DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 1 - Curryficados

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

Integrante	LU	Correo electrónico
Castro Eguren, Aitor José	292/24	aitorcastro05@gmail.com
Polo López, David	629/24	davidpolo235@gmail.com
Ginsberg, Mario Ezequiel	145/14	ezequielginsberg@gmail.com
Medrano, Lisette Sandra	1240/23	lisettemedrano84@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar/

Índice

1. Introducción: Calculadora Incierta

El presente trabajo práctico consiste en completar algunos módulos de la implementación de una Calculadora Incierta. Esta calculadora opera sobre expresiones *especiales*, que además de poder contener números, puede contener rangos numéricos. Las operaciones que admiten estas expresiones son las básicas: suma ($+$), resta ($-$), multiplicación ($*$) y división ($/$).

A la hora de calcular el valor de una expresión, vamos a tener que reemplazar cada rango numérico por un valor que estará dentro de dicho rango con una confianza del 95 % y cuya distribución será la de una $\mathcal{N}(a, b)$, siendo $a < b :: \texttt{Float}$ los extremos del rango.

Para determinar el resultado de la expresión y quitar la ambigüedad de los rangos numéricos, se tomará un gran número de muestras aleatorias usando la variable aleatoria con distribución normal mencionada anteriormente. Al combinar los resultados de todas estas muestras, se creará un histograma para ver cómo se distribuyen los resultados.

a. Consignas

Vale la pena escribir las consignas? Por un lado quedaría como un informe autocontenido, que está bueno, pero por otro lado hay que ponerse a copiar, pegar y darle formato a todos los ejercicios XD

b. Forma de trabajo

Nos resultó cómodo realizar juntos, como equipo, las funciones del módulo `Util` (ejercicios 1 y 2), y la demostración (ejercicio 12), mientras que el resto de los ejercicios fueron divididos de forma equitativa. Al finalizar, hicimos una puesta en común para que todos estemos al tanto de lo realizado por nuestros compañeros.

2. Ejercicios 1 al 11

Estos ejercicios fueron resueltos en código y por lo tanto no hay demasiado para aclarar en este informe más que el hecho de que se pueden encontrar en los archivos `src/Util.hs`, `src/Histograma.hs`, `src/Expr.hs` y `test/Main.hs`

Podríamos contar cada uno qué quizo testear con los tests de cada ejercicio, se me ocurre.

3. Ejercicio 12 - Demostración

a. Predicado Unario

Sea $\mathcal{P}(e) = \texttt{cantLit}\ e\ \texttt{=}\ \texttt{S}\ \texttt{(cantOp}\ e\texttt{)}$, vamos a hacer inducción sobre $e :: \texttt{Expr}$

b. Esquema de inducción

Casos base

1. $\forall x :: \texttt{Expr.}\ \mathcal{P}(\texttt{Const}\ x)$
2. $\forall a, b :: \texttt{Float.}\ \mathcal{P}(\texttt{Rango}\ a\ b)$

Pasos inductivos

1. $\forall e_1, e_2 :: \texttt{Expr.}\ \mathcal{P}(e_1) \land \mathcal{P}(e_2) \Rightarrow \mathcal{P}(\texttt{Suma}\ e_1\ e_2)$
2. $\forall e_1, e_2 :: \texttt{Expr.}\ \mathcal{P}(e_1) \land \mathcal{P}(e_2) \Rightarrow \mathcal{P}(\texttt{Resta}\ e_1\ e_2)$
3. $\forall e_1, e_2 :: \texttt{Expr.}\ \mathcal{P}(e_1) \land \mathcal{P}(e_2) \Rightarrow \mathcal{P}(\texttt{Mult}\ e_1\ e_2)$
4. $\forall e_1, e_2 :: \texttt{Expr.}\ \mathcal{P}(e_1) \land \mathcal{P}(e_2) \Rightarrow \mathcal{P}(\texttt{Div}\ e_1\ e_2)$

c. Demostración

Casos base

1. Sea $x :: \texttt{Float}$, queremos ver que vale $\mathcal{P}(\texttt{Const}\ x)$:

$$\mathcal{P}(\texttt{Const}\ x) = \texttt{cantLit}\ (\texttt{Const}\ x)\ \texttt{=}\ \texttt{S}\ (\texttt{cantOp}\ (\texttt{Const}\ x))$$

Lado izquierdo:

$$\texttt{cantLit}\ (\texttt{Const}\ x) = \texttt{S}\ \texttt{Z} \quad \{\texttt{L1}\}$$

Lado derecho:

$$\texttt{S}\ (\texttt{cantOp}\ (\texttt{Const}\ x)) = \texttt{S}\ \texttt{Z} \quad \{\texttt{O1}\}$$

2. Sean $a, b :: \texttt{Float}$, queremos ver que vale $\mathcal{P}(\texttt{Rango}\ a\ b)$:

$$\mathcal{P}(\texttt{Rango}\ a\ b) = \texttt{cantLit}\ (\texttt{Rango}\ a\ b)\ \texttt{=}\ \texttt{S}\ (\texttt{cantOp}\ (\texttt{Rango}\ a\ b))$$

Lado izquierdo:

$$\texttt{cantLit}\ (\texttt{Rango}\ a\ b) = \texttt{S}\ \texttt{Z} \quad \{\texttt{L2}\}$$

Lado derecho:

$$\texttt{S}\ (\texttt{cantOp}\ (\texttt{Rango}\ a\ b)) = \texttt{S}\ \texttt{Z} \quad \{\texttt{O2}\}$$

Pasos inductivos

En este caso vamos a demostrar únicamente el paso inductivo para el constructor `Suma`, por simplicidad y porque el resto de casos son análogos. Entendemos que para una demostración válida y completa habría que demostrar todos los pasos inductivos.

