Guía Rápida de Haskell

Esta guía rápida abarca los elementos fundamentales del lenguaje Haskell: sintaxis, palabras clave y otros elementos. Se presenta como un archivo ejecutable de Haskell y también como un documento para impresión. Cargue la fuente en su intérprete favorito para jugar con los ejemplos de código mostrados.

Sintaxis Básica

Comentarios

Un comentario de una sola línea comienza con '`--`' y se extiende hasta el final de la línea. Los comentarios de varias líneas comienzan con '`{-`' y se extienden hasta '`-}`'. Los comentarios pueden ser anidados.

Los comentarios antes de las definiciones de función deben comenzar con '`{- |`' y los que están junto a los tipos de parámetros con '`-- ^`' para que sean compatibles con Haddock, un sistema para documentar código en Haskell.

Palabras Reservadas

Las siguientes palabras están reservadas para Haskell. Es un error de sintaxis darle a una variable o a una función uno de estos nombres.

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

Cadenas

- `"abc"` – Cadena Unicode string, azúcar sintáctica de `['a','b','c']`.
- `'a'` – Un solo caracter.

Cadenas en varias líneas Normalmente, es un error de sintaxis si una cadena contiene caracteres de fin de línea. Eso es, esto es un error de sintaxis:

```
string1 = "My long
string."
```

Se pueden emplear barras diagonales inversas ('\') para hacer "escape" de un fin de línea:

```
string1 = "My long \
\string."
```

El área entre las barras diagonales inversas es ignorada. Los fines de línea *en* la cadena deben ser representados explícitamente:

```
string2 = "My long \n\
\string."
```

Eso es, `string1` evalúa como

```
My long string.
```

Mientras `string2` evalúa como:

```
My long
string.
```

Códigos de escape Los siguientes códigos de escape pueden ser utilizados en caracteres o cadenas:

- `\n`, `\r`, `\f`, etc. – Los códigos estándar para fin de línea, retorno de carro, avance de línea, etc.
- `\72`, `\x48`, `texto110` – Un caracter con el valor 72 en decimal, hexadecimal y octal, respectivamente.
- `\&` – El caracter de escape "null", es utilizado para que los códigos de escape numéricos puedan aparecer junto de las literales numéricas. Es equivalente a "" y por lo tanto no puede ser utilizado en literales de caracter.

Números

- `1` – Entero o valor de punto flotante.
- `1.0, 1e10` – Valor de punto flotante.
- `0o1, 0O1` – Valor octal.
- `0x1, 0X1` – Valor hexadecimal.
- `-1` – Número negativo; el signo de menos ("-") no puede ir separado del número.

Enumeraciones

- `[1..10]` – Lista de números – `1, 2, ..., 10`.
- `[100..]` – Lista infinita de números – `100, 101, 102, ...` .
- `[110..100]` – Lista vacía; los rangos solamente avanzan hacia adelante.
- `[0, -1 ..]` – Enteros negativos.
- `[-110..-100]` – Error de sintaxis; necesita `[-110.. -100]` por los negativos.
- `[1,3..99], [-1,3..99]` – Lista de 1 a 99, de 2 en 2; y de -1 a 99, de 4 en 4.

jgbailey@codeslower.com

De hecho, se puede usar cualquier valor que esté en la clase `Enum`:

- `['a' .. 'z']` – Lista of caracteres – `a, b, ..., z`.
- `[1.0, 1.5 .. 2]` – `[1.0,1.5,2.0]`.
- `[UppercaseLetter ..]` – Lista de valores `GeneralCategory` (en `Data.Char`).

Listas & Tuplas

- `[]` – Lista vacía.
- `[1,2,3]` – Lista de tres números.
- `1 : 2 : 3 : []` – Forma alterna de escribir listas usando "cons" (`:`) y "nil" (`[]`).
- `"abc"` – Lista de tres caracteres (las cadenas son listas).
- `'a' : 'b' : 'c' : []` – Lista de caracteres (lo mismo que `"abc"`).
- `(1,"a")` – Tupla de dos elementos, un número y una cadena.
- `(head, tail, 3, 'a')` – Tupla de cuatro elementos, dos funciones, un número y un caracter.

Regla "Layout", llaves y punto y comas

Se puede escribir Haskell utilizando llaves y punto y comas, igual que en C. Sin embargo, nadie lo hace. En lugar de eso se emplea la regla "layout", donde se emplea espacio en blanco para separar contextos. La regla general es: siempre usar sangrías. Cuando el compilador se queje, usar más.

Llaves y punto y comas Los paréntesis finalizan una expresión, y las llaves representan contextos. Pueden ser utilizados después de varias palabras clave: `where`, `let`, `do` y `of`. No pueden ser utilizados al definir el cuerpo de una función. Por ejemplo, esto no compila:

```
square2 x = { x * x; }
```

Sin embargo, esto funciona bien:

```
square2 x = result
    where { result = x * x; }
```

Definición de funciones Aplique una sangría de al menos un espacio a partir del nombre de la función:

```
square x  =
  x * x
```

A menos que esté presente una cláusula `where`. En ese caso, aplique la sangría de la cláusula `where` al menos un espacio a partir del nombre de la función y todos los cuerpos de la función al menos a un espacio a partir de la palabra clave `where`:

```
square x =
    x2
  where x2 =
    x * x
```

Let Aplique sangría sobre el cuerpo del `let` al menos un espacio a partir de la primera definición en el `let`. Si el `let` aparece por sí solo en una línea, el cuerpo de cualquier definición debe aparecer en la columna después del `let`:

```
square x =
  let x2 =
        x * x
  in x2
```

Como se puede ver arriba, la palabra clave `in2` debe también estar en la misma columna que el `let`. Finalmente, cuando se van múltiples definiciones, todos los identificadores deben aparecer en la misma columna.

Declaraciones, Etc.

La siguiente sección describe las reglas para la declaración de funciones, las listas por comprensión, y otras áreas del lenguaje.

Definición de Funciones

Las funciones se definen declarando su nombre, los argumentos, y un signo de igual:

```
square x = x * x
```

Todos los nombres de función deben comenzar con letra minúscula o "_". Es un error de sintaxis de cualquier otra forma.

