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 I/O 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 lazy functional languages.

(The Haskell 98 Report)

Valores & Expressões

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:

```
>  5
5

>  3.5  +  6.7
10.2

>  2 < 35
True

>  not True
False

>  not ((3.5+6.7) > 23)
True
```


Tipos

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

Informalmente, podemos associar a noção de *"tipo"* à noção de *"conjunto"*, e a noção de *"ter tipo"* à noção de *"pertença"*.

Exemplos:

```
58      :: Int
'a'     :: Char
[3,5,7] :: [Int]
(8,'b') :: (Int,Char)
```

Inteiro
Caracter
Lista de inteiros
Par com um inteiro e um caracter

O Haskell é uma linguagem fortemente tipada, com um sistema de tipos muito evoluído (como veremos). Em Haskell, a verificação de tipos é feita durante a compilação.

Tipos Básicos

Bool Boleanos: `True, False`

Char Caracteres: `'a', 'b', 'A', '1', '\n', '2', ...`

Int Inteiros de tamanho limitado: `1, -3, 234345, ...`

Integer Inteiros de tamanho ilimitado: `2, -7, 75756850013434682, ...`

Float Números de vírgula flutuante: `3.5, -6.53422, 51.2E7, 3e4, ...`

Double Núm. vírg. flut. de dupla precisão: `3.5, -6.5342, 51.2E7, ...`

() *Unit* `()` é o seu único elemento do tipo *Unit*.

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.

Exemplos:

```
(1,5)         :: (Int,Int)
('a',6,True)  :: (Char,Int,Bool)
```

Listas `[T]`

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

Exemplos:

```
[2,5,6,8]          :: [Integer]
['h','a','s']      :: [Char]
[3.5,86.343,1.2]   :: [Float]
```

Funções `T1 -> T2`

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

Exemplos:

```
not :: Bool -> Bool
ord :: Char -> Int
```


Funções

A operação mais importante das funções é a sua **aplicação**.

Se `f :: T1 -> T2` e `a :: T1` então `f a :: T2`

Exemplos:

```
> not True
False :: Bool

> ord 'a'
97 :: Int

> ord 'A'
65 :: Int

> chr 97
'a' :: Char
```

Preservação de Tipos

O tipo de cada expressão é preservado ao longo do processo de cálculo.

Qual será o tipo de chr ?

Novas definições de funções deverão que ser escritas num ficheiro, que depois será carregado no interpretador.

Definições

Uma definição associa um nome a uma expressão. *nome* **=** *expressão*

nome tem que ser uma palavra começada por letra minúscula.

A definição de funções pode ainda ser feita por um conjunto de **equações** da forma:

nome arg1 arg2 ... argn **=** *expressão*

Quando se define uma função podemos incluir *informação sobre o seu tipo*. No entanto, essa informação não é obrigatória.

Exemplos:

```
pi = 3.1415

areaCirc x = pi * x * x

areaQuad = \x -> x*x

areaTri b a = (b*a)/2

volCubo :: Float -> Float
volCubo y = y * y * y
```


Pólimorfismo

O tipo de cada função é inferido automáticamente pelo interpretador.

Exemplo:

Para a função g definida por: `g x = not (65 > ord x)`

O tipo inferido é `g :: Char -> Bool`

Porquê ?

Mas, há funções às quais é possível associar *mais do que um* tipo concreto.

Exemplos:

```
id x = x

nl y = '\n'
```

O que fazem estas funções ?

Qual será o seu tipo ?

O problema é resolvido recorrendo a ***variáveis de tipo***.

Uma variável de tipo representa um tipo qualquer.

```
id :: a -> a

nl :: a -> Char
```

Em Haskell:

- As variáveis de tipo representam-se por nomes começados por letras minúsculas (normalmente a, b, c, ...).
- Os tipos concretos usam nomes começados por letras maiúsculas (ex: Bool, Int, ...).

Quando as funções são usadas, as variáveis de tipos são substituídas pelos tipos concretos adquados.

Exemplos:

```
id True
id 'a'
nl False
nl (volCubo 3.2)
```

```
id :: Bool -> Bool
id :: Char -> Char
nl :: Bool -> Char
nl :: Float -> Char
```


Funções cujos tipos têm variáveis de tipo são chamadas **funções polimórficas**.

