PROGRAMANDO EN HASKELL

Capítulo 4 - Definiendo Funciones

Expresiones Condicionales

Como en la mayoría de lenguajes de programación, las funciones pueden definirse usando expresiones condicionales.

```
abs :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs toma un entero n y regresa n si no es negativo, de lo contrario regresa -n.

Las expresiones condicionales pueden anidarse:

```
signum :: Int → Int
signum n = if n < 0 then -1 else
                 if n == 0 then 0 else 1
```

Nota:

- En Haskell, las expresiones condicionales siempre deben tener una rama else, lo que evita posibles problemas de ambigüedad con condicionales anidadas.

Ecuaciones con Guardas

Como alternativa a las condicionales, las funciones también pueden definirse con ecuaciones con guardas.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

abs, usando ecuaciones con guardas.

Las ecuaciones con guardas también pueden ser usadas para hacer mas legibles las definiciones que involucran múltiples condiciones:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Nota:

- otherwise, la condición que atrapa todo, está definida en la librería como otherwise = True.

Emparejando Patrones

Muchas funciones tienen particularmente una clara definición si se busca emparejar patrones sobre sus argumentos.

```
not :: Bool → Bool
not False = True
not True  = False
```

not mapea False a True, y True a False.

A menudo las funciones pueden ser definidas de diferentes formas mediante el emparejamiento de patrones. Ejemplo

```
(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

puede ser definida compactamente por

```
True && True = True
_    && _    = False
```

Sin embargo, la siguiente definición es mas eficiente, porque evita evaluar el segundo argumento si el primer argumento es False:

```
True  && b = b
False && _ = False
```

Nota:

- El guión bajo _ es un patrón comodín que empareja cualquier valor de argumento.

- Los patrones se emparejan en orden. Por ejemplo, la siguiente definición siempre regresa False:

```
_     && _     = False
True && True = True
```

- Los patrones no deberán repetir variables. Por ejemplo, la siguiente definición da un error:

```
b && b = b
_ && _ = False
```

Patrones de Lista

Internamente, cada lista no vacía se construye con el uso repetido de un operador (:) llamado "cons" que añade un elemento al inicio de una lista.

```
[1,2,3,4]
```

Significa 1:(2:(3:(4:[]))).

Las funciones sobre listas pueden definirse usando patrones x:xs.

```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

head y tail mapean cualquier lista no vacía a su primer elemento y a su restante.

Nota:

- Los patrones x:xs solo emparejan listas no vacías:

```
> head []
*** Exception: empty list
```

- Los patrones x:xs deben llevar paréntesis, porque la aplicación tiene prioridad sobre (:). Ejemplo, la siguiente definición da un error:

```
head x:_ = x
```

Expresiones Lambda

Las funciones pueden construirse sin ser nombradas, al utilizar expresiones lambda.

$$\lambda x \rightarrow x + x$$

la función anónima que toma un número 'x' y regresa el resultado x + x.

Nota:

- El símbolo λ es la letra Griega lambda, y se escribe en el teclado como una diagonal \.

- En matemáticas, las funciones anónimas usualmente se denotan con el símbolo $\mapsto$, ejemplo, $x \mapsto x + x$.

- En Haskell, el uso del símbolo λ para funciones anónimas viene del cálculo lambda, la teoría de funciones sobre el cual se basa Haskell.

¿Por qué son útiles las expresiones lambda?

Las expresiones lambda se pueden utilizar para dar un significado formal a las funciones currificadas.

Por ejemplo:

```
add :: Int → Int → Int
add x y = x + y
```

significa

```
add :: Int → (Int → Int)
add = λx → (λy → x + y)
```

Las expresiones lambda pueden ser usadas para evitar el nombrado de funciones que solo serán referidas una sola vez.

Por ejemplo:

```
odds n = map f [0..n-1]
         where
             f x = x*2 + 1
```

puede simplificarse a

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

Secciones de Operador

Un operador escrito entre sus dos argumentos puede convertirse a una función currificada escrita antes de sus dos argumentos utilizando paréntesis.

Por ejemplo:

```
> 1+2
3

> (+) 1 2
3
```

Esta convención también permite que uno de los argumenos del operador sea incluído dentro de los paréntesis.

Por ejemplo:

```
> (1+) 2
3

> (+2) 1
3
```

En general, si $\oplus$ es un operador entonces las funciones de la forma ($\oplus$), (x$\oplus$) y ($\oplus$y) se llaman secciones.

¿Por Qué Son Útiles Las Secciones De Operador?

Algunas veces, se pueden lograr funciones útiles de una forma simple utilizando secciones. Por ejemplo:

`(1+)` - sucesor de un número

`(1/)` - recíproco de un número

`(*2)` - el doble de un número

`(/2)` - la mitad de un número

Ejercicios

(1) Considera una función safetail que se comporta igual que tail, excepto que safetail mapea la lista vacía a la lista vacía, mientras tail para este caso da un error. Define safetail con:

   (a) una expresión condicional;
   (b) ecuaciones con guardas;
   (c) emparejado de patrones.

   Pista: la función null :: [a] $\rightarrow$ Bool en la librería, puede utilizarse para checar si una lista está vacía.

(2) Da tres definiciones posibles para el operador lógico or (||) usando emparejado de patrones.

(3) Redefine la siguiente versión de (&&) usando condicionales en lugar de patrones:

```
True && True = True
_    && _    = False
```

(4) Haz lo mismo con la siguiente versión:

```
True  && b = b
False && _ = False
```

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com