Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
CC3001- 1 Algoritmo y Estructura de datos

Cálculo de derivadas usando árboles binarios

Informe Tarea N° 3

Nombre: Sebastián Sepúlveda
Profesor: Patricio Pobrete
Auxiliares: Gabriel Flores M.
Sven Reisenegger M.
Fecha de entrega: 28/05/2018

Índice de Contenidos

Lista de Códigos

Introducción

El problema que se va a abordar en este informe será la construcción de un árbol de expresión aritmetica a partir de una expresión textual (contenida en un String) escrita en notación polaca inversa, calcular su derivada respecto a la variable solicitada por el usuario y publicar el resultado.

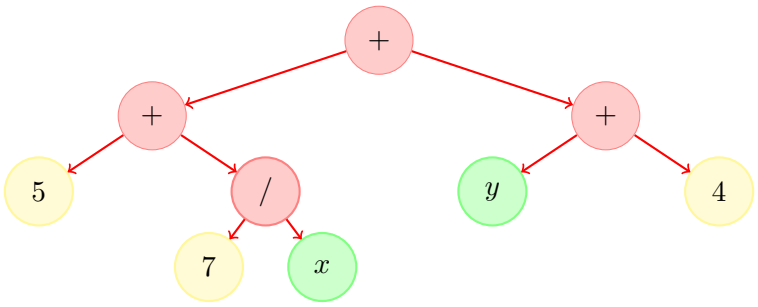
La notación polaca es una notación matemática en la que los operandos preceden a los operadores, esta notación no requiere uso de paréntesis, ya que no contiene ambigüedades.
Por ejemplo la expresión

$$>> \ (5/(7*x)) + (y+4)$$

Se representaria en notación polaca de la siguiente manera:

5	7	x	$*$	$/$	y	4	$+$	$+$

Según el problema propuesto para la elaboración de este informe, el caso anterior quedaría representado por el siguiente árbol de expresión:



Esta última expresión junto con el uso de pilas, será la que nos ayudará a obtener el resultado final requerido. El output del programa daria un resultado como el siguiente, si derivamos con respecto a la variable x:

$$>> \ 7*(5-y)/(5-y)*(5-y)$$

En el resultado final anterior se considero que $y \neq 5$.

Análisis del problema

Construcción del árbol de expresión

Para comenzar con la creación de la solución al problema, se buscó el método de leer el input que ingresaba el usuario, por lo que se asumió que *el usuario ingresará la expresión solamente con la notación polaca inversa, y con espacio entre cada término que ingrese.*

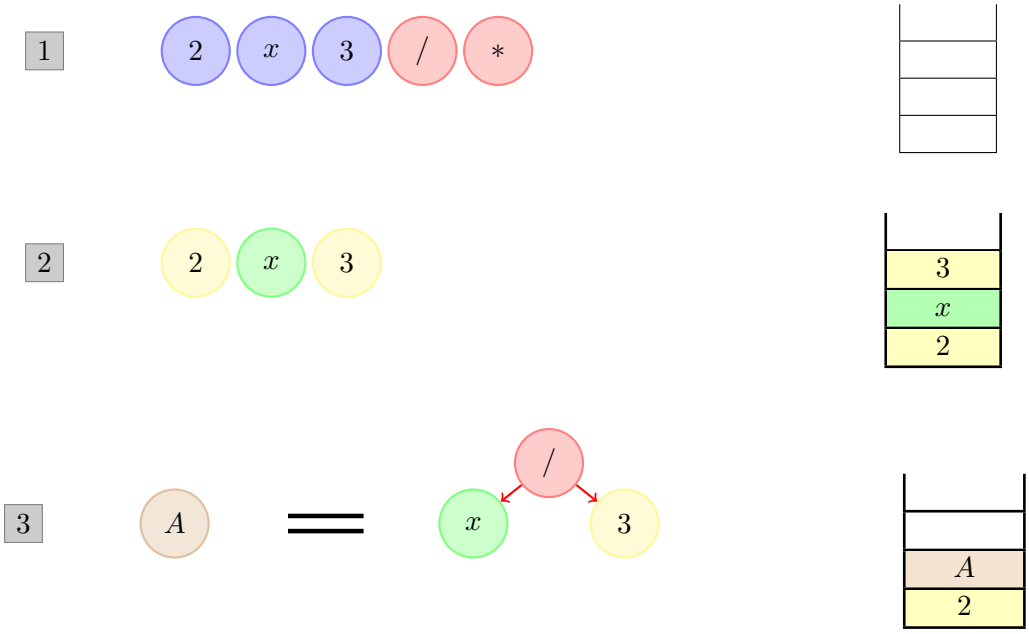
Se le pedirá al usuario que ingrese primero la expresión en matemática que desea derivar en polaca inversa, y luego que ingrese la variable que desea derivar. Es importante señalar que para nuestro programa se pensó en el uso de cualquier letra del abecedario castellano como variable que puede utilizar el usuario, teniendo como restricción la utilización de variables de sólo un carácter.