Sean $e_1, e_2 :: \texttt{Expr}$, asumimos que valen las `HI`: $\mathcal{P}(e_1)$ y $\mathcal{P}(e_2)$, y queremos ver que vale:

$$\mathcal{P}(\texttt{Suma}\ e_1\ e_2) = \texttt{cantLit}\ (\texttt{Suma}\ e_1\ e_2) = \texttt{S}\ (\texttt{cantOp}\ (\texttt{Suma}\ e_1\ e_2))$$

Lado izquierdo

$$\begin{aligned}
\texttt{cantLit}\ (\texttt{Suma}\ e_1\ e_2) &= \texttt{suma}\ (\texttt{cantLit}\ e_1)\ (\texttt{cantLit}\ e_2) && \{L3\}\\
&= \texttt{suma}\ (\texttt{S}\ (\texttt{cantOp}\ e_1))\ (\texttt{S}\ (\texttt{cantOp}\ e_2)) && \{\texttt{HI}\}\\
&= \texttt{S}\ (\texttt{suma}\ (\texttt{cantOp}\ e_1)\ (\texttt{S}\ (\texttt{cantOp}\ e_2))) && \{S2\}\\
&= \texttt{S}\ (\texttt{suma}\ (\texttt{S}\ (\texttt{cantOp}\ e_2))\ (\texttt{cantOp}\ e_1)) && \{\texttt{CONMUT}\}\\
&= \texttt{S}\ (\texttt{S}\ (\texttt{suma}\ (\texttt{cantOp}\ e_2)\ (\texttt{cantOp}\ e_1))) && \{S2\}
\end{aligned}$$

Lado derecho

$$\begin{aligned}
\texttt{S}\ (\texttt{cantOp}\ (\texttt{Suma}\ e_1\ e_2)) &= \texttt{S}\ (\texttt{S}\ (\texttt{suma}\ (\texttt{cantOp}\ e_1)\ (\texttt{cantOp}\ e_2))) && \{\texttt{O3}\}\\
&= \texttt{S}\ (\texttt{S}\ (\texttt{suma}\ (\texttt{cantOp}\ e_2)\ (\texttt{cantOp}\ e_1))) && \{\texttt{CONMUT}\}
\end{aligned}$$

Dado que se llego a la misma expresión desde ambos lados de la igualdad, queda demostrado el paso inductivo para el generador recursivo `Suma`.

Asumiendo que el resto de pasos inductivos son análogos como se mencionó en la consigna del ejercicio, y habiendo probado tanto todos los casos base como uno de los pasos inductivos, queda demostrada la propiedad:

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

□

4. Apéndice

a. Módulo Util

```haskell
module Util where

-- | @alinearDerecha n s@ agrega espacios a la izquierda de @s@ hasta
   ↪ que su longitud sea @n@.
-- Si @s@ ya tiene longitud @>= n@, devuelve @s@.
alinearDerecha :: Int -> String -> String
alinearDerecha n s = foldr (:) s listaEspaciosNecesarios
  where
    listaEspaciosNecesarios = replicate cantidadEspacios ' '
    cantidadEspacios = max 0 (n - length s)
-- creamos una lista con la cant de ' ' necesarios
-- vamos haciendo ' ' : (' ' : (... : s)) con foldr

-- | Dado un índice y una función, actualiza 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.
actualizarElem :: Int -> (a -> a) -> [a] -> [a]
actualizarElem n f xs = zipWith (\a i -> if i == n then f a else a) xs
   ↪ [0..]
-- creamos la lista [0,1,2, ...] que representa los indices
-- con zipWith aplicamos una funcion que nos dice si es o no el indice
   ↪ que buscamos.
-- si lo es, aplicamos la funcion f a ese elemento

-- | infinito positivo (Haskell no tiene literal para +infinito)
infinitoPositivo :: Float
infinitoPositivo = 1 / 0

-- | infinito negativo (Haskell no tiene literal para -infinito)
infinitoNegativo :: Float
infinitoNegativo = -(1 / 0)
```

