Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Trabajo Final

Análisis de Lenguajes de Programación

HASKLAB

Rodríguez Jeremías
R-3887/3

12 de febrero de 2016

Índice

1. Introducción

Hasklab es un programa desarrollado en Haskell destinado a la manipulación de funciones reales de una variable real.

Su alcance abarca la definición de funciones (desde un archivo o en modo interactivo), ploteo, derivación, composición, aproximación de raíces, aproximación de integrales y cálculo de polinomios de aproximación (Taylor, interpolación).

Consiste en una interfaz que recibe comandos. Cada comando es parseado, interpretado, ejecutado y se exponen los resultados correspondientes.

Ofrece información, ayuda, ejemplos y mensajes de error detallados sobre cada comando para hacer más intuitiva la interacción con el usuario, y no requiere que éste tenga conocimiento alguno de programación.

Para dar una idea general del trabajo, la siguiente sección ilustrará cómo utilizar Hasklab mediante ejemplos.

Para describir la implementación y el trabajo en mayor profundidad, en las demás secciones se describirán los distintos módulos que componen el programa.

2. Usando Hasklab

2.1. Instalación y Ejecución

Se utiliza el módulo Graphics.EasyPlot, que simplifica funcionalidades de gnuplot.

```
sudo apt-get install gnuplot-x11
cabal install easyplot
```

En la carpeta src se encuentran los distintos módulos que conforman este trabajo. El módulo principal, main.hs, ya se encuentra compilado.

```
$ ./Main
Abriendo Prelude.mn...

MINI HASKLAB
Métodos Numéricos en Haskell.
Escriba :help para recibir ayuda.
HL>
```

Ahora ya podemos comenzar a usar HaskLab.

2.2. Ayuda

Hasklab acepta comandos de una línea. Los nombres de los comandos no utilizan letras mayúsculas. Para consultar la lista de comandos existentes y su sintaxis, podemos ingresar:

```
HL> :help
```

Si se quiere información detallada de un comando (sintaxis, descripción, ejemplos), se debe utilizar el comando :?.

```
HL> :? bisection
```

2.3. Definición

Para manipular una función, ésta debe ser previamente definida. Una primera opción es cargar un archivo de definiciones usando :load. En cada línea deberá ir una definición y se aceptan comentarios (--). Otra opción es definir funciones interactivamente.

La definición de funciones admite constantes reales, variables x; operadores $+$, $-$, $*$, $/$, $\hat{}$(**); funciones básicas cos, sin(sen), tan(tg), $cosh$, $sinh$, $tanh$, $acos$, $asin$, $atan$, ln(log) y las constantes pi y e. También se permite utilizar otra función previamente definida en la definición de una nueva función.

```
HL> def sinc(x) = sin x / x
HL> def fs(x) = sinc (cos x)
HL> def zeta(x) = (2*x*sin(2*x)+cos(2*x)-1)/(2*x)
HL> disp sinc

AST:
FCoc (FSin FVar) FVar
Se muestra como :
sin(x)/x
```

Observaciones:

Como vemos, el comando disp nos muestra la ley de una función y su AST en la representación interna.

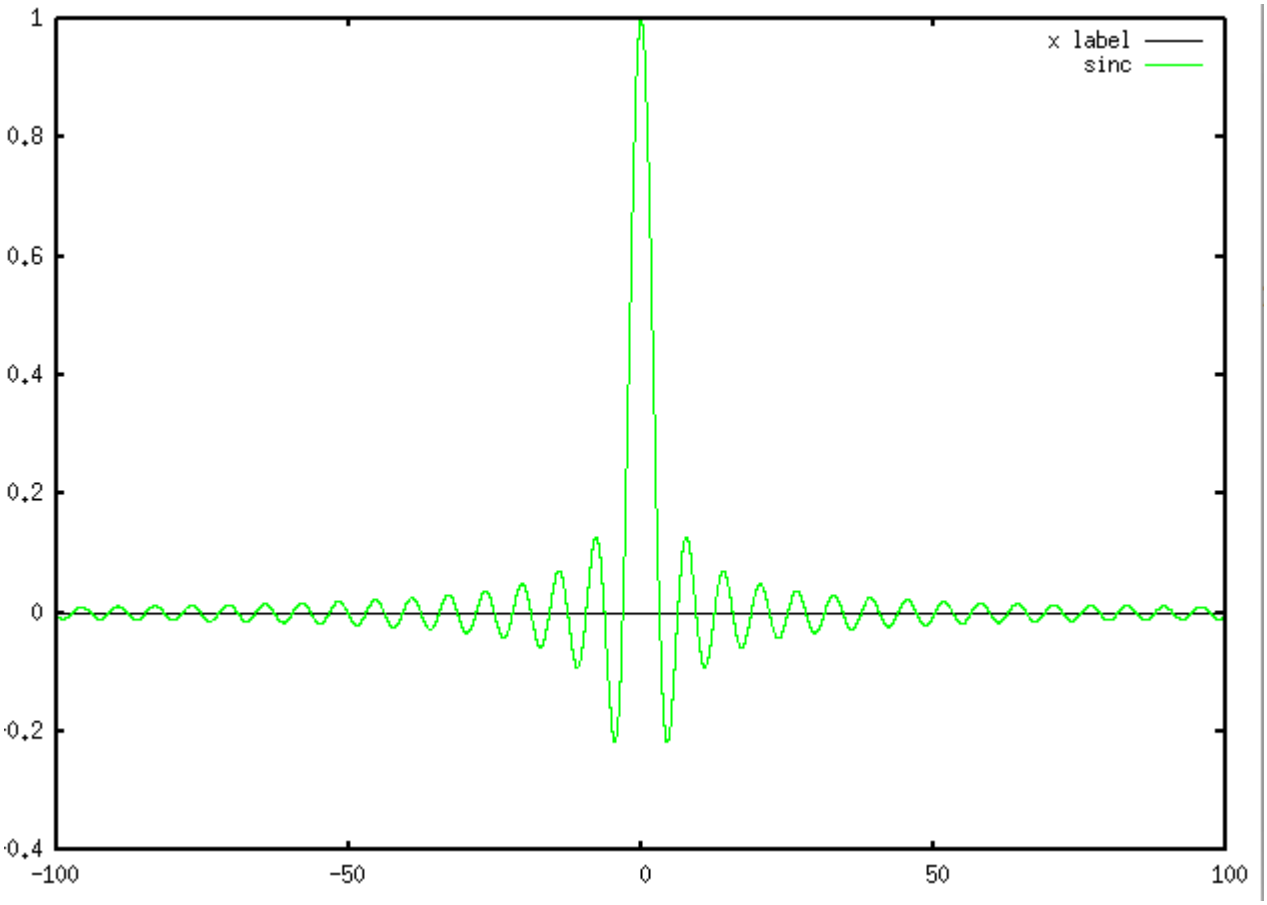
El comando :browse nos muestra los nombres de funciones definidas en scope.

Los nombres de funciones deben comenzar con una letra, solo pueden incluir los símbolos «_» y «'»; y no pueden ser palabras reservadas como *cos* o x.