Comparación de patrones Se pueden definir varias "cláusulas" de una función haciendo "comparación de patrones" en los valores de los argumentos. Aquí, la función `agree` tiene cuatro casos separados:

```
-- Coincide cuando se da la cadena "y".
agree1 "y" = "Great!"
-- Coincide cuando se da la cadena "n".
agree1 "n" = "Too bad."
-- Coincide cuando se da una cadena que
-- comienza con 'y'.
agree1 ('y':_) = "YAHOO!"
-- Coincide con cualquier otro valor.
agree1 _ = "SO SAD."
```

Nótese que el caracter '_' es un comodín y coincide con cualquier valor.

La comparación de patrones se puede extender a valores anidados. Asumiendo esta declaración de dato:

```
data Bar = Bil (Maybe Int) | Baz
```

y recordando la definición de `Maybe` de la página 8 podemos hacer coincidir en valores `Maybe` anidados cuando `Bil` está presente:

```
f (Bil (Just _)) = ...
f (Bil Nothing) = ...
f Baz = ...
```

La comparación de patrones también permite que valores sean asociados a variables. Por ejemplo, esta función determina si la cadena dada es o no vacía. Si no, el valor asociado a `str` es convertido a minúsculas:

```
toLowerStr [] = []
toLowerStr str = map toLower str
```

Nótese que aquí `str` es similar a `_` en que va a coincidir con lo que sea; la única diferencia es que al valor que coincide también se le da un nombre.

$n+k$ Patterns Esta (a veces controversial) comparación de patrones hace fácil coincidir con ciertos tipos de expresiones numéricas. La idea es definir un caso base (la porción "n") con un número constante para que coincida, y después definir las coincidencias (la porción "k") como sumas sobre el caso base. Por ejemplo, esta es una forma ineficiente de determinar si un número es o no par:

```
isEven 0 = True
isEven 1 = False
isEven (n + 2) = isEven n
```

Captura de Argumentos La captura de argumentos es útil para comparar un patrón y utilizarlo, sin declarar una variable extra. Utilice un símbolo '`@`' entre el patrón a coincidir y la variable a la cual asociar el valor. Este mecanismo se utiliza en el siguiente ejemplo para asociar el primer elemento de la lista en `l` para mostrarlo, y al mismo tiempo asociar la lista completa a `ls` para calcular su longitud:

```
len ls@(l:_) = "List starts with " ++
  show l ++ " and is " ++
  show (length ls) ++ " items long."
len [] = "List is empty!"
```

Guardas Las funciones booleanas se pueden utilizar como "guardas" en definicione de función al mismo tiempo que la comparación de patrones. Un ejemplo sin comparación de patrones:

```
which n
  | n == 0 = "Cero"
  | even n = "Par"
  | otherwise = "Impar"
```

Note el `otherwise` – siempre evalúa verdadero y puede ser utilizado para especificar un caso por "default".

Las guardas se pueden utilizar con patrones. El siguiente ejemplo es una función que determina si el primer caracter en una cadena es mayúscula o minúscula:

```
what [] = "Cadena vacía"
what (c:_)
  | isUpper c = "Mayúscula"
  | isLower c = "Minúscula"
  | otherwise = "No es letra"
```

Comparación & Orden de las Guardas La comparación de patrones procede en un orden de arriba hacia abajo. De la misma forma, las expresiones con guarda son evaluadas de la primera a la última. Por ejemplo, ninguna de estas funciones sería muy interesante:

```
allEmpty _ = False
allEmpty [] = True

alwaysEven n
  | otherwise = False
  | n `div` 2 == 0 = True
```

Sintaxis de Registros Normalmente la comparación de patrones ocurre basándose en la posición en los argumentos del valor a coincidir. Los tipos que se declaran con sintaxis de registro, sin embargo, pueden coincidir basándose en los nombres en el registro. Dado el siguiente tipo de datos:

```
data Color = C { red
  , green
  , blue :: Int }
```

podemos hacer coincidir solamente `green`:

```
isGreenZero (C { green = 0 }) = True
isGreenZero _ = False
```

Es posible capturar argumentos con esta sintaxis, aunque se vuelve incómodo. Continuando con el

ejemplo, podemos definir un tipo `Pixel` y una función que reemplace con negro valores con componente `green` diferente de cero:

```
data Pixel = P Color

-- El valor del color no se
-- modifica si green es 0
setGreen (P col@(C {green = 0})) = P col
setGreen _ = P (C 0 0 0)
```

Patrones Perezosos Esta sintaxis, también conocida como patrones *irrefutables*, permite hacer comparación de patrones que siempre coincida. Esto significa que cualquier cláusula utilizando el patrón tendrá éxito, pero si trata de utilizar el valor que ha coincidido puede ocurrir un error. Esto es generalmente útil cuando se debe realizar una acción basándose en el *tipo* de un valor en particular, aún si el valor no está presente.

Por ejemplo, defina una clase para valores por default:

```
class Def a where
  defValue :: a -> a
```

La idea es que le dé a `defValue` un valor del tipo correcto y regrese un valor por default para ese tipo. Definir instancias para tipos básicos es fácil:

```
instance Def Bool where
  defValue _ = False

instance Def Char where
  defValue _ = ' '
```

`Maybe` es un poco más complicado, porque queremos obtener un valor por default para el tipo, pero el constructor puede ser `Nothing`. La siguiente definición podría funcionar, pero no es óptima porque obtenemos `Nothing` cuando se le pasa `Nothing`.

```
instance Def a => Def (Maybe a) where
  defValue (Just x) = Just (defValue x)
  defValue Nothing = Nothing
```

Preferiríamos mejor obtener un valor `Just` (*valor por default*). Aquí es donde un patrón perezoso ayuda – podemos aparentar que hemos hecho coincidir con `Just x` y usar eso para obtener un valor por default, aún si entra `Nothing`:

```
instance Def a => Def (Maybe a) where
  defValue ~(Just x) = Just (defValue x)
```

Mientras el valor `x` no sea evaluado, estamos a salvo. Ninguno de los tipos base necesita inspeccionar `x` (ver la coincidencia con "_" que usan), así que funcionará bien.