b. Módulo Histograma

```haskell
-- | Un `Histograma` es una estructura de datos que permite contar cuá
   ↪ ntos valores hay en cada rango.
-- @vacio n (a, b)@ devuelve un histograma vacío con n+2 casilleros:
--
-- * @(-inf, a)@
-- * @[a, a + tamIntervalo)@
-- * @[a + tamIntervalo, a + 2*tamIntervalo)@
-- * ...
-- * @[b - tamIntervalo, b)@
-- * @[b, +inf)@
--
```

```haskell
-- ‘vacio‘, ‘agregar‘ e ‘histograma‘ se usan para construir un
   ↪ histograma.
module Histograma
  ( Histograma, -- No se exportan los constructores
    vacio,
    agregar,
    histograma,
    Casillero (..),
    casMinimo,
    casMaximo,
    casCantidad,
    casPorcentaje,
    casilleros,
  )
where

import Util ( actualizarElem, infinitoNegativo, infinitoPositivo )
import Data.List (zipWith4)

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

-- | Inicializa un histograma vacío con @n@ casilleros para representar
-- valores en el rango y 2 casilleros adicionales para los valores fuera
   ↪  del rango.
-- Require que @l < u@ y @n >= 1@.
vacio :: Int -> (Float, Float) -> Histograma
vacio n (l, u) = Histograma l t cs
  where
  t = ((u - l) / fromIntegral n) -- 't' es el tamaño de cada intervalo
  cs = (replicate (n + 2) 0) -- 'cs' es la lista con la cantidad de
     ↪ valores de cada casillero (se inicializan en 0)

-- | Agrega un valor al histograma.
agregar :: Float -> Histograma -> Histograma
agregar x (Histograma i t cs) = Histograma i t (actualizarElem
   ↪ obtenerIndice (+1) cs)
  where
  obtenerIndice = entreRango (1 + fromIntegral (floor ((x - i) / t)))
  entreRango indice = min (length cs - 1) (max 0 indice) -- Asegura que
     ↪ el indice no se vaya del rango [0, length cs)

-- | Arma un histograma a partir de una lista de números reales con la
   ↪ cantidad de casilleros y rango indicados.
histograma :: Int -> (Float, Float) -> [Float] -> Histograma
histograma n (a,b) = foldr agregar (vacio n (a, b))

-- | Un ‘Casillero‘ representa un casillero del histograma con sus lí
   ↪ mites, cantidad y porcentaje.
-- Invariante: Sea @Casillero m1 m2 c p@ entonces @m1 < m2@, @c >= 0@,
   ↪ @0 <= p <= 100@
data Casillero = Casillero Float Float Int Float
  deriving (Show, Eq)
```

```haskell
-- | Mínimo valor del casillero (el límite inferior puede ser @-inf@)
casMinimo :: Casillero -> Float
casMinimo (Casillero m _ _ _) = m

-- | Máximo valor del casillero (el límite superior puede ser @+inf@)
casMaximo :: Casillero -> Float
casMaximo (Casillero _ m _ _) = m

-- | Cantidad de valores en el casillero. Es un entero @>= 0@.
casCantidad :: Casillero -> Int
casCantidad (Casillero _ _ c _) = c

-- | Porcentaje de valores en el casillero respecto al total de valores
   ↪ en el histograma. Va de 0 a 100.
casPorcentaje :: Casillero -> Float
casPorcentaje (Casillero _ _ _ p) = p

-- | Dado un histograma, devuelve la lista de casilleros con sus límites
   ↪ , cantidad y porcentaje.
casilleros :: Histograma -> [Casillero]
casilleros (Histograma i t cs) = zipWith4 Casillero limiteInf limiteSup
   ↪ cs porcentaje
  where
    cantidadesTotal = sum cs
    n = length cs - 2
    limiteInf = [infinitoNegativo] ++ [ i + fromIntegral k * t | k <- [0
       ↪  .. n]]
    limiteSup = [ i + fromIntegral k * t | k <- [0 .. n] ] ++ [
       ↪ infinitoPositivo]
    porcentaje = map (\c -> (fromIntegral c / fromIntegral (max 1
       ↪ cantidadesTotal)) * 100) cs -- el (max 1 cantidadesTotal) es
       ↪ para que no se indefina la división
```

c. Módulo Expr

```haskell
module Expr
  ( Expr (..),
    recrExpr,
    foldExpr,
    eval,
    armarHistograma,
    evalHistograma,
    mostrar,
  )
where

import Generador ( G, Gen, dameUno, muestra, rango95,
   ↪ genNormalConSemilla )
import Histograma ( Histograma, histograma )

-- | Expresiones aritméticas con rangos
```

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

-- recrExpr :: ... anotar el tipo ...
recrExpr :: (Float->b) -> (Float->Float->b) -> (b->b->Expr->Expr->b) ->
   ↪ (b->b->Expr->Expr->b) -> (b->b->Expr->Expr->b) -> (b->b->Expr->
   ↪ Expr->b) -> Expr -> b
recrExpr cConst cRango cSuma cResta cMult cDiv e = case e of
  Const x -> cConst x
  Rango a b -> cRango a b
  Suma e1 e2 -> cSuma (rec e1) (rec e2) e1 e2
  Resta e1 e2 -> cResta (rec e1) (rec e2) e1 e2
  Mult e1 e2 -> cMult (rec e1) (rec e2) e1 e2
  Div e1 e2 -> cDiv (rec e1) (rec e2) e1 e2
  where rec = recrExpr cConst cRango cSuma cResta cMult cDiv