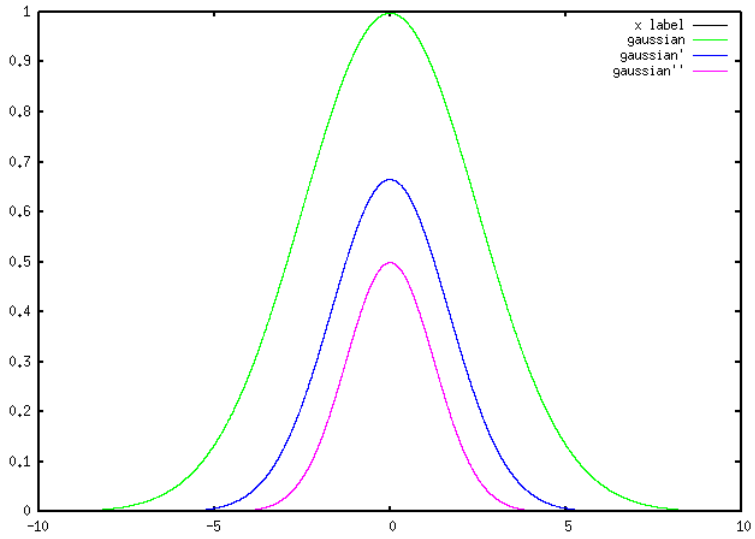
2.4. Ploteo

El ploteo utiliza la librería EasyPlot. Los plots son mostrados en una nueva ventana. Podemos plotear una sola función, o varias a la vez:

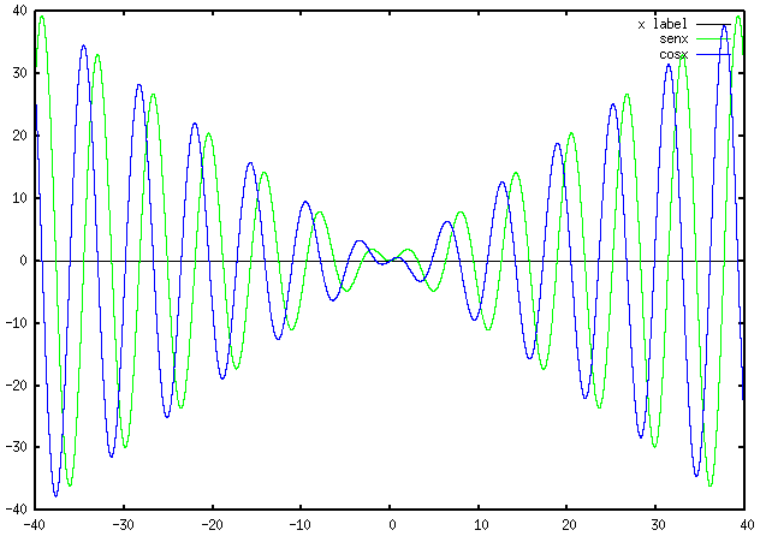
```
HL> plot sinc -100 100 0.01
```



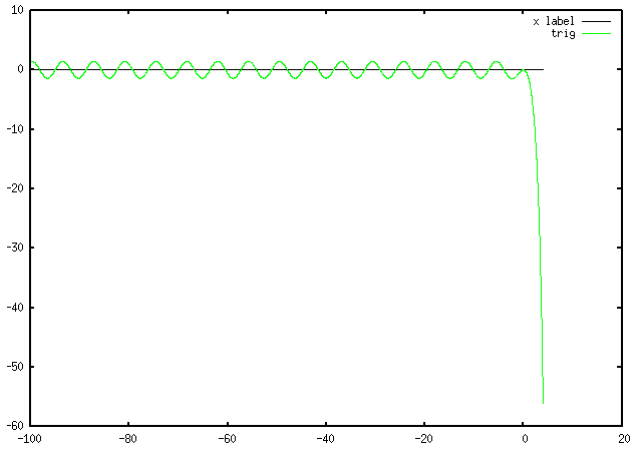

```
HL> def gaussian(x) = 1 / (2*pi*0.40**2)**0.5 * e ** (-x**2/2*0.40**2)
HL> def gaussian'(x) = 1 / (2*pi*0.60**2)**0.5 * e ** (-x**2/2*0.60**2)
HL> def gaussian''(x) = 1 / (2*pi*0.80**2)**0.5 * e ** (-x**2/2*0.80**2)
HL> plot [gaussian,gaussian',gaussian''] -10 10 0.001
```




```
HL> def senx(x) = x*sin x
HL> def cosx(x) = x*cos x
HL> plot [senx,cosx] -40 40 0.001
```



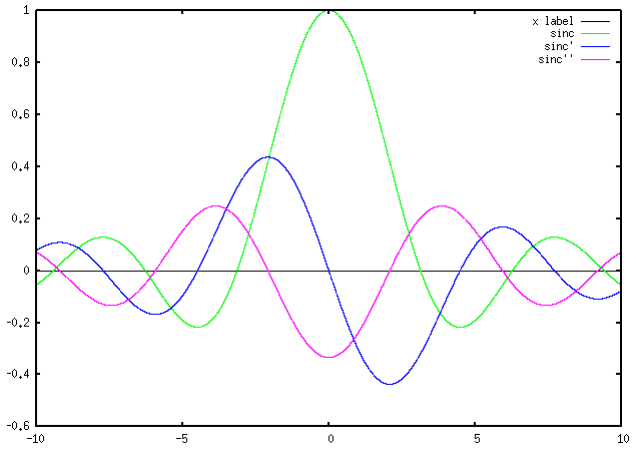
```
HL> def trig(x) = - cosh x - senh x + cos x + sen x
HL> plot trig -100 4 0.01
```



2.5. Derivación

Para hallar la función derivada, se utiliza el comando derivate. La función resultante (al igual que en todos los comandos que generan una nueva función) queda guardada en el indentificador temporal ans.

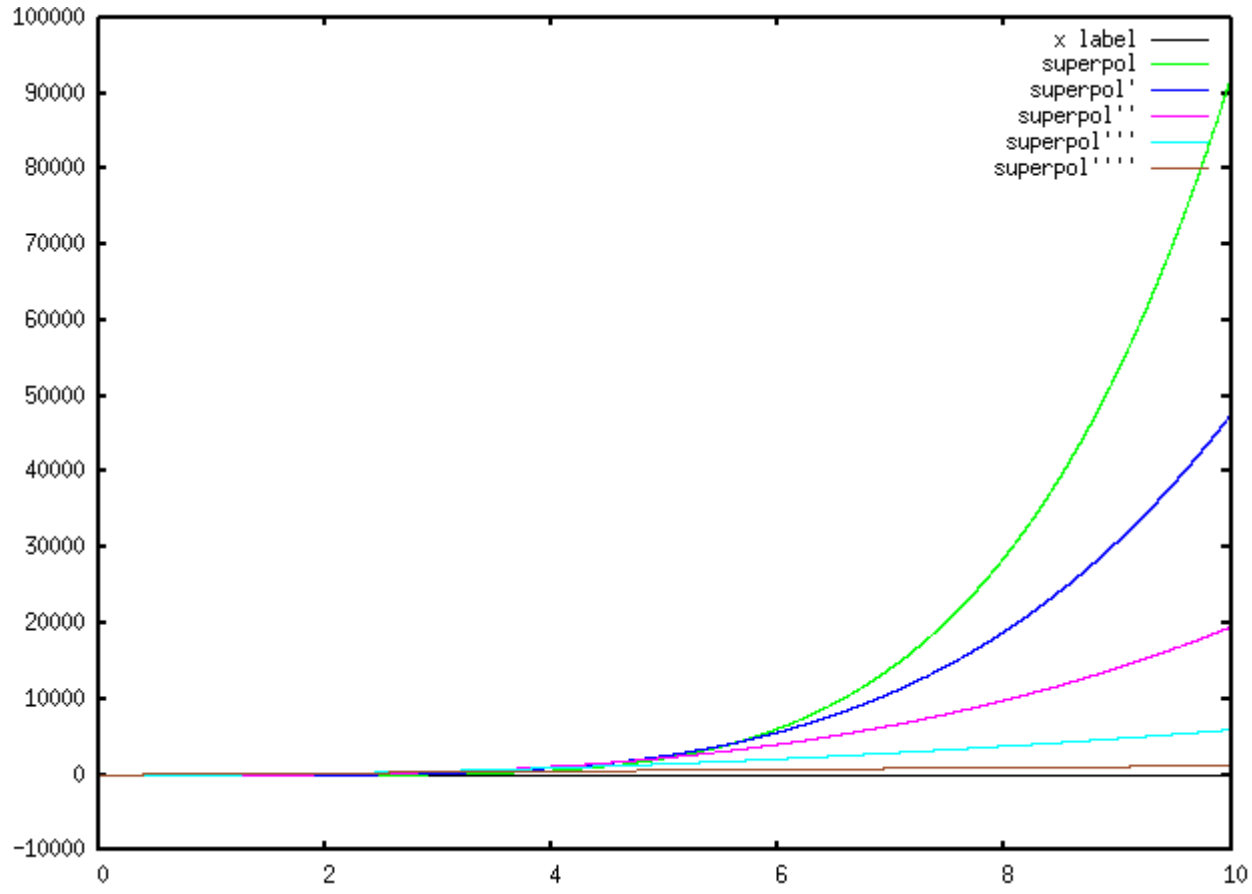
```
HL> derivate sinc
sinc(x)=FCoc (FSin FVar) FVar
sinc'(x)=FCoc (FDif (FMul (FMul (FConst 1.0) (FCos FVar)) FVar) (FMul (FConst 1.0) (F ...
HL> def sinc'(x)=ans(x)
HL> derivate sinc'
(...)
HL> def sinc''(x)=ans(x)
HL> plot [sinc,sinc',sinc''] -10 10 0.001
```



