Capítulo 3

Definição de Funções

O módulo `Prelude.hs` contêm um grande número de funções, entre elas temos as funções (operadores) referentes aos diferentes tipos de dados. Este módulo é imediatamente carregado quando se invoca o interpretador, sendo que esse facto é salientado no arranque e pela linha de comando.

```
...
Reading file "/usr/local/lib/hugs/lib/Prelude.hs":
...
Prelude>
```

Para definir novas funções é necessário utilizar um editor de texto para escrever as definições correspondentes num ficheiro, por exemplo `novas.hs` que posteriormente será carregado no interpretador através do comando `:load novas`.

Por exemplo, pode-se escrever um dado ficheiro de texto `novas.hs` contendo as seguintes linhas de texto:

```
quadrado x = x*x
dobro x = 2*x
```

depois torna-se necessário carregar estas novas definições no interpretador através do comando `:l novas`:

```
Prelude> :l novas
Reading file "novas.hs":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
novas.hs
Main> dobro 3
6
Main> quadrado 3
9
Main>
```

[1](Versão 1.4)

A mudança da linha de comando reflecte o facto de o módulo `Main` ser o módulo em que, por omissão, são feitas as definições. Ou seja a menos que o nome de um novo módulo seja explicitamente definido o módulo em que são feitas as definições "soltas" é o módulo `Main`.

3.1 Definições Simples

A definição de uma função passa pela escrita de uma "equação" que determine o seu comportamento. A sintaxe é a seguinte:

$$< nome\_da\_função > < argumento > \texttt{=} < expressão >$$

por exemplo

```
cubo x = x*x*x
```

A definição de funções deve ser entendida como uma regra de re-escrita da esquerda para a direita, a qual está implicitamente quantificada universalmente nas suas variáveis. Ou seja a leitura da definição de `cubo` é a seguinte:

$$\forall_{x \in T}\ cubo\ x \rightarrow x \times x \times x$$

com T um dos tipos da linguagem *Haskell*, e $\rightarrow$ o operador de redução.

A avaliação de uma expressão que envolva a função `cubo` passa pela redução da expressão à sua forma normal, utilizando, entre outras, a regra de redução escrita acima. Por exemplo:

```
Main> cubo 3
27
```

teve como passos da redução $cubo\ 3 \rightarrow (3 \times 3) \times 3 \rightarrow 9 \times 3 \rightarrow 27$.

Qual é o tipo da função `cubo`

```
Main> :t cubo
cubo :: Num a => a -> a
```

A resposta aparentemente confusa tem a seguinte leitura, `cubo` é uma função de domínio A e co-domínio A (`a -> a`) em que A é um qualquer tipo numérico (`Num a =>`). Ou seja `cubo` é uma função polimórfica cujo tipo (`a->a`) pertence à classe dos tipos numéricos (`Num a`). A definição de uma instância concreta para a função `cubo` só acontece no momento da avaliação.

Um outro ponto que importa salientar é que as funções em *Haskell* têm um, e um só, argumento. Como proceder então para definir uma função a aplicar a dois argumentos?

Existem duas soluções possíveis: Em primeiro podemos agrupar os argumentos num n-uplo. Por exemplo:

```
soma1 :: (Int,Int) -> Int
soma1 (x,y) = x + y
```

A outra solução é dada pela definição de uma função *curry*.

```
soma2 :: Int -> (Int -> Int)
soma2 x y = x + y
```

Estas duas definições são equivalentes dado serem a expressão da propriedade universal das funções. É de relembrar que a aplicação de funções associa à esquerda, ou seja `soma x y` deve ler-se como `(soma x) y`.

A vantagem da utilização de funções *curry* em detrimento das funções *não-curry* é dupla: por um lado utilizam-se argumentos não estruturados tornando deste modo a escrita mais simples; por outro lado ao aplicar-se uma função *curry* a um dos seus argumentos obtêm-se uma outra função que pode por sua vez ser um objecto interessante.

