Programação Funcional CC

Lic. Ciências da Computação - 1º Ano

2006 / 2007

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 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.
- Haskell: the craft of functional programming.
  Simon Thompson. Addison-Wesley.
- A Gentle Introduction to Haskell.
  Paul Hudak, John Peterson and Joseph Fasel.

- Apontamentos da aulas teóricas e fichas práticas.

  Disponíveis em **www.di.uminho.pt/~mjf/PFcc/2006-07**

  Acessíveis a partir de **lcc.di.uminho.pt**

Critérios de Avaliação

A avaliação tem uma **componente teórica** e uma **componente prática**, ambas **obrigatórias**.

Nota Final = 0.4 **NP** + 0.6 **NT**

sendo:

NT a nota teórica (**NT** $\geq$ 9.5, obrigatóriamente), obtida através da realização de uma prova individual escrita;

NP a nota prática (**NP** $\geq$ 9.5, obrigatóriamente), resultante da realização de dois trabalhos e da avaliação do desempenho nas aulas laboratoriais.

Os trabalhos práticos são resolvidos em grupo (de 3 alunos), mas a nota prática é individual.

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
```


Transparência Referencial

- No paradigma funcional, as expressões:
  - são a representação concreta da informação;
  - podem ser associadas a nomes (definições);
  - denotam valores que são determinados pelo interpretador da linguagem.
- No âmbito de um dado contexto, todos os nomes que ocorrem numa expressão têm um valor único e imotável.
- O valor de uma expressão depende *unicamente* dos valores das sub-expressões que a constituem, e essas podem ser substituidas por outras que possuam o mesmo valor.

A esta caracteristica dá-se o nome de **transparência referencial**.

Linguagens Funcionais

- O nome de *linguagens funcionais* advém do facto de estas terem como operações básicas a definição de funções e a aplicação de funções.
- Nas linguagens funcionais as 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.

Isto dá às linguagens funcionais uma **grande flexibilidade**, **capacidade de abstração** e **modularização do processamento de dados**.

- As linguagens funcionais fornecem um alto nivel de abstração, o que faz com que os programas funcionais sejam mais **concisos**, mais **fáceis de entender / manter** e mais **rápidos de desenvolver** do que programas imperativos.
- No entanto, em certas situações, os programas funcionias podem ser mais penalizadores em termos de eficiência.

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**.
- 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 ...)

www.haskell.org

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 **mostra** 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 con 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          Inteiro
'a'     :: Char         Caracter
[3,5,7] :: [Int]        Lista de inteiros
(8,'b') :: (Int,Char)   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 compilaor 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` calcula o primeiro elemento da lista.

`tail :: [a] -> [a]` calcula a lista sem o primeiro elemento.

`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` calcula 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 ∈ ℕ ∧ 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]
```

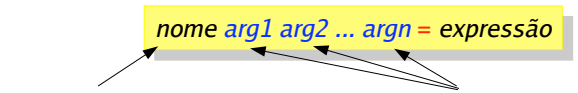

Equações e Funções

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 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	Não padrões
`x`	`a`	`[x,'a',1]`
`True`	`Bool`	`(4*6, y)`
`4`	`Int`	
`(x,y,(True,b))`	`(a,b,(Bool,c))`	
`('A',False,x)`	`(Char,Bool,a)`	
`[x,'a',y]`	`[Char]`	

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 paso)

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)*.