Observación:
La función derivate puede ser ineficiente si se aplica a una función muy grande o repetidas veces, pues genera expresiones muy grandes. Más adelante se volverá sobre el tema.

```
HL> def superpol(x) = x**5 - 8 * x **3 + 10 * x + 6
```

Si derivamos superpol cuatro veces como hicimos con sinc y luego ploteamos, obetenemos la siguiente gráfica:

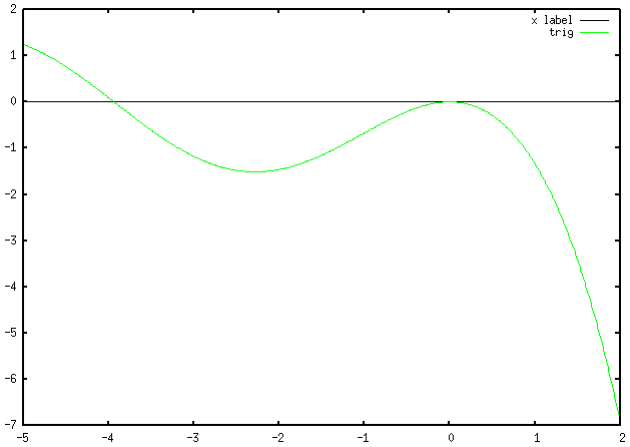


Si se intenta, se verificará el (inmenso) tamaño que alcanza la ley de la derivada cuarta; cuando en realidad es sólo una función lineal.

2.6. Aproximando Raíces

Ilustraré dos ejemplos usando dos métodos distintos, ver con :help la lista de métodos disponibles para aproximar raíces.

```
HL> def trig(x) = - cosh x - senh x + cos x + sen x
HL> plot trig -5 2 0.01
```



Como vemos, esta función tiene una raíz entre -5 y -2.5. En ese intervalo se cumplen las hipótesis del método de la bisección (contínua y signos opuestos), por lo que podemos hallar una buena aproximación de esa raíz:

```
HL> bisection trig -5 -2.5 0.000001

Resultado: -3.9407330751419067
Causa de terminación: Se alcanzó la tolerancia deseada en el eje x
Número de iteraciones realizadas: 21
```

El último argumento de bisection es la tolerancia deseada (máxima diferencia entre la raíz real y la aproximada).

Si deseamos aproximar la otra raíz que se ve en la gráfica, bisection no nos será útil pues no estan dadas las hipótesis. En cambio, podemos aplicar el método de Newton Raphson.

Se ofrecen dos variantes de este método: uno requiere la derivada de trig como argumento, otro la aproxima usando diferencias finitas. En nuestro caso, como no tenemos idea (o ganas de calcular) la derivada, usamos la segunda: *hnewton_raphson*.

Este método precisa de una aproximación incial, una tolerancia en el eje x, una tolerancia en el eje y, un coeficiente h para aproximar la derivada y un límite de iteraciones. (:? *hnewton_raphson* para detalles).

Veamos qué sucede para distintas tolerancias y puntos iniciales.

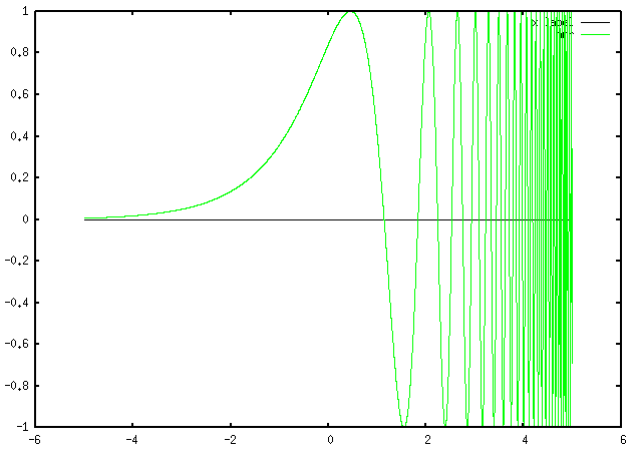
```
HL> hnewton_raphson trig -1 0.0001 0.0001 0.001 1000

Resultado: -8.919970458375023e-3
Causa de terminación: Se alcanzó la tolerancia deseada en el eje y
Número de iteraciones realizadas: 5

HL> hnewton_raphson trig -1 0.0001 0.000000001 0.001 1000

Resultado: 3.1657363254032454e-4
Causa de terminación: Se alcanzó la tolerancia deseada en el eje x
Número de iteraciones realizadas: 14

HL> hnewton_raphson trig 5 0.0001 0.000000001 0.001 1000
```

```
Resultado: 3.153093490117077e-4
Causa de terminación: Se alcanzó la tolerancia deseada en el eje x
Número de iteraciones realizadas: 18

HL> hnewton_raphson trig 100 0.0001 0.000000001 0.001 1000

Resultado: 3.188649866876466e-4
Causa de terminación: Se alcanzó la tolerancia deseada en el eje x
Número de iteraciones realizadas: 113
```

Tener en cuenta que los métodos como Newton Raphson y Secante pueden diverger, o arribar a raíces lejanas (no necesariamente la más cercana a la aproximación inicial brindada por el usuario).

2.7. Integración

```
HL> def  mmr(x) = sin(e**x)
HL> plot mmr -5 5 0.001
```



Deseamos estimar el area bajo esa curva en [-4,1].