Um tipo pode conter diferentes variáveis de tipo.

Exemplo:

```
fst (x,y) = x

fst :: (a,b) -> a
```

Inferência de tipos

O tipo de cada função é inferido automáticamente pelo compilador de Haskell.
O compilador infere sempre o *tipo mais preciso* de qualquer expressão *(o seu tipo principal)*.

É possivel associar a uma função um tipo *mais específico* do que o tipo inferido automaticamente.

Exemplo:

```
seg :: (Bool,Int) -> Int
seg (x,y) = y
```


O Haskell tem um enorme conjunto de definições (que está no módulo **Prelude**) que é carregado por omissão e que constitui a base da linguagem Haskell.

Alguns operadores:

Lógicos: **&&** (e), **||** (ou), **not** (negação)

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

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

Condicional: **if** ... **then** ... **else** ...

:: Bool **:: a**

Exemplo:

```
> if (3>=5) then [1,2,3] else [3,4]
[3,4]
> if (ord 'A' == 65) then 2 else 3
2
```


Módulos

Um programa Haskell está organizado em *módulos*.

Cada **módulo** é uma colecção de funções e tipos de dados, definidos num ambiente fechado.

Um módulo pode exportar todas ou só algumas das suas definições. (...)

module *Nome* (*nomes_a_exportar*) **where**

... definições ...

Ao arrancar o interpretador do GHC, **ghci**, este carrega o módulo **Prelude** (que contém um enorme conjunto de declarações) e fica à espera dos pedidos do utilizador.

ghci

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.2.1, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude>
```

O utilizador pode fazer dois tipos de pedidos ao interpretador **ghci**:

- Calcular o valor de uma expressão.

```
Prelude> 3+5
8
Prelude> (5>=7) || (3^2 == 9)
True
Prelude> fst (40/2,'A')
20.0
Prelude> pi
3.141592653589793
Prelude> aaa

<interactive>:1: Variable not in scope: `aaa'
Prelude>
```

- Executar um comando.
  - Os comandos do **ghci** começam sempre por dois pontos (**:**).
  - O comando **:?** lista todos os comandos existentes

```
Prelude> :?
 Commands available from the prompt:

 ...
```


Alguns comandos úteis:

`:quit` ou `:q` termina a execução do **ghci**.

`:type` ou `:t` indica o tipo de uma expressão.

```
Prelude> :type (2>5)
(2>5) :: Bool
Prelude> :t not
not :: Bool -> Bool
Prelude> :q
Leaving GHCi.
```

`:load` ou `:l` carrega o programa (o módulo) que está num dado ficheiro.

Exemplo: Considere o seguinte programa guardado no ficheiro Temp.hs

Temp.hs

```
module Temp where

celFar c = c * 1.8 + 32

kelCel k = k - 273

kelFar k = celFar (kelCel k)
```

Os programas em Haskell têm normalmente extensão **.hs** (de ***haskell script***)

Depois de carregar um módulo, os nomes definidos nesse módulo passam a estar disponíveis no ambiente de interpretação

```
Prelude> kelCel 300

<interactive>:1: Variable not in scope: `kelCel'
Prelude> :load Temp
Compiling Temp             ( Temp.hs, interpreted )
Ok, modules loaded: Temp.
*Temp> kelCel 300
27
*Temp>
```

Inicialmente, apenas as declarações do módulo Prelude estão no ambiente de interpretação. Após o carregamento do ficheiro Temp.hs, ficam no ambiente todas a definições feitas no módulo Temp e as definições do Prelude.

Um módulo constitui um *componente de software* e dá a possibilidade de gerar bibliotecas de funções que podem ser reutilizadas em diversos programas Haskell.

Exemplo: Muitas funções sobre caracteres estão definidas no módulo **Char** do GHC.

Para se utilizarem declarações feitas noutros módulos, que não o Prelude, é necessário primeiro fazer a sua importação através da instrução:

import *Nome_do_módulo*

Exemplo.hs

```
module Exemplo where

import Char

letra :: Int -> Char
letra n = if (n>=65 && n<=90) || (n>=97 && n<=122)
             then chr n
             else ' '

numero :: Int -> Char
numero n = if (n>=48 && n<=57)
             then chr n
             else ' '
```

Comentários

É possível colocar comentários num programa Haskell de duas formas:

-- O texto que aparecer a seguir a -- até ao final da linha é ignorado pelo interpretador.

{- ... **-}** O texto que estiver entre {- e -} não é avaliado pelo interpretador. Podem ser várias linhas.