3.2 Definições Condicionais

A definição de uma função através de uma única equação não nos permite definir funções por ramos. Temos duas possibilidades para o fazer: através de expressões condicionais.

```
menor1 :: (Int,Int) -> Int
menor1 (x,y) = if x <= y then x else y
```

ou então através de equações com "guardas".

```
menor2 :: (Int,Int) -> Int
menor2 (x,y)
   | x <= y  = x
   | x > y   = y
```

As duas definições são equivalentes, no segundo caso as "guardas" são avaliados e a primeira a tomar o valor lógico de verdade determina o valor da função.

A segundo definição podia ser re-escrita do seguinte modo:

```
menor3 :: (Int,Int) -> Int
menor3 (x,y)
   | x > y = y
   | otherwise = x
```

dado que o valor lógico de `otherwise` é *Verdade*, então este padrão determina o valor da função para todos os casos em que os outros ramos que o precedem não tenham o valor lógico de *Verdade*.

3.3 Definições Recursivas

A definição de funções recursivas em *Haskell* não só é possível, como a sua sintaxe é muito simples, e segue de perto a sintaxe matemática. Por exemplo a definição da função factorial é dada por:

```
fact :: Integer -> Integer
fact n
   | n == 0 = 1
   | n >  0 = n*fact(n-1)
```

Na definição desta função levanta-se uma questão: e no caso em $n < 0$?

Para os valores de $n < 0$ a função factorial não está definida, ela é uma função parcial no conjunto dos inteiros, torna-se então necessário saber lidar com funções parciais. Em *Haskell* a forma de lidar com funções parciais passa por especificar situações de erro.

```
fact1 :: Integer -> Integer
fact1 n
   | n < 0  = error "A função fact não está definida para n<0"
   | n == 0 = 1
   | n > 0  = fact1(n-1)*n
```

No entanto, e dado o tipo da função `fact1`, a função `error` tem de ser `String -> Integer`. O tipo da função `error` é na verdade `String -> A`, com `A` um tipo *Haskell*, ou seja `error` é uma função polimórfica cujo tipo do co-domínio depende da situação em que é aplicada.

O mesmo se passa com a função `‘*’`, o seu tipo é `A -> A -> A`, sendo depois instanciada para uma dada função concreta no momento da aplicação.

3.3.1 O uso de Acumuladores

Embora a definição recursiva de funções seja muito apelativa, o seu uso revela-se computacionalmente pesado. Vejamos um exemplo disso utilizando a definição acima escrita da função factorial, e vejamos o desenvolvimento do cálculo de `fact 4`.

```
fact 4
-> 4*(fact 3)
     -> 3*(fact 2)
          -> 2*(fact 1)
               -> 1*(fact 0)
                    -> 1
                    <- 1
               <- 1*1
          <- 2*1
     <- 3*2
<- 4*6
24
```

Temos então uma primeira fase em que há um simples desdobrar da definição de `fact`, e uma segunda fase em que são executados os cálculos. Este forma de proceder tem como consequência o criar de uma expressão que vai crescendo à medida que o valor do argumento cresce, sendo que os cálculos são feitos depois desta fase de desdobramento.

Uma outra forma de escrever a definição de `fact` faz uso de um acumulador, isto é, de uma variável auxiliar que serve para ir guardando os resultados parciais. Vejamos como proceder:

```
fact2 :: Int -> Int
fact2 n
   | n < 0 = error "A função fact não está definida para n<0"
   | otherwise = factac 1 n
```

```
factac :: Int -> Int -> Int
factac ac n
   | n==0 = ac
   | otherwise = factac ac*n (n-1)
```

Para o calcular `fact2 4` temos agora:

```
fact2 4
-> factac 1 4
   -> factac (1*4) 3
      -> factac (4*3) 2
         -> factac (12*2) 1
            -> factac (24*1) 0
            <- 24
         <- 24
      <- 24
   <- 24
<- 24
```

ou seja, na nova definição os cálculos são feitos imediatamente, não sendo necessário esperar pelo desdobrar de toda a definição.

Um exemplo extremos da diferença de eficiência entre os dois tipos de definição é dado pela implementação da função de *Fibonnacci*.

