Trabajo Práctico 1
Programación Funcional

versión 1.0

Paradigmas ~~de Lenguajes~~ de Programación
2^{do} cuatrimestre 2025

Fecha de entrega: 16 de septiembre

Introducción: Calculadora Incierta

Vamos a hacer una calculadora que opere con cierta incertidumbre. La idea es que además de números concretos y las operaciones aritméticas $+, -, \times, /$, vamos a trabajar con rangos numéricos.

Un rango numérico se representa como un par de números reales $a \sim b$. El rango nos indica un número que no sabemos exactamente cuál va a ser, pero sabemos que va a estar entre a y b con una confianza del 95 %. Esto quiere decir que hay un 5 % de probabilidad de que esté por fuera del rango. La distribución de los números es normal: las chances de obtener un número en el medio del rango son mayores, y decrecen a medida que nos alejamos. No hace falta saber estadística.

Entonces, cuando queramos evaluar una expresión, como por ejemplo $10000 * 0.7 \sim 0.8 + 2000 \sim 5000$, vamos a obtener a veces un valor y a veces otro.

Para llegar a una conclusión sobre el resultado de la expresión, vamos a realizar muchas muestras, donde cada una nos dará un resultado distinto. Juntando todas las muestras vamos a construir un histograma que nos permita ver cómo se distribuyen los resultados.

El histograma a continuación nos va a decir además que con un 95 % de confianza el resultado va a estar entre 9415.63 y 12580.79.

```
% make run
incierticalc> 10000 * 0.7∼0.8 + 2000∼5000
    12580.79 - +inf |▄▄▄▄▄
12293.05 - 12580.79 |▄▄▄▄▄
12005.31 - 12293.05 |▄▄▄▄▄▄▄▄
11717.56 - 12005.31 |▄▄▄▄▄▄▄▄▄▄▄
11429.82 - 11717.56 |▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
11142.08 - 11429.82 |▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
10854.34 - 11142.08 |▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄ 14.14%
10566.60 - 10854.34 |▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
10278.86 - 10566.60 |▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
 9991.11 - 10278.86 |▄▄▄▄▄▄▄▄▄▄▄
  9703.37 - 9991.11 |▄▄▄▄▄▄▄▄
  9415.63 - 9703.37 |▄▄▄▄▄
     -inf - 9415.63 |▄▄▄▄▄

incierticalc> :q
Adiós!
```

Los ejercicios a continuación tienen como objetivo implementar las partes faltantes de la aplicación `incierticalc`.

El código base cuenta con varios archivos, algunos completos y otros con funciones a implementar. La estructura es la siguiente:

- `app/Main.hs` es el punto de entrada de la aplicación interactiva. **Este archivo está completo**.
- `src/App.hs` define la aplicación interactiva. **Este archivo está completo**.
- `src/Expr/Parser.hs` define un parser para convertir un `String` en una expresión del tipo `Expr`. **Este archivo está completo**.
- `src/Generador.hs` define funciones para generar números aleatorios. **Este archivo está completo**.
- `src/Util.hs` define funciones auxiliares. Hay ejercicios para completar este módulo.
- `src/Histograma.hs` define el tipo abstracto de datos `Histograma`. Hay ejercicios para completar este módulo.
- `src/Expr.hs` define el tipo algebraico de datos `Expr` y varias funciones sobre él. Hay ejercicios para completar este módulo.
- `test/Main.hs` es el punto de entrada para los tests. Hay algunos ejemplos de tests. **Deberán agregar más tests**. Con `make test` o `make watch` pueden ejecutarlos.

Módulo `Util`

En este módulo definiremos funciones que nos serán útiles para resolver el resto del trabajo. Dentro se encuentran definidas las constantes `infinitoPositivo` e `infinitoNegativo` debido a que Haskell no tiene literal para ∞.

Ejercicio 1

Definir la función `alinearDerecha :: Int → String → String` que, dados `n` un entero y `s` un String, agrega espacios a la izquierda de `s` hasta que su longitud sea `n`. Si `s` ya es de longitud mayor o igual a `n`, devuelve `s`. **No se permite recursión explícita**. Por ejemplo:

```
ghci> alinearDerecha 6 "hola"
"  hola"

ghci> alinearDerecha 10 "incierticalc"
"incierticalc"
```

Ejercicio 2

Definir la función `actualizarElem :: Int → (a → a) → [a] → [a]` que dados un índice, una función y una lista, actualiza en la lista el elemento en la posición del índice aplicando la función al valor actual. Si el índice está fuera de los límites de la lista, devuelve la lista sin cambios. El primer elemento de la lista es el índice 0. Por ejemplo:

```
ghci> actualizarElem 1 (+10) [0, 1, 2, 3, 4]
[0,11,2,3,4]

ghci> actualizarElem (-1) (+10) [0, 1, 2, 3, 4]
[0,1,2,3,4]
```

Es importante que la función sea **total por construcción**, deben evitar usar funciones parciales como `(!!)`. **No se permite recursión explícita**.