-- foldExpr :: ... anotar el tipo ...
foldExpr :: (Float->b) -> (Float->Float->b) -> (b->b->b) -> (b->b->b) ->
   ↪  (b->b->b) -> (b->b->b) -> Expr -> b
foldExpr cConst cRango cSuma cResta cMult cDiv e = case e of
  Const x -> cConst x
  Rango a b -> cRango a b
  Suma e1 e2 -> cSuma (rec e1) (rec e2)
  Resta e1 e2 -> cResta (rec e1) (rec e2)
  Mult e1 e2 -> cMult (rec e1) (rec e2)
  Div e1 e2 -> cDiv (rec e1) (rec e2)
  where rec = foldExpr cConst cRango cSuma cResta cMult cDiv

-- | Evaluar expresiones dado un generador de números aleatorios
eval :: Expr -> G Float
eval = foldExpr (\x g ->(x, g)) (\a b g -> dameUno (a,b) g) (operacion
   ↪ (+)) (operacion (-)) (operacion (*)) (operacion (/))

operacion :: (Float -> Float -> Float) -> G Float -> G Float -> Gen -> (
   ↪ Float, Gen)
operacion op fl fr g0 = (\(vL, g1) ->(\(vR, g2) -> (op vL vR, g2)) (fr
   ↪ g1)) (fl g0)

-- | @armarHistograma m n f g@ arma un histograma con @m@ casilleros
-- a partir del resultado de tomar @n@ muestras de @f@ usando el
   ↪ generador @g@.
armarHistograma :: Int -> Int -> G Float -> G Histograma --
   ↪ armarHistograma :: Int -> Int -> (Gen -> (Float, Gen)) -> (Gen ->
   ↪ (Histograma, Gen))
armarHistograma m n f g = (\(xs,g1)-> (\(a,b)-> (histograma m (a,b) xs,
   ↪ g1) ) (rango95 xs)) (muestra f n g)
```

```haskell
-- | @evalHistograma m n e g@ evalúa la expresión @e@ usando el
   ↪ generador @g@ @n@ veces
-- devuelve un histograma con @m@ casilleros y rango calculado con
   ↪ @rango95@ para abarcar el 95% de confianza de los valores.
-- @n@ debe ser mayor que 0.
evalHistograma :: Int -> Int -> Expr -> G Histograma
evalHistograma m n e = armarHistograma m n (eval e)

-- | Mostrar las expresiones, pero evitando algunos paréntesis
   ↪ innecesarios.
-- En particular queremos evitar paréntesis en sumas y productos
   ↪ anidados.
mostrar :: Expr -> String
mostrar e = recrExpr show (\a b -> show a ++ "~" ++ show b)
                      (\str1 str2 e1 e2 -> maybeParen (
                         ↪ esConstrRecursivoDistintoDe e1 CESuma) str1
                         ↪ ++ " + " ++ maybeParen (
                         ↪ esConstrRecursivoDistintoDe e2 CESuma) str2)
                      (\str1 str2 e1 e2 -> maybeParen (esConstrRecursivo
                         ↪ e1) str1 ++ " - " ++ maybeParen (
                         ↪ esConstrRecursivo e2) str2)
                      (\str1 str2 e1 e2 -> maybeParen (
                         ↪ esConstrRecursivoDistintoDe e1 CEMult) str1
                         ↪ ++ " * " ++ maybeParen (
                         ↪ esConstrRecursivoDistintoDe e2 CEMult) str2)
                      (\str1 str2 e1 e2 -> maybeParen (esConstrRecursivo
                         ↪ e1) str1 ++ " / " ++ maybeParen (
                         ↪ esConstrRecursivo e2) str2)
                      e
  where
    esConstrRecursivo e = constructor e /= CEConst && constructor e /=
       ↪ CERango
    esConstrRecursivoDistintoDe e c = esConstrRecursivo e && constructor
       ↪  e /= c

    -- Para los constructores no recursivos usamos la función show dada
    -- En los constructores recursivos diferenciamos los casos de
       ↪ Multiplicación y Suma para no poner paréntesis innecesarios

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

-- | Indica qué constructor fue usado para crear la expresión.
constructor :: Expr -> ConstructorExpr
constructor (Const _) = CEConst
constructor (Rango _ _) = CERango
constructor (Suma _ _) = CESuma
constructor (Resta _ _) = CEResta
constructor (Mult _ _) = CEMult
constructor (Div _ _) = CEDiv

-- | Agrega paréntesis antes y después del string si el Bool es True.
```

```
maybeParen :: Bool -> String -> String
maybeParen True s = "(" ++ s ++ ")"
maybeParen False s = s
```

d. Tests

