Taller 3: Reconocimiento de patrones

Juan Francisco Díaz Frias Emily Núñez

Septiembre 2024

1. Ejercicios de programación

El objetivo de este ejercicio es implementar el método de Newton para hallar raíces de una función cualquiera (ver, por ejemplo, (este enlace)). Una función estará representada por una expresión construida a partir de un *trait* denominado *Expr* definido así:

```
trait Expr
case class Numero (d:Double) extends Expr
case class Atomo (x:Char) extends Expr
case class Suma(e1:Expr, e2:Expr) extends Expr
case class Prod(e1:Expr, e2:Expr) extends Expr
case class Resta(e1:Expr, e2:Expr) extends Expr
case class Div(e1:Expr, e2:Expr) extends Expr
case class Expo(e1:Expr, e2:Expr) extends Expr
case class Logaritmo(e1:Expr) extends Expr
```

Se pueden crear expresiones que representen un flotante, una variable, una operación binaria (la suma, resta, multiplicación, división y exponenciación) o una operación unaria (para el logaritmo natural).

Así por ejemplo la función $f(x) = 5k + \ln(3)/(8-x)^x$ podría ser representada en Scala así:

```
Suma(Prod(Numero(5.0), Atomo('k')),
          Div(Logaritmo(Numero(3.0)),
                Expo(Resta(Numero(8.0), Atomo('x')), Atomo('x'))))
```

Ahora, si tenemos una función $f(x)$ y queremos hallar sus ceros (es decir $\{x : f(x) = 0\}$), podemos aplicar el método de Newton (iterativo):

- Sea x_0 una aproximación inicial.
- 1. Si $f(x_i)$ está suficientemente cerca de 0, x_i es una solución de $f(x) = 0$.
  2. Sino, sea $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. Vuelva al item anterior.

En este taller usted implementará el algoritmo de Newton para hallar la raíz de una función representada por una expresión de tipo *Expr*. **El objetivo es adquirir destreza para escribir código funcional usando la técnica de reconocimiento de patrones.**

1.1. Mostrando expresiones

Implemente una función `mostrar` que dada una expresión e devuelve una cadena que representa la definición simbólica de la función, sin ambigüedades (coloque paréntesis donde sea necesario; no suponga prioridades de las operaciones).

```
def mostrar(e:Expr):String = {...}
```

Por ejemplo, para las siguiente expresiones:

```
val expr1=Suma(Atomo('x'), Numero(2))
val expr2=Prod(Atomo('x'), Atomo('x'))
val expr3= Suma(expr1, Expo(expr2,Numero(5)))
val expr4= Logaritmo(Atomo('x'))
val expr5=Prod(Div(expr1, expr2), Resta(expr3,expr4))
val expr6=Expo(Atomo('x'),Numero(3))
```

los valores resultantes de invocar *mostrar* son, respectivamente:

```
val res1: String = (x + 2.0)
val res2: String = (x * x )
val res3: String = ((x + 2.0) + ((x * x ) ^ 5.0))
val res4: String = (lg(x ))
val res5: String = (((x + 2.0) / (x * x )) * (((x + 2.0) + ((x * x ) ^ 5.0)) - (lg(x ))))
val res6: String = (x ^ 3.0)
```

1.2. Derivando expresiones

Implemente una función `derivar` que dada una función representada por la expresión f y un átomo a (que representa la variable de la función), retorne la expresión correspondiente a su derivada [1] con respecto a esa variable; todos los demás átomos de f deben ser considerados como constantes.

```
def derivar(f:Expr, a:Atomo):Expr = {...}
```

Por ejemplo, a las siguientes invocaciones:

```
mostrar(derivar(expr6, Atomo('x')))
mostrar(derivar(expr2, Atomo('x')))
mostrar(derivar(expr2, Atomo('y')))
mostrar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x')))
```

la función responde así:

```
val res6: String = ((x ^ 3.0) * (((1.0 * 3.0) / x ) + (0.0 * (lg(x )))))
val res7: String = ((1.0 * x ) + (x * 1.0))
val res8: String = ((0.0 * x ) + (x * 0.0))
val res9: String = (0.0 + ((0.0 * x ) + (3.0 * 1.0)))
```

[1] Tenga en cuenta las reglas del Cuadro 1.

$$
\begin{aligned}
c &\overset{'}{\rightarrow} 0 \qquad (c \text{ es constante}) && (1)\\
x &\overset{'}{\rightarrow} 1 && (2)\\
f+g &\overset{'}{\rightarrow} f'+g' && (3)\\
f-g &\overset{'}{\rightarrow} f'-g' && (4)\\
f\cdot g &\overset{'}{\rightarrow} f'\cdot g + f\cdot g' && (5)\\
\frac{f}{g} &\overset{'}{\rightarrow} \frac{f'\cdot g - f\cdot g'}{g^2} && (6)\\
\ln(f) &\overset{'}{\rightarrow} \frac{f'}{f} && (7)\\
f^g &\overset{'}{\rightarrow} f^g * \left(\frac{f'\cdot g}{f} + g'\cdot \ln f\right) && (8)
\end{aligned}
$$

Cuadro 1: Reglas de derivación usadas. Tenga en cuenta que f y g son funciones de la variable de derivación. También tenga en cuenta que $f'(x) = 0$ si f es una constante.