Dos métodos muy precisos son trapecio compuesto y Simpson compuesto. Sólo debemos proveerles la función, el intervalo a integrar y el número de subintervalos. (:? para más detalles)

```
HL> trapecio_compuesto mmr -4 1 20
Resultado: 1.7896892347812685
HL> trapecio_compuesto mmr -4 1 100
Resultado: 1.802204775630908
HL> simpson_compuesto mmr -4 1 10000
Resultado: 1.8027739909241658
```

Los primeros decimales del valor real son 1.802724971600. Estos métodos funcionan aplicando otros métodos sencillos a subintervalos, que integran polinomios interpoladores.

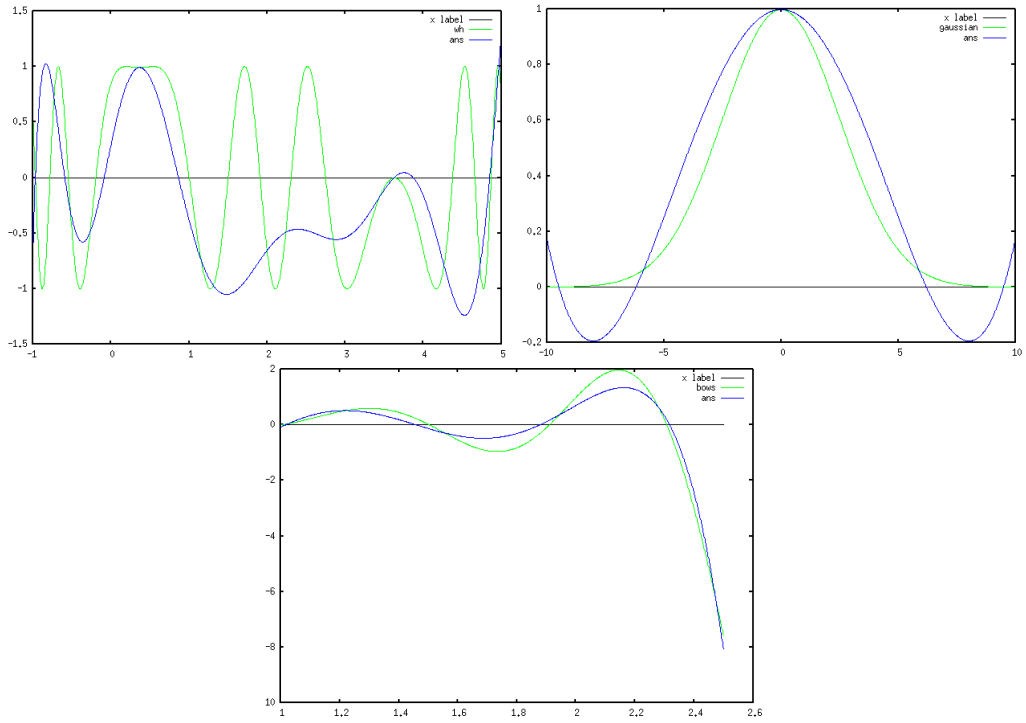
2.8. Aproximando Funciones

Dada una función, podemos hallar otras funciones que sirvan para aproximarla -por lo general, polinomios de aproximación-

Veamos algunos ejemplos, comparando las gráficas de las funciones y sus funciones de aproximación:

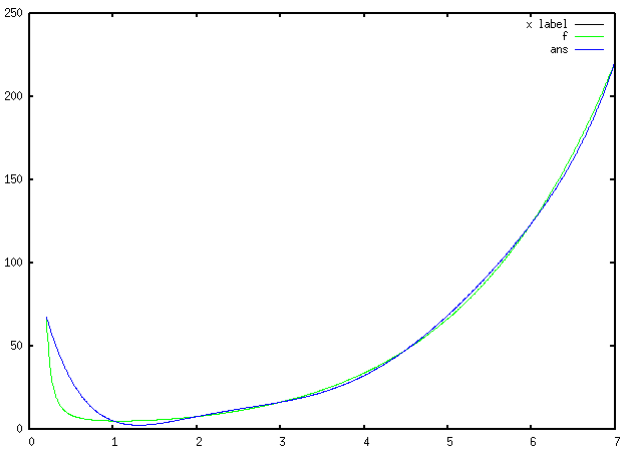
chebyshev f a b n genera un polinomio de grado n que aproxima a f en [a,b]. El polinomio interpola a f en n+1 nodos elegidos de forma óptima.

```
HL> def gaussian(x) = 1 / (2*pi*0.40**2)**0.5 * e ** (-x**2/2*0.40**2)
HL> chebyshev gaussian -10 10 4

HL> def wh(x) = cos ( x^3 - 6*x^2 + 4*x + 12)
HL> chebyshev wh -1 5 10

HL> def bows(x) =wh(x)/(x*sen x*ln (ln x))
HL> chebyshev bows 1 2.5 5
```



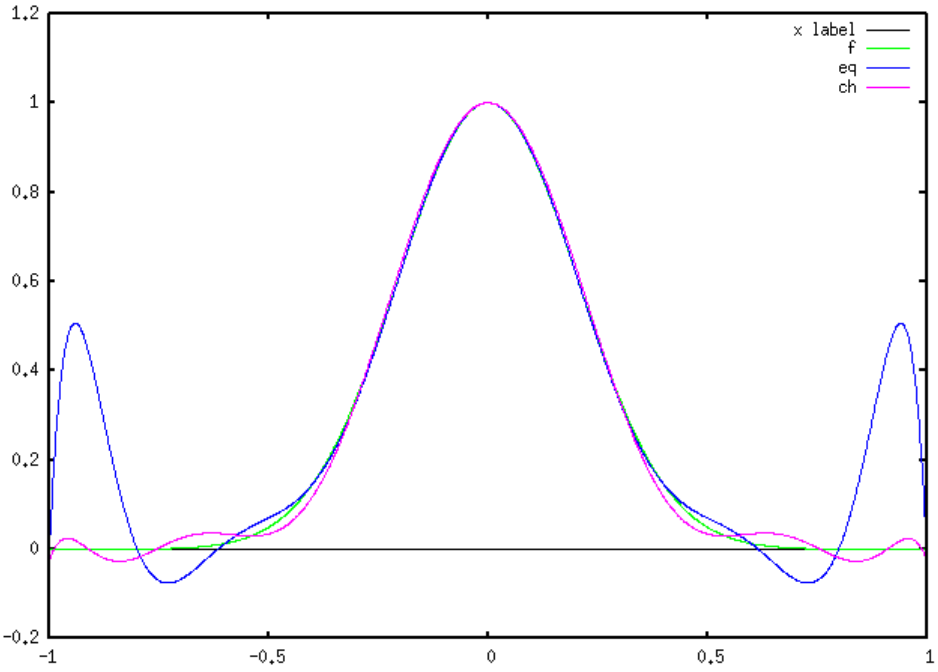
Podemos aproximar a f mediante un polinomio que interpole ciertos nodos elegidos manualmente. Dada la función y los nodos, el polinomio es único. Hay dos estrategias para hallarlo, la interpolación de Lagrange y la de Newton:

```
HL> def f(x) = sen(cos x) / cosh x + 5*x**log x
HL> interpolacion_newton f [0.2, 1, 2, 3 , 4.5, 6, 7]
```



Veamos un caso en que se comparan la interpolación usando nodos equiespaciados y nodos de Chebyshev, interpolando una función mediante un polinomio de grado 10.

```
HL> def f(x) = e ** (-12 * x**2)
HL> interpolacion_newton f [-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]
HL> def eq(x) = ans(x)
HL> chebyshev f -1 1 10
HL> def ch(x) = ans(x)
HL> plot [f,eq,ch] -1 1 0.001
```