```
module Main (main) where

import App
import Expr
import Expr.Parser
import GHC.Stack (HasCallStack)
import Generador
import Histograma
import Test.HUnit
import Util

main :: IO ()
main = runTestTTAndExit allTests

-- | Función auxiliar para marcar tests como pendientes a completar
completar :: (HasCallStack) => Test
completar = TestCase (assertFailure "COMPLETAR")

allTests :: Test
allTests =
  test
    [ "Ej 1 - Util.alinearDerecha" ~: testsAlinearDerecha,
      "Ej 2 - Util.actualizarElem" ~: testsActualizarElem,
      "Ej 3 - Histograma.vacio" ~: testsVacio,
      "Ej 4 - Histograma.agregar" ~: testsAgregar,
      "Ej 5 - Histograma.histograma" ~: testsHistograma,
      "Ej 6 - Histograma.casilleros" ~: testsCasilleros,
      "Ej 7 - Expr.recrExpr" ~: testsRecr,
      "Ej 7 - Expr.foldExpr" ~: testsFold,
      "Ej 8 - Expr.eval" ~: testsEval,
      "Ej 9 - Expr.armarHistograma" ~: testsArmarHistograma,
      "Ej 10 - Expr.evalHistograma" ~: testsEvalHistograma,
      "Ej 11 - Expr.mostrar" ~: testsMostrar,
      "Expr.Parser.parse" ~: testsParse,
      "App.mostrarFloat" ~: testsMostrarFloat,
      "App.mostrarHistograma" ~: testsMostrarHistograma

    ]

-- Ej 1
testsAlinearDerecha :: Test
testsAlinearDerecha =
  test
    [ alinearDerecha 6 "hola" ~?= "  hola",
      alinearDerecha 10 "incierticalc" ~?= "incierticalc",
      alinearDerecha 0 "noAgregaEspacios" ~?= "noAgregaEspacios",
```

```
      alinearDerecha (-1) "negativo" ~?= "negativo",
      alinearDerecha 14 "espaciosJustos" ~?= "espaciosJustos",
      alinearDerecha 1 "" ~?= " ",
      alinearDerecha 0 "" ~?= "",
      alinearDerecha 1 " " ~?= " ",
      alinearDerecha 12 " conEspacio" ~?= "  conEspacio"
    ]

-- Ej 2
testsActualizarElem :: Test
testsActualizarElem =
  test
    [ actualizarElem 0 (+ 10) [1, 2, 3] ~?= [11, 2, 3],
      actualizarElem 1 (+ 10) [1, 2, 3] ~?= [1, 12, 3],
      actualizarElem 2 (+ 10) [1, 2, 3] ~?= [1, 2, 13],
      actualizarElem 3 (+ 10) [1, 2, 3] ~?= [1, 2, 3],
      actualizarElem (-1) (+ 10) [1, 2, 3] ~?= [1, 2, 3],
      actualizarElem 0 (+ 1) [1, 2, 3] ~?= [2, 2, 3],
      actualizarElem 0 (+ 1) [] ~?= []
    ]

-- Ej 3
testsVacio :: Test
testsVacio =
  test
    [ casilleros (vacio 1 (0, 10))
        ~?= [ Casillero infinitoNegativo 0 0 0,
              Casillero 0 10 0 0,
              Casillero 10 infinitoPositivo 0 0
            ],
      casilleros (vacio 3 (0, 6))
        ~?= [ Casillero infinitoNegativo 0 0 0,
              Casillero 0 2 0 0,
              Casillero 2 4 0 0,
              Casillero 4 6 0 0,
              Casillero 6 infinitoPositivo 0 0
            ],
      casilleros (vacio 2 (-5, -1)) -- caso rangos negativos
        ~?= [ Casillero infinitoNegativo (-5) 0 0,
              Casillero (-5) (-3) 0 0,
              Casillero (-3) (-1) 0 0,
              Casillero (-1) infinitoPositivo 0 0
            ],
        casilleros (vacio 2 (1.5, 5.5)) -- caso con rangos con coma
        ~?= [ Casillero infinitoNegativo 1.5 0 0,
              Casillero 1.5 3.5 0 0,
              Casillero 3.5 5.5 0 0,
              Casillero 5.5 infinitoPositivo 0 0
            ]
    ]