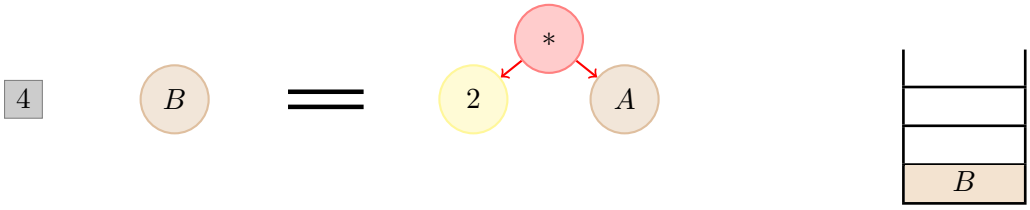
Luego, se buscó la manera de obtener el árbol binario que represente la expresión entregada por el usuario, construyendo una pila de nodos convenientemente definidos tal que al final del algoritmo la pila contenga un solo elemento que será precisamente el árbol por encontrar.

- Si el símbolo que se lee es un número o una variable, se crea un nodo con ese valor y se hace *push* a la pila.
- Si el símbolo que se lee es una operación, entonces se crea un nodo, con el valor de esa operación, tal que sus hijos son los elementos del stack que le siguen. El siguiente elemento será su nodo derecho y el otro el izquierdo.

En el programa, luego de devolver el `String` reprentante de la expresión, se señalará si este posee una división por cero, donde será necesario que el usuario ingrese nuevamente la expresión correcta que desee derivar para encontrar el resultado requerido.

Un ejemplo de lo que realiza el método `StackConstructor` se visualiza en la siguiente secuencia de figuras, donde finalmente B es el árbol de la expresión ingresada en [1]:





Construcción árbol representante de la derivada

Para construcción del árbol que contiene la expresión de la derivada de la expresión original basta aplicar las reglas de la derivación normales, las cuales son recursivas:

- Si el nodo raíz contiene un número o una variable que no es respecto a la cual se deriva la expresión, el resultado es un nodo que contiene un cero.
- Si el nodo raíz contiene la variable respecto a la cual se deriva la expresión el resultado es un nodo con el número 1
- Si el nodo raíz contiene una operación entonces el resultado es un árbol que contiene la expresión correspondiente a la regla de la derivada para la operación:

$$(f \pm g)' = f' \pm g'$$

$$(fg)' = f'g \ \pm fg'$$

$$\left(\frac{f}{g}\right)' = \left(\frac{f'g \pm fg'}{g * g}\right)$$

Es importante destacar que para el diseño de nuestro programa no se considero que las expresiones algebraicas fueran simplificadas en la división. También se consideran los casos de bordes en los cuales se divide por cero, donde se enviará un mensaje al usuario, luego de ingresar la expresión, que el código presenta un error de división por cero, que al derivar aún mantiene.

En el caso que el usuario desee derivar expresiones como $1/x$ el signo menos que arroja la derivada de la expresión estará representado por un $(0 - 1)$ donde no se simplifica el signo menos para mantener la lógica del árbol del expresión.

Construcción de la expresión simplificada e imprimir en pantalla

Para devolver el resultado simplificado de la expresión dada primero nos aseguramos que la expresión entregada ya esté simplificada, aunque se asume que el usuario entregaría esta expresión ya resulta desde antes, necesatando solamente su derivada.

