Programação Funcional e em Lógica
Aula 1

2022

Conteúdo e objetivos

- Introdução à programação funcional usando a linguagem Haskell
- Objetivos de aprendizagem:
  - definir funções usando equações com padrões e guardas
  - definir novos tipos algébricos para estruturas de dados
  - definir algoritmos recursivos elementares
  - decompor problemas de programação elementares em funções
  - utlizar funções de ordem superior e *lazy evaluation*
  - escrever programas com I/O usando monads e notação-*do*

Bibliografia recomendada

- *Programming in Haskell*, Graham Hutton, Cambridge University Press, 2007-2016.
- *Introduction to Functional Programming*, Richard Bird & Philip Wadler, Prentice-Hall International, 1988.

Outros livros

- *Thinking functionally with Haskell*, Richard Bird. Cambridge University Press, 2015.
- *Learn you a Haskell for great good!*, Miran Lipovača. `http://learnyouahaskell.com/`

O que é a programação funcional?

- Um paradigma: uma conceção sobre o que é um programa
- Programas em C ou Java são normalmente imperativos: sequências de comandos que modificam variáveis em memória
- No paradigma funcional, um programa é um conjunto de definições de funções que usamos para exprimir um algoritmo por composição
- Num programa puramente funcional nunca modificamos variáveis: só aplicamos funções!
- Um programa funcional é uma função

$$\textit{dados} \xrightarrow{\text{programa}} \textit{resultado}$$

expressa como composição de funções mais simples

Linguagens funcionais

- Podemos programar num estilo funcional em quase todas as linguagens...
- ...mas as linguagens explicamente funcionais suportam melhor este paradigma
- Exemplos: Scheme, ML, OCaml, Haskell, F#, Scala

Exemplo

Para exemplificar a distinção de paradigmas vamos ver dois pequenos programas que calculam

$$1^2 + 2^2 + 3^2 + \cdots + 10^2$$

em C e em Haskell.

Somar os quadrados — versão imperativa

```
/* Programa imperativo em C */
int main() {
  int total = 0;
  for (int i=1; i<=10; ++i) {
     total = total + i*i;
  }
  printf("%d\n", total);
}
```

- O programa é uma sequência de instruções
- O resultado é calculado modificando as variáveis
- Compreender o programa significa compreender como os valores mudam ao longo do tempo

Execução passo-a-passo

Inspecionando valores de variáveis à entrada do ciclo:

passo	0	1	2	3	4	5	6	7	8	9	final
`i`	1	2	3	4	5	6	7	8	9	10	11
`total`	0	1	5	14	30	55	91	140	204	285	385

Somar quadrados — versão funcional

```
-- Programa funcional em Haskell
main = print (sum (map (^2) [1..10]))
```

- `[1..10]` é a sequência de inteiros de 1 a 10
- `map (^2)` calcula o quadrado de cada valor
- `sum` soma a sequência
- `print` imprime o resultado

Redução passo-a-passo I

A execução de um programa funcional é redução da expressão até obter um resultado que não pode ser mais simplificado.

```
  sum (map (^2) [1..10])
=
  sum [1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2]
=
  1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 +
  10^2
=
  1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
  385
```

Redução passo-a-passo II

Ao contrário do programa imperativo: podemos efetuar reduções por outra ordem e obter o mesmo resultado.

```
  sum (map (^2) [1..10])
=
  sum [1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2]
=
  sum [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
=
  1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
  385
```

Porquê aprender programação funcional? I

Pensar a um nível mais alto

- programas mais concisos
- próximos duma especificação matemática
- mais enfoque na análise do problema e menos em "debugging"
- ajuda a programar melhor em qualquer linguagem!

> *A language that doesn't affect the way you think about programming is not worth knowing.*

Alan Perlis (1922–1990), pioneiro norte-americano da ciência de computadores

Porquê aprender programação funcional? II

Mais modularidade

- decompor problemas em componentes pequenas e re-utilizáveis

Garantias de correção

- demonstrações de correção usando provas matemáticas
- maior facilidade em fazer testes automáticos

Concorrencia/paralelismo

- a ordem de execução não afecta os resultados

Desvantagens da programação funcional

Maior distância do *hardware*

- os compiladores e interpretadores são mais complexos
- difícil prever os custos de execução (tempo/espaço)
- alguns programas de baixo-nível necessitam de controlo preciso de tempo/espaço
- alguns algoritmos são mais eficientes quando implementados de forma imperativa