-- Ej 4
testsAgregar :: Test
```

```
testsAgregar =
  let h0 = vacio 3 (0, 6)
   in test
        [ casilleros (agregar 0 h0)
            ~?= [ Casillero infinitoNegativo 0 0 0,
                  Casillero 0 2 1 100, -- El 100% de los valores están
                     ↪ acá
                  Casillero 2 4 0 0,
                  Casillero 4 6 0 0,
                  Casillero 6 infinitoPositivo 0 0
                ],
          casilleros (agregar 2 h0)
            ~?= [ Casillero infinitoNegativo 0 0 0,
                  Casillero 0 2 0 0,
                  Casillero 2 4 1 100, -- El 100% de los valores están
                     ↪ acá
                  Casillero 4 6 0 0,
                  Casillero 6 infinitoPositivo 0 0
                ],
          casilleros (agregar (-1) h0)
            ~?= [ Casillero infinitoNegativo 0 1 100, -- El 100% de los
               ↪ valores están acá
                  Casillero 0 2 0 0,
                  Casillero 2 4 0 0,
                  Casillero 4 6 0 0,
                  Casillero 6 infinitoPositivo 0 0
                ],
          casilleros (agregar 6 h0) --Caso extremo [4 , +infinito)
           ~?= [  Casillero infinitoNegativo 0 0 0,
                  Casillero 0 2 0 0,
                  Casillero 2 4 0 0,
                  Casillero 4 6 0 0,
                  Casillero 6 infinitoPositivo 1 100 -- El 100% de los
                     ↪ valores están acá
                ],
          casilleros (agregar 1 (agregar 2 h0)) --Caso aplicado 2 veces
             ↪ la funcion
            ~?= [ Casillero infinitoNegativo 0 0 0,
                  Casillero 0 2 1 50, --El 50% de los valores están acá
                  Casillero 2 4 1 50, --El 50% de los valores están acá
                  Casillero 4 6 0 0,
                  Casillero 6 infinitoPositivo 0 0
                ]

        ]

-- Ej 5
testsHistograma :: Test
testsHistograma =
  test
    [ histograma 4 (1, 5) [1, 2, 3] ~?= agregar 3 (agregar 2 (agregar 1
       ↪ (vacio 4 (1, 5)))),
```

```
      histograma 2 (1.5, 3) [1.5, 3.75, 2.225] ~?= agregar 2.225 (
         ↪ agregar 3.75 (agregar 1.5 (vacio 2 (1.5, 3)))), --Caso con
         ↪ numeros con coma en rangos y lista

      histograma 3 (-6, -1) [-8, -4, 1] ~?= agregar 1 (agregar (-4) (
         ↪ agregar (-8) (vacio 3 (-6, -1)))), --Caso negativos en rango
         ↪  y lista

      histograma 3 (1, 3) [] ~?= vacio 3 (1, 3) -- Caso lista vacia
    ]

-- Ej 6
testsCasilleros :: Test
testsCasilleros =
  test
    [ casilleros (vacio 3 (0, 6))
        ~?= [ Casillero infinitoNegativo 0.0 0 0.0,
              Casillero 0.0 2.0 0 0.0,
              Casillero 2.0 4.0 0 0.0,
              Casillero 4.0 6.0 0 0.0,
              Casillero 6.0 infinitoPositivo 0 0.0
            ],
      casilleros (agregar 2 (vacio 3 (0, 6)))
        ~?= [ Casillero infinitoNegativo 0.0 0 0.0,
              Casillero 0.0 2.0 0 0.0,
              Casillero 2.0 4.0 1 100.0,
              Casillero 4.0 6.0 0 0.0,
              Casillero 6.0 infinitoPositivo 0 0.0
            ],
      casilleros (histograma 2 (1,5) [3]) --Caso con la funcion
         ↪ histograma
        ~?= [ Casillero infinitoNegativo 1.0 0 0.0,
              Casillero 1.0 3.0 0 0.0,
              Casillero 3.0 5.0 1 100.0,
              Casillero 5.0 infinitoPositivo 0 0.0
            ],
       casilleros (histograma 2 (1,5) [3, 2, -1, -2]) --Caso con dif
          ↪ porcentajes
        ~?= [ Casillero infinitoNegativo 1.0 2 50.0,
              Casillero 1.0 3.0 1 25.0,
              Casillero 3.0 5.0 1 25.0,
              Casillero 5.0 infinitoPositivo 0 0.0
            ]
    ]

-- Ej 7
testsRecr :: Test
testsRecr =
  test
    [ -- True: en la raíz los dos hijos son iguales
      hayNodoConSubarbolesIguales (Suma (Const 1) (Const 1)) ~?= True,

      -- False: en la raíz (2) != (3)
```

```
      hayNodoConSubarbolesIguales (Mult (Const 2) (Const 3)) ~?= False,

      -- True: más profundo; los dos SUMA internos son iguales
      hayNodoConSubarbolesIguales
        (Suma (Suma (Const 1) (Const 2)) (Suma (Const 1) (Const 2)))
        ~?= True
    ]
  where
    hayNodoConSubarbolesIguales :: Expr -> Bool
    hayNodoConSubarbolesIguales = recrExpr (const False) (\_ _ -> False)
       ↪  operacion operacion operacion operacion
                                  where
                                    operacion reci recd i d =  reci ||
                                       ↪ recd || (i==d)

-- Ej 7
testsFold :: Test
testsFold =
  test
    [ tamFold (Const 5) ~?= 1,
      tamFold (Suma (Resta (Const 1) (Const 2)) (Mult (Const 3) (Const
         ↪ 4))) ~?= 7,
      tamFold (Div (Suma (Const 1) (Const 2)) (Rango 0 1)) ~?= 5
    ]
  where
    tamFold :: Expr -> Int
    tamFold = foldExpr (const 1) (\_ _ -> 1) operacion operacion
       ↪ operacion operacion
                                  where
                                    operacion reci recd = 1 + reci +
                                       ↪ recd