Dos posibles caminos para resolver este ejercicio son completar alguna de las siguientes implementaciones:

a) `actualizarElem n f = zipWith _ [0 ..]`

b) `actualizarElem n f xs = actualizarElem' f xs n`

```
actualizarElem' :: (a → a) → [a] → (Int → [a])
actualizarElem' f = foldr _ (const [])
```

Módulo `Histograma`

En este módulo se define el tipo `Histograma`. Es un tipo abstracto, no se exportan sus constructores. Se dispone de operaciones para crear un `Histograma` y una función `casilleros` para poder observarlo.

Un histograma se puede ver como una serie de casilleros. Cada casillero tiene un intervalo y cuenta cuántos números caen dentro de dicho intervalo. El primer casillero empieza en $-\infty$. El último termina en $+\infty$. El tamaño del intervalo de todos los casilleros intermedios es el mismo. A partir de esto resulta que un `Histograma` se define de la siguiente forma:

```
data Histograma = Histograma Float Float [Int]
  deriving (Show, Eq)
```

- El primer `Float` es el inicio del intervalo de la segunda casilla.
- El segundo `Float` es el tamaño del intervalo de cada casillero, que debe ser mayor que 0.
- La lista de enteros representa la cuenta de cuántos elementos caen en el intervalo de cada casillero. Tiene al menos 3 elementos.

Sea `Histograma i t cs :: Histrograma`, sabemos

- `cs !! 0` indica cuantos números caen en $(-\infty, i)$
- `cs !! 1` indica cuantos números caen en $[i, i + t)$
- `cs !! 2` indica cuantos números caen en $[i + t, i + 2 \times t)$
- `cs !! 3` indica cuantos números caen en $[i + 2 \times t, i + 3 \times t)$
- ...

Nota: Dado que el tipo `Histograma` es abstracto, no se pueden acceder directamente a sus campos en `test/Main.hs`. En el ejercicio 6 se pide definir la función `casilleros` que permite observar un `Histograma`. Dicha función es útil para definir los casos de prueba para los ejercicios 3, 4 y 5. Aunque también necesitan los ejercicios 3 y 4 para hacer los casos de prueba del ejercicio 6.

Ejercicio 3

Definir la función `vacio :: Int → (Float, Float) → Histograma` que inicializa un histograma con todos sus casilleros vacíos. `vacio n (l, u)` devuelve un histograma con `n` casilleros finitos para representar valores en el rango `(l, u)` y 2 casilleros adicionales para los valores fuera del rango. Requiere que `l < u` y `n` $\geq$ `1` (se puede asumir como verdadero sin necesidad de verificarlo). La función `fromIntegral :: Int → Float` puede ser útil.

Ejercicio 4

Definir la función `agregar :: Float → Histograma → Histograma` que agrega el valor indicado al casillero correspondiente del histograma.

Pueden utilizar `actualizarElem` para generar la nueva lista de cantidades.

Las funciones `fromIntegral :: Int → Float`, `floor :: Float → Int`, `min` y `max` pueden ser útiles.

Ejercicio 5

Definir la función `histograma :: Int → (Float, Float) → [Float] → Histograma` que construye un histograma a partir de una lista de números reales con la cantidad de casilleros finitos y rango indicados. **No se permite recursión explícita**.

En el módulo se define el tipo algebraico de datos `Casillero` y funciones convenientes para observarlo.

```
data Casillero = Casillero Float Float Int Float
  deriving (Show, Eq)

casMinimo :: Casillero → Float
casMaximo :: Casillero → Float
casCantidad :: Casillero → Int
casPorcentaje :: Casillero → Float
```

Cada función se corresponde con un campo del tipo `Casillero`. El porcentaje es un número entre 0 y 100 y representa la cantidad de valores en el casillero respecto al total de valores en el histograma.

Ejercicio 6

Definir la función `casilleros :: Histograma → [Casillero]` que, dado un histograma, devuelve la lista de casilleros con sus límites, cantidad y porcentaje. El primer casillero es el que va desde $-\infty$ hasta el mínimo del rango del histograma.

Esta función es útil para poder escribir tests, ya que nos permite observar un `Histograma`. **No se permite recursión explícita**.