A definição matemática é nos dada pela seguinte expressão:

$$Fib(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ Fib(n-1) + Fib(n-2), & n \geq 2 \end{cases}$$

Uma possível implementação recursiva (clássica) desta função é dada por:

```
fib :: Int -> Int
fib n
  | n==0 = 1
  | n==1 = 1
  | otherwise = fib(n-1) + fib(n-2)
```

Uma implementação recursiva, recorrendo a acumuladores é dada por:

```
fib1 :: Int -> Int
fib1 n = fibac 1 1 n

fibac :: Int -> Int -> Int -> Int
fibac ac1 ac2 n
  | n==0 = ac2
  | n==1 = ac2
  | otherwise = fibac ac2 (ac1+ac2) (n-1)
```

Para nos convencer-mo-nos da grande diferença de eficiência entre as duas definições, podemos calcular o valor de $Fib(30)$, através das duas implementações acima descritas, antes de o fazer vai-se modificar o comportamento do interpretados através do comando `:set +s`.

```
Fact> fib 35
14930352
(281498375 reductions, 404465770 cells, 4321 garbage collections)
Fact>
Fact> fib1 35
14930352
(358 reductions, 608 cells)
Fact>
```

Escusado será dizer que a diferença de tempo (real) entre os dois cálculos foi muito significativa.

3.4 Definições Polimórficas

A definição de funções polimórficas em *Haskell* é possível dado existirem variáveis de tipo, isto é variáveis cujo domínio de variação é o conjunto de todos os tipos em *Haskell*.

Podemos então re-definir a função factorial como uma função polimórfica capaz de aceitar argumentos dos tipos `Int`, `Integer` e mesmo do tipo `Float`.

```
fact2 :: (Ord a, Num a) => a -> a
fact2 n
   | n < 0  = error "A função fact não está definida para n<0"
   | n == 0 = 1
   | n > 0  = fact2(n-1)*n
```

Ou seja a função `fact2` é uma função polimórfica cujo tipo é `A -> A`, com o tipo `A` pertencente à classe dos tipos numéricos e com ordenação.

Outro exemplo

```
identidade :: a -> a
identidade x = x
```

3.5 Associação de Padrões

Em alguns casos na definição de uma função por ramos, os diferentes casos a considerar são reduzidos. Por exemplo na definição da função de procura de um dado elemento numa lista. Existem dois casos diferentes a considerar: ou a lista está vazio, ou não. No último caso ou o elemento é igual à cabeça da lista, ou temos que considerar a procura no resto da lista. Temos então

```
procura :: (a,[a]) -> Bool
procura (elem,l) = if l==[] then
                      False
```

```
                else if elem == head(l) then
                        True
                     else
                        procura(elem,tail(l))
```

com `head` e `tail` duas funções pré-definidas no módulo `Prelude` com o significado óbvio. No entanto a definição de `procura` pode ficar mais clara e concisa se se usar os construtores próprios de uma lista de uma forma explícita.

```
procura1 (elem,[]) = False
procura1 (elem,hd:tl) = if elem == hd then
                           True
                        else
                           procura(elem,tl)
```

Nesta definição usou-se a técnica de *associação de padrões* ("pattern matching").

A associação de padrões está ligada aos tipos de dados definidos, por utilização dos seus construtores. No caso do tipo lista temos dois casos, padrões, a explorar: uma lista ou é a lista vazia; ou é uma lista constituída por um elemento (a cabeça da lista) e por uma lista (a cauda da lista). Os construtores definidos são `[]`, para a lista vazia, e `:` para a junção de um elemento a uma lista.

Sendo assim a definição de uma função cujo domínio de definição seja o tipo lista terá de explorar esses dois casos, definindo desse modo duas equações, uma para cada um dos padrões possíveis. Uma para o padrão `[]`,

```
procura1 (elem,[]) = False
```

e outra para o padrão `:`,

```
procura1 (elem,hd:tl) = ...
```

É importante notar que as equações (e por isso os padrões) são lidos sequencialmente e como tal o primeiro "encaixe" define qual das equações é que se deve aplicar.

Por exemplo, qual das definições seguintes é a correcta, e porquê?

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

ou

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

Como fica claro no exemplo anterior os padrões possíveis também se aplicam aos construtores 0-ários, ou seja aos valores concretos de um dado tipo.