```
module Temp where

-- de Celcius para Farenheit
celFar c = c * 1.8 + 32

-- de Kelvin para Celcius
kelCel k = k - 273

-- de Kelvin para Farenheit
kelFar k = celFar (kelCel k)

{- dado valor da temperatura em Kelvin, retorna o triplo com
o valor da temperatura em Kelvin, Celcius e Farenheit -}

kelCelFar k = (k, kelCel k, kelFar k)
```


As funções test e test' são muito parecidas mas há uma diferença essencial:

```
test (x,y) = [ (not x), (y || x), (x && y) ]

test' x  y = [ (not x), (y || x), (x && y) ]
```

Têm tipos diferentes !

A função test recebe **um único** argumento (que é um par de booleanos) e devolve uma lista de booleanos.

```
test :: (Bool,Bool) -> [Bool]

> test (True,False)
```

A função test' recebe **dois** argumentos, cada um do tipo Bool, e devolve uma lista de booleanos.

```
test' :: Bool -> Bool -> [Bool]

> test' True False
```

A função test' recebe um valor de cada vez. Realmente, o seu tipo é:

```
test' :: Bool -> (Bool -> [Bool])

> (test' True) False
```

Mas os parentesis podem ser dispensados !

- **O tipo função associa à direita.**

  Isto é, `f :: T1 -> T2 -> ... -> Tn -> T`

  é uma forma abreviada de escrever

  `f :: T1 -> (T2 -> (... -> (Tn -> T)...))`

- **A aplicação de funções é associativa à esquerda.**

  Isto é, `f x1 x2 ... xn`

  é uma forma abreviada de escrever

  `(...((f x1) x2) ...) xn`

Exercício:

Considere a seguinte declaração das funções fun1, fun2 e fun3.

```
fun1 (x,y) = (not x) || y

fun2 a b = (a||b, a&&b)

fun3 x y z = x && y && z
```

Qual será o tipo de cada uma destas funções ?

Dê exemplos da sua invocação.

Lista e String

[a] é o tipo das listas cujos elementos *são todos* do tipo a .

Exemplos:

```
[2,5,6,8]                    :: [Integer]
[(1+3,'c'),(8,'A'),(4,'d')] :: [(Int,Char)]
[3.5, 86.343, 1.2*5]         :: [Float]
['O','l','a']                :: [Char]
```

Atenção !

```
['A', 4, 3, 'C']

[(1,5), 9, (6,7)]
```

Não são listas bem formadas, porque os seus elementos não têm todos o mesmo tipo !

String O Haskell tem pré-definido o tipo **String** como sendo **[Char]**.

Os valores do tipo String também se escrevem de forma abreviada entre " ".

Exemplo: `"haskell"` é equivalente a `['h','a','s','k','e','l','l']`

```
> "Ola" == ['O','l','a']
True
```


Algumas funções sobre listas definidas no Prelude.

`head :: [a] -> a` dá o primeiro elemento da lista (a cabeça da lista).

`tail :: [a] -> [a]` dá a lista sem o primeiro elemento (a cauda da lista).

`take :: Int -> [a] -> [a]` dá um segmento inicial de uma lista.

`drop :: Int -> [a] -> [a]` dá um segmento final de uma lista.

`reverse :: [a] -> [a]` calcula a lista invertida.

`last :: [a] -> a` dá o último elemento da lista.

Exemplos:

```
Prelude> head [3,4,5,6,7,8,9]
3
Prelude> tail ['a','b','c','d']
['b','c','d']
Prelude> take 3 [3,4,5,6,7,8,9]
[3,4,5]
```