Después de analizar si el resultado entregado ya está simplificado, se comienza la derivación, y luego se simplifica nuevamente la expresión arrojada por la derivada, que en caso de seguir incluyendo ceros y/o unos que son despreciables en el resultado, son eliminados por el método `toStringArbol` el cual es el mismo que devuelve la expresión en String dado un árbol de expresión matemática.

Finalmente la impresión en pantalla será el resultado del `String` dado, en formato in-fijo y el resultado de la derivada de la expresión.

El esqueleto del programa quedaría de la siguiente manera:

Código 1: Esqueleto del programa

```
class Arbol{//...
}
class Lista{// ...
}
class Stack{// ...
}
//Clase principal
public class Mathematicas{//...
   public static void main(String[] args){// ...
   }
   public static Arbol StackConstructor(String dato){//...
   }
   private static boolean esVariable(String s){// ...
   }
   private static boolean esNumero(String s){//...
   }
   private static boolean esOperador(String s){//...
   }
   private static Arbol derivar(Arbol arbolNormal, String variable){// ...
   }
   private static Arbol simplificar(Arbol arbol){// ...
   }
   private static String toStringArbol(Arbol arbol){//...
   }
}
```

Implementación

En el desarrollo del trabajo, son necesarias 3 clases esenciales, `class Arbol`, `class Lista` y `class Stack`, las cuales están detalladas en el apunte.

En respuesta a los datos ingresados, el programa responde con la expresión in-fijo de la expresión, y luego con la expresión in-fijo de la derivada de la expresión. Para simplificar problemas mayores a los solicitados se siguieron las siguientes reglas:

- Las expresiones ingresadas tendrán las operaciones: suma (+), resta (-), multiplicación (*) y división (/).
- Las multiplicaciones de un término por 0 se reemplazan por 0.
- Las multiplicaciones de un término por 1 se reemplazan por el mismo término.
- Las sumas o restas de un término con 0 se reemplazan por el mismo término.
- Las divisiones de un término por 1 se reemplazan por el término.
- La expresión final omite los paréntesis que no son necesarios. Por ejemplo:

$$(((a * b) * c) + d)/e = (a * b * c + d)/e$$

  Un ejemplo de salida simplificada es:

$$(2 * x) + 1 + 0 = 2 * x + 1$$

Para lograr esto se siguió la secuencia señalada en la sección anterior:

1. **Construcción del árbol de la expresión utilizando una pila:**

   Para ello, en nuestro programa se creó el método `StackConstructor`(Código [2]), el cuál recibe un elemento tipo `String` y devuelve el árbol de expresión.

Código 2: Método `StackConstructor`

```
String[] ldato = dato.split(" "); //crea una lista con los Strings de la expresion dada.
Stack stack = new Stack();
Arbol error = new Arbol("Error al ingresar expresión");
for (String miembro : ldato){
   if (esNumero(miembro)||esVariable(miembro)){
      Arbol a = new Arbol(miembro);
      stack.apilar(a);} //push
   else if (esOperador(miembro)){
      Arbol aux1 = stack.desapilar();
      Arbol aux2 = stack.desapilar();
      Arbol arbol = new Arbol(miembro, aux2, aux1);
      stack.apilar(arbol);}
   else return error;
}
Arbol a = stack.desapilar();
return a;
```

2. **Construcción del árbol de la expresión derivada:**

   Antes de empezar a derivar el árbol entregado por el método `StackConstructor`, creamos 3 métodos *booleanos* que nos ayudarán a identificar si el nodo en donde nos encontremos `esVariable, esNumero or esOperador`, según las restricciones que analizamos en la sección anterior.

   Para la creación de este árbol se creó el método `derivar`(Código [3]), el cual, dado un `Arbol` de expresión, devuelve otro árbol que representa la derivada del árbol ingresado.

