Diseño y Análisis de Algoritmos Camilo Vargas Cabrera (200612197)
G11 - Proyecto Final Diciembre 1 / 2011

Descripción del Problema:

El objetivo del proyecto final es diseñar e implementar un software que sea capaz de despejar un parámetro específico de una ecuación que recibe como entrada. La ecuación puede contener operadores binarios como 'suma', 'resta', 'multiplicación' y 'división' y operadores unarios como 'logaritmo', 'exponenciación', 'elevar al cuadrado' y 'raíz cuadrada'.

Estructura de clases:

El sistema se ha estructurado de la siguiente manera:

- proyectoFinal
  - Parser: Clase encargada de la lectura de los archivos de entrada y generación de los archivos de salida. Contiene el método Main().
  - proyectoFinal.ecuacion
    - Ecuacion: Clase encargada de la generación del árbol sintáctico de la ecuación y del proceso de resolución. Procesa las cadena que representa la ecuación en **notación posfija.**
    - Nodo: Clase que representa un nodo de la ecuación, todas las demás clases extienden de ella.
    - OperadorBinario: Nodo que representa un operador binario. Tiene dos nodos hijos.
    - OperadorUnario: Nodo que representa un operador unario. Tiene un solo nodo hijo.
    - Parametro: Nodo que representa cualquier parámetro de la ecuación. Todas las hojas del árbol deben ser parámetros.
    - proyectoFinal.ecuacion.operadores
      - Cuadrado: Nodo que representa la operación elevar al cuadrado (^2). Extiende de OperadorUnario. Inverso de 'Raiz'.
      - Raiz: Nodo que representa la operación raíz cuadrada (rz). Extiende de OperadorUnario. Inverso de 'Cuadrado'.
      - Exponenciación: Nodo que representa la operación exponenciar (2^). Extiende de OperadorUnario. Inverso de 'Logaritmo'.
      - Logaritmo: Nodo que representa la operación logaritmo base 2 (log). Extiende de OperadorUnario. Inverso de 'Exponenciación'.
      - Suma: Nodo que representa las operaciones suma y resta (+ -). Extiende de OperadorBinario.
      - Multiplicación: Nodo que representa las operaciones de multiplicación y división (* /). Extiende de OperadorBinario.
      - Igualdad: Nodo que representa una igualdad (=). Extiende de OperadorBinario. Contiene los **algoritmos heurísticos**.
  - proyectoFinal.excepciones: Paquete que contiene múltiples excepciones desarrolladas para facilitar el debug del proyecto.

Para simplificar la implementación de varios de los algoritmos recursivos, las operaciones de resta y división se trabajan como casos particulares de 'Suma' y 'Multiplicación'. Para este efecto 'Nodo' contiene los atributos booleanos 'negativo' y 'divisor' y los métodos

'cambiarSigno()' e 'invertirNodo()' para operar sobre estos atributos.

Para facilitar el debug de la aplicación, la clase 'Nodo' contiene el método abstracto 'recorrerInOrden()', el cual permite que los archivos de salida se generen utilizando notación estándar.

Resolución y Tipos de Transformaciones:

La resolución de una ecuación se descompone en múltiples transformaciones secuenciales, las cuales se han clasificado en 5 tipos principales:

- Simplificación: Representa operaciones de simplificación aritmética estándar. Estas operaciones las ejecuta cada nodo considerando únicamente a sus hijos.
- Aritmética: Caso particular de 'Simplificación' en el cual el resultado es un parámetro numérico. Aquí se incluyen la suma y multiplicación de dos números, así como el cálculo de las operaciones unarias cuyo hijo es un número.
- Atracción: Este tipo de transformación representa el proceso de colocar todas las incógnitas en el lado izquierdo de la ecuación.
- Aislamiento: Estas transformaciones representan el proceso de colocar todos los parámetros que no contienen incógnitas al lado derecho de la ecuación.
- Colección: Este tipo representa las transformaciones orientadas a agrupar parámetros similares. En general es un tipo de simplificación, pero debe operar sobre todo un subárbol.