-- Ej 8
testsEval :: Test
testsEval =
  test
    [ fst (eval (Suma (Rango 1 5) (Const 1)) genFijo) ~?= 4.0,
      fst (eval (Suma (Rango 1 5) (Const 1)) (genNormalConSemilla 0))
         ↪ ~?= 3.7980492,
      -- el primer rango evalua a 2.7980492 y el segundo a 3.1250308
      fst (eval (Suma (Rango 1 5) (Rango 1 5)) (genNormalConSemilla 0))
         ↪ ~?= 5.92308,
      --
      fst (eval (Div (Const 8) (Const 4)) genFijo) ~?= 2.0,

      fst (eval (Suma (Const 1) (Rango 1 5)) (genNormalConSemilla 0))
         ↪ ~?= 3.7980492,

      -- Probamos que un rango de x a x es lo mismo que evaluar
         ↪ directamente x
      fst (eval (Const 2) (genNormalConSemilla 4)) ~?=
      fst (eval (Rango 2 2) (genNormalConSemilla 4))
    ]
```

```
-- Ej 9
testsArmarHistograma :: Test
testsArmarHistograma =
  test
    [ casilleros (fst (armarHistograma 3 3 (dameUno (1,5)) (
       ↪ genNormalConSemilla 5))) ~?=[Casillero infinitoNegativo
       ↪ 1.9035962 0 0.0,
       Casillero 1.9035962 2.8586945999999998 1 (1/3 * 100),
       Casillero 2.8586945999999998 3.813793 1 (1/3 * 100),
       Casillero 3.813793 4.7688914 1 (1/3 * 100),
       Casillero 4.7688914 infinitoPositivo 0 0.0],

      -- Probamos que con muchos valores si llega a cubrir los valores
         ↪ fuera de rango
      -- a diferencia de con pocos valores que queda con 0 elementos
      casilleros (fst (armarHistograma 3 10000 (dameUno (1,5)) (
         ↪ genNormalConSemilla 7))) ~?=[Casillero infinitoNegativo
         ↪ 1.0287318 263 (263/10000 * 100),
       Casillero 1.0287318 2.3566919000000004 2304 (2304/10000 * 100),
       Casillero 2.3566919000000004 3.6846520000000003 4852 (4852/10000
          ↪ * 100),
       Casillero 3.6846520000000003 5.01261213 2338 (2338/10000 * 100),
       Casillero 5.01261213 infinitoPositivo 243 (243/10000 * 100)],

      -- Probamos que si ponemos siempre el mismo valor, caen todos en
         ↪ el mismo casillero
      casilleros (fst (armarHistograma 4 3 (\x -> (2,x)) (
         ↪ genNormalConSemilla 7))) ~?=[Casillero infinitoNegativo 1.0
         ↪ 0 0.0,
       Casillero 1.0 1.5 0 0.0,
       Casillero 1.5 2.0 0 0.0,
       Casillero 2.0 2.5 3 100.0,
       Casillero 2.5 3.0 0 0.0,
       Casillero 3.0 infinitoPositivo 0 0.0]
    ]

-- Ej 10
testsEvalHistograma :: Test
testsEvalHistograma =
  test
    [-- Probamos que se distribuyen os valores correctamente en
       ↪ diferentes casilleros
    -- si la expresion contiene un rango
    casilleros (fst (evalHistograma 3 4 (Rango 1 3) (genNormalConSemilla
       ↪  4))) ~?=
    [Casillero infinitoNegativo 1.2331247 0 0.0,
     Casillero 1.2331247 1.57619244 1 (1/4 * 100),
     Casillero 1.57619244 1.9192603 2 (2/4 * 100),
     Casillero 1.9192603 2.26232792 1 (1/4 * 100),
     Casillero 2.26232792 infinitoPositivo 0 0.0],
```

```
    -- Probamos que si la expresion no tiene un rango (es constante) los
       ↪  valores caen siempre en el mismo casillero
    casilleros (fst (evalHistograma 3 4 (Const 2) (genNormalConSemilla
       ↪ 4))) ~?=
    [Casillero infinitoNegativo 1.0 0 0.0,
     Casillero 1.0 1.6666667 0 0.0,
     Casillero 1.6666667 2.3333334 4 100.0,
     Casillero 2.3333334 3.0000001 0 0.0,
     Casillero 3.0000001 infinitoPositivo 0 0.0]

    -- Aclaracion: No hacemos tests con expresiones mas complejas como
       ↪ sumas, restas, etc
    -- porque nos interesa ver como caen los valores finales en el
       ↪ histograma
    -- y no que se calculen bien, ya que eso es parte de los tests de la
       ↪  funcion eval
    -- que evalua expresiones
    ]