Pueden usar la función `zipWith4` para construir la lista de casilleros. Esta se debe importar desde `Data.List` agregando `import Data.List (zipWith4)` al principio del archivo.

Módulo `Generador`

El módulo `Generador` nos permite generar números aleatorios dentro de un rango con 95 % de confianza. **Este módulo está completo**. Los generadores se representan con el tipo `Gen`.

Se dispone de la función `dameUno :: (Float, Float) → Gen → (Float, Gen)` que genera un número aleatorio y devuelve el generador actualizado.

¿Por qué `dameUno` devuelve un generador actualizado? Porque los valores (incluso los generadores) son *inmutables* y cada vez que se genera un número aleatorio, debemos dejar de usar el generador original. Si no lo hacemos, el generador original seguirá generando el mismo número una y otra vez (no muy aleatorio, ¿no?).

Si `g :: Gen`, `dameUno (a, b) g` reduce a `(x, g')` donde `x` es un número entre `a` y `b` con un 95 % de confianza y `g'` es un nuevo `Gen` que debe ser usado para obtener el siguiente número a `x`.

Hay 3 formas de crear un generador:

- `genFijo` es un generador que siempre devuelve el valor medio del rango.

- `genNormalConSemilla n` con `n :: Int` es un generador que produce números aleatorios con distribución normal. La semilla `n` permite reproducibilidad dado que determina cuáles van a ser los números aleatorios a generar[1].
- `genNormal` es un generador que produce números aleatorios con distribución normal usando una semilla aleatoria.

La última función no es pura. Dada una función `f :: Gen → a`, podemos evaluarla usando `f genFijo`, `f (genNormalConSemilla 0)`, `f (genNormalConSemilla 1)`, etc.

Si queremos evaluar `f` con un generador aleatorio podemos usar `conGenNormal f` que devuelve el resultado de `f`. `conGenNormal` tampoco es una función pura, pero nos permite aplicar funciones puras en forma cómoda.

En `ghci` podemos probar:

```
ghci> dameUno (1, 5) genFijo
(3.0,<Gen>)

ghci> dameUno (1, 5) (genNormalConSemilla 0)
(2.7980492,<Gen>)

ghci> conGenNormal (dameUno (1, 5))
(1.2677777,<Gen>)
ghci> conGenNormal (dameUno (1, 5))
(0.3479743,<Gen>)
ghci> conGenNormal (dameUno (1, 5))
(1.4352515,<Gen>)
```

Nuevamente, siempre que se trabaje con generadores como parámetro de una función vamos a querer devolver otro generador como resultado. Si no perderíamos el "nuevo estado" del generador y solo podríamos generar el mismo número nuevamente.

Por ejemplo `dameUno :: (Float, Float) → Gen → (Float, Gen)`. La última parte indica que toma un generador y devuelve un número y el generador resultante.

Se dispone del tipo `G a` que es un sinónimo de tipo para `Gen → (a, Gen)`. Usando `G a` el tipo de `dameUno` queda `dameUno :: (Float, Float) → G Float`, ¡pero es el mismo tipo!

El tipo `G a` es una función que toma un generador y devuelve un valor de tipo `a` y el generador resultante.

Dada una función `f :: G a` y `n :: Int`, entonces `muestra f n :: G [a]` aplica `f` exactamente `n` veces devolviendo los resultados en una lista (y el generador resultante).

La función `rango95 :: [Float] → (Float, Float)` toma una lista finita de números reales, que si provienen de un generador de distribución normal, devuelve un rango que cubre el 95 % de los valores que producirá el generador. Si todos los números son iguales, devuelve el rango `(valor-1, valor+1)`. La lista debe tener al menos un elemento.

Nota: Dado el caracter aleatorio de un generador es esperable que algunos casos de prueba usen los números que sabemos serán generados. Saber qué números serán generados requiere experimentación.

```
ghci> fst (dameUno (1, 5) (genNormalConSemilla 0)) == 2.7980492
True

ghci> muestra (dameUno (1, 5)) 2 (genNormalConSemilla 0)
([2.7980492,3.1250308],<Gen>)
```

[1] https://es.wikipedia.org/wiki/Semilla_aleatoria

Módulo `Expr`

Las expresiones de `incierticalc` se representan con el siguiente tipo algebraico:

```
data Expr = Const Float
          | Rango Float Float
          | Suma Expr Expr
          | Resta Expr Expr
          | Mult Expr Expr
          | Div Expr Expr
          deriving (Show, Eq)
```

A continuación algunos ejemplos de expresiones:

- $1 + 2 \times 3$ representa como `Suma (Const 1.0) (Mult (Const 2.0) (Const 3.0))`.
- $(1 + 2) \times 3$ representa como `Mult (Suma (Const 1.0) (Const 2.0)) (Const 3.0)`.