```
Prelude> drop 3 [3,4,5,6,7,8,9]
[6,7,8,9]
Prelude> reverse [3,4,5,6,7,8,9]
[9,8,7,6,5,4,3]
Prelude> last ['a','b','c','d']
'd'
```


Funções sobre String definidas no Prelude.

`words :: String -> [String]` dá a lista de palavras de um texto.

`unwords :: [String] -> String` constrói um texto a partir de uma lista de palavras.

`lines :: String -> [String]` dá a lista de linhas de um texto (i.e. parte pelo *'\n'*).

Exemplos:

```
Prelude> words "aaaa bbbb cccc\tddddd eeee\nffff  gggg hhhh"
["aaaa","bbbb","cccc","ddddd","eeee","ffff","gggg","hhhh"]

Prelude> unwords ["aaaa","bbbb","cccc","ddddd","eeee","ffff","gggg","hhhh"]
"aaaa bbbb cccc ddddd eeee ffff gggg hhhh"

Prelude> lines "aaaa bbbb cccc\tddddd eeee\nffff  gggg hhhh"
["aaaa bbbb cccc\tddddd eeee","ffff  gggg hhhh"]
```

```
Prelude> reverse "programacao funcional"
"lanoicnuf oacamargorp"
```


Listas por Compreensão

Inspirada na forma de definir conjuntos por compreensão em linguagem matemática, a linguagem Haskell tem também mecanismos para definir listas por compreensão.

{ 2x | x ∈ {10,3,7,2} } `[ 2*x | x <- [10,3,7,2] ]` = [20,6,14,4]

{ n | n ∈ {9,8,-2,-10,3} ∧ 0 ≤ n+2 ≤ 10 }

`[ n | n <- [9,8,-2,-10,3] , 0<=n+2, n+2<=10 ]` = [8,-2,3]

{4, 7, ... , 19} `[4,7..19]` = [4,7,10,13,16,19]

`[1..7]` = [1,2,3,4,5,6,7]

{ (x,y) | x ∈ {3,4,5} ∧ y ∈ {9,10} }

`[ (x,y) | x <- [3,4,5], y <- [9,10] ]`

= [(3,9),(3,10),(4,9),(4,10),(5,9),(5,10)]

Listas infinitas

{5,10, ... } `[5,10..]` = [5,10,15,20,25,30,35,40,45,50,55,...

{ x^3 | $x \in \mathbb{N} \wedge par(x)$ }

`[ x^3 | x <- [0..], even x ]` = [0,8,46,216,...

Mais exemplos:

```
Prelude> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Prelude> ['A','C'..'X']
"ACEGIKMOQSUW"
Prelude> [50,45..(-20)]
[50,45,40,35,30,25,20,15,10,5,0,-5,-10,-15,-20]
Prelude> drop 20 ['a'..'z']
"uvwxyz"
Prelude> take 10 [3,3..]
[3,3,3,3,3,3,3,3,3,3]
```


Equações e Funções

Uma função pode ser definida por equações que relacionam os seus argumentos com o resultado pretendido.

Exemplos:

```
triplo x = 3 * x
dobro y = y + y
perimCirc r = 2*pi*r
perimTri x y z = x+y+z
minimo x y = if x>y them y else x
```

As equações definem regras de cálculo para as funções que estão a ser definidas.

nome arg1 arg2 ... argn = expressão

Nome da função
(iniciada por letra minúscula).

Argumentos da função.
Cada argumento é um ***padrão***.
(cada variável não pode ocorrer mais do que uma vez)

O tipo da função é *inferido* tendo por base que ambos os lados da equação têm que ter o mesmo tipo.

Padrões *(patterns)*

Um **padrão** é uma **variável**, uma **constante**, ou um **"esquema" de um valor atómico**
(isto é, o resultado de aplicar construtores básicos dos valores a outros padrões).

No Haskell, um padrão **não** pode ter variáveis repetidas *(padrões lineares)*.

Exemplos:

Padrões	Tipos
`x`	`a`
`True`	`Bool`
`4`	`Int`
`(x,y,(True,b))`	`(a,b,(Bool,c))`
`('A',False,x)`	`(Char,Bool,a)`
`[x,'a',y]`	`[Char]`

Não padrões

```
[x,'a',1]
(4*6, y)
```

Porquê ?

Quando não nos interessa dar nome a uma variável, podemos usar **_** que representa uma *variável anónima nova*.

Exemplos:

```
snd (_,x) = x
segundo (_,y,_) = y
```


Exemplos:

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

outro modo seria

```
soma w z = ((fst w)+z, (snd w)+z)
```

Qual é mais legível ?