   Código 3: Método `derivar`

```
Arbol derivada = new Arbol("");
String v = arbolNormal.valor;
Arbol i = arbolNormal.izq;
Arbol d = arbolNormal.der;
Arbol cero = new Arbol("0");
Arbol uno = new Arbol("1");

if (esNumero(v)) derivada = cero;
else if (esVariable(v)){
   if (v.equals(variable)) derivada = uno;
   else derivada = cero;
}
else if (esOperador(v)){
   if (esNumero(i.valor) && esNumero(d.valor)) return cero;
   else if (v.equals("+")) derivada = new Arbol(v, derivar(i, variable),derivar(d, variable));
   else if(v.equals("-")) derivada = new Arbol(v, derivar(i, variable),derivar(d, variable));
   else if(v.equals("*")){
      Arbol a = new Arbol(v, derivar(i, variable), d);
      Arbol b = new Arbol(v, i, derivar(d, variable));
      derivada = new Arbol("+", a, b);
   }
   else if(v.equals("/")){
      Arbol numa = new Arbol("*", derivar(i, variable), d);
      Arbol numb = new Arbol("*", i, derivar(d,variable));
      Arbol num = new Arbol("-", numa, numb);
      Arbol div = new Arbol("*", d, d);
      derivada = new Arbol("/", num, div);
   }
}
return derivada;
```

   El árbol entregado por `derivar` no es el árbol totalmente simplificado, si no que una expresión que representa la derivada total del `Arbol` ingresado inicialmente.

3. **Simplificar el árbol de la expresión resultante:**

   Dado que el usuario al comienzo nos podría ingresar un árbol con expresiones innecesarias, y que la expresión del árbol que obtenemos del método `derivar` nos puede arrojar varios ceros y unos que tampoco son necesarios en el resultado final, se hizo necesario crear el método `simplificar` (Código [4]), que dado un Arbol de expresión, devuelve un `Arbol` simplificado hasta al menos una de sus hojas, donde en caso de presentarse casos (`Arbol` ± 0) o (`Arbol` $*0$), por ejemplo, el método `toStringArbol` será el encargado de simplificar esta expresión final, simplificando así de mejor manera la expresión.

Código 4: Método `simplificar` - Casos principales

```
1     if (arbol==null) return arbol;
2     Arbol resumen = new Arbol("");
3     String v = arbol.valor;
4     Arbol i = arbol.izq;
5     Arbol d = arbol.der;
6     Arbol cero = new Arbol("0"); //en caso de multiplicación por 0 por ejemplo
7     Arbol error = new Arbol("error");
8     if (esNumero(v)) resumen = cero;
9     else if (esVariable(v)) resumen = arbol;
10    else if(esOperador(v)){
11       //En caso que las hojas sean numeros o variables
12       if((esVariable(i.valor)||esVariable(d.valor)) || (esNumero(d.valor)||esNumero(i.valor))){ //...
13       }
14       //En caso que solo la hoja del lado izquierdo sea operador
15       if(esOperador(i.valor)){ // ...
16       }
17       //En caso que solo la hoja del lado derecho sea operador
18       if(esOperador(d.valor)){ // ...
19       }
20       //En caso de que ambas hojas sean operadores
21       if(esOperador(i.valor) && esOperador(d.valor)){ //...
22       }
23       //Si se salta todos los casos no es necesario simplificar, se devuelve el arbol ingresado
24       resumen = new Arbol(v,i,d);
25       return resumen;
26    }
27    return arbol;
```

Finalmente se genera el `output` del programa y se imprime en pantalla el resultado obtenido de la derivación.

Modo de uso

Para el uso correcto del programa se le recomienda al usuario seguir los siguientes pasos y considerar los supuestos expuestos en las secciones anteriores:

- Ingresar expresión matemática en polaca inversa, ya simplificada en lo posible (resuelta, sin operadores ni variables innecesarias) utilizando como variables las letras del Abecedario Castellano [1]. Presionar Enter.
- Indicar la variable por la cual se piensa derivar. Presionar Enter.
- Luego de que recibir el resultado, puede ingresar otra expresión, ingresar la expresión anterior bien expresada, ingresar la misma expresión o salir del programa apretando `Ctrl + C`, recibiendo un mensaje "EOF".

