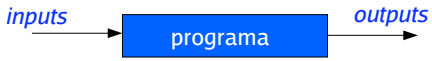
Programação Funcional

2012/2013

Maria João Frade (mjf@di.uminho.pt)
Departamento de Informática
Universidade do Minho

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: Pascal, C, Java, ...)

-- díficil 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: Prolog, 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

$$0! = 1$$
$$n! = n * (n-1)! \text{ , se } n>0$$

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

C

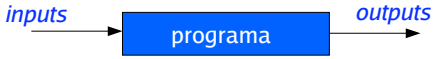
```
int factorial(int n)
{ int i, r;

  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.

Programa Resumido

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

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

Bibliografia

- **Fundamentos da Computação, Livro II: Programação Funcional.** José Manuel Valença e José Bernardo Barros. Universidade Aberta, 1999.
- Introduction to Functional Programming using Haskell. Richard Bird. Prentice-Hall, 1998.
- Programming in Haskell. Graham Hutton. Cambridge University Press, 2007,
- Haskell: the craft of functional programming. Simon Thompson. Addison-Wesley, 1999.
- A Gentle Introduction to Haskell. Paul Hudak, John Peterson and Joseph Fasel.
- www.haskell.org

Apontamentos das aulas teóricas e fichas práticas

elearning.uminho.pt

Avaliação

- A avaliação na disciplina consiste em duas componentes, I (12 valores) e II (8 valores), sendo que a nota mínima na componente I, que visa os resultados mínimos de aprendizagem, é de 8 valores em 12.
- A avaliação da componente II será sempre efectuada num teste final ou exame.
- A componente I poderá ser obtida em regime contínuo durante o semestre (método A) ou no teste ou exame final (método B). Para este efeito (método A) terão lugar nas aulas teórico-práticas mini-testes periódicos, resultando numa nota entre 0 e 12 valores.
- Os alunos que entreguem pelo menos um mini-teste serão considerados automaticamente inscritos no método A. Caso obtenham nota inferior a 8 valores na componente I, apenas poderão ser avaliados no exame, ficando-lhes impossibilitado o acesso ao teste final. Caso obtenham nota superior ou igual a 8 valores nessa componente, serão avaliados no teste final apenas na componente II.
- Os alunos que não entreguem qualquer mini-teste consideram-se inscritos no método B, podendo ser avaliados em ambas as componentes I e II no teste final ou no exame.

Avaliação

	Parte I (12 valores)	**Parte II** (8 valores)
Método A	4 mini-testes	teste final
Método B	teste final	teste final

- Datas previstas para os mini-testes: aulas TP
  - de 15-Out a 20-Out
  - de 5-Nov a 10-Nov
  - de 26-Nov a 3-Dez
  - de 7-Jan a 12-Jan
- Nota mínima da Parte I: 8 valores.
- Os alunos que entreguem pelo menos um mini-teste serão considerados automaticamente inscritos no método A.
- Não é autorizada a passagem do método A para o método B.
- Qualquer aluno que não fique aprovado no teste, pode ir a exame.

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

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.

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