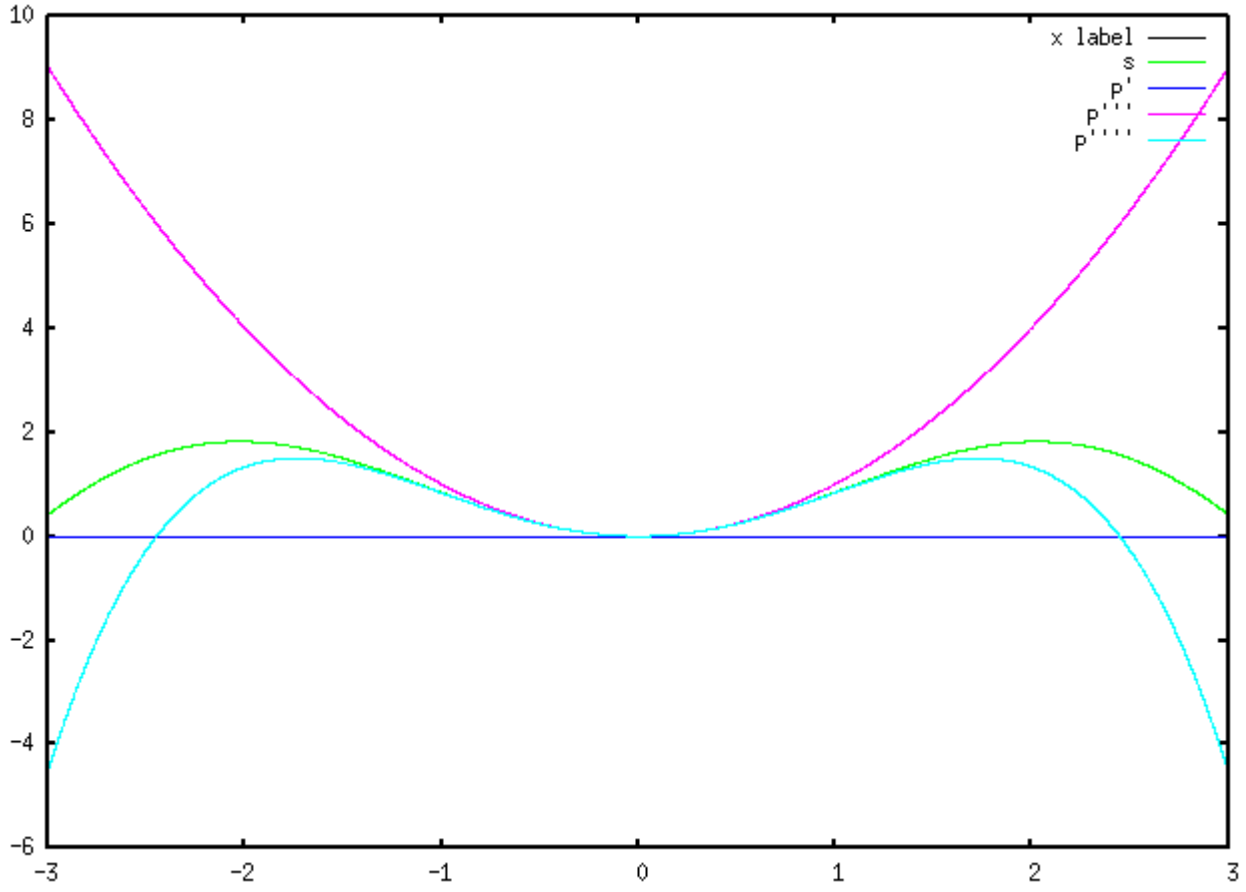


Como último ejemplo, el comando taylor f x h n es utilizado para general el polinomio de taylor de f aldedor de x de grado n. El real h es usado para estimar las n primeras derivadas. (Obviamente la función debe ser derivable n veces en un entorno de x)

Esta aproximación es útil cuando deseamos estimar valores de una función cerca de un punto.

```
HL> def s(x) = x*sen x
HL> taylor s 0 0.0001 5
```

En la siguiente imágen vemos la comparación de los polinomios de taylos de grado 1,3 y 4.



3. Common.hs

A partir de aquí, se analizarán los distintos módulos que conforman Hasklab. Primero se describen Common.hs y RealFC.hs, pues son utilizados por todos los demás módulos.

En Common.hs se reúnen las distintas definiciones de tipo que se usarán en los demás módulos.

3.1. Funciones Reales

Como el trabajo se centra en la manipulación de funciones reales, comenzaré exponiendo el tipo de datos que utilicé para representarlas. Lo elegí porque representa la estructura de la ley de la función.

```
data FReal = FConst Double
           | FVar
           | FUmin FReal
           | FSum FReal FReal
           | FDif FReal FReal
           | FMul FReal FReal
           | FCoc FReal FReal
           | FPot FReal FReal
           | FSin FReal
           | FCos FReal
           | FTan FReal
           | FSinh FReal
           | FCosh FReal
           | FTanh FReal
           | FAsin FReal
           | FAtan FReal
           | FAcos FReal
           | FLog FReal
           | FId String FReal    --- g(x**2)
      deriving (Eq,Show)
```

Este tipo de datos será usado, en primer lugar, por el parser de funciones. Toda función real ingresada por el usuario se representará internamente como una FReal. Por ejemplo:

$$f(x) = 2 * x ** 2 + \log (\cos x) + x$$

Será representada como:

(FSum (FSum (FMul (FConst 2.0) (FPot FVar (FConst 2.0))) (FLog (FCos FVar))) FVar)

Representar a las funciones de esta forma también me permitió calcular derivadas de acuerdo a su estructura, componer funciones y manejar errores en caso de evaluar una función en un punto donde no está definida.

3.2. Comandos

Un tipo de dato stmt es utilizado para representar comandos que involucran funciones, luego de ser parseados. Los comandos de ayuda, carga de archivos, etc no utilizan este tipo de datos (:help, :load, :reload)

```
data Stmt i = Def String i
            | Plot [String] Double Double Double
            | Disp String
            | Derivate String
            | HDerivate String Double Double
            | Compose String String
            | Bisection String Double Double Double
            | Eval String Double
            | RegulaFalsi String Double Double Double Double Int
            | NewtonRaphson String String Double Double Double Int
            | HNewtonRaphson String Double Double Double Double Int
            | Secante String String Double Double Double Double Int
            | HSecante String Double Double Double Double Double Int
            | Trapecio String Double Double
            | TrapecioC String Double Double Int
            | Simpson String Double Double
            | SimpsonC String Double Double Int
            | Taylor String Double Double Int
            | InterpolacionL String [Double]
            | InterpolacionN String [Double]
            | Chebyshev String Double Double Int
    deriving (Show)
```

Como vemos, los distintos comandos que serán reconocidos consistirán en definir, plotear, mostrar, derivar, aproximar raíces, integrales, y polinomios de aproximación; dada una función real y algunos argumentos adicionales.

Las constantes String hacen referencia al identificador de una función, ya que durante la ejecución de los comandos se irá llevando un entorno con todos los nombres de funciones definidas y su correspondiente ley.

3.3. Mónada para Métodos Numéricos

En tercer lugar, definí un tipo de datos utilizado como tipo de retorno para los distintos métodos numéricos. Teniendo en cuenta que necesitaba tener manejo de mensajes de error, contar cantidad de iteraciones, etc, utilicé el siguiente tipo de datos:

```
data Code = Empty | Maxit | Delta | Epsilon | Reached  deriving (Show,Eq)
data NumMonad e a = E e Int | R Int a Code  deriving Show
```

Al ejecutar un método numérico, se iterará hasta que alguna condición de terminación se cumpla. Estas pueden ser alcanzar un máximo de iteraciones predefinido, alcanzar una tolerancia deseada en el eje x, en el eje y, o alcanzar la raíz más exacta posible usando números en punto flotante. Estos motivos de terminación son representados en el tipo de datos Code (Empty representa que ninguna condición se cumple o que no es relevante en ese momento).