Um pouco de história I

1930s Alonzo Church desenvolve o cálculo-λ, um formalismo matemático para exprimir computação usando funções

1950s Inspirado no cálculo-λ, John McCarthy desenvolve o LISP, uma das primeiras linguagens de programação

1960s Peter Landin desenvolve o ISWIM, a 1ª linguagem

1970s–1980s Robin Milner desenvolve o Standard ML, a primeira linguagem funcional com *polimorfismo* e *inferência de tipos*

1970s–1980s David Turner desenvolve várias linguagens que empregam *lazy evaluation*, culminando na linguagem Miranda

1987 Um comité académico inicia o desenvolvimento do Haskell, uma linguagem funcional padronizada com *lazy evaluation*

Um pouco de história II

2003 Publicação do *Haskell 98*, uma definição padronizada da linguagem

2010 Publicação do padrão da linguagem *Haskell 2010*

2021 Criação da *Haskell Foundation* uma organização para promover a adopção do Haskell

Linguagem Haskell

`http://www.haskell.org`

- Uma linguagem funcional pura de uso genérico
- Nomeada em homenagem ao matemático americano Haskell Curry (1900–1982)
- Concebida para ensino e também para o desenvolvimento de aplicações reais
- Resultado de trinta anos de investigação por uma comunidade académica muito activa
- Utilização industrial crescente nos últimos 10–15 anos
- Implementação principal livre: *Glasgow Haskell Compiler* (GHC)

Haskell na indústria

Galois Investigação aplicada em segurança e sistemas críticos
`https://galois.com/`

Facebook Sistema de deteção de *Spam*
`https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/`

Github Projeto *Semantic* para análise de código-fonte de várias linguagens
`https://github.com/github/semantic`

Cardano Plataforma de cripto-moeda e *smart contracts*
`https://iohk.io/projects/cardano/`

Mais exemplos:
`http://www.haskell.org/haskellwiki/Haskell_in_industry`

Haskell em *open-source*

GHC o compilador de Haskell é escrito em Haskell
`https://www.haskell.org/ghc/`

Darcs um sistema distribuido para gestão de código-fonte
`http://darcs.net/`

Pandoc conversor entre formatos de "markup" de documentos
`https://pandoc.org/`

Codex um dos sistemas para exercícios de programação on-line usando no DCC ☺
`https://github.com/pbv/codex`

Glasgow Haskell Compiler (GHC)

- Compilador de Haskell código-máquina nativo
- Suporta Haskell 98, Haskell 2010 e bastantes extensões
- Otimização de código, interfaces a outras linguagens, *profilling*, grande conjunto de bibliotecas, etc.
- Inclui também um interpretador interativo `ghci` (útil para experimentação)
- Disponível em `http://www.haskell.org/ghc`

Primeiros passos

Linux/Mac OS: executar comando `ghci`

Windows: executar aplicação *WinGHCi* ou comando `ghci`

```
$ ghci
GHCi, version 8.6.5:
http://www.haskell.org/ghc/ :? for help
Prelude>
```

Uso do interpretador

O interpretador *lê uma expressão* do terminal, *calcula o seu valor* e *imprime-o*.[1]

```
Prelude> 2+3*5
17

Prelude> (2+3)*5
25

Prelude> sqrt (3^2 + 4^2)
5.0
```

[1] Em inglês: *read-eval-print-loop* ou *REPL*.

Alguns operadores e funções aritméticas

`+`	adição
`-`	subtração
`*`	multiplicação
`/`	divisão fracionária
`^`	potência (expoente inteiro)
`div`	quociente de divisão inteira
`mod`	resto de divisão inteira
`sqrt`	raiz quadrada
`==`	comparação igualdade
`/=`	negação da igualdade (diferente)
`< > <= >=`	comparações de ordem

Algumas convenções sintáticas I

- Os argumentos de funções são separados por espaços
- A aplicação tem maior precendência do que qualquer operador

Haskell	notação usual
`f x`	$f(x)$
`f (g x)`	$f(g(x))$
`f (g x) (h x)`	$f(g(x), h(x))$
`f x y + 1`	$f(x, y) + 1$
`f x (y+1)`	$f(x, y + 1)$
`sqrt x + 1`	$\sqrt{x} + 1$
`sqrt (x + 1)`	$\sqrt{x + 1}$