Resultados

Los casos de prueba que se hicieron al programa fueron los ejemplos dados en el enunciado de la tarea, junto con casos bordes como multiplicación de ceros, unos o mostrar cuando se divide por cero:

1. `input`:
   $>>\ 2\ x\ 3\ /\ *\ y\ x\ -\ +$
   $>>\ x$
   `output`:
   1.- Expresion in-fijo del arbol: $2\ *\ x\ /\ 3\ +\ (y\ -\ x)$
   2.- Derivada de la expresion respecto a x : $2\ *\ 3\ /\ 3\ *\ 3\ +\ 1$

2. `input`:
   $>>\ 2\ x\ 3\ /\ *\ y\ x\ -\ +$
   $>>\ y$
   `output`:
   1.- Expresion in-fijo del arbol: $2\ *\ x\ /\ 3\ +\ (y\ -\ x)$
   2.- Derivada de la expresion respecto a y : 1

3. `input`:
   $>>\ x\ 0\ +\ y\ 1\ /\ *$
   $>>\ x$
   `output`:
   1.- Expresion in-fijo del arbol: $x\ *\ y$
   2.- Derivada de la expresion respecto a x : y

4. `input`:
   $>>\ 0\ 0\ 1\ /\ +\ 0\ 1\ -\ +$

$>>\ x$
`output`:
1.- Expresion in-fijo del arbol: 1
2.- Derivada de la expresion respecto a x : 0

5. `input`:
   $>>\ 1\ x\ x\ *\ /$
   $>>\ x$
   `output`:
   1.- Expresion in-fijo del arbol: $1/x * x$
   2.- Derivada de la expresion respecto a x : $(0 - (x + x))/x * x * x * x$

Conclusión

Finalmente, podemos analizar que el trabajo realizado en conjunto con las clases `Arbol`, `Lista`, `Stack`, que fueron vitales para poder alcanzar los resultados obtenidos, es aprovechado con mayor eficacia con el uso de las propiedades de los arboles binarios, destacando el corto periodo de tiempo que tarda en hacer la recursión cada método, a pesar de la gran cantidad de `if` o `else if`, `else` que se colocaban en ellos.

Sin embargo, es posible que el programa sea más eficaz de lo que ya es eliminando de alguna forma la recursividad en alguno de los métodos creados y lograr que el tiempo de ejecución para una expresión más larga de las que probamos en la sección anterior tarde menos tiempo.