Algoritmos considerados para la aplicación de las transformaciones:

Una ecuación cualquiera genera un árbol sintáctico de profundidad arbitraria, el cual solo puede procesarse efectivamente con algoritmos recursivos correctamente estructurados. La forma en la cual se implementen estos algoritmos determina la complejidad temporal y espacial de la solución propuesta. El desarrollo de estos algoritmos ha representado una dificultad significativa, la cual fue subestimada al inicio del proyecto.

Estimación de costos:

Para este proyecto los costos se estimarán en función del número de nodos del árbol sintáctico generado (n) y del número de apariciones de la incógnita en la ecuación (i). Cada uno de los caracteres en la especificación posfija de la ecuación genera un nodo, por lo cual el número inicial de nodos será igual número de caracteres en la ecuación.

La unidad de medida utilizada para la complejidad temporal será 'visita a un nodo', es decir que cada consulta a un nodo cualquiera costará 1. Para la complejidad espacial se utilizará el número de objetos tipo 'Nodo' utilizados.

Si la ecuación tiene solución, en general se puede afirmar que cada transformación realizada reduce en 1 el número de nodos del árbol sintáctico, por lo cual si no se crean nodos adicionales en las transformaciones la complejidad espacial será cero. En un caso ideal, en el cual para realizar cada transformación solo se debe consultar un nodo, la complejidad temporal del algoritmo sería $O(n)$. Por el contrario si para cada transformación se deben consultar todos los nodos la complejidad temporal será $O(n!)$.

Alternativas Estudiadas:

En principio se consideró un algoritmo de resolución *centralizado* cuya lógica en general era:

- Realizar un recorrido completo de la ecuación, identificando la ubicación de las incógnitas en el árbol sintáctico. CT = n.
- Calcular la distancia *d* en número de nodos entre dos de las incógnitas. *CT = O(d).*
- Realizar transformaciones sobre el subárbol que contiene las dos incógnitas y calcular nuevamente las distancias. *CE = O(d)* en la medida que se debe duplicar el árbol para realizar las transformaciones. $CT = O(d^2)$ ya que para cada subárbol creado debo calcular la nueva distancia.
- Elegir el árbol que reduce la distancia y repetir el proceso hasta llegar a *i = 1*. El costo total del proceso sería entonces $CT = O(i*d^2 + n)$ y $CE = O(d^2)$ aproximadamente, donde *d* depende de la separación de las incógnitas en la ecuación.

Proceso de resolución descentralizado:

Tras recrear el proceso de resolución esperado para algunos casos se observó que ciertos procesos se pueden implementar de forma tal que operen autónomamente. A este modo de operación se le denomina *descentralizado* en la medida que cada 'Nodo' determina como operará a sus hijos de forma independiente. En estos casos el *CT = n* y *CE = 0.* Si fuera posible implementar los cinco procesos de esta manera, la solución se podría alcanzar en *CT=O(n)* y *CE=0.*

De las recreaciones realizadas se pudo concluir que en general los procesos de 'simplificación' y 'atracción' se pueden implementar de forma descentralizada. Por su parte, los procesos de 'colección' y 'aislamiento' pueden operar de forma descentralizada sobre subárboles que representan combinaciones lineales.

Se sospecha que la resolución de ecuaciones que involucran operadores unarios sin simplificaciones evidentes, podría requerir de algoritmos *centralizados* con lógicas similares a la propuesta inicialmente.

Arquitectura

La clase 'Nodo' declara métodos abstractos para cada uno de los cinco tipos de transformaciones propuestos, así como algunos métodos auxiliares para ciertos procesos. Estos métodos deben devolver un objeto tipo 'Nodo' que representa la ecuación reducida o simplificada, este objeto se asignará a la referencia sobre la cual que se invocó el método.

Cada nodo invoca el mismo método sobre sus hijos antes de proceder con su simplificación, lo cual garantiza que él va a trabajar sobre la versión más simplificada de sus subárboles. La terminación de la recurrencia se garantiza en las hojas del árbol, es decir en la clase 'Parámetro'. Si un determinado nodo no puede realizar ninguna simplificación, se retorna a sí mismo, con lo cual el árbol se mantiene inalterado.