El tipo de datos NumMonad es similar al tipo de datos Either. Si ocurre algún error, el método retornará un valor (E e n), dónde e es un mensaje de error y n es el número de iteraciones hasta que se produjo el error. Si el método termina exitosamente, se retorna un valor (R n a c) donde n es el número de iteraciones realizadas, c el motivo de terminación y a el valor (en este caso real) aproximado.

```
instance Monad (NumMonad e) where
  return x           = R 0 x Empty
```

```
    (E e ni) >>= f = E e ni
    (R ni x code)   >>= f = case (f x) of
                (E e ni)        -> E e ni
                (R ni' x' code') -> R (ni + ni') x' (if code==Empty then code' else code)



  raise :: String -> NumMonad String a
  raise msg = E msg 0

  tick :: NumMonad String ()
  tick = R 1 () Empty

  set_code :: Code -> NumMonad String ()
  set_code c = R 0 () c

  remove :: NumMonad e a -> a
  remove (R _ x _ ) = x
```

La instancia mónada permite utilizar la notación do para emprolijar el código y hace más sencilla la adaptación de los métodos imperativos a haskell.

4. RealFC.hs

Este módulo implementa distintas funciones generales sobre el tipo FReal.

4.1. eval

La función eval es utilizada permanentemente ya que permite evaluar una FReal en un punto determinado. Un fragmento de su definición es el siguiente:

```
eval :: FReal -> Double -> NumMonad String Double
eval (FConst c)  _  = return c
eval FVar   x       = return x
eval (FSum f g ) x  = do fv <- eval f x
                         gv <- eval g x
                         return (fv + gv)
eval (FCoc f g ) x  = do fv <- eval f x
                         gv <- eval g x
                         if gv == 0 then raise "Divisón por cero."
                                    else return (fv / gv)
```

Se utiliza la mónada NumMonad pues eval será luego utilizada dentro de las funciones que implementan métodos numéricos. Si se quiere hacer alguna operación ilegal, se devuelve un error indicando el motivo, por ejemplo división por cero, arcsin de un valor mayor que 1, etc.

4.2. composition

La composición de funciones es muy sencilla de implementar, sustituyendo las variables de la segunda función por la definición de la primera.

4.3. derivate

La función derivate recibe como argumento una FReal y, utilizando las reglas de derivación, devuelve otra FReal que es su función derivada.

Utiliza distintas reglas como la derivada del producto y derivación logarítima para potencias, que aumentan considerablemente el tamaño de la expresión resultante, por lo que el resultado será probablemente más ineficiente cuanto mayor sea la función a derivar.

Para mejorar un poco esta situación, definí una función *simplificar_ceros* que simplifica un poco la expresión de la función resultante, eliminando sumas de constantes cero, o términos multiplicados por cero.

Dado que los métodos numéricos no precisan de esta función, pues la derivada es aproximada más eficientemente usando diferencias finitas, decidí no optimizar demasiado esta función.

La función resultante queda guardada en un identificador ans (que podrá ser reescrito si no se guarda con otro nombre)

4.4. hderivate

```
hderivate :: FReal -> Double -> Double -> NumMonad String Double
hderivate f x h =  do fx  <- eval f x
                      fxh <- eval f (x+h)
                      return ((fxh - fx) / h)
```

Esta función aproxima el valor de la derivada de una función f en un punto determinado mediante diferencias finitas, utilizando una constante h para realizar la aproximación.

5. Rootfinding.hs

Este módulo implementa métodos numéricos para aproximación de raíces. Utiliza los dos módulos ya vistos. A continuación describiré brevemente cada método, el significado de sus argumentos y en algunos casos detalles de su implementación.

El objetivo de esta sección no es dar fundamentos teóricos sobre los métodos, la referencia [1] explica cláramente las bases matemáticas y los pseudocódigos imperativos en MatLab. Sólo pretendo introducir las distintas funciones, sus argumentos, y algunas explicaciones puntuales sobre mi implementación en Haskell.

5.1. Bisection

El método de la bisección es ampliamente conocido, dada una función contínua f y un intervalo [a,b] tal que $f(a).f(b) \leqslant 0$, retorna una aproximación a una raíz de la función con un error menor a un delta dado.

Su definición es:

```
bisection :: FReal -> Double -> Double -> Double -> NumMonad String Double

bisection f a b delta = do fa <- eval f a
                           fb <- eval f b
                           let nits = 1 + round ((log (b-a) - (log delta) )/ (log 2))
                           if fa==0 then
                               do { (set_code Reached) ; return a }
                           else if fb==0 then
                               do { (set_code Reached) ; return b }
```

```
                                else if fa*fb > 0 then
                                    raise "f(a)*f(b) must be negative"
                                else
                                    bisection' f a b nits


bisection' f a b it_rest = do let c = (b + a)/2
                              if it_rest==1 then
                                 do { (set_code Delta) ; return c }
                              else
                                 do fa <- eval f a
                                    fb <- eval f b
                                    fc <- eval f c
                                    if fc == 0 then
                                        do { (set_code Reached) ; return c }
                                    else if fa * fc < 0  then
                                        do { tick ; bisection' f a c (it_rest - 1)}
                                    else
                                        do { tick ; bisection' f c b (it_rest - 1)}
```

En primer lugar, se chequea que los argumentos cumplan las hipótesis y que ninguno de los extremos sea ya una raíz. En caso de haber un error, se reporta usando la función *raise*. En caso de haber terminado exitosamente, se indica el criterio de terminación satisfecho usando *set_code*.

En caso de ser los argumentos legales, *bisection′* se encarga de repetir el proceso dicotómico hasta alcanzar la tolerancia deseada. En este caso, tenemos la ventaja de saber de antemano cuántas iteraciones necesitaremos, y usamos esto para hacer mas sencilla la implementación. En cada iteración exitosa, llamamos a la función *tick* para aumentar el contador de iteraciones realizadas.

5.2. Regula Falsi

El método regula falsi es muy similar al de la bisección, pero en cada iteración, en vez de elegir el punto medio del intervalo, elige un punto más conveniente. Como desventaja al implementarla, no podemos saber exáctamente cuántas iteraciones haremos.

Por este motivo, debemos ir chequeando en cada iteración si se ha alcanzado alguna condición de terminación. El código puede verse en RootFinding.hs.

Su signatura es:

```
regula_falsi :: FReal -> Double -> Double -> Double -> Double -> Int -> NumMonad String Double
```

Donde los argumentos corresponden a $regula\_falsi$ f a b delta epsilon maxit. [a,b] es el intervalo dónde buscamos la raíz. Si r es la raíz real y x_n es una nueva aproximación hallada en una dada iteración, entonces:

- Si $|x_n - r| < delta$ la función retornará x_n como aproximación final.
- Si $|f(x_n)| < epsilon$ la función retornará x_n como aproximación final.
- Si ya se han hecho maxit iteraciones, la función retornará x_n como aproximación final.

5.3. Newton Raphson

El método de Newton-Raphson también nos permite aproximar raíces, pero a diferencia de los dos anteriores, no precisamos de un intervalo dado, sino de una aproximación inicial de la raíz a buscar.