Anexo

Código 5: Código Final

```
27  import java.util.*;
28
29  class Arbol{//revisar en apunte
30  }
31  class Lista{//revisar en apunte
32  }
33  class Stack{//revisar en apunte
34  }
35  //Clase principal
36 public class Mathematicas{
37  public static void main(String[] args){
38     Scanner sc= new Scanner(System.in);
39     System.out.println("Ingrese primero la notacion polaca inversa:");
40     while(sc.hasNextLine()){
41        String s = sc.nextLine();
42        System.out.println("Ingrese variable que desea derivar:");
43        String variable = sc.nextLine();
44        //ver que nos este dando un valor cierto
45        int largo = s.length();
46        int largov = variable.length();
47        if(largo == 0){
48           System.out.println("Error al ejecutar, ingrese todo de nuevo ");
49        }
50        else if (largov == 0){
51           System.out.println("Error al ejecutar, ingrese todo de nuevo ");
52        }
53        else{
54           Arbol arbol = StackConstructor(s);
55           Arbol simplificadoArbol = simplificar(arbol);
56           String simpArbol = toStringArbol(simplificadoArbol);
57           System.out.println("1.- Expresion in-fijo del arbol: " +simpArbol);
58           Arbol derivada_arbol = derivar(simplificadoArbol, variable);
59           Arbol simple = simplificar(derivada_arbol);
60           String sim = toStringArbol(simple);
61           System.out.println("2.- Derivada de la expresion " + "respecto a " + variable + " : " +sim);
62        }
63     }
64     System.out.println("EOF");//para chequear que se acaba el while pueden poner un print acá
65  }
66  public static Arbol StackConstructor(String dato){
67     String[] ldato = dato.split(" ");
68     Stack stack = new Stack();
69     Arbol error = new Arbol("Error al ingresar expresión");
70     for (String miembro : ldato){
71        //si el simbolo que se lee es una variable o un numero se crea un nodo, ie un arbol de expresion, con ese valor
72        //y haremos push
73        if (esNumero(miembro)||esVariable(miembro)){
74           Arbol a = new Arbol(miembro);
75           stack.apilar(a);
76        }
77        else if (esOperador(miembro)){
78           Arbol aux1 = stack.desapilar();
79           Arbol aux2 = stack.desapilar();
80           Arbol arbol = new Arbol(miembro, aux2, aux1);
81           stack.apilar(arbol);
82        }
83        else return error;
```

```
84      }
85      Arbol a = stack.desapilar();
86      return a;
87    }
88    private static boolean esVariable(String s){
89      String alfa = "aqwsedrftgyhujikolpñzxcvbnm";
90      char car = s.charAt(0);
91      for(int i=0; i<alfa.length();i++){
92        char aux = alfa.charAt(i);
93        if(car==aux){
94          return true;
95        }
96      }
97      return false;
98    }
99    private static boolean esNumero(String s){
100     for(int i = 0; i<s.length(); i++){
101       if(!Character.isDigit(s.charAt(i))){
102         return false;
103       }
104     }
105     return true;
106   }
107   private static boolean esOperador(String s){
108     if(s.equals("+") || s.equals("/") || s.equals("*") || s.equals("-")) return true;
109     return false;
110   }
111   private static Arbol derivar(Arbol arbolNormal, String variable){
112     Arbol derivada = new Arbol("");
113     String v = arbolNormal.valor;
114     Arbol i = arbolNormal.izq;
115     Arbol d = arbolNormal.der;
116     Arbol cero = new Arbol("0");
117     Arbol uno = new Arbol("1");
118     if (esNumero(v)) derivada = cero;
119     else if (esVariable(v)){
120       if (v.equals(variable)) derivada = uno;
121       else derivada = cero;
122     }
123     else if (esOperador(v)){
124       if (esNumero(i.valor) && esNumero(d.valor)) return cero;
125       else if (v.equals("+")) derivada = new Arbol(v, derivar(i, variable),derivar(d, variable));
126       else if(v.equals("-")) derivada = new Arbol(v, derivar(i, variable),derivar(d, variable));
127       else if(v.equals("*")){
128         Arbol a = new Arbol(v, derivar(i, variable), d);
129         Arbol b = new Arbol(v, i, derivar(d, variable));
130         derivada = new Arbol("+", a, b);
131       }
132       else if(v.equals("/")){
133         Arbol numa = new Arbol("*", derivar(i, variable), d);
134         Arbol numb = new Arbol("*", i, derivar(d,variable));
135         Arbol num = new Arbol("-", numa, numb);
136         Arbol div = new Arbol("*", d, d);
137         derivada = new Arbol("/", num, div);
138       }
139     }
140     return derivada;
141   }
142   private static Arbol simplificar(Arbol arbol){
143     if (arbol==null) return arbol;
144     Arbol resumen = new Arbol("");
145     String v = arbol.valor;
146     Arbol i = arbol.izq;
```

```
147     Arbol d = arbol.der;
148     Arbol cero = new Arbol("0");
149     Arbol error = new Arbol("error");
150     if (esNumero(v)) resumen = cero;
151     else if (esVariable(v)) resumen = arbol;
152     else if(esOperador(v)){
153        if((esVariable(i.valor)||esVariable(d.valor)) || (esNumero(d.valor)||esNumero(i.valor))){
154           String vi = i.valor;
155           String vd = d.valor;