Algumas convenções sintáticas II

- Um operador pode ser usando como uma função escrevendo-o entre parêntesis
- Reciprocamente: uma função pode ser usada como operador escrevendo-a entre aspas esquerdas

```
(+) x y   ≡   x+y
(*) y 2   ≡   y*2

x`mod`2   ≡   mod x 2
f x `div` n   ≡   div (f x) n
```

O prelúdio-padrão (*standard Prelude*)

O módulo *Prelude* contém um grande conjunto de funções pré-definidas:

- os operadores e funções aritméticas
- funções genéricas sobre listas
- . . . entre muitas outras

O prelúdio-padrão é carregado automaticamente pelo interpretador/compilador e pode ser usado em qualquer programa Haskell.

Algumas funções do prelúdio I

```
> head [1,2,3,4]
1
> head "banana"
'b'
```

obter o 1º elemento

```
> tail [1,2,3,4]
[2,3,4]
> tail "banana"
"anana"
```

remover o 1º elemento

```
> length [1,2,3,4,5]
5
> length "banana"
6
```

comprimento

Algumas funções do prelúdio II

obter um prefixo

```
> take 3 [1,2,3,4,5]
[1,2,3]
> take 3 "banana"
"ban"
```

remover um prefixo

```
> drop 3 [1,2,3,4,5]
[4,5]
> drop 3 "banana"
"ana"
```

concatenar

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> "aba" ++ "cate"
"abacate"
```

Algumas funções do prelúdio III

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
> reverse "abacate"
"etacaba"
```

inverter a ordem

```
> [1,2,3,4,5] !! 3
4
> "abacate" !! 3
'c'
```

indexação a partir de 0

```
> sum [1,2,3,5]
11
```

soma dos valores

```
> product [1,2,3,5]
30
```

produto dos valores

Definir novas funções

- Vamos definir novas funções num ficheiro de texto
- Usamos um editor de texto externo (e.g. Emacs)
- O nome do ficheiro deve terminar em `.hs` (*Haskell script*)[2]

[2] Alternativa: `.lhs` (*literate Haskell script*)

Criar um ficheiro de definições

teste.hs

```
dobro x = 2*x

quadruplo x = dobro (dobro x)
```

Usamos o comando *:load* para carregar estas definições no GHCi.

```
$ ghci
...
> :load teste.hs
[1 of 1] Compiling Main ( teste.hs, interpreted )
Ok, modules loaded: Main.
```

Exemplos de uso

```
> dobro 2
4

> quadruplo 2
8

> take (quadruplo 2) [1..100]
[1,2,3,4,5,6,7,8]
```

Modificar o ficheiro

Acrescentamos novas definições e gravamos.

teste.hs

```
factorial n = product [1..n]

media x y = (x+y)/2
```

Usamos *:reload* no GHCi para carregar as modificações.

```
> :reload
> factorial 10
3628800
> media 2 3
2.5
```

Comandos úteis do interpretador

`:load` *ficheiro*	carregar um ficheiro
`:reload`	re-carregar modificações
`:edit`	editar o ficheiro actual
`:set editor` *prog*	definir o editor
`:type` *expr*	mostrar o tipo duma expressão
`:help`	obter ajuda
`:quit`	terminar a sessão

Podem ser abreviados:

- `:l` em vez de `:load`
- `:r` em vez de `:reload`
- `:t` em vez de `:type`
- `:q` em vez de `:quit`

Identificadores

Os nomes de funções e variáveis devem começar por letras mínusculas e podem incluir letras, dígitos, sublinhados e apóstrofes:

```
fun1    x_2    y'    fooBar
```

As seguintes palavras reservadas não podem ser usadas como identificadores:

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

Definições locais

Podemos fazer definições locais usando `where`.

```
a = b+c
  where b = 1
        c = 2
d = a*2
```

A indentação indica o âmbito das declarações; também podemos usar agrupamento explícito:

```
a = b+c
  where {b = 1;
         c = 2}
d = a*2
```

Indentação

Todas as definições num mesmo âmbito devem começar na mesma coluna.

```
a = 1
 b = 2
c = 3
```

ERRADO

```
  a = 1
b   = 2
  c = 3
```

ERRADO

```
a = 1
b = 2
c = 3
```

OK

A ordem entre as definições não é importante.