Dependiendo de la aproximación brindada (por el usuario), el método puede converger o diverger.

Además, este método precisa conocer el valor de la función derivada en ciertos puntos. Por este motivo, implementé dos versiones: una recibe como argumento la función a analizar y su derivada (*newton_raphson*), y la otra no requiere la derivada como argumento, pues la aproxima mediante diferencias finitas usando una constante h brindada por el usuario (*hnewton_raphson*).

la implementación de la primera es:

```
newton_raphson :: FReal -> FReal -> Double -> Double -> Double -> Int -> NumMonad String Double
newton_raphson f f' p0 delta epsilon maxit = do f0  <- eval f  p0
                                                f0' <- eval f' p0
                                                let p1 = p0 - f0 / f0'
                                                f1  <- eval f p1
                                                if f1==0 then
                                                  do { (set_code Reached) ; return p1 }
                                                else if maxit==0 then
                                                  do { (set_code Maxit) ; return p1 }
                                                else if (abs f1) < epsilon then
                                                  do { (set_code Epsilon) ; return p1 }
                                                else if  abs (p1-p0) < delta then
                                                  do { (set_code Delta) ; return p1 }
                                                else
                                                  do tick
                                                     newton_raphson f f' p1 delta epsilon (maxit-1)
```

La interpretación de los parámetros delta, epsilon y maxit es análoga a regula falsi.

La implementación de *hnewton_raphson* es análoga a la de *newton_raphson*, se puede ver en RootFinding.hs:

```
hnewton_raphson :: FReal -> Double -> Double -> Double -> Double -> Int -> NumMonad String Double
hnewton_raphson f p0 delta epsilon h maxit = .........
```

5.4. Secante

El método de la secante es similar al de Newton-Raphson pues también requiere la derivada de la función a analizar y de aproximaciones iniciales, dos en vez de una.

Por el mismo motivo que mencioné en Newton-Raphson, implementé dos versiones: secante (si el usuario conoce la derivada exacta) y hsecante (si el usuario desconoce la derivada o desea aproximarla).

Las signaturas y argumentos son:

```
secante :: FReal -> FReal -> Double -> Double -> Double -> Double -> Int -> NumMonad String Double
secante f f' p0 p1 delta epsilon maxit = ........

hsecante :: FReal -> Double -> Double -> Double -> Double -> Double -> Int -> NumMonad String Double
hsecante f h p0 p1 delta epsilon maxit = ........
```

Ejecutando en Hasklab el comando :? <comm>, se mostrará información de cualquiera de estos métodos, ejemplos de como invocarlos, etc.

6. Integration.hs

Este módulo implementa cuatro funciones para aproximación de integrales definidas de funciones reales. Al igual que en la sección anterior, los fundamentos teóricos pueden consultarse en [1].

6.1. Trapecio, Trapecio Compuesto

Dada una función f y un intervalo [a,b], el método del trapecio calcula la integral definida de la función lineal que pasa por (a,f(a)) y (b,f(b)). Es una aproximación muy inexacta.

```
trapecio :: FReal -> Double -> Double -> NumMonad String Double
trapecio f a b = do fa <- eval f a
                    fb <- eval f b
                    let h = b-a
                    return ((fb+fa)*h/2)
```

Una forma de optimizar esta aproximación es subdividir [a,b] en n subintervalos del mismo tamaño y aplicar el método del trapecio en cada uno de ellos. Esto es lo que hace el método del trapecio compuesto:

```
trapecio_compuesto :: FReal -> Double -> Double -> Int -> NumMonad String Double
trapecio_compuesto f a b n = let points = subdivide a b n in trapecio_compuesto' f points


trapecio_compuesto' :: FReal -> [Double] -> NumMonad String Double
trapecio_compuesto' f ps | length ps < 2  = return 0
trapecio_compuesto' f (pn:pn':ps)          = do tn    <- trapecio f pn pn'
                                                trest <- trapecio_compuesto' f (pn':ps)
                                                return (tn + trest)

subdivide :: Double -> Double -> Int -> [Double]
subdivide a b n = [ a + h * (fromIntegral k) | k <- [0..n] ]
    where h = (b-a)/(fromIntegral n)
```

En este caso, no utilizamos código de retorno ni contador de iteraciones pues no es relevante (el método siempre terminará excepto que se evalúe la función en un punto en que no está definida).

6.2. Simpson y Simpson Compuesto

El método de simpson es análogo al del trapecio, pero aproxima la integral de f en [a,b] por la integral de la función cuadrática que interpola a (a,f(a)), (b,f(b)) , (c,f(c)) , c=$a + \frac{b-a}{2}$.

El método de simpson compuesto consiste en subdividir nuevamente [a,b] en n subintervalos y aplicar simpson en cada uno de ellos.

Las signaturas son:

```
simpson :: FReal -> Double -> Double -> NumMonad String Double
simpson f a b = ....

simpson_compuesto :: FReal -> Double -> Double -> Int -> NumMonad String Double
simpson_compuesto f a b n = .....
```

7. Approximation.hs

Este módulo consiste en cuatro funciones. Tres de ellas permiten, dada una función f y ciertos puntos, hallar un polinomio interpolador que puede ser utilizado para aproximar a f. El cuarto (Taylor) nos permite hallar un polinomio que permitirá aproximar una función alrededor de un punto elegido.

7.1. Taylor

Dada una función f, un punto p, un natural n y un valor h para aproximar sus derivadas; se puede aproximar a f en un entorno de p mediante su polinomio de taylor de grado n alrededor de p.

```
taylor :: FReal -> Double -> Double -> Int -> FReal
taylor f a h n = ...
```

El polinomio resultante queda guardado asociado a un identificador temporal ans. Si se quiere guardar la ley del polinomio, habrá que redefinir la función: def tf(x)=ans(x).

7.2. Interpolación de Newton/Lagrange

Dada una función f y un conjunto de puntos $\{ x_i \mid 0 \leqslant i \leqslant n\}$ de su dominio, se puede aproximar a f mediante el polinomio de grado menor o igual a n que interpola los puntos $\{(x_i, f(x_i))\}$

Hay dos formas de hallar este polinomio, que, por el teorema fundamental del álgebra es único: el método de Lagrange y el método de Newton, que son implementados con las siguientes funciones:

```
interpolacion_lagrange :: FReal -> [Double] -> FReal
interpolacion_newton :: FReal -> [Double] -> FReal
```

Al igual que en taylor, el polinomio resultante queda resguardado en el identificador ans.

7.3. Chebyshev

En este caso, nuevamente se desea aproximar una funcion f. Pero en vez de proporcionar los puntos (nodos de interpolación), se brinda un intervalo y la cantidad de puntos deseados. Los nodos son hallados de tal forma que la suma del error cuadrado sea mínimo a medida que aumenta el grado del polinomio, usando polinomios de Chebyshev. (La teoría puede leerse en [5]).

```
chebyshev :: FReal -> Double -> Double -> Int -> FReal
chebyshev f a b n = ...
```

El polinomio resultante se guarda en ans.

8. Plotting.hs

Este módulo importa la librería EasyPlot [2] (que a su vez utiliza GnuPlot). EasyPlot brinda una forma muy práctica y sencilla de graficar funciones y puntos, suficientemente precisa para los alcances de este trabajo.

Básicamente la librería permite graficar una lista de puntos (x,y) (permitiendo elegir detalles como colores, etc). Usando esta herramienta, implemente 4 opciones a la hora de plotear. Mi trabajo consistió en crear la lista de pares (x,f x) adecuada para que en el gráfico se pueda apreciar la forma de la gráfica de la función. Luego, invocar a las funciones provistas por EasyPlot con la lista de puntos creada.