156           if(v.equals("*")){
157              Arbol auxi = i;
158              Arbol auxd = d;
159              if((vi.equals("0") && esVariable(vd))||(esVariable(vi) && vd.equals("0"))){
160                 if (vi.equals("0")) return cero;
161                 else if (vd.equals("0")) return cero;
162              }
163              else if((vi.equals("1") && esVariable(vd))||(esVariable(vi) && vd.equals("1"))){
164                 if (esVariable(vi)) return auxi;
165                 else if (esVariable(vd)) return auxd;
166              }
167              else if((vi.equals("0") && esNumero(vd))||(esNumero(vi) && vd.equals("0"))){
168                 if (vi.equals("0")) return cero;
169                 else if (vd.equals("0")) return cero;
170              }
171              else if((esNumero(vi) && vd.equals("1"))||(vi.equals("1") && esNumero(vd))){
172                 if (esNumero(vi) && !vi.equals("1")) return auxi;
173                 else if (esNumero(vd) && !vd.equals("1")) return auxd;
174              }
175              //se acabo con *
176           }
177           // no se considera 0 - x, porque debe quedar asi (unica manera de representar los numeros negativos en el programa)
178           else if(v.equals("+")){
179              Arbol aux = i;
180              Arbol auxd = d;
181              if((esVariable(vi) && vd.equals("0"))||(vi.equals("0") && esVariable(vd))){
182                 if (esVariable(vi)) return i;
183                 else if (esVariable(vd)) return d;
184                 }
185              if((esNumero(vi) && vd.equals("0"))||(vi.equals("0") && esNumero(vd))){
186                 if (esNumero(vi) && !vi.equals("0")) return i;
187                 else if (esNumero(vd) && !vd.equals("0")) return d;
188                 else if (vi.equals("0") && vd.equals("0")) return cero;
189                 }
190           }
191           else if(v.equals("-")){
192              Arbol aux = i;
193              Arbol auxd = d;
194              if((esVariable(vi)||esNumero(vi)) && vd.equals("0")) return i;
195           }
196           else if(v.equals("/")){
197              Arbol aux = i;
198              Arbol auxd = d;
199              if(vi.equals("0")){
200                 if(esVariable(vd)) return cero;
201                 else if(esNumero(vd)) return cero;
202              }
203              else if(vd.equals("0")){
204                 if (esVariable(vi)){
205                    System.out.println("ERROR FATAL: Division Zero");
206                    return error;
207                 }
208                 else if (esNumero(vi)){
209                    System.out.println("ERROR FATAL: Division Zero");
```

```
                    return error;
                }
            }
            else if(vd.equals("1")){
                if(esVariable(vi)) return i;
                else if (esNumero(vi)) return i;
            }
        }
    }
    if(esOperador(i.valor)){
        if(esNumero(d.valor)){
        Arbol auxi = i;
        Arbol auxd = d;
            if (v.equals("*")){
                if(d.valor.equals("1")) return simplificar(i);
                else if(d.valor.equals("0")) return cero;
            }
            else if (v.equals("+") || v.equals("-")){
                if(d.valor.equals("0")) return simplificar(i);
            }
            else if(v.equals("/")){
                if(d.valor.equals("1")) return simplificar(i);
                else if (d.valor.equals("0")){
                    System.out.println("ERROR FATAL: Divizion Zero");
                    return error;
                }
            }
            else{
                Arbol izquierdo = simplificar(i);
                resumen = new Arbol(v,izquierdo,d);
                return resumen;
                }
        }
    }
    if(esOperador(d.valor)){
        if(esNumero(i.valor)){
        Arbol auxi = i;
        Arbol auxd = d;
        if(v.equals("/")){
            if(i.valor.equals("0")) return new Arbol("0");
        }
        if(v.equals("*")){
            if(i.valor.equals("0")) return new Arbol("0");
            else if(i.valor.equals("1")){
                resumen = simplificar(auxd);
                return resumen;
            }
        }
        if(v.equals("+")){
            if(i.valor.equals("0")){
                resumen = simplificar(d);
                return resumen;
            }
        }
        else {
            Arbol derecho = simplificar(d);
            resumen = new Arbol(v,i,derecho);
            //return simplificar(resumen);
            return resumen;
            }
        }
    }
    if(esOperador(i.valor) && esOperador(d.valor)){
```

```
273             Arbol izquierdo = simplificar(i);
274             Arbol derecho = simplificar(d);
275             resumen = new Arbol(v,simplificar(izquierdo), simplificar (derecho));
276             return resumen;
277          }
278          resumen = new Arbol(v,i,d);
279          return resumen;
280       }
281       return arbol;
282    }
283    private static String toStringArbol(Arbol arbol){
284       if (arbol == null) return null;
285       String v = arbol.valor;