1.3. Evaluando expresiones

Implemente una función `evaluar` que dada una función representada por la expresión f y un átomo a (que representa la variable de la función), y un valor v flotante, calcule la evaluación de la función en v. Nota: Suponga que la fórmula de entrada solamente tendrá un tipo átomo, es decir que no dependerá de varias variables, además que siempre podrá ser evaluada.

```
def evaluar(f:Expr, a:Atomo, v:Double):Double = {...}
```

Por ejemplo, a las siguientes invocaciones:

```
mostrar(Numero(5.0))
evaluar(Numero(5.0),Atomo('x'), 1.0)
mostrar(Atomo('x'))
evaluar(Atomo('x'),Atomo('x'), 5.0)
mostrar(Suma(expr1,expr2))
evaluar(Suma(expr1,expr2),Atomo('x'), 5.0)
mostrar(Prod(expr1,expr2))
evaluar(Prod(expr1,expr2),Atomo('x'), 5.0)
mostrar(Resta(expr1,expr2))
evaluar(Resta(expr1,expr2),Atomo('x'), 5.0)
mostrar(Div(expr1,expr2))
evaluar(Div(expr1,expr2),Atomo('x'), 5.0)
mostrar(Expo(expr1,expr2))
evaluar(Expo(expr1,expr2),Atomo('x'), 5.0)
mostrar(Logaritmo(expr1))
evaluar(Logaritmo(expr1),Atomo('x'), 5.0)
```

la función responde así:

```
val res10: String = 5.0
val res11: Double = 5.0
val res12: String = "x  "
val res13: Double = 5.0
```

```
val res14: String = ((x   + 2.0) + (x   * x  ))
val res15: Double = 32.0
val res16: String = ((x   + 2.0) * (x   * x  ))
val res17: Double = 175.0
val res18: String = ((x   + 2.0) - (x   * x  ))
val res19: Double = -18.0
val res20: String = ((x   + 2.0) / (x   * x  ))
val res21: Double = 0.28
val res22: String = ((x   + 2.0) ^ (x   * x  ))
val res23: Double = 1.341068619663965E21
val res24: String = (lg((x   + 2.0)))
val res25: Double = 1.9459101490553132
```

1.4. Limpiando expresiones

Implemente una función `limpiar` que dada una función representada por la expresión f, retorne una fórmula equivalente pero que no contenga ceros ni unos innecesarios.

```
  def limpiar(f:Expr):Expr = {...}
```

Por ejemplo, a las siguientes invocaciones:

```
limpiar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x')))
mostrar(limpiar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x'))))
```

la función responde así:

```
val res26: Newton.Expr = Numero(3.0)
val res27: String = 3.0
```

1.5. Hallando raíces

Implemente una función iterativa `raizNewton` que que dada una función representada por la expresión f y un átomo a (que representa la variable de la función), y un valor x_0 flotante, candidato a raíz de f, y una función booleana ba retorne una raíz, r, de la función usando el método de Newton (ba es una función que recibe la expresión f, y el átomo a y el valor candidato a raíz, x_0, y devuelve *true* si $f(x_0)$ está suficientemente cerca de 0).

```
  def raizNewton(f:Expr, a:Atomo, x0:Double,
                 ba:(Expr, Atomo, Double)=>Boolean):Double = {...}
```

Por ejemplo, a las siguientes invocaciones:

```
def buenaAprox (f:Expr, a:Atomo, d:Double):Boolean = {
  evaluar(f,a,d) < 0.001
}

val e1= Resta(Prod(Atomo('x'),Atomo('x')), Numero(2.0))
val e2= Resta(Prod(Atomo('x'),Atomo('x')), Numero(4.0))
val e3 = Suma(Resta(Prod(Atomo('x'),Atomo('x')), Numero(4.0)), Prod(Numero(3.0),Atomo('x')))
raizNewton(e1, Atomo('x'), 2.0, buenaAprox)
raizNewton(e2, Atomo('x'), 2.0, buenaAprox)
raizNewton(e3, Atomo('x'), 2.0, buenaAprox)
```

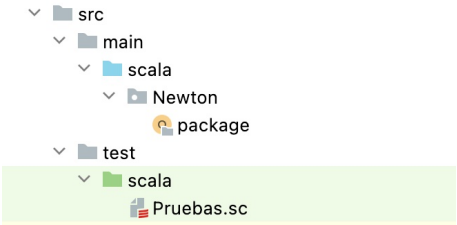
la función responde así:

```
def buenaAprox(f: Newton.Expr, a: Newton.Atomo, d: Double): Boolean



val e1: Newton.Resta = Resta(Prod(Atomo(x),Atomo(x)),Numero(2.0))
val e2: Newton.Resta = Resta(Prod(Atomo(x),Atomo(x)),Numero(4.0))
val e3: Newton.Suma = Suma(Resta(Prod(Atomo(x),Atomo(x)),Numero(4.0)),Prod(Numero(3.0),Atomo(x)))
val res28: Double = 1.4142156862745099
val res29: Double = 2.0
val res30: Double = 1.0000029768726761
```