Ejercicio 7

Definir las funciones `recrExpr` y `foldExpr` que corresponden con los esquemas de recursión primitiva y estructural respectivamente del tipo `Expr`. Indicar el tipo de las funciones junto a sus definiciones. **Se permite usar recursión explícita en estas funciones**.

Ejercicio 8

Definir la función `eval` que permite obtener una posible evaluación de una expresión dado un generador de números aleatorios. **No se permite recursión explícita**.

El tipo deseado es `eval :: Expr → Gen → (Float, Gen)` para poder obtener el generador resultante después de generar algunos números aleatorios para evaluar los rangos. Puede escribirse usando `G a` como `eval :: Expr → G Float`.

Se debe usar `dameUno` para determinar el valor de un rango.

Ejercicio 9

Definir la función `armarHistograma :: Int → Int → G Float → G Histograma`.

`armarHistograma m n f g` arma un histograma con `m` casilleros finitos a partir del resultado de tomar `n` muestras de `f` usando el generador `g`. El rango del histograma debe ser calculado con `rango95` para que abarque el 95 % de confianza de los valores que producirá `f`. `n` y `m` deben ser mayor que 0.

El tipo `Expr` no aparece en la signatura de la función, porque va a ser usado junto con `eval e` en el próximo ejercicio, dado un valor `e :: Expr`. Notar que `eval e :: G Float`.

Dicho de otra manera, esta función toma un generador de números `f :: G Float` y genera un histograma a partir de tomar `n` muestras de `f`.

Las funciones `muestra`, `histograma` y `rango95` van a ser útiles.

Ejercicio 10

Definir la función `evalHistograma :: Int → Int → Expr → G Histograma`.

`evalHistograma m n e g` evalúa la expresión `e` usando el generador `g` un total de `n` veces. `n` y `m` deben ser mayor que 0.

El histograma se arma según `armarHistograma` y tendrá `m` casilleros finitos.

Ejercicio 11

Definir la función `mostrar :: Expr → String` para convertir una expresión en un `String` con la notación infija de operadores usuales, y evitando algunos paréntesis innecesarios. En particular queremos evitar paréntesis en sumas dentro de sumas, y en productos dentro de productos. **No se permite recursión explícita**.

Disponemos para esto de `show :: Float → String` definida en `Prelude` y de las siguientes definiciones no exportadas en el módulo `Expr`:

```
data ConstructorExpr = CEConst
                     | CERango
                     | CESuma | CEResta | CEMult | CEDiv
  deriving (Show, Eq)

constructor :: Expr → ConstructorExpr
maybeParen :: Bool → String → String
```

- `constructor e` devuelve el constructor de la expresión `e`.
- `maybeParen b s` devuelve `s` entre paréntesis si `b` es `True`, y `s` sin cambios si `b` es `False`.

Por ejemplo:

```
ghci> mostrar (Suma (Suma (Suma (Const 1) (Const 2)) (Const 3)) (Const 4))
"1.0 + 2.0 + 3.0 + 4.0"

ghci> mostrar (Suma (Const 1) (Suma (Const 2) (Suma (Const 3) (Const 4))))
"1.0 + 2.0 + 3.0 + 4.0"

ghci> mostrar (Div (Suma (Rango 1 5) (Mult (Const 3) (Rango 100 105))) (Const 2))
"(1.0~5.0 + (3.0 * 100.0~105.0)) / 2.0"

ghci> mostrar (Resta (Resta (Const 1) (Const 2)) (Resta (Const 3) (Const 4)))
"(1.0 - 2.0) - (3.0 - 4.0)"

ghci> mostrar (Resta (Resta (Resta (Const 1) (Const 2)) (Const 3)) (Const 4))
"((1.0 - 2.0) - 3.0) - 4.0"
```

Módulo `Expr.Parser`

El módulo `Expr.Parser` nos permite convertir `String` en `Expr`. **Este módulo está completo** y puede ser cómodo para escribir algunos tests, aunque no es obligatorio usarlo.

- `parse :: String → Expr` devuelve un `Expr` si el `String` es una expresión válida o lanza da error en otro caso. **Es una función parcial**.
- `parseEither :: String → Either String Expr` similar a la anterior, pero devuelve un `String` describiendo el error si la expresión no es válida. **Es una función total**.

Módulo `App`

El módulo `App` define la aplicación `incierticalc`. **Este módulo está completo**. Define además la función auxiliar `probarHistograma` que puede usarse para probar el histograma de una expresión a partir de un generador directamente.