286       Arbol i = arbol.izq;
287       Arbol d = arbol.der;
288       Arbol cero = new Arbol("0");
289       Arbol error = new Arbol("error");
290       if (esNumero(v)){
291          int algo = Integer.parseInt(v);
292          if(algo == 0) return "0";
293          return v;
294       }
295       else if(esVariable(v)){
296          return v;
297       }
298       else if (esOperador(v)){
299          String si = i.valor;
300          String sd = d.valor;
301          if (v.equals("*") || v.equals("/")){
302             if(si.equals("0") && sd.equals("0")){
303                if(v.equals("/")){
304                   System.out.println("ERROR FATAL: Division Zero");
305                   return toStringArbol(error);
306                }
307                return toStringArbol(cero);
308             }
309             else if(si.equals("0")) return toStringArbol(cero);
310             else if(sd.equals("0")){
311                if(v.equals("/")){
312                   System.out.println("ERROR FATAL: Division Zero");
313                   return toStringArbol(error);
314                }
315                return toStringArbol(cero);
316             }
317             else if(si.equals("+") || si.equals("-")){
318                if (sd.equals("0")) return toStringArbol(cero);
319                else if (sd.equals("1")) return toStringArbol(i);
320                else if (sd.equals("+") || sd.equals("-")){
321                   return "(" + toStringArbol(i) + ") " + v + " (" + toStringArbol(d) + ")";
322                }
323                return "(" + toStringArbol(i) + ") " + v + " " +toStringArbol(d);
324             }
325             else if(sd.equals("+") || sd.equals("-")){
326                if (si.equals("0")) return toStringArbol(cero);
327                else if (si.equals("1")) return toStringArbol(d);
328                if (si.equals("+") || si.equals("-")){
329                   return "(" + toStringArbol(i) + ") " + v + " (" + toStringArbol(d) + ")";
330                }
331                return toStringArbol(i) + " " + v + " (" + toStringArbol(d) + ")";
332             }
333          }
334          else if (v.equals("+") || v.equals("-")){
335             //dos lados cero
```

```
336             if(si.equals("0") && sd.equals("0")){
337                return toStringArbol(cero);
338             }
339             //lado izquierdo no es cero
340             else if (sd.equals("0")){
341                if(esOperador(si)) return toStringArbol(i);
342                else if (esVariable(si) || esNumero(si)) return toStringArbol(i);
343             }
344             //lado derecho no es cero
345             else if(si.equals("0")){
346                String derecho = toStringArbol(d);
347                int der_largo = derecho.length();
348                if(esOperador(sd) && !v.equals("-") ) return toStringArbol(d);
349                else if (esVariable(sd) || esNumero(sd)) return toStringArbol(d);
350                else if (der_largo==1) return toStringArbol(i) + " " + v + " " + toStringArbol(d);
351                else if (v.equals("-")) return toStringArbol(i) + " " + v + " (" + toStringArbol(d) + ")";
352             }
353             else if(si.equals("/") || si.equals("*")){
354                if(sd.equals("+") || sd.equals("-")){
355                   String derecho = toStringArbol(d);
356                   int der_largo = derecho.length();
357                   if (der_largo==1) return toStringArbol(i) + " " + v + " " + toStringArbol(d);
358                   return toStringArbol(i) + " " + v + " (" + toStringArbol(d) + ")";
359                }
360             }
361             else if(sd.equals("/") || sd.equals("*")){
362                if(si.equals("+") || si.equals("-")){
363                   String izquierdo = toStringArbol(i);
364                   int izq_largo = izquierdo.length();
365                   if (izq_largo==1) return toStringArbol(i) + " " + v + " " + toStringArbol(d);
366                   return "(" + toStringArbol(i) + ") " + v + " " + toStringArbol(d);
367                }
368             }
369          }
370          else if ((si.equals("/") || sd.equals("/")) || (si.equals("*") || sd.equals("*"))){
371             if(esNumero(si) || esVariable(si)) return toStringArbol(i) + " " + v + " (" + toStringArbol(d) + ")";
372             else if(esNumero(sd) || esVariable(sd)) return "(" + toStringArbol(i) + ") " + v + " " + toStringArbol(d);
373             return "("+ toStringArbol(i) + ") " + v + " (" + toStringArbol(d) + ")";
374          }
375          return toStringArbol(i) + " " + v + " " + toStringArbol(d);
376       }
377       return "(" + toStringArbol(i) + " " + v + " " + toStringArbol(d) + ")";
378    }
379 }
```

Referencias

[1] Ejemplos de Abecedarios *Abecedario Castellano*
http://aliso.pntic.mec.es/agalle17/cultura_clasica/alfabetos.htm

[2] Notación Polaca *Notación polaca inversa*
https://es.wikipedia.org/wiki/Notaci%C3%B3n_polaca_inversa

[3] Apunte Algoritmo y Estructuras de datos. *Tipos de datos abstractos - Universidad de Chile*
https://users.dcc.uchile.cl/~bebustos/apuntes/cc3001/TDA/