - **GHC** *Glasgow Haskell Compiler* (é o que vamos usar ...)

The Haskell Platform contém o GHC

www.haskell.org

O Paradigma Funcional de Programação

- Um **programa** é um conjunto de definições.
- Uma **definição** associa um ***nome*** a um ***valor***.
- **Programar** é definir estruturas de dados e funções para resolver um dado problema.
- O **interpretador** (da linguagem funcional) actua como uma máquina de calcular:

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

Exemplo: Um programa para converter valores de temperaturas em graus *Celcius* para graus *Farenheit*, e de graus *Kelvin* para graus *Celcius*.

```
celFar c = c * 1.8 + 32
kelCel k = k - 273
```

Depois de carregar este programa no interpretador Haskell, podemos fazer os seguintes testes:

```
> celFar 25
77.0
> kelCel 0
-273
>
```

- A um conjunto de associações *nome-valor* dá-se o nome de **ambiente** ou **contexto** (ou *programa*).

- As expressões são avaliadas no âmbito de um contexto e podem conter ocorrências dos nomes definidos nesse contexto.

- O interpretador usa as definições que tem no contexto (programa) como regras de cálculo, para simplificar (calcular) o valor de uma expressão.

Exemplo: Este programa define três funções de conversão de temperaturas.

```
celFar c = c * 1.8 + 32
kelCel k = k - 273
kelFar k = celFar (kelCel k)
```

No interpretador ...

```
> kelFar 300
80.6
```

É calculado pelas regras estabelecidas pelas definições fornecidas pelo programa.

```
kelFar 300 ⇨ celFar (kelCel 300)
           ⇨ (kelCel 300) * 1.8 + 32
           ⇨ (300 - 273) * 1.8 + 32
           ⇨ 27 * 1.8 + 32
           ⇨ 80.6
```


Valores, expressões e seus tipos

Os **valores** são as entidades básicas da linguagem Haskell. São os elementos atómicos.

As **expressões** são obtidas aplicando funções a valores ou a outras expressões.

O interpretador Haskell actua como uma *calculadora ("read - evaluate - print loop")*:

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

Exemplos:

```
> 3.5 + 6.7
10.2

> 2 < 35
True
```

Os tipos servem para **classificar** entidades (de acordo com as suas caracteristicas).

Em Haskell *toda a expressão tem um **tipo** associado.*

`e :: T` significa que a expressão e *tem* tipo T
é do tipo

Tipos

- **Tipos básicos:** Char, Bool, Int, Integer, Float, Double, ()

- **Tipos compostos:**

  Produtos Cartesianos `(T1,T2, ...,Tn)`

  `(T1,T2,...,Tn)` é o tipo dos tuplos com o 1º elemento do tipo `T1`, 2º elemento do tipo `T2`, etc.

  Listas `[T]`

  `[T]` é o tipo da listas cujos elementos *são todos* do tipo `T`.

  Funções `T1 -> T2`

  `T1 -> T2` é o tipo das funções que *recebem* valores do tipo `T1` e *devolvem* valores do tipo `T2`.

data types novos que podemos definir ...

Demo