2. Entrega

2.1. Paquete *Newton* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las funciones correspondientes a cada ejercicio, `mostrar, derivar, evaluar, limpiar` y `raizNewton` deben ser implementadas en un paquete de Scala denominado `Newton`. **Todas, salvo la última, deben usar reconocimiento de patrones esencialmente.** En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Newton {
  trait Expr
  case class Numero (d:Double) extends Expr
  case class Atomo (x:Char) extends Expr
  case class Suma(e1:Expr, e2:Expr) extends Expr
  case class Prod(e1:Expr, e2:Expr) extends Expr
  case class Resta(e1:Expr, e2:Expr) extends Expr
  case class Div(e1:Expr, e2:Expr) extends Expr
  case class Expo(e1:Expr, e2:Expr) extends Expr
  case class Logaritmo(e1:Expr) extends Expr


  def mostrar(e:Expr):String = {...
  }

  def derivar(f:Expr, a:Atomo):Expr = {...
  }

  def evaluar(f:Expr, a:Atomo, v:Double):Double ={...
 }

  def limpiar(f:Expr):Expr = {...
}

  def raizNewton(f:Expr, a:Atomo, x0:Double,
                 ba:(Expr, Atomo, Double)=>Boolean):Double ={...
}
```

```
}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import Newton._


val expr1=Suma(Atomo('x'), Numero(2))
val expr2=Prod(Atomo('x'), Atomo('x'))
val expr3= Suma(expr1, Expo(expr2,Numero(5)))
val expr4= Logaritmo(Atomo('x'))
val expr5=Prod(Div(expr1, expr2), Resta(expr3,expr4))
val expr6=Expo(Atomo('x'),Numero(3))
mostrar(expr1)
mostrar(expr2)
mostrar(expr3)
mostrar(expr4)
mostrar(expr5)
mostrar(expr6)



mostrar(derivar(expr6, Atomo('x')))
mostrar(derivar(expr2, Atomo('x')))
mostrar(derivar(expr2, Atomo('y')))
mostrar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x')))


mostrar(Numero(5.0))
evaluar(Numero(5.0),Atomo('x'), 1.0)
mostrar(Atomo('x'))
evaluar(Atomo('x'),Atomo('x'), 5.0)
mostrar(Suma(expr1,expr2))
evaluar(Suma(expr1,expr2),Atomo('x'), 5.0)
mostrar(Prod(expr1,expr2))
evaluar(Prod(expr1,expr2),Atomo('x'), 5.0)
mostrar(Resta(expr1,expr2))
evaluar(Resta(expr1,expr2),Atomo('x'), 5.0)
mostrar(Div(expr1,expr2))
evaluar(Div(expr1,expr2),Atomo('x'), 5.0)
mostrar(Expo(expr1,expr2))
evaluar(Expo(expr1,expr2),Atomo('x'), 5.0)
mostrar(Logaritmo(expr1))
evaluar(Logaritmo(expr1),Atomo('x'), 5.0)


limpiar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x')))
mostrar(limpiar(derivar(Suma(Atomo('k'), Prod(Numero(3.0), Atomo('x'))), Atomo('x'))))



def buenaAprox (f:Expr, a:Atomo, d:Double):Boolean = {
  evaluar(f,a,d) < 0.001
}

val e1= Resta(Prod(Atomo('x'),Atomo('x')), Numero(2.0))
val e2= Resta(Prod(Atomo('x'),Atomo('x')), Numero(4.0))
val e3 = Suma(Resta(Prod(Atomo('x'),Atomo('x')), Numero(4.0)), Prod(Numero(3.0),Atomo('x')))
raizNewton(e1, Atomo('x'), 2.0, buenaAprox)
raizNewton(e2, Atomo('x'), 2.0, buenaAprox)
raizNewton(e3, Atomo('x'), 2.0, buenaAprox)
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de uso del reconocimiento de patrones, informe de corrección y conclusiones.

2.2.1. Informe de uso del reconocimiento de patrones

Tal como se ha visto en clase, el reconocimiento de patrones es una herramienta my poderosa y expresiva para programar. En esta sección usted debe hacer una tabla indicando en cuáles funciones usó la técnica y en cuáles no. Y en las que no, indicar por qué no lo hizo.

También se espera una apreciación corta de su parte, sobre el uso del reconocimiento de patrones como técnica de programación.

2.2.2. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección Para cada función, `mostrar`, `derivar`, `evaluar`, `limpiar` y `raizNewton`, argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Casos de prueba Para cada función se requieren mínimo 5 casos de prueba donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión.

Una descripción de los casos de prueba diseñados, sus resultados y una argumentación de si cree o no que los casos de prueba son suficientes para confiar en la corrección de cada uno de sus programas, los registra en esta sección del informe. Obviamente, esta parte del informe debe ser coherente con el archivo de pruebas entregado, `pruebas.sc`.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 3 de octubre de 2024, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación funcional*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce

como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.