Un inconveniente con esto es que debemos proporcionar anotaciones de tipo en el intérprete o en el código cuando usemos un constructor `Nothing`. `Nothing` tiene tipo `Maybe a` pero, a falta de información adicional, se debe informar a Haskell qué es `a`. Algunos ejemplos de valores por default:

```
-- Return "Just False"
defMB = defValue (Nothing :: Maybe Bool)
-- Return "Just ' '"
defMC = defValue (Nothing :: Maybe Char)
```

Listas por Comprensión

Una lista por comprensión consiste de cuatro tipos de elementos: *generadores*, *guardas*, *asociaciones locales*, y *objetivos*. Una lista por comprensión crea una lista de valores objetivo basados en los generadores y en las guardas proporcionados. Esta comprensión genera todos los cuadrados:

```
squares = [x * x | x <- [1..]]
```

`x <- [1..]` genera una lista de todos los valores enteros positivos y los coloca en `x`, uno por uno. `x * x` crea cada elemento de la lista multiplicando `x` por sí mismo.

Las guardas permiten que algunos elementos sean omitidos. El ejemplo a continuación muestra cómo calcular los divisores (excluyendo a él mismo) para cierto número. Notar cómo se usa el mismo `d` en la guarda y en la expresión objetivo.

```
divisors n =
  [d | d <- [1..(n `div` 2)]
     , n `mod` d == 0]
```

Las asociaciones locales proveen nuevas definiciones para usar en la expresión generada o en las guardas y generadores que las siguen. Debajo, `z` es empleado para representar el mínimo de `a` y `b`:

```
strange = [(a,z) | a <-[1..3]
                 , b <-[1..3]
                 , c <- [1..3]
                 , let z = min a b
                 , z < c ]
```

Las comprensiones no están limitadas a números. Funcionan con cualquier lista. Se pueden generar todas las letras mayúsculas:

```
ups =
  [c | c <- [minBound .. maxBound]
     , isUpper c]
```

O, para encontrar todas las apariciones de un valor `br` en una lista de palabras (con índices desde cero):

```
idxs palabras br =
  [i | (i, c) <- zip [0..] palabras
     , c == br]
```

Un aspecto único de las listas por comprensión es que los errores en la comparación de patrones no causan un error; simplemente son omitidos de la lista resultante.

Operadores

Hay muy pocos "operadores" predefinidos en Haskell—muchos que parecen estar predefinidos en realidad son sintaxis (e.g. "="). En lugar de eso, los operadores son simplemente funciones que toman dos argumentos y tienen un soporte sintáctico especial. Cualquier así llamado operador puede ser aplicado como una función prefijo usando paréntesis:

```
3 + 4 == (+) 3 4
```

Para definir un nuevo operador, simplemente defínalo como una función normal, excepto que el operador aparezca entre los dos argumentos. Este es uno que inserta una coma entre dos cadenas y asegura que no aparezcan espacios adicionales:

```
first ## last =
  let trim s = dropWhile isSpace
        (reverse (dropWhile isSpace
          (reverse s)))
  in trim last ++ ", " ++ trim first
```

```
> "  Haskell " ## " Curry "
Curry, Haskell
```

Por supuesto, comparación de patrones, guardas, etc. están disponibles en esta forma. La declaración de tipos es un poco diferentes. El operador "nombre" debe aparecer entre paréntesis:

```
(##) :: String -> String -> String
```

Los símbolos que se permite usar para definir operadores son:

```
# \$ % & * + . / < = > ? @ \ ^ | - ~
```

Sin embargo, hay varios "operadores" que no pueden ser redefinidos. Estos son `<-`, `->` y `=`. En sí mismos no se les puede asignar nueva funcionalidad, pero pueden ser utilizados como parte de un operador multicaracter. La función "bind", `>>=`, es un ejemplo.

Precedencia & Asociatividad La precedencia y asociatividad, colectivamente llamadas *fijidad*, de cualquier operador, pueden ser establecidos a través de las palabras clave `infix`, `infixr` e `infixl`. Éstas pueden ser aplicadas a funciones en el nivel superior o a definiciones locales. La sintaxis es:

`infix` | `infixr` | `infixl` *precedencia op*

donde *precedencia* varía de 0 a 9. *Op* puede ser cualquier función que tome dos argumentos (i.e., cualquier operación binaria). Que el operador sea asociativo por la izquierda o por la derecha está especificado por `infixl` o `infixr`, respectivamente. La declaración `infix` no tiene asociatividad.

La precedencia y la asociatividad hacen que muchas de las reglas de la aritmética funcionen "como se espera". Por ejemplo, considere las siguientes modificaciones menores a la precedencia de la suma y multiplicación:

```
infixl 8 `plus1`
plus1 a b = a + b
infixl 7 `mult1`
mult1 a b = a * b
```

Los resultados son sorpresivos:

```
> 2 + 3 * 5
17
> 2 `plus1` 3 `mult1` 5
25
```

Revertir la asociatividad también tiene efectos interesantes. Redefiniendo la división como asociativa por la derecha:

```
infixr 7 `div1`
div1 a b = a / b
```

Obtenemos resultados interesantes:

```
> 20 / 2 / 2
5.0
> 20 `div1` 2 `div1` 2
20.0
```

Aplicación parcial

En Haskell las funciones no tienen que recibir todos sus argumentos de una vez. Por ejemplo, considere la función `convertOnly`, que solamente convierte ciertos elementos de una cadena dependiendo de una prueba:

```
convertOnly test change str =
    map (\c -> if test c
               then change c
               else c) str
```

Usando `convertOnly` podemos escribir la función `l33t` que convierte ciertas letras a números:

```
l33t = convertOnly isL33t toL33t
  where
    isL33t 'o' = True
    isL33t 'a' = True
    -- etc.
    isL33t _ = False
    toL33t 'o' = '0'
    toL33t 'a' = '4'
    -- etc.
    toL33t c = c
```

Nótese que `l33t` no tiene argumentos especificados. También, que el argumento final de `convertOnly` no es proporcionado. Sin embargo, la declaración de tipos de `l33t` cuenta la historia completa:

```
l33t :: String -> String
```

Eso es, `l33t` toma una cadena y produce una cadena. Es "contante" en el sentido de que `l33t` siempre regresa un valos que es una función que toma una cadena y produce una cadena. `l33t` regresa una versión "currificada" de `convertOnly`, donde solamente dos de sus tres argumentos han sido provistos.