Hasta este punto cada nodo es llamado una vez por cada una de las transformaciones invocadas (*CT = O(n)*) y las transformaciones se han realizado sobre el mismo árbol (*CE=0*). Lo anterior aplica sin modificaciones para las transformaciones de tipo 'Simplificación', las cuales operan únicamente sobre sus hijos inmediatos. Las operaciones de 'Atracción' y 'Colección', las cuales deben operar sobre todos sus descendientes, realizan llamados a métodos auxiliares o subrutinas cuya complejidad

temporal es O(n).

Transformación Simplificación:

La lógica de implementación es la misma en todas las clases, adaptada al caso específico de cada operador:

- Pedir a mis hijos que se simplifiquen.
- Si mis hijos son números, los opero y devuelvo el resultado.
- Si no son números:
  - Para operadores binarios miro si son iguales y de serlo los simplifico:
    - Suma: `a + a = 2a`
    - Multiplicación: `a * a = a^2`
    - Igualdad:
      - `a*b = c*b <> a = c`
      - `a + b = c + b <> a = c`
      - `op_unario(a) = op_unario(b) <> a = b`
  - Para operadores unarios verifico el inverso:
    - Logaritmo / Exponenciación: `log(2^a) = a || 2^log(a) = a`
    - Raíz / Cuadrado: `rz(a^2) = a || (rz(a))^2 = a`
- Si no puedo realizar ninguna simplificación, me devuelvo a mi mismo.

Para la entrada:

```
10 6 - 1 3 + / 2^ a 2 * 2^ log 2 a * log 2^ * rz 2^ =
```

El resultado obtenido es:

```
[0] 2^((10.0 + -6.0)*(1/1.0 + 3.0)) = 2^(rz(log(2^(a*2.0))*2^(log(2.0*a))))
= <Aritmética: a + b = c, a,b,c : R>
[1] 2^(4.0*(1/1.0 + 3.0)) = 2^(rz(log(2^(a*2.0))*2^(log(2.0*a))))
= <Aritmética: a + b = c, a,b,c : R>
[2] 2^(4.0*1/4.0) = 2^(rz(log(2^(a*2.0))*2^(log(2.0*a))))
= <Aritmética: a*b = c, a,b,c : R>
[3] 2^(1.0) = 2^(rz(log(2^(a*2.0))*2^(log(2.0*a))))
= <Aritmética: 2^n = m, n,m : R>
[4] 2.0 = 2^(rz(log(2^(a*2.0))*2^(log(2.0*a))))
= <Simplificación: log(2^u) = u>
[5] 2.0 = 2^(rz((a*2.0)*2^(log(2.0*a))))
= <Simplificación: 2^(log(u)) = u>
[6] 2.0 = 2^(rz((a*2.0)*(2.0*a)))
= <Simplificación: a*a = a^2, a : R>
[7] 2.0 = 2^(rz((a*2.0)^2))
= <Simplificación: rz(u^2) = u>
[8] 2.0 = 2^(a*2.0)
```

Transformación Atracción:

El objetivo de este proceso es pasar a la izquierda todas las apariciones de la incógnita en el lado derecho. Es invocado por la clase 'Igualdad' sobre su hijo derecho hasta recibir como resultado una contenedora vacía. La firma del método es

```
public Nodo atraccion(Stack<Nodo> contenedora)
```

Donde contenedora almacena la referencia al nodo que está siendo atraído y el método devuelve el Nodo a asignar en la referencia sobre la cual se realizó el llamado.

- **Parámetro:** Verifica si él es incógnita:
  - Si sí es incógnita, se agrega a la contenedora y devuelve null.
  - Si no es incógnita se devuelve a sí mismo, dejando la contenedora intacta.
- **Suma:** Invoca el método sobre su hijo izquierdo:
  - Verifica la contenedora:
    - Si la contenedora no regresa vacía transforma su contenido:

```
if(esNegativo()) contenedora.peek().cambiarSigno();
```

    - Si la contenedora regresa vacía invoca el método sobre su hijo derecho y repite el proceso.
  - Verifica la referencia:
    - Si la referencia es nula, se elimina a si mismo retornando su otro hijo transformado.