Cuando carguen `ghci` mediante `make repl` se va a cargar este módulo automáticamente, lo que les permite usar todos los módulos anteriores directamente.

Demostración

Ejercicio 12

Necesitamos demostrar que toda expresión tiene un literal más que su cantidad de operadores. Los literales son las constantes y los rangos. Para esto se dispone de las siguientes definiciones[2]:

```
data Nat = Z | S Nat

suma :: Nat → Nat → Nat
suma Z m = m                  -- {S1}
suma (S n) m = S (suma n m)   -- {S2}

cantLit :: Expr → Nat
cantLit (Const _)   = S Z                              -- {L1}
cantLit (Rango _ _) = S Z                              -- {L2}
cantLit (Suma a b)  = suma (cantLit a) (cantLit b)     -- {L3}
cantLit (Resta a b) = suma (cantLit a) (cantLit b)     -- {L4}
cantLit (Mult a b)  = suma (cantLit a) (cantLit b)     -- {L5}
cantLit (Div a b)   = suma (cantLit a) (cantLit b)     -- {L6}

cantOp :: Expr → Nat
cantOp (Const _)   = Z                                  -- {O1}
cantOp (Rango _ _) = Z                                  -- {O2}
cantOp (Suma a b)  = S (suma (cantOp a) (cantOp b))     -- {O3}
cantOp (Resta a b) = S (suma (cantOp a) (cantOp b))     -- {O4}
cantOp (Mult a b)  = S (suma (cantOp a) (cantOp b))     -- {O5}
cantOp (Div a b)   = S (suma (cantOp a) (cantOp b))     -- {O6}
```

La propiedad a demostrar queda expresada de la siguiente manera:

$$\forall \texttt{e :: Expr} \cdot \texttt{cantLit e} = \texttt{S (cantOp e)}$$

Se pide:

a) Definir el predicado unario correspondiente a una demostración por inducción estructural (¿en qué estructura?) de esta propiedad.

b) Definir el esquema formal de inducción estructural correspondiente a dicha demostración. Incluir todos los cuantificadores necesarios (los cuantificadores son los ∀s y los ∃s).

c) Demostrar los casos correspondientes a los casos base y al constructor `Suma`. Los demás casos inductivos son análogos a este último, y por eso les pedimos que no los escriban para este trabajo práctico. En general en la materia siempre tendrán que escribir todos los casos, aunque sean análogos o similares, excepto que les digamos explícitamente que no es necesario.

   - Todos los pasos de la demostración deben estar debidamente justificados usando las herramientas que vimos en clase.
   - Pueden asumir el siguiente lema como válido. No hace falta demostrarlo:

$$\{\texttt{CONMUT}\} \quad \forall \texttt{n}, \texttt{m :: Nat} \cdot \texttt{suma n m} = \texttt{suma m n}$$

Pautas de Entrega

Se debe entregar a través del campus un único archivo llamado "tp1.zip" conteniendo el código con la implementación de las funciones pedidas. Para eso, ya se encuentra disponible la entrega "TP1 - Programación Funcional" en la solapa "TPs" (configurada de forma grupal para que sólo

[2] Estas funciones están definidas usando pattern matching y recursión explícita para facilitar la demostración, sin embargo no dejen de practicar cómo seria una demostración donde estén definidas usando `foldExpr`.

una persona haga la entrega en nombre del grupo). El código entregado **debe** incluir tests que permitan probar las funciones definidas. El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aquellas disponibles en la sección Útil del campus y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.List`, `Data.Map`, `Data.Function`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Tests: cada ejercicio debe contar con uno o más ejemplos que muestren que exhibe la funcionalidad solicitada. Para esto se recomienda la codificación de tests usando el paquete HUnit https://hackage.haskell.org/package/HUnit. El esqueleto provisto incluye algunos ejemplos de cómo utilizarlo para definir casos de test para cada ejercicio.

También se dispone de un `Makefile` que permite compilar y ejecutar los tests de los ejercicios y descargar HUnit directamente.

- `make test` compila y ejecuta todos los tests.
- `make repl` inicia GHCi con los módulos cargados.
- `make watch` inicia GHCid y ejecuta los tests cada vez que se modifica un archivo.
- `make run` compila y ejecuta la aplicación interactiva.

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de Haskell o disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., chat-GPT, copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la versión del lenguaje Haskell al 2010, disponible online en https://www.haskell.org/onlinereport/haskell2010.

- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en https://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en https://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en https://www.haskell.org/hoogle.