Esto puede ser llevado más lejos. Digamos que queremos escribir una función que solamente cambie letras mayúsculas. Sabemos cual es la prueba a aplicar, `isUpper`, pero no queremos especificar la conversión. Esa función puede ser escrita como:

```
convertUpper = convertOnly isUpper
```

Que tiene la declaración de tipos:

```
convertUpper :: (Char -> Char)
  -> String -> String
```

Eso es, `convertUpper` puede tomar dos argumentos. El primero es la función de conversión que convierte caracteres individuales y el segundo es la cadena que se va a convertir.

Se pueden crear una forma currificada de cualquier función que toma múltiples argumentos. Una forma de pensar esto es que cada "flecha" en la declaración de tipos de la función representa una nueva función que puede ser creada al proveer más argumentos.

Secciones Los operadores son funciones, y pueden ser currificados como cualquier otro. Por ejemplo, una versión currificada de "+" se puede escribir como:

```
add10 = (+) 10
```

Sin embargo esto es incómodo y difícil de leer. Las "secciones" son operadores currificados, usando paréntesis. Este es `add10` usando secciones:

```
add10 = (10 +)
```

El argumento provisto puede estar del lado izquierdo o derecho, lo que indica qué posición debe tomar. Esto es importante para operaciones como la concatenación:

```
onLeft str = (++ str)
onRight str = (str ++)
```

Que produce resultados diferentes:

```
> onLeft "foo" "bar"
"barfoo"
> onRight "foo" "bar"
"foobar"
```

"Actualizando" Valores y la Sintaxis de Registros

Haskell es un lenguaje puro y, como tal, no tiene estado mutable. Eso es, una vez que un valor ha sido establecido nunca cambia. "Actualizar" es en realidad una operación de copiado, con valores nuevos en los campos que "cambiaron". Por ejemplo, usando el tipo `Color` definido antes, podemos escribir una función que establece a cero el campo `green` fácilmente:

```
noGreen1 (C r _ b) = C r 0 b
```

Esto es algo extenso y puede ser vuelto a escribir con sintaxis de registro. Este tipo de "actualización" solamente establece valores para los campos especificados y copia los demás:

```
noGreen2 c = c { green = 0 }
```

Aquí capturamos el valor `Color` en `c` y devolvemos un nuevo valor `Color`. Ese valor resulta tener el mismo valor para `red` y `blue` que `c` y su componente `green` es 0. Podemos combinar esto con comparación de patrones para establecer los campos `green` y `blue` para que sean iguales al campo `red`:

```
makeGrey c@(C { red = r }) =
  c { green = r, blue = r }
```

Nótese que debemos usar captura de argumentos ("`c@`") para obtener el valor de `Color` y comparar con sintaxis de registro ("`C { red = r}`") para obtener el campo interno `red`.

Funciones Anónimas

Una función anónima (i.e., una *expresión lambda* o simplemente *lambda*) es una función sin nombre. Pueden ser definidas en cualquier momento de la siguiente forma:

```
\c -> (c, c)
```

que define una función que toma un argumento y regresa un tuple conteniendo ese argumento en ambas posiciones. Éstas son útiles para funciones simples donde no necesitamos un nombre. El ejemplo siguiente determina si una cadena consiste solamente de letras mayúsculas o minúsculas y espacio en blanco.

```
mixedCase str =
  all (\c -> isSpace c ||
             isLower c ||
             isUpper c) str
```

Por supuesto, las lambdas pueden ser regresadas también de otras funciones. Este clásico regresa una función que multiplicará su argumento por el que se ha dado originalmente:

```
multBy n = \m -> n * m
```

Por ejemplo:

```
> let mult10 = multBy 10
> mult10 10
100
```

Declaración de tipos

Haskell cuenta con inferencia de tipos, lo que significa que casi nunca es necesario declarar los tipos. Indicarlos es todavía útil al menos por dos razones.

Documentación—Aún si el compilador puede inferir los tipos de sus funciones, otros programadore o aún usted mismo podría no ser capaz de hacerlo más tarde. Declarar los tipos en todas las funciones del nivel principal se considera una buena práctica.

Especialización—Las clases de tipos permiten sobrecargar funciones. Por ejemplo, una función que hace la negación de una lista de números tiene la declaración de tipos:

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

Sin embargo, si por eficiencia o por otras razones solamente desea permitir tipos `Int`, puede hacerlo declarando los tipos:

```
negateAll :: [Int] -> [Int]
```

Los tipos pueden aparecer en funciones del nivel superior y en definiciones `let` o `where` anidadas. En general esto es útil para hacer documentación, aunque en algunos casos es requerido para prevenir el polimorfismo. Una declaración de tipos es primero el nombre del item, seguido de `::`, seguido de los tipos.

Las declaraciones de tipos no necesitan aparecer directamente sobre su implementación. Pueden ser especificadas en cualquier parte del módulo que las contiene (aún debajo). Se pueden definir al mismo tiempo varios items que tengan la misma declaración de tipos:

```
pos, neg :: Int -> Int

...

pos x | x < 0 = negate x
      | otherwise = x

neg y | y > 0 = negate y
      | otherwise = y
```

Anotaciones de Tipo Algunas veces Haskell no puede determinar qué tipo se debe aplicar. La demostración clásica de esto es el denominado problema "`show . read`":

```
canParseInt x = show (read x)
```

Haskell no puede compilar la función porque no conoce el tipo de `read x`. Debemos restringir el tipo por medio de una anotación:

```
canParseInt x = show (read x :: Int)
```

Las anotaciones tienen la misma sintaxis que las declaraciones de tipo, pero pueden adornar cualquier expresión. Nótese que la anotación en el ejemplo arriba está en la expresión `read x`, no en la variable `x`. Solamente la aplicación de función (e.g., `read x`) asocia más fuertemente que las anotaciones. Si ese no fuera el caso, habría sido necesario escribir `(read x) :: Int`.

Unidad

`()` – tipo "unidad" y valor "unidad". El valor y tipo que no representa información útil.