Comentários

Simples: começam por `--` até ao final de uma linha

Multi-linha: delimitados por `{-` e `-}`

```
-- calcular o factorial de um inteiro
factorial n = product [1..n]

-- calcular a média de dois valores
media x y = (x+y)/2

{-  as definições seguintes estão comentadas
dobro x = x+x
quadrado x = x*x
-}
```

Um exemplo maior: *Quicksort* I

Em Java:

```
public class Quicksort {
  public static void qsort(double[] a) {
    qsort(a, 0, a.length - 1);
  }
  public static void qsort(double[] a,
                           int left, int right) {
      if (right <= left) return;
      int i = partition(a, left, right);
      qsort(a, left, i-1);
      qsort(a, i+1, right);
  }
  private void swap(double[] a, int i, int j) {
    double tmp = a[i]; a[i] = a[j]; a[j] = tmp;
  }
```

Um exemplo maior: *Quicksort* II

```
    private static int partition(double[] a,
                                 int left, int right) {
        int i = left;
        int j;
        for(j=left+1; j<=right; ++j) {
            if(a[j] < a[left]) {
              ++i;
              swap(a, i, j);
            }
          }
        swap(a, i, left);
        return i;
    }
}
```

Um exemplo maior: *Quicksort* III

Em Haskell:

```
qsort [] = []
qsort (x:xs) = qsort xs1 ++ [x] ++ qsort xs2
      where xs1 = [x' | x'<-xs, x'<=x]
            xs2 = [x' | x'<-xs, x'>x]
```

Tipos

Um tipo é um nome para uma coleção de valores relacionados.

Por exemplo, o tipo `Bool` contém os dois valores lógicos:

```
True          False
```

Erros de tipos

Algumas operações só fazem sentido com valores de determinados tipos.

Exemplo: não faz sentido somar números e valores lógicos.

```
> 1 + False
<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: 1 + False
      In an equation for ‘it’: it = 1 + False
```

Em Haskell, estes erros são detetados classificando as expressões com o tipo do resultado.

Tipos em Haskell

Escrevemos

```
e :: T
```

para indicar que a expressão *e* admite o tipo *T*.

- Se *e* :: *T*, então o resultado de *e* será um valor de tipo *T*
- O interpretador/compilador verifica tipos indicados pelo programador e infere os tipos omitidos
- Os erros de tipos são assinalados antes da execução

Tipos básicos

Bool valores lógicos
`True`, `False`

Char carateres simples
`'A'`, `'B'`, `'?'`, `'\n'`

String sequências de carateres
`"Abacate"`, `"UB40"`

Int inteiros de precisão fixa (tipicamente 64-*bits*)
`142`, `-1233456`

Integer inteiros de precisão arbitrária
(apenas limitados pela memória do computador)

Float vírgula flutuante de precisão simples
`3.14154`, `-1.23e10`

Double vírgula flutuante de precisão dupla

Listas

Uma *lista* é uma sequência de tamanho variável de elementos dum mesmo tipo.

```
[False,True,False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

Em geral: `[`*T*`]` é o tipo de listas cujos elementos são de tipo *T*.

Caso particular: `String` é equivalente a `[Char]`.

```
"abcd" == ['a','b','c','d']
```

Tuplos

Um *tuplo* é uma sequência de tamanho fixo de elementos de tipos possivelmente diferentes.

```
(42,'a') :: (Int,Char)
(False,'b',True) :: (Bool,Char,Bool)
```

Em geral: $(T_1, T_2, \ldots, T_n)$ é o tipo de tuplos com n componentes de tipos T_i para i de 1 a n.

Observações I

- Listas de tamanhos diferentes podem ter o mesmo tipo.
- Tuplos de tamanhos diferentes têm tipos diferentes.

```
['a'] :: [Char]

['b','a','b'] :: [Char]

('a','b') :: (Char,Char)

('b','a','b') :: (Char,Char,Char)
```

Observações II

Os elementos de listas e tuplos podem ser quaisquer valores, inclusivé outras listas e tuplos.

```
[['a'], ['b','c']] :: [[Char]]

(1,('a',2)) :: (Int,(Char,Int))

(1, ['a','b']) :: (Int,[Char])
```

Observações III

- A lista vazia `[]` admite qualquer tipo `[`*T*`]`
- O tuplo vazio `()` é o único valor do *tipo unitário* `()`
- Não existem tuplos com apenas um elemento

Tipos funcionais I

Uma função faz corresponder valores de um tipo em valores de outro um tipo.