```
exemplo :: (Bool,Float) -> ((Float,Int), Float) -> Float
exemplo (True,y) ((x,_),w) = y*x + w
exemplo (False,y) _ = y
```

em alternativa, poderiamos ter

```
exemplo a b = if (fst a) then (snd a)*(fst (fst b)) + (snd b)
                         else (snd a)
```

Redução

O cálculo do valor de uma expressão é feito usando as equações que definem as funções como regras de cálculo.

Uma **redução** é um passo do processo de cálculo (é usual usar o símbolo ⇒ denotar esse passo)

Cada redução resulta de substituir a *instância* do lado esquerdo da equação (o redex) pelo respectivo lado direito (o contractum).

Exemplos: Relembre as seguintes funções

```
triplo x = 3 * x
dobro y = y + y
snd (_,x) = x
nl x = '\n'
```

Exemplos: triplo 7 ⇒ 3*7 ⇒ 21

A instância de (triplo x) resulta da *substituição* [7/x].

snd (9,8) ⇒ 8

A instância de snd (_,x) resulta da *substituição* [9/_,8/x].

A expressão `dobro (triplo (snd (9,8)))` pode reduzir de três formas distintas:

```
dobro (triplo (snd (9,8))) ⇒ dobro (triplo 8)

dobro (triplo (snd (9,8))) ⇒ dobro (3*(snd (9,8)))

dobro (triplo (snd (9,8))) ⇒ (triplo (snd (9,8)))+(triplo (snd (9,8)))
```

A estratégia de redução usada para o cálculo das expressões é uma característica essencial de uma linguagem funcional.

O **Haskell** usa a estratégia ***lazy evaluation*** *(call-by-name)*, que se caracteriza por escolher para reduzir sempre o redex mais externo. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda *(outermost; leftmost)*.

Uma outra estratégia de redução conhecida é a ***eager evaluation*** *(call-by-value)*, que se caracteriza por escolher para reduzir sempre o redex mais interno. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda *(innermost; leftmost)*.

Lazy Evaluation *(call-by-name)*

```
dobro (triplo (snd (9,8))) ⇒ (triplo (snd (9,8)))+(triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (3*(snd (9,8)))
                           ⇒ (3*8) + (3*(snd (9,8)))
                           ⇒ 24 + (3*(snd (9,8)))
                           ⇒ 24 + (3*8)
                           ⇒ 24 + 24
                           ⇒ 48
                           ⇒
```

Com a estrategia *lazy* os parametros das funções só são calculados se o seu valor fôr mesmo necessário.

```
nl (triplo (dobro (7*45)) ⇒ '\n'
```

A *lazy evaluation* faz do Haskell uma linguagem **não estrita**. Esto é, uma função aplicada a um valor indefinido pode ter em Haskell um valor bem definido.

```
nl (3/0) ⇒ '\n'
```

A *lazy evaluation* também vai permitir ao Haskell lidar com *estruturas de dados infinitas*.

Podemos definir uma função recorrendo a várias equações.

Exemplo:

```
h :: (Char,Int) -> Int
h ('a',x) = 3*x
h ('b',x) = x+x
h (_,x) = x
```

Todas as equações têm que ser bem tipadas e de tipos coincidentes.

Cada equação é usada como regra de redução. Quando uma função é aplicada a um argumento, a equação que é selecionada como regra de redução é a 1ª equação (a contar de cima) cujo padrão que tem como argumento **concorda** com o argumento actual (***pattern matching***).

Exemplos:

```
h ('a',5) ⇒ 3*5 ⇒ 15
h ('b',4) ⇒ 4+4 ⇒ 8
h ('B',9) ⇒ 9
```

Note: Podem existir *várias* equações com padrões que concordam com o argumento actual. Por isso, a ordem das equações é importante, pois define uma prioridade na escolha da regra de redução.

O que acontece se alterar a ordem das equações que definem h ?

Funções Totais & Funções Parciais

Uma função diz-se **total** se está definida para todo o valor do seu domínio.

Uma função diz-se **parcial** se há valores do seu domínio para os quais ela não está definida (isto é, não é capaz de produzir um resultado no conjunto de chegada).

Exemplos:

```
conjuga :: (Bool,Bool) -> Bool
conjuga (True,True) = True
conjuga (x,y) = False
```

Função total

```
parc :: (Bool,Bool) -> Bool
parc (True,False) = False
parc (True,x) = True
```

Função parcial

Porquê ?

Tipos Sinónimos

O Haskell pode renomear tipos através de declarações da forma:

type *Nome p1 ... pn* **=** *tipo*

parâmetros (*variáveis de tipo*)

Exemplos:

```
type Ponto = (Float,Float)
type ListaAssoc a b = [(a,b)]
```

Note que não estamos a criar tipos novos, mas apenas nomes novos para tipos já existentes. Esses nomes devem contribuir para a compreensão do programa.

Exemplo:

```
distOrigem :: Ponto -> Float
distOrigem (x,y) = sqrt (x^2 + y^2)
```

O tipo **String** é outro exemplo de um tipo sinónimo, definido no Prelude.

```
type String = [Char]
```


Definições Locais

Uma definição associa um nome a uma expressão.

Todas as definições feitas até aqui podem ser vistas como **globais**, uma vez que elas são visíveis no *módulo* do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração.

Em Haskell há duas formas de fazer definições **locais**: utilizando expressões let ... in ou através de cláusulas where junto da definição equacional de funções.

Exemplos:

```
let  c = 10
     (a,b) = (3*c, f 2)
     f x = x + 7*c
in f a + f b
```

⟹ 242

Porquê ?

```
testa y = 3 + f y + f a + f b
  where  c = 10
         (a,b) = (3*c, f 2)
         f x = x + 7*c
```

```
> testa 5
320

> c
Variable not in scope: `c'

> f a
Variable not in scope: `f'
Variable not in scope: `a'
```

As declarações locais podem ser de funções e de identificadores (fazendo uso de padrões).

Layout

Ao contrário de quase todas as linguagens de programação, o Haskell não necessita de marcas para delimitar as diversas declarações que constituem um programa.

Em Haskell a *identação do texto* (isto é, a forma como o texto de uma definição está disposto), tem um significado bem preciso.

Regras fundamentais:

1. Se uma linha começa mais à frente do que começou a linha anterior, então ela deve ser considerada como a continuação da linha anterior.
2. Se uma linha começa na mesma coluna que a anterior, então elas são consideradas definições independentes.
3. Se uma linha começa mais atrás do que a anterior, então essa linha não pretence à mesma lista de definições.

Ou seja: *definições do mesmo género devem começar na mesma coluna*

Exemplo:

```
exemplo :: Float -> Float -> Float
exemplo x 0 = x
exemplo x y = let a = x*y
                  b = if (x>=y) then x/y
                                else y*x
                  c = a-b
              in (a+b)*cCou
```

Operadores

Operadores infixos como o + , * , && , ... , não são mais do que funções.

Um operador infixo pode ser usado como uma função vulgar (i.e., usando notação prefixa) se estiver entre parentesis.

Exemplo: (+) 2 3 é equivalente a 2+3

Note que
```
(+) :: Int -> Int -> Int
```

Podem-se definir novos operadores infixos.

```
(+>) :: Float -> Float -> Float
x +> y = x^2 + y
```

Funções binárias podem ser usadas como um operador infixo, colocando o seu nome entre ` `.

Exemplo:
```
mod :: Int -> Int -> Int
```

3 `**mod**` 2 é equivalente a mod 3 2

Cada operador tem uma prioridade e uma associatividade estipulada.

Isto faz com que seja possível evitar alguns parentesis.

Exemplo: x + y + z é equivalente a (x + y) + z

x + 3 * y é equivalente a x + (3 * y)

A aplicação de funções tem prioridade máxima e é associativa à esquerda.

Exemplo: f x * y é equivalente a (f x) * y

É possível indicar a prioridade e a associatividade de novos operadores através de declarações.

infixl *num op*
infixr *num op*
infix *num op*

Funções com Guardas

Em Haskell é possível definir funções com alternativas usando **guardas**.

Uma guarda é uma expressão booleana. Se o seu valor for True a equação correspondente será usada na redução (senão tenta-se a seguinte).

Exemplos:
```
sig x y = if x > y then 1
                   else if x < y then -1
                                 else 0
```

é equivalente a