Palabras Clave

Las palabras clave en Haskell están listadas a continuación, en orden alfabético.

Case

`case` es similar a la declaración `switch` en C# o Java, pero puede hacer comparación de un patrón: la forma del valor siendo inspeccionado. Considere un tipo de datos simple.

```
data Choices = First String | Second |
  Third | Fourth
```

`case` puede ser utilizado para determinar qué opción se seleccionó:

```
whichChoice ch =
  case ch of
    First _ -> "1st!"
    Second -> "2nd!"
    _ -> "Something else."
```

Igual que en la comparación de patrones, el token '_' es un "comodín" que coincide con cualquier valor.

Anidado & Captura Se permite hacer comparación y asociación anidadas.

```
data Maybe a = Just a | Nothing
```

Figure 1: La definición de `Maybe`

Usando `Maybe` podemos determinar si alguna opción fue proporcionada utilizando una comparación anidada:

```
anyChoice1 ch =
  case ch of
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
    _ -> "Something else."
```

Se puede asociar un nombre al valor comparado para poder manipularlo:

```
anyChoice2 ch =
  case ch of
    Nothing -> "No choice!"
    Just score@(First "gold") ->
      "First with gold!"
    Just score@(First _) ->
      "First with something else: "
        ++ show score
    _ -> "Not first."
```

Orden de Comparación La comparación procede de arriba hacia abajo. Si el orden de `anyChoice` se modifica de la siguiente forma, el primer patrón siempre tendrá éxito:

```
anyChoice3 ch =
  case ch of
    _ -> "Something else."
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
```

Guardas Las guardas, o comparaciones condicionales, se pueden utilizar en casos igual que en la definición de funciones. La única diferencia es el uso de `->` en lugar de `=`. Esta es una función que hace comparación de cadenas sin importar si las letras son mayúscula o minúscula:

```
strcmp s1 s2 = case (s1, s2) of
  ([], []) -> True
  (s1:ss1, s2:ss2)
    | toUpper s1 == toUpper s2 ->
        strcmp ss1 ss2
    | otherwise -> False
  _ -> False
```

Clases

Una función en Haskell es definida para funcionar con un cierto tipo o conjunto de tipos y no puede ser definida más de una vez. Muchos lenguajes cuentan con el concepto de "sobrecarga", donde una función puede tener diferente comportamiento dependiendo del tipo de sus argumentos. Haskell implementa sobrecarga a través de declaraciones de `clase` y de `instancia`. Una

`clase` define una o más funciones que pueden ser aplicadas a cualquier tipo que sea miembro (i.e. instancia) de esa clase. Una clase es análoga a una interface en Java o C#, y, las intancias, a una implementación concreta de la interface.

Una clase debe ser declarada con una o más variables de tipo. Técnicamente, Haskell 98 solamente permite una variable de tipo, pero muchas implementaciones de Haskell implementan *tipos de clase multi-parámetro*, que permiten más de una variable de tipo.

Podemos definir una clase que provee un "sabor" para un tipo dado:

```
class Flavor a where
  flavor :: a -> String
```

Nótese que la declaración solamente da la declaración de tipos de la función—no se proporciona la implementación aquí (con algunas excepciones, ver "Defaults" en la página 9). Continuando, podemos definir varias instancias:

```
instance Flavor Bool where
  flavor _ = "dulce"

instance Flavor Char where
  flavor _ = "agrio"
```

Evaluating `flavor True` gives:

```
> flavor True
"dulce"
```

While `flavor 'x'` gives:

```
> flavor 'x'
"agrio"
```

Defaults Se pueden dar implementaciones por default para las funciones en una clase. Éstas son útiles cuando ciertas funciones pueden ser definidas en términos de otras en la clase. Un default es definido dadno un cuerpo a una de las funciones miembro. El ejemplo canónico es `Eq`, que define `/=` (no igual) en términos de `==`. :

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  (/=) a b = not (a == b)
```

Se pueden crear definiciones recursivas. Continuando con el ejemplo de `Eq`, `==` puede ser definido entérminos de `/=`:

```
  (==) a b = not (a /= b)
```

Sin embargo, si las instancias no proveen implementaciones suficientemente concretas de las funciones miembro, el programa que use esas instancias puede entrar en ciclo infinito.

Data

Los *tipos de datos algebraicos* pueden ser declarados de la siguiente forma:

```
data MyType = MyValue1 | MyValue2
```

`MyType` es el *nombre* del tipo. `MyValue1` y `MyValue2` son *valores* del tipo y son llamados *constructores*. Se pueden declarar varios constructores, que se separan con el caracter '`|`'. Nótese que los nombres de tipo y de constructor *deben* iniciar con letra mayúscula. Es un error de sintaxis de cualquier otra forma.

Constructores con Argumentos El tipo arriba noes muy interesante excepto como una enumeración. Se pueden declarar constructores que tomen argumentos, permitiendo que se almacene más información:

```
data Point = TwoD Int Int
  | ThreeD Int Int Int
```

Note que los argumentos para cada constructor son nombres de *tipo*, no constructores. Eso significa que una declaración como la siguiente es ilegal:

```
data Poly = Triangle TwoD TwoD TwoD
```

En lugar de eso se debe usar el tipo `Point`:

```
data Poly = Triangle Point Point Point
```

Nombres de Tipos y de Constructores Los tipos y constructores pueden tener el mismo nombre, porque nunca serán utilizados de forma que pudiera causar conflicto. Por ejemplo:

```
data User = User String | Admin String
```

que declara un tipo llamado `User` con dos constructores, `User` y `Admin`. Usando ese tipo en una función hace clara la diferencia:

```
whatUser (User _) = "normal user."
whatUser (Admin _) = "admin user."
```

Cierta literatura se refiere a esta práctica como *"type punning"*.

jgbailey@codeslower.com

Variables de Tipo Declarar tipos de datos *polimórficos* es tan fácil como agregar variables de tipo en la declaración:

```
data Slot1 a = Slot1 a | Empty1
```

Esto declara un tipo `Slot1` con dos constructores, `Slot1` y `Empty1`. El constructor `Slot1` puede tomar un argumento de *cualquier* tipo, que es representado por la variable de tipo `a` arriba.

También podemos mezclar variables de tipo y tipos específicos en los constructores:

```
data Slot2 a = Slot2 a Int | Empty2
```

Arriba, el constructor `Slot2` puede tomar un valor de cualquier tipo y un valor `Int`.

Sintaxis de Registro Se pueden declarar los argumentos del constructor ya sea por su posición, como se hace arriba, o utilizando sintaxis de registros, que le da un nombre a cada argumento. Por ejemplo, aquí declaramos un tipo `Contact` con nombres para los argumentos apropiados:

```
data Contact = Contact { ctName :: String
       , ctEmail :: String
       , ctPhone :: String }
```

A esos nombres se les llama funciones *selector* o *accesor* y son eso, funciones. Deben empezar con minúscula o guión bajo y no pueden tener el mismo nombre que otra función en el mismo contexto. Por eso el prefijo "`ct`" en el ejemplo arriba. Varios constructores (del mismo tipo) pueden utilizar la misma función accesor para valores del mismo tipo, pero esto puede ser peligroso si el accesor no es utilizado por todos los constructores. Considere el siguiente ejemplo:

```
data Con = Con { conValue :: String }
  | Uncon { conValue :: String }
  | Noncon

whichCon con = "convalue is " ++
  conValue con
```

Si `whichCon` es invocado con un valor `Noncon`, ocurrirá un error.

Finalmente, como se explica en otras partes, esos nombres se pueden utilizar para comparación de patrones, captura y "actualización".

Restricciones de Clase Se pueden declarar tipos de datos con restricciones de clase en las variables de tipo, pero en general esta práctica es desaprobada. Generalmente es mejor ocultar los constructores de datos empleando el sistema de módulos y exportar constructores "inteligentes" que apliquen las restricciones apropiadas. En cualquier caso, la sintaxis es:

```
data (Num a) => SomeNumber a = Two a a
  | Three a a a
```

Esto declara un tipo `SomeNumber` que tiene un argumento de variable de tipo. Los tipos válidos son los que están en la clase `Num`.

Deriving Muchos tipos tienen operaciones en común que son tediosas para definir aunque necesarias, como la habilidad de convertir de y a cadenas, comparar igualdad, u ordenar en secuencia. Esas capacidades están definidas como clases de tipos en Haskell.

Como siete de estas operaciones son muy comunes, Haskell provee la palabra clave `deriving` que automáticamente implementa la clase de tipos al tipo asociado. Las siete clases de tipos que permiten hacerlo son: `Eq`, `Read`, `Show`, `Ord`, `Enum`, `Ix`, y `Bounded`.

Hay dos formas de usar `deriving`. La primera se utiliza cuando un tipo solamente deriva una clase:

```
data Priority = Low | Medium | High
  deriving Show
```

El segundo es usado cuando se derivan múltiples clases:

```
data Alarm = Soft | Loud | Deafening
  deriving (Read, Show)
```

Es un error de sintaxis especificar `deriving` para ninguna otra clase además de las indicadas.

Deriving

Vea la sección en `deriving` bajo la palabra clave `data` en la página 10.

Do

La palabra clave `do` indica que el código a continuación estará en un contexto *monádico*. Las declaraciones están separadas por saltos de línea, la "asignación" es indicada por `<-`, y se puede emplear una forma de `let` que no requiere la palabra clave `in`.

If con IO `if` puede ser complicado cuando se utiliza con IO. Conceptualmente no es diferente de un `if` en cualquier otro contexto, pero intuitivamente es difícil de asimilar. Considere la función `doesFileExists` de `System.Directory`:

```
doesFileExist :: FilePath -> IO Bool
```

La declaración `if` tiene esta pseudo-"declaración de tipos":

```
if-then-else :: Bool -> a -> a -> a
```

Eso es, toma un valor `Bool` y evalúa a algún otro valor con base en la condición. De la declaración de tipos está claro que `doesFileExist` no puede ser utilizado directamente por `if`:

```
wrong fileName =
  if doesFileExist fileName
    then ...
    else ...
```

Eso es, `doesFileExost` resulta en un valor `IO Bool`, mientras que `if` quiere un valor `Bool`. El valor correcto debe ser "extraído" ejecutando la acción IO:

```
right1 fileName = do
  exists <- doesFileExist fileName
  if exists
    then return 1
    else return 0
```

Note el uso de `return`. Como `do` nos coloca "dentro" de la mónada `IO`, no podemos "salir"excepto a través de `return`. Nótese que no tenemos que usar `if` directamente aquí—también podemos usar `let` para evaluar la condición y obtener un resultado primero:

```
right2 fileName = do
  exists <- doesFileExist fileName
  let result =
        if exists
          then 1
          else 0
  return result
```

Una vez más, notar donde está `return`. No lo ponemos en la declaración `let`. En lugar de eso lo usamos una vez al final de la función.

`do`'s Multiples Al usar `do` con `if` o `case`, se requiere otro `do` is cualquier rama tiene múltiples declaraciones. Un ejemplo con `if`:

```
countBytes1 f =
  do
    putStrLn "Enter a filename."
    args <- getLine
    if length args == 0
      -- no 'do'.
      then putStrLn "No filename given."
      else
        -- multiple statements require
        -- a new 'do'.
        do
          f <- readFile args
          putStrLn ("The file is " ++
            show (length f)
            ++ " bytes long.")
```

Y uno con `case`:

```
countBytes2 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do
       f <- readFile file
       putStrLn ("The file is " ++
         show (length f)
         ++ " bytes long.")
```

Una sintaxis alternativa usa llaves y punto y coma. Todavía se requiere un `do`, pero la sangría es innecesaria. Este código muestra un ejemplo de `case`, pero el principio aplica igual con `if`:

```
countBytes3 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do { f <- readFile file;
       putStrLn ("The file is " ++
         show (length f)
         ++ " bytes long."); }
```

Export

Vea la sección `module` en la página 12.

If, Then, Else

Recuerde, `if` siempre "devuelve" un valor. Es una expresión, no solamente una declaración de control de flujo. Esta función revisa si la cadena dada inicia con letra minúscula, y, de ser así, la convierte a mayúscula:

lower case letter and, if so, converts it to upper case:

```
-- Usa comparación de patrones
-- para obtener el primer caracter
sentenceCase (s:rest) =
 if isLower s
   then toUpper s : rest
   else s : rest
-- Cualquier otro caso es
-- sobre cadena vacía
sentenceCase _ = []
```

Import

Ver la sección module en la página 12.

In

Ver let en la página 12.

Infix, infixl e infixr

Ver la sección operators en la página 5.

Instance

Ver la sección class en la página 8.

Let

Se pueden definir funciones localmente dentro de una función usando let. La palabra clave let debe siempre ser seguida por in. El in debe aparecer en la misma columna que la palabra clave let. Las funciones definidas tienen acceso a todas las demás funciones y variables dentro del mismo contexto (incluyendo las definidas por let). En este ejemplo, mult multiplica su argumento n por x, que fue recibido del multiples original. mult es usado por map para dar los múltiplos de x por 1 hasta 10:

```
multiples x =
  let mult n = n * x
  in map mult [1..10]
```

Las "funciones" let sin argumentos son en realidad constantes y, una vez que son evaluadas, no serán evaluadas otra vez. Esto es útil para capturar porciones comunes de su función y reutilizarlas. El siguiente es un ejemplo que da la suma de una lista de números, su promedio, y su mediana:

```
listStats m =
  let numbers = [1,3 .. m]
      total = sum numbers
      mid = head (drop (m `div` 2)
                       numbers)
  in "total: " ++ show total ++
     ", mid: " ++ show mid
```

Deconstrucción El lado izquierdo de una definición let puede también desestructurar su argumento, en caso de que se requiera acceso a sus sub-componentes. Esta definición extraerá los primeros tres caracteres de una cadena:

```
firstThree str =
  let (a:b:c:_) = str
  in "Initial three characters are: " ++
      show a ++ ", " ++
      show b ++ ", and " ++
      show c
```

Note que esto es diferente de lo que sigue, que solamente funciona si la cadena tiene exactamente tres caracteres:

```
onlyThree str =
  let (a:b:c:[]) = str
  in "The characters given are: " ++
      show a ++ ", " ++ show b ++
      ", and " ++ show c
```

Of

Vea la sección case en la página 8.

Module

Un módulo es una unidad de compilación que exporta funciones, tipos, clases, instancias, y otros módulos. Un módulo solamente se puede definir en un solo archivo, aunque lo que exporte puede provenir de varias fuentes. Para convertir un archivo Haskell en módulo basta con agregar una declaración de módulo hasta arriba:

```
module MyModule where
```

Los nombres de módulo deben comenzar con letra mayúscula pero pueden contener puntos, números y guiones bajos. Los puntos se usan para dar un sentido de estructura, y los compiladores Haskell los utilizarán como indicadores del directorio en el que está un archivo fuente en particular, pero fuera de eso no tienen significado.

La comunidad Haskell ha estandarizado un conjunto de nombres de módulo como Data, System, Network, etc. Asegúrese de consultarlos para seleccionar un lugar apropiado para su propio módulo si planea liberarlo al público.

Import Las bibliotecas estándar Haskell están divididas en un número de módulos. Se accede a la funcionalidad provista por esas bibliotecas importándolas en el programa fuente. Para importar todo lo que exporta una biblioteca, simplemente use el nombre del módulo:

```
import Text.Read
```

Todo significa *todo*: funciones, tipos de datos y constructores, declaraciones de clase, y aún otros módulos importados y luego exportados por ese módulo. Para importar selectivamente se pasa una lista de nombres qué importar. Pr ejemplo, aquí importamos algunas funciones de `Text.Read`:

```
import Text.Read (readParen, lex)
```

Los tipos de datos pueden ser importados de varias formas. Podemos solamente importar el tipo sin constructores:

```
import Text.Read (Lexeme)
```

Por supuesto, esto impide que nuestro módulo haga comparación de patrones con los valores de tipo `Lexeme`. Podemos importar uno o más constructores explícitamente:

```
import Text.Read (Lexeme(Ident, Symbol))
```

Se pueden importar todos los constructores para un tipo dado:

```
import Text.Read (Lexeme(..))
```

Podemos también importar tipos y clases definidos en el módulo:

```
import Text.Read (Read, ReadS)
```

En el caso de las clases, podemos importar las funciones definidas para una clase usando una sintaxis similar a importar constructores para tipos de datos:

```
import Text.Read (Read(readsPrec
                       , readList))
```

Note que, a diferencia de los tipos de datos, todas las funciones de clase son importadas a menos que sean explícitamente excluídas. Para importar *sólo* las clases, usamos esta sintaxis:

```
import Text.Read (Read())
```

Exclusiones Si la mayoría, pero no todos los nombres van a ser importados de un módulo, sería tedioso listarlos. Por esa razón, también se pueden omitir nombres con la palabra clave `hiding` (“ocultar”):

```
import Data.Char hiding (isControl
                         , isMark)
```

Excepto por las declaraciones de instancia, cualquier tipo, función o clase pueden ser ocultos.

Declaraciones de Instancia Debe notarse que las declaraciones de instancia (`instance`) *no pueden* ser excluídas al importar; todas las declaraciones `instance` en un módulo serán importadas cuando el módulo sea importado.

Import Qualified Los nombres exportados por un módulo (i.e., funciones, tipos, operadores, etc.) pueden tienen un prefijo asociado a través de importación calificada. Esto es particularmente útil para módulos que tienen un gran número de funciones con el mismo nombre que funciones del `Prelude`. `Data.Set` es un buen ejemplo:

```
import qualified Data.Set as Set
```

Esta forma requiere que cualquier función, tipo, constructor u otro nombre exportado por `Data.Set` tenga el prefijo del *alias dado (i.e.,* `Set`*). La siguiente es una forma de remover todos los duplicados de una lista:*

```
removeDups a =
  Set.toList (Set.fromList a)
```

Una segunda forma no crea un alias. En lugar de ello, el prefijo se convierte en el nombre del módulo. Podemos escribir una función que verifique si una cadena está escrita en todo mayúsculas:

```
import qualified Char

allUpper str =
  all Char.isUpper str
```

Excepto por el prefijo especificado, la importación calificada emplea la misma sintaxis que una importación normal. Los nombres importados se pueden limitar de las mismas formas descritas arriba.

Export *Si no se provee una lista de exportaciones, entonces todas las funciones, tipos, constructores, etc. estarán disponibles siempre que se importe el módulo. Note que los módulos importados* no *son exportados en este caso. Limitar los nombres exportados se consigue agregando una lista entre paréntesis de los nombres antes de la palabra clave* `where`*:*

```
module MyModule (MyType
  , MyClass
  , myFunc1
  ...)
where
```

La misma sintaxis que se usa para importar puede ser usada para especificar qué funciones, tipos, constructores, y clases son exportados, con unas pocas diferencias. Si un módulo importa otro módulo, puede también exportar ese módulo:

```
module MyBigModule (module Data.Set
  , module Data.Char)
where

import Data.Set
import Data.Char
```

Un módulo puede también re-exportarse a sí mismo, lo que puede ser útil cuando todas las definiciones locales y un módulo importado dado van a se exportados. A continuación nos exportamos a nosotros mismos y a `Data.Set`*, pero no a* `Data.Char`*:*

```
module AnotherBigModule (module Data.Set
  , module AnotherBigModule)
where

import Data.Set
import Data.Char
```

Newtype

Mientras que `data` *agrega nuevos valores y* `type` *solamente crea sinónimos,* `newtype` *está en un punto medio. La sintaxis para* `newtype` *está más restringida—solamente se puede definir un constructor, y ese constructor solamente puede tomar un argumento. Continuando con el ejemplo de arriba, podemos definir un tipo* `Phone` *de la forma que sigue:*

```
newtype Home = H String
newtype Work = W String
data Phone = Phone Home Work
```

En contraste con `type`*, los "valores"* `H` *y* `W` *en* `Phone` no *son solamente valores* `String`*. El verificador de tipos los trata como tipos completamente nuevos. Eso significa que nuestra función* `lowerName` *arriba no compilaría. Esto produce un error de tipos:*

```
lPhone (Phone hm wk) =
  Phone (lower hm) (lower wk)
```

En lugar de eso, debemos usar comparación de patrones para llegar a los "valores" a los que queremos aplicar `lower`*:*

```
lPhone (Phone (H hm) (W wk)) =
  Phone (H (lower hm)) (W (lower wk))
```

La observación clave es que esta palabra clave no crea un valor nuevo; en lugar de eso crea un tipo nuevo. Esto proporciona dos propiedades muy útiles:

- *No hay costo en tiempo de ejecución asociado al tipo nuevo, porque en realidad no produce valores nuevos. En otras palabras,* `newtype` *es absolutamente gratis en desempeño cuando el programa es ejecutado.*
- *El verificador de tipos puede hacer que tipos comunes como* `Int` *o* `String` *se usen de formas restringidas, especificadas por el programador.*

Finalmente, se debe notar que cualquier cláusula `deriving` *que puede ser anexada a una declaración* `data` *puede también ser usada al declarar un* `newtype`*.*

Return

Ver `do` *en la página 10.*

Type

Esta palabra clave define un sinónimo de tipo *(i.e., alias). Esta palabra clave no define un tipo nuevo, como* `data` *o* `newtype`*. Es útil para documentar código pero, además de eso, no tiene efecto en el tipo de una función o valor dados. Por ejemplo, un tipo de datos* `Person` *puede ser definido como:*

```
data Person = Person String String
```

donde el argumento del primer constructor representa al nombre y el segundo al apellido. Sin embargo, el orden y significado de los dos argumentos no es muy claro. Una declaración con `type` *puede ayudar:*

```
type FirstName = String
type LastName = String
data Person = Person FirstName LastName
```

Como `type` *crea un sinónimo, la verificación de tipos no se afecta. La función* `lower`*, definida como:*

```
lower s = map toLower s
```

que tiene tipo

```
lower :: String -> String
```

puede ser usada igual en valores con tipo `FirstName` *o* `LastName`*:*

```
lName (Person f l ) =
  Person (lower f) (lower l)
```

Como `type` *es solamente un sinónimo, no puede declarar múltiples constructores de la forma que* `data` *puede. Se pueden usar variables de tipo, pero no puede haber más que las las variables de tipo declaradas con el tipo original. Eso significa que un sinónimo como el siguiente es válido:*

```
type NotSure a = Maybe a
```

pero este no:

```
type NotSure a b = Maybe a
```

Nótese que se pueden usar menos *variables de tipo, lo que es útil en ciertas circunstancias.*

Where

De la misma forma que `let`*,* `where` *define funciones y constantes locales. El contexto de una definición* `where` *es la función actual. Si una función está dividida en definiciones múltiples con comparación de patrones, entonces el contexto de una cláusula* `where` *en particular solamente aplica a esa definición. Por ejemplo, la función* `result` *a continuación tiene un significado diferente dependiendo de los argumentos proporcionados a la función* `strlen`*:*

```
strlen [] = result
  where result = "No string given!"
strlen f = result ++ " characters long!"
  where result = show (length f)
```

Where vs. Let *Una cláusula* `where` *solamente puede ser definida al nivel de una definición de función. Usualmente, eso es idéntico al contexto de una definición* `let`*. La única diferencia es en cuándo son utilizadas las guardas. En contraste, el contexto de una expresión* `let` *es solamente la cláusula actual de la función y la guarda, de existir.*

Traducción al Español

Jaime Soffer, jaime.soffer@gmail.com

Contributors

My thanks to those who contributed patches and useful suggestions: Dave Bayer, Paul Butler, Elisa Firth, Marc Fontaine, Cale Gibbard, Stephen Hicks, Kurt Hutchinson, Johan Kiviniemi, Adrian Neumann, Barak Pearlmutter, Lanny Ripple, Markus Roberts, Holger Siegel, Adam Vogt, Leif Warner, and Jeff Zaroyko.

Version

This is version 2.0. The source can be found at GitHub (`http://github.com/m4dc4p/cheatsheet`*). The latest released version of the PDF can be downloaded from* `http://cheatsheet.codeslower.com`*. Visit CodeSlower.com (*`http://blog.codeslower.com/`*) for other projects and writings.*

Traducción al Español version 0.01 sobre la versión original mencionada arriba. Repositorio en http://github.com/jsoffer/cheatsheet