```
not :: Bool -> Bool
```

```
isDigit :: Char -> Bool
```

Em geral:

$$A \texttt{ -> } B$$

é o tipo das funções que fazem corresponder valores do tipo A em valores do tipo B.

Tipos funcionais II

Os argumento e resultado duma função podem ser listas, tuplos ou de quaisquer outros tipos.

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

contar :: Int -> [Int]
contar n = [0..n]
```

Funções de vários argumentos

Uma função de vários argumentos toma um argumento de cada vez.

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

incr :: Int -> Int
incr = soma 1
```

Ou seja: `soma 1` é a função que a cada y associa $1 + y$.

NB: a esta forma de tratar múltiplos argumentos chama-se *currying* (em homenagem a Haskell B. Curry).

Tuplos vs. *currying*

Função de dois argumentos (*curried*)

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

Função de um argumento (par de inteiros)

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

Porquê usar *currying*?

Funções *curried* são mais flexíveis do que funções usando tuplos porque podemos aplicá-las parcialmente.

Exemplos

```
soma 1 :: Int -> Int                    -- incrementar

take 5 :: [Char] -> [Char]              -- primeiros 5 elms.

drop 5 :: [Char] -> [Char]              -- retirar 5 elms.
```

É preferível usar *currying* exceto quando queremos explicitamente construir tuplos.

Convenções sintáticas

Duas convenções que reduzem a necessidade de parêntesis:

- a seta `->` associa à direita
- a aplicação associa à esquerda

```
   Int -> Int -> Int -> Int
=  Int -> (Int -> (Int -> Int))

  f x y z
= (((f x) y) z)
```

Funções polimórficas I

Certas funções operam com valores de qualquer tipo; tais funções admitem tipos com variáveis.

Uma função diz-se polimórfica ("de muitas formas") se admite um tipo com variáveis.

Exemplo

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

A função *length* calcula o comprimento duma lista de valores de qualquer tipo *a*.

Funções polimórficas II

Ao aplicar funções polimórficas, as variáveis de tipos são automaticamente substituidas pelos tipos concretos:

```
> length [1,2,3,4]                          -- Int
4
> length "abacate"                          -- Char
7
> length [False,True]                       -- Bool
2
> length [(2,'A'),(3,'C')]                  -- (Int,Char)
2
```

As variáveis de tipo devem começar por uma letra minúscula; é convencional usar *a*, *b*, *c*, ...

Funções polimórficas III

Muitas funções do prelúdio-padrão são poliformas:

```
null :: [a] -> Bool

head :: [a] -> a

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

fst :: (a,b) -> a

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

O polimorfismo permite utilizar estas funções em vários contextos.

Sobrecarga (*overloading*) I

Certas funções operam sobre vários tipos mas não sobre *quaisquer* tipos.

```
> sum [1,2,3]
6

> sum [1.5, 0.5, 2.5]
4.5

> sum ['a', 'b', 'c']
  No instance for (Num Char) ...

> sum [True, False]
  No instance for (Num Bool) ...
```

Sobrecarga (*overloading*) II

O tipo mais geral da função `sum` é:

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

- "`Num a => ...`" é uma restrição da variável *a*.
- Indica que `sum` opera apenas sobre tipos numéricos
- Exemplos: `Int`, `Integer`, `Float`, `Double`

Sobrecarga (*overloading*) III

As variáveis com restrições só podem ser substituidas por tipos concretos apropriados.

```
> sum [1, 2, 3]                                    -- Int
6

> sum [0.5, 1.0, 1.5]                              -- Float
3.0

> sum ['a', 'b', 'c']
ERRO: Char não é um tipo numérico
```

Algumas classes pré-definidas

Num	tipos numéricos (ex: Int, Integer, Float, Double)
Integral	tipos com divisão inteira (ex: Int, Integer)
Fractional	tipos com divisão fracionária (ex: Float, Double)
Eq	tipos com igualdade
Ord	tipos com comparações de ordem total

Exemplos:

```
(+) :: Num a => a -> a -> a
mod :: Integral a => a -> a -> a
(/) :: Fractional a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
max :: Ord a => a -> a -> a
```

Hierarquia de classes I

Algumas classes estão organizadas numa hierarquica:

- Ord é uma subclasse de Eq
- Fractional e Integral são subclasses de Num

Hierarquia de classes II

B é subclasse de *A* sse todas as operações de *A* existem para tipos em *B*. Exemplos:

- `==` está definida para tipos em Ord
- `+`, `-` e `*` estão definidas para Fractional e Integral

Constantes numéricas

Em Haskell, as constantes também podem ser usadas com vários tipos:

```
1 :: Int
1 :: Float
1 :: Num a => a                    -- tipo mais geral

3.0 :: Float
3.0 :: Double
3.0 :: Fractional a => a           -- tipo mais geral
```

Logo, as expressões seguintes são correctamente tipadas:

```
1/3 :: Float
(1 + 1.5 + 2) :: Float
```

Misturar tipos numéricos

Vamos tentar definir uma função para calcular a média aritmética duma lista de números.

```
media xs = sum xs / length xs
```

Parece correta, mas tem um erro de tipos!

```
Could not deduce (Fractional Int) ...
```

Misturar tipos numéricos (cont.)

Problema

```
(/) :: Fractional a => a -> a -> a  -- divisão fracionária
length xs :: Int                    -- não é fracionário
```

Solução: usar uma conversão explícita

```
media xs = sum xs / fromIntegral (length xs)
```

`fromIntegral` converte qualquer tipo inteiro para qualquer outro tipo numérico.

Quando usar anotações de tipos I

- Podemos escrever definições e deixar o interpretador inferir os tipos.
- Mas é recomendado anotar sempre tipos de definições de funções:

```
media :: [Float] -> Float
media xs = sum xs / fromIntegral(length xs)
```

- Benefícios:
  - serve de documentação
  - ajuda a escrever as definições
  - por vezes ajuda a tornar as mensagens de erros mais compreensíveis

Quando usar anotações de tipos II

- Pode ser mais fácil começar com um tipo concreto e depois generalizar
- O interpretador assinala um erro de tipos se a generalização for errada

```
media :: Num a => [a] -> a          -- ERRO
media xs = sum xs / fromIntegral(length xs)

media :: Fractional a => [a] -> a   -- OK
media xs = sum xs / fromIntegral(length xs)
```

Definição de funções

Podemos definir novas funções simples como expressões usando outras funções pré-definidas.

```
minuscula :: Char -> Bool
minuscula c = c>='a' && c<='z'

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

Expressões condicionais

Podemos exprimir uma condição com duas alternativas usando 'if...then...else'.

```
-- valor absoluto: x se x>=0; -x se x<0
absoluto :: Float -> Float
absoluto x = if x>=0 then x else -x
```

As expressões condicionais podem ser imbricadas:

```
sinal :: Int -> Int
sinal x = if x>0 then 1 else
            (if x==0 then 0 else -1)
```

Ao contrário do C ou Java:

- o 'if...then ...else' é uma expressão e não um comando
- a alternativa 'else' é obrigatória

Alternativas com guardas I

Podemos usar guardas em vez de expressões condicionais:

```
absoluto :: Float -> Float
absoluto | x>=0      = x
         | otherwise = -x


sinal :: Int -> Int
sinal x | x>0       = 1
        | x==0      = 0
        | otherwise = -1
```

Alternativas com guardas II

Caso geral:

```
f x y ... | condição 1 = expressão 1
          | condição 2 = expressão 2
            ...
          | condição N = expressão N
```

- As condições são testada pela ordem indicada
- O resultado é dado pela expressão da primeira alternativa verdadeira
- A função é indefinida se nenhuma condição for verdadeira (dá erro durante a execução)
- A condição 'otherwise' é um sinónimo para `True`

Alternativas com guardas III

Definições locais abrangem todas as alternativas se a palavra 'where' estiver alinhada com as guardas.

Exemplo: raizes de uma equação do 2º grau.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
     | delta>0   = [(-b+sqrt delta)/(2*a),
                    (-b-sqrt delta)/(2*a)]
     | delta==0  = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Alternativas com guardas IV

Também podemos definir variáveis locais usando 'let...in...'.
Neste caso o âmbito da definição não inclui outras alternativas.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
     | delta>0   = let r = sqrt delta
                   in [(-b+r)/(2*a),(-b-r)/(2*a)]
        -- r só está definido na expressão acima
     | delta==0  = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Encaixe de padrões I

Podemos usar várias equações com padrões para distinguir casos.

```
not :: Bool -> Bool
not True = False
not False = True

(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

Nota: estas funções estão pré-definidas no prelúdio-padrão.