```
sig x y | x > y  = 1
        | x < y  = -1
        | x == y = 0
```

ou a

```
sig x y
    | x > y     = 1
    | x < y     = -1
    | otherwise = 0
```

otherwise é equivalente a True.

Exemplo: Raizes reais do polinómio $a\,x^2 + b\,x + c$

```
raizes :: (Double,Double,Double) -> (Double,Double)
raizes (a,b,c) = (r1,r2)
  where r1 = (-b + r) / (2*a)
        r2 = (-b – r) / (2*a)
        d = b^2 – 4*a*c
        r | d >= 0 = sqrt d
          | d < 0 = error “raizes imaginarias”
```

error é uma função pré-definida que permite indicar a mensagem de erro devolvida pelo interpretador. Repare no seu tipo

```
error :: String -> a
```

```
> raizes (2,10,3)
(-0.320550528229663,-4.6794494717703365)

> raizes (2,3,4)
*** Exception:  raizes imaginarias
```

Listas

[T] é o tipo das listas cujos elementos são todos do tipo T -- *listas homogéneas* .

```
[3.5^2, 4*7.1, 9+0.5] :: [Float]
[(253,"Braga"), (22,"Porto"), (21,"Lisboa")] :: [(Int,String)]
[[1,2,3], [1,4], [7,8,9]] :: [[Integer]]
```

Na realidade, as listas são construidas à custa de dois construtores primitivos:

- a lista vazia **[]**
- o construtor (**:**), que é um operador infixo que dado um elemento x de tipo a e uma lista l de tipo [a], constroi uma nova lista com x na 1ª posição seguida de l.

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

[1,2,3] é uma abreviatura de 1:(2:(3:[])) que é igual a 1:2:3:[]
porque (:) é associativa à direita.

Portanto: [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

Os padrões do tipo lista são expressões envolvendo apenas os construtores **:** e **[]** (*entre parentesis*), ou a representação abreviada de listas.

```
head (x:xs) = x
```

```
tail (x:xs) = xs
```

```
null [] = True
null (x:xs) = False
```

```
soma3 :: [Integer] -> Integer
soma3 [] = 0
soma3 (x:y:z:t) = x+y+z
soma3 (x:y:t) = x+y
soma3 (x:t) = x
```

Qual o tipo destas funções ?

As funções são totais ou parciais?

```
> head [3,4,5,6]
3
> tail "HASKELL"
"ASKELL"
> head []
*** exception
> null [3.4, 6.5, -5.5]
False
> soma3 [5,7]
13
```

Em soma3 a ordem das equações é importante ? Porquê ?

Recorrência

Como definir a função que calcula o comprimento de uma lista ?

Temos dois casos:

- Se a lista fôr vazia o seu comprimento é zero.
- Se a lista não fôr vazia o seu comprimento é um mais o comprimento da cauda da lista.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Esta função é **recursiva** uma vez que se invoca a si própria (aplicada à cauda da lista).

A função termina uma vez que as invocações recursivas são feitas sobre listas cada vez mais curtas, e vai chegar ao ponto em que a lista é vazia.

length [1,2,3] **=** length (1:[2,3])
⇒ 1 + length [2,3]
⇒ 1 + (1 + length [3])
⇒ 1 + (1 + (1 + length []))
⇒ 1 + (1 + (1 + 0))
⇒ 3

Em linguagens funcionais, a recorrência é a forma de obter ciclos.

Mais alguns exemplos de funções já definidas no módulo Prelude:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
last [x] = x
last (_:xs) = last xs
```

```
elem x [] = False
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

```
(++) :: [a] -> [a] -> [a]
[] ++ l = l
(x:xs) ++ l = x : (xs ++ l)
```

Qual o tipo destas funções ?

São totais ou parciais ?

Podemos trocar a ordem das equações ?

Considere a função zip já definida no Perlude:

```
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```

Qual o seu tipo ? É total ou parcial ?
Podemos trocar a ordem das equações ?
Podemos dispensar alguma equação ?
Será que podemos definir zip com menos equações ?