testsParse :: Test
testsParse =
  test
    [ parse "1" ~?= Const 1.0,
      parse "-1.7 ~ -0.5" ~?= Rango (-1.7) (-0.5),
      parse "1+2" ~?= Suma (Const 1.0) (Const 2.0),
      parse "1 + 2" ~?= Suma (Const 1.0) (Const 2.0),
      parse "1 + 2 * 3" ~?= Suma (Const 1.0) (Mult (Const 2.0) (Const
         ↪ 3.0)),
      parse "1 + 2 + 3" ~?= Suma (Suma (Const 1.0) (Const 2.0)) (Const
         ↪ 3.0),
      parse "1 + (2 + 3)" ~?= Suma (Const 1.0) (Suma (Const 2.0) (Const
         ↪ 3.0)),
      parse "1 + 2 ~ 3 + 4" ~?= Suma (Suma (Const 1.0) (Rango 2.0 3.0))
         ↪ (Const 4.0),
      parse "1 - 2 - 3 - 4" ~?= Resta (Resta (Resta (Const 1.0) (Const
         ↪ 2.0)) (Const 3.0)) (Const 4.0),
      parse "(((1 - 2) - 3) - 4)" ~?= Resta (Resta (Resta (Const 1.0) (
         ↪ Const 2.0)) (Const 3.0)) (Const 4.0),
      parse "1 " ~?= Const 1.0,
      parse "   1    " ~?= Const 1.0
    ]

-- Ej 11
testsMostrar :: Test
testsMostrar =
  test
    [ 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",
      mostrar (Suma (Suma (Suma (Const 1) (Const 2)) (Const 3)) (Const
         ↪ 4))
        ~?= "1.0 + 2.0 + 3.0 + 4.0",
```

```
      mostrar (Suma (Const 1) (Suma (Const 2) (Suma (Const 3) (Const 4))
        ↪ ))
        ~?= "1.0 + 2.0 + 3.0 + 4.0",
      mostrar (Suma (Suma (Const 1) (Const 2)) (Suma (Const 3) (Const 4)
        ↪ ))
        ~?= "1.0 + 2.0 + 3.0 + 4.0",
      mostrar (Mult (Mult (Mult (Const 1) (Const 2)) (Const 3)) (Const
        ↪ 4))
        ~?= "1.0 * 2.0 * 3.0 * 4.0",
      mostrar (Mult (Const 1) (Mult (Const 2) (Mult (Const 3) (Const 4))
        ↪ ))
        ~?= "1.0 * 2.0 * 3.0 * 4.0",
      mostrar (Mult (Mult (Const 1) (Const 2)) (Mult (Const 3) (Const 4)
        ↪ ))
        ~?= "1.0 * 2.0 * 3.0 * 4.0",
      mostrar (Resta (Resta (Const 1) (Const 2)) (Resta (Const 3) (Const
        ↪  4)))
        ~?= "(1.0 - 2.0) - (3.0 - 4.0)",
      mostrar (Resta (Resta (Resta (Const 1) (Const 2)) (Const 3)) (
        ↪ Const 4))
        ~?= "((1.0 - 2.0) - 3.0) - 4.0",
      mostrar (Suma (Mult (Suma (Const 1) (Const 2)) (Const 3)) (Const
        ↪ 4))
        ~?= "((1.0 + 2.0) * 3.0) + 4.0",
      mostrar (Mult (Suma (Suma (Const 1) (Const 2)) (Const 3)) (Const
        ↪ 4))
        ~?= "(1.0 + 2.0 + 3.0) * 4.0",

      -- Tests propios
      mostrar (parse "1") ~?= "1.0",
      mostrar (parse "1+2") ~?= "1.0 + 2.0",
      mostrar (parse "1*(1~2)") ~?= "1.0 * 1.0~2.0",
      mostrar (parse "-1 / (3-1)") ~?= "-1.0 / (3.0 - 1.0)"
    ]

testsMostrarFloat :: Test
testsMostrarFloat =
  test
    [ mostrarFloat 0.0 ~?= "0.00",
      mostrarFloat 1.0 ~?= "1.00",
      mostrarFloat (-1.0) ~?= "-1.00",
      -- Redondeo
      mostrarFloat 3.14159 ~?= "3.14",
      mostrarFloat 2.71828 ~?= "2.72",
      mostrarFloat 0.000001 ~?= "1.00e-6",
      mostrarFloat 100000 ~?= "100000.00",
      -- Infinitos
      mostrarFloat infinitoPositivo ~?= "+inf",
      mostrarFloat infinitoNegativo ~?= "-inf"
    ]

testsMostrarHistograma :: Test
testsMostrarHistograma =
```

```
  let h0 = vacio 3 (0, 6)
      h123 = agregar 1 (agregar 2 (agregar 3 h0))
   in test
        [ lines (mostrarHistograma h123)
            ~?= [ "6.00 - +inf |",
                  "4.00 - 6.00 |",
                  "2.00 - 4.00 |
                     ↪
                     ↪  66.67%",
                  "0.00 - 2.00 |
                     ↪                                                  ",
                  "-inf - 0.00 |"
                ],
          lines (mostrarHistograma (agregar 1 (vacio 3 (0, 1000))))
            ~?= [ "  1000.00 - +inf |",
                  "666.67 - 1000.00 |",
                  " 333.33 - 666.67 |",
                  "   0.00 - 333.33 |
                     ↪
                     ↪  100.00%",
                  "     -inf - 0.00 |"
                ]
        ]
```