Encaixe de padrões II

Uma definição alternativa:

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && x = x
```

- O padrão "_" encaixa qualquer valor
- As variáveis no padrão podem ser usadas no lado direito
- A definição acima ignora o segundo argumento se o primeiro for `False`

Encaixe de padrões III

Não podemos repetir variáveis nos padrões:

```
x && x = x                    -- ERRO
_ && _ = False
```

Alternativa: podemos usar uma guarda para impor a condição de igualdade.

```
x && y | x==y = x             -- OK
_ && _        = False
```

Padrões sobre tuplos

Exemplos: as funções `fst` (*first*) e `snd` (*second*) dão-nos o primeiro e segundo elemento de um par.

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

Estas funções também estão pré-definidas no prelúdio-padrão.

Construtor de listas

Qualquer lista é construida acrescentando elementos um-a-um a uma lista vazia usando "`:`"[3].

```
[1, 2, 3, 4]   =   1 : (2 : (3 : (4 : [])))
```

[3] Lê-se "*cons*" de "*construtor*".

Padrões sobre listas I

Podemos também usar um padrão `x:xs` para decompor uma lista.

```
head :: [a] -> a
head (x:_) = x                    -- 1º elemento

tail :: [a] -> [a]
tail (_:xs) = xs                  -- restantes elementos
```

Padrões sobre listas II

O padrão `x:xs` só encaixa listas não-vazias:

```
> head []
```
ERRO

Necessitamos de parêntesis à volta do padrão (porque a aplicação têm maior precedência):

```
head x:_ = x                    -- ERRO

head (x:_) = x                  -- OK
```

Padrões sobre inteiros I

Exemplo: testar se um inteiro é 0, 1 ou 2.

```
small :: Int -> Bool
small 0    = True
small 1    = True
small 2    = True
small _    = False
```

A última equação encaixa todos os casos restantes.

Expressões-case I

Em vez de equações podemos usar ‘case. . . of. . . ’:

Exemplo:

```
null :: [a] -> Bool
null xs = case xs of
              [] -> True
              (_:_) -> False
```

Expressões-case II

Os padrões são tentados pela ordem das alternativas.

Logo, a esta definição é equivalente à anterior:

```
null :: [a] -> Bool
null xs = case xs of
            [] -> True
            _  -> False
```

Expressões-lambda I

Podemos definir uma *função anónima* (i.e. sem nome) usando uma expressão-lambda.

Exemplo:

```
\x -> 2*x+1
```

é a função que a cada x faz corresponder $2x + 1$.

Esta notação é baseada no *cálculo-*λ, o formalismo matemático que é a base teórica da programação funcional.

Expressões-lambda II

Podemos usar uma expressão-lambda aplicando-a a um valor (tal como o nome de uma função).

```
> (\x -> 2*x+1) 1
3

> (\x -> 2*x+1) 3
7
```

Para que servem as expressões-lambda? I

As expressões-lambda permitem definir funções cujos resultados são outras funções.

Em particular: as expressões-lambda permitem compreender o uso de *"currying"* para funções de múltiplos argumentos.

Exemplo:

```
soma x y = x+y
```

é equivalente a

```
soma = \x -> (\y -> x+y)
```

Para que servem as expressões-lambda? II

As expressões-lambda são também úteis para evitar dar nomes a funções curtas.

Um exemplo usando `map` (que aplica uma função a todos os elementos duma lista): em vez de

```
quadrados = map f [1..10]
      where f x = x^2
```

podemos escrever

```
quadrados = map (\x->x^2) [1..10]
```

Operadores e secções I

Qualquer operador binário (+, ∗, etc.) pode ser usado como função de dois argumentos colocando-o entre parentêsis.

Exemplo:

```
> 1+2
3

> (+) 1 2
3

> (++) "Abra" "cadabra!"
"Abracadabra!"
```

Operadores e secções II

- Expressões da forma `(x ⊗)` e `(⊗ x)` chamam-se secções
- Definem a função resultante de aplicar um dos argumentos do operador ⊗

```
> (+1) 2
3

> (/2) 1
0.5

> (++"!!!") "Bang"
"Bang!!!"
```

Exemplos

`(1+)`	sucessor
`(2*)`	dobro
`(^2)`	quadrado
`(1/)`	recíproco
`(++"!!")`	concatenar `"!!"` ao final

Assim podemos re-escrever o exemplo

```
quadrados = map (\x -> x^2) [1..10]
```

de forma ainda mais sucinta:

```
quadrados = map (^2) [1..10]
```