Exercícios:

- Indique todos os passos de redução envolvidos no cálculo da expressão:

  `zip [1,2] "LEI"`

- Defina a função que faz o "zip" de 3 listas.

- Defina a função unzip :: [(a,b)] -> ([a],[b])

Padrões sobre números naturais (Atenção: só no Haskell 98)

O Haskell aceita como um padrão sobre números naturais, expressões da forma:

(*variável* + *número_natural*)

Exemplos:

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

```
decTres (x+3) = x
```

```
> fact 4
24
> fact (-2)
*** Exception: Non-exhaustive patterns in function fact
```

```
> decTres 5
2
> decTres 10
7
> decTres 2
*** Exception: Non-exhaustive ...
```

Atenção: expressões como

(n*5), (x-4) ou (2+n)

não são padrões !

Mais alguma funções sobre listas pré-definidas no Prelude.

```
(x:_) !! 0  = x
(_:xs) !! (n+1) = xs !! n
```

```
init [x] = []
init (x:xs) = x : init xs
```

O que fazem estas funções ?

Qual o seu tipo ?

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

As funções take e drop estão pré-definidas no Prelude da seguinte forma:

```
take                :: Int -> [a] -> [a]
take n _   | n <= 0  = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs
```

```
drop                :: Int -> [a] -> [a]
drop n xs  | n <= 0  = xs
drop _ []            = []
drop n (_:xs)        = drop (n-1) xs
```

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

Defina funções equivalentes utilizando padrões de números naturais.

nome@padrão

nome@padrão é uma forma de fazer uma definição local ao nível de um argumento de uma função.

Exemplos:

A função
```
fun :: (Int,String) -> (Char,(Int,String))
```
pode ser definida, equivalentemente, por:

```
fun (n,(x:xs)) = (x,(n,(x:xs))
```

ou

```
fun par@(n,(x:xs)) = (x,par)
```

ou

```
fun (n,(x:xs)) = let par = (n,(x:xs))
                 in (x,par)
```

```
{- Esta função vai retirando os elementos de uma lista até
encontrar um elemento não positivo -}

dropWhilePos []                      = []
dropWhilePos lis@(x:xs) | x > 0      = dropWhilePos xs
                        | otherwise = lis
```


Funções e listas por compreensão

Pedem-se usar listas por compreensão na definição de funções.

Exemplo: Máximo divisor comum de dois números.

```
divisores n = [ x | x <- [1..n], (n `mod` x) == 0 ]
```

```
divisoresComuns x y = [ n | n <- divisores x, (y `mod` n) == 0 ]
```

```
mdc n m = maximum (divisoresComuns n m)
```


O crivo de Eratosthenes

Esta função deixa ficar numa lista o primeiro elemento e todos os que não são múltiplos desse argumento, repetindo em seguida esta operação para a restante lista.

```
crivo []     = []
crivo (x:xs) = x : (crivo ys)
  where ys = [ n | n <- xs , n `mod` x /= 0 ]
```

A lista dos números primos não superiores a um dado número.

```
primos_ate' x = crivo [2..x]
```

Lista dos números primos.

```
seqPrimos = crivo [2..]
```

Calcular os n primeiros primos.

```
primeirosPrimos n = take n seqPrimos
```


Algoritmos de Ordenação

A ordenação de um conjunto de valores é um problema muito frequente, e muito útil na organização de informação.

Para o problema de ordenação de uma lista de valores, existem diversos algoritmos:

- ***Insertion Sort***
- ***Quick Sort***
- ***Merge Sort***
- ***...***

Vamos apresentar estes algoritmos, para *ordenar uma lista de valores por ordem crescente*, de acordo com os operadores relacionais <, <=, >, e >=
(que implicítamente assumimos estarem definidos para os tipos desses valores).