Implementé cuatro opciones a la hora de plotear:

- `sfPlot :: (FReal,String) -> Double -> Double -> Double -> IO Bool`

  sfPlot (fLey,fNombre) a b step, crea un gráfico de f en el intervalo [a,b] usando puntos equiespaciados a distancia step. El nombre de la función se utiliza para colocar una etiqueta que indique a que función corresponde la gráfica.

- `mfPlot :: [(FReal,String)] -> Double -> Double -> Double -> IO Bool`

  mfPlot es una modificiación de la anterior para plotear muchas funciones a la vez en un mismo plano.

- `xPlot :: [(FReal,String)] -> [(Double,Double)] -> Double -> Double -> Double -> IO Bool`

  xPlot agrega la posibilidad de graficar una serie de puntos, en forma de cruces. Se utiliza cuando se hallan raíces, para ilustrar la función y el punto hallado como raíz.

- `autoPlot :: [(FReal,String)] -> IO Bool`

  autoPlot sólo recibe como argumento una lista de funciones y sus identificadores. Automáticamente determina en que dominio hacer el gráfico, y con qué step. Este trabajo es hecho por la librería, en ciertos casos el gráfico es erróneo, en especial cuando la función tiene asíntotas horizontales o no es contínua.

9. Main.hs

El módulo principal comenzó como una adaptación del intérprete de lambda cálculo que desarrollamos en el trabajo práctico 3 de ALP [4]. Al ir agregando funcionalidades o encontrar incompatibilidades, todas las funciones fueron modificadas.

Básicamente, al ejecutarse, se permite al usuario ingresar comandos por teclado. Se ingresa un comando, se ejecutan las acciones correspondientes, y se vuelve a esperar otro comando hasta que ocurra un error o el usuario decida terminar (:quit).

En todo momento se lleva un entorno con los nombres de funciones definidas y las leyes correspondientes, que puede consultarse con (:browse).

Pueden cargarse archivos de definiciones, y automáticamente se cargará el archivo prelude.mn. (:load, :reload)

Un esquema del funcionamiento de main es el siguiente:

1. Se cargan los archivos con definiciones provistos como argumento y prelude.mn . ($compileFiles$)
2. Se lee una línea de la entrada. ($readline, getline$)
3. Se determina el tipo de comando del que se trata ($interpretCommand$)
4. Se toman las acciones adecuadas acorde al comando. Esto implica el uso de un parser generado por happy, para parsear comandos referidos a funciones y sus argumentos; llamar a las funciones de los otros módulos que sea necesario, por ejemplo para aplicar un método numérico; y exponer al usuario el resultado deseado. En caso de ser un comando inexistente, se le notifica un error de parseo. ($handleCommand$)
5. Repetir (1).

En este módulo también se encuentran muchas funciones necesarias por $handleCommand$, por ejemplo, funciones que proporcionan textos de ayuda si el comando fue :?.

Notar que en este módulo se utilizan funciones importadas desde Parse.hs, generado a partir de Happy desde Parse.y. En concreto se utiliza para parsear leyes de funciones, comandos, y secuencias de definiciones en archivos. Para esto modifiqué el parser usado en el trabajo práctico y tuve que aprender un poco más sobre el funcionamiento de happy en [3].

10. Limitaciones

Algunas limitaciones, posibles extensiones y autocríticas son:

- No es posible definir funciones a trozos. Si bien la gramática de funciones es bastante expresiva, no incluí pisos/techos, factorial, más funciones trigonométricas, exp, logaritmos de cualquier base, etc.
- El parser de funciones no permite hacer cosas como def f = ans (en vez de f(x)=ans(x)); también sería cómodo poder hacer plot cos 0 10 0.001 (para todas las funciones conocidas, sin necesidad de que hayan sido definidas). En general, flexibilizar un poco más los comandos (lo cual complicaría bastante el parser) sería mucho más cómodo para el usuario.
- La única variable utilizable es x
- Las funciones que pueden ser simplificadas (por ejemplo con términos x/x, términos opuestos, constantes operando entre si) no son optimizadas. Se parsea el input pero no se busca optimizar la expresión de la función.
- Los plots pueden fallar si hay asíntotas verticales, si hay valores muy grandes, si por algún motivo una constante NaN o Infinity no fue detectada, o símplemente si la función es discontínua en algun valor. (Ejemplo: plotear tangente)
- Hay muchísimos métodos numéricos más eficientes y más complejos de implementar.
- Los métodos que buscan raíces son recursivos. Es decir que, si se itera demasiado, puede haber un stack overflow. Por ejemplo, en mi computadora, pedir más de 70000 iteraciones en el método de Newton-Raphson ocasiona un stack overflow. No me preocupé demasiado por esto porque para llegar a estos extremos, se necesitaban pedir tolerancias que exceden la precisión de los números en punto flotante.
- Hay muchas operaciones entre valores muy grandes/pequeños (en especial potencias) que arrojan como resultado Infinity (desconozco si por cuestiones de Haskell, por hardware, o por algún error mío), lo cual limita la manipulación de algunas funciones que tienen valores extremos (por ejemplo asíntotas verticales).
- Si bien realicé muchos test con todo tipo de funciones que se me ocurrieron (ver prelude.mn donde dejé varias de ellas), es muy probable que se me hayan pasado funciones o argumentos erróneos que ocasionen malos comportamiento.

11. Aprendizajes

Habiendo expuesto el esquema general del trabajo final, enumero una lista de aprendizajes que obtuve luego de terminarlo. En general este trabajo me sirvió para afianzar muchos conceptos que desarrollamos en EDII y ALP:

1. Aprendí en bastante detalle a usar el generador de parsers Happy, y en general entendí cómo es el proceso que se lleva a cabo.
2. Aprendí a utilizar más los errores que expone Haskell, lo cual ayuda a solucionarlos más rápido, ya sea googleando o por haber lidiado con ese error antes.
3. Me familiaricé con buscar librerías, instalarlas, solucionar los (muchos) problemas de dependencias que surgieron al querer instalar una libería adecuada para plotear funciones.

4. Reforcé la idea de que las mónadas sirven para hacer más claro el código, ya que en un principio empecé a trabajar sin mónadas y en especial el manejo de errores era muy tedioso. En un par de ocasiones cambié detalles que, de no haber usado mónadas, hubieran implicado reescribir gran parte del código.

5. El principal desafío fue entender y modificar el intérprete de lambda cálculo, porque se involucran muchas funciones y módulos que desconocía. Pude apreciar muchas utilidades al usar mónadas, en especial al manejar entrada/salida, y luego de entenderlas pude usarlas para adaptar Main a mis necesidades.

Referencias

[1] John N. Mathews *Métodos Numéricos con MATLAB.*

[2] `https://hackage.haskell.org/package/easyplot-1.0/docs/Graphics-EasyPlot.html`, *EasyPlot*

[3] `https://www.haskell.org/happy/doc/html/`, *Happy User Guide.*

[4] `http://www.fceia.unr.edu.ar/lcc/r313/archivos/2015.ALP.TP3.zip` *Trabajo Práctico 3, Análisis de Lenguajes de Programación, FCEIA*

[5] Stefano Nasini, *Minimizar el error de interpolación considerando las raíces del polinomio de Chebyshev* , `http://www-eio.upc.es/~nasini/Blog/ChebyshevPolinomio.pdf`