Em alguns casos o valor de um padrão, ou de parte dele, não é importante para a definição da função, nesses casos é possível definir o *padrão livre* "`_`" ("wildcard pattern"). Por exemplo na definição da função que serve para determinar se uma dada lista está vazia ou não.

```
estavazia :: [a] -> Bool
estavazia [] = True
estavazia _  = False
```

Para o caso em que a lista não é vazia, isto é não houve encaixe com o padrão `[]`, os valores da cabeça e da cauda da lista não são importantes, então basta colocar o padrão livre para completar a definição. O padrão livre encaixa com qualquer valor, como tal terá de ser sempre o último caso a ser considerado.

3.5.1 A construção `case`

A utilização da técnica de associação de padrões pode ser alargada a outros valores que não os argumentos das funções. Suponhamos que, dado uma lista de expressões numéricas simples dadas sob a forma de uma sequência de caracteres contendo sempre um só sinal de operação, queremos calcular cada um dos valores associados às diferentes expressões contidas na lista obtendo deste modo a lista de todos os resultados.

Uma forma elegante de resolver este problema seria o seguinte:

```
calc :: [String] -> [Int]
calc [] = []
calc (hd:tl) =
    case operador(hd) of
             '+' -> (operando1(hd) + operando2(hd)):calc tl
             '*' -> (operando1(hd) * operando2(hd)):calc tl
             '-' -> (operando1(hd) - operando2(hd)):calc tl
             '/' -> (operando1(hd) `div` operando2(hd)):calc tl
             _   -> error "Operador não considerado"
```

com `operador`, `operando1`, e `operando2` funções de reconhecimento para as várias componentes das expressões simples.

Como vemos no exemplo a associação de padrões é, neste caso, feita sobre o resultado de uma expressão e não directamente sobre os argumentos da função.

A forma genérica da construção `case` é a seguinte:

```
case e of
  p1 -> e1
  p2 -> e2
  ...
  pn -> en
```

com os e_is expressões e os p_is padrões.

3.6 Definições Locais

Em muitas situações o cálculo do valor de uma função depende de um determinado cálculo auxiliar. Embora se possa desenvolver esse cálculo auxiliar de uma forma independente, e depois usá-lo aquando da avaliação da função essa não é, em geral, a solução pretendida dado a utilidade do cálculo auxiliar se esgotar no cálculo do valor da função. Por exemplo na definição da função que calcula as raízes reais de uma equação do segundo grau.

```
raizes_reais :: (Float,Float,Float) -> (Float,Float)
raizes_reais(a,b,c) = if b^2-4*a*c>=0 then
     ((-b+sqrt(b^2-4*a*c))/(2*a),(-b-sqrt(b^2-4*a*c))/(2*a))
                     else
     error "Raízes Imaginárias"
```

embora correcto esta definição é "desajeitada"; o cálculo do binómio discriminante é feito por três vezes; a definição da função resulta pouco clara.

A seguinte definição resolve esses problemas

```
raizes_reais1 :: (Float,Float,Float) -> (Float,Float)
raizes_reais1 (a,b,c)
         | delta < 0  = error "Raízes Imaginárias"
         | delta >= 0 = (r1,r2)
         where d = 2*a
               delta = b^2-4*a*c
               r1 = (-b+sqrt(delta))/d
               r2 = (-b-sqrt(delta))/d
```

De uma forma geral tem-se

<*definição de função*> `where` <*definições locais*>

O contexto das definições locais em `where ...` é a expressão no lado direito da definição de uma dada função, restritas à equação (condicional) em que aparecem.

As definições que se seguem ao identificador `where` não são visíveis no exterior da definição.

Este exemplo serve também para ilustrar a técnica da avaliação de expressões usada em Hashell, e que é designada por "lazy evaluation", avaliação preguiçosa numa tradução literal, e que eu vou traduzir por avaliação a pedido.

Vejamos em que consiste a avaliação a pedido, e em que isso interfere na definição que acabamos de escrever.

Numa linguagem em que a avaliação é forçada ("strict evaluation") o cálculo das expressões:

```
               r1 = (-b+sqrt(delta))/d
               r2 = (-b-sqrt(delta))/d
```

estaria errada para todos os valores de `a`, `b` e `c` que tornem `delta` negativo. Ou seja numa linguagem em que todas as componentes de uma expressão têm de ser avaliadas antes de ser calculado o valor final da função `raizes_reais1` está errada.

Em contrapartida em *Haskell* uma expressão só é avaliada quando é necessário. Ora o cálculo de `r1` (e de `r2`) só é necessário quando `delta >= 0`. Só nessa situação é que o valor de `r1` (e de `r2`) é calculado e como tal a definição de `raizes_reais1` está correcta.

Outra forma de escrever definições locais a uma função passa pela utilização do mecanismo `let ... in ...`. Podemos então re-escrever a definição da função `raizes_reais` da seguinte forma:

```
raizes_reais2 :: (Float,Float,Float) -> (Float,Float)
raizes_reais2 (a,b,c)
            | b^2-4*a*c < 0 = error  error "Raízes Imaginárias"
            | b^2-4*a*c >= 0 = let
                               d = 2*a
                               delta = b^2-4*a*c
                               r1 = (-b+sqrt(delta))/d
                               r2 = (-b-sqrt(delta))/d
                              in (r1,r2)
```

De uma forma geral tem-se

`let` <*definições_locais*> `in` <*expressão*>

Temos então que o contexto das definições locais é dado pela expressão que se segue ao identificador `in`.

3.7 Funções de Ordem-Superior

Numa linguagem funcional as funções são objectos de primeira ordem, isto é, podem ser manipuladas tais como outros quaisquer objectos, nomeadamente podemos ter funções como argumentos de outras funções, assim como funções como resultado de outras funções. Um exemplo do que se acabou de dizer é-nos dado pela composição de funções, em *Haskell* podemos escrever `f.g` para deste modo obtermos a função composição de `f` com `g` (f após g).

Se num dos interpretadores de *Haskell* normalmente usados fizermos:

```
Main> :type (.)
```

obtemos como resposta o tipo da função composição:

```
forall b c a. (b -> c) -> (a -> b) -> a -> c
```

ou seja, estamos perante uma função que recebe duas funções componíveis (vejam-se os tipos das funções) como argumento e que produz como resultado a função que é a composição das duas primeiras.

As funções que manipulam outras funções são designadas por funções de ordem-superior, ou mais usualmente, funcionais. Dentro destas é importante destacar algumas que pela sua utilidade são muito usadas em programação funcional, são elas:

map funcional que aplica uma dada função a todos os elementos de uma estrutura (em geral uma lista).

No Haskell temos a função `map` cujo tipo e definição são os seguintes:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (hd:tl) = f hd : map f tl
```

Como exemplo podemos ter uma função que adiciona 2 a todos os elementos de uma dada lista `mapm2 l = map (+2) l`

folding funcional que aplica de forma cumulativa uma dada função binária aos elementos de uma lista.

No Haskell temos a função `foldr1` cujo tipo e definição são os seguintes:

```
foldr1 ::  (a -> a -> a) -> [a] -> a
foldr1 f [a] = a
foldr1 f (a:b:tl) = f a (fold f (b:tl))
```

Duas notas em relação a esta definição:

- pela definição da função `foldr1` fica claro que a as sucessivas aplicações da operação são feitas pela direita, o que implica um operador associativo à direita. Existe uma versão que faz a associação pela esquerda, é ela a função `foldl1`.
- a definição só está bem definida para listas não vazias. Se queremos considerar também este caso temos de dar mais um argumento, o qual especifica qual é o valor final no caso da lista ser vazia, as funções que implementam este caso são as funções `foldl` e `foldr` cujo tipo é: `forall a b.  (a -> b -> b) -> b -> [a] -> b`

Um exemplo de uma aplicação deste funcional é-nos dado pela implementação dos somatórios, `somatório l = foldr (+) 0 l`.

Filtros funcionais que aplicam um dado predicado a todos os elementos de uma estrutura (em geral uma lista) seleccionando deste modo todos os elementos que satisfazem um dado critério.

No Haskell temos a função `filter` cujo tipo e definição são os seguintes:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (hd:tl)
    | p a = a : filter p tl
    | otherwise = filter p tl
```

Como exemplo de aplicação deste funcional temos a função `impares l = filter odd l` a qual nos dá todos os impares de uma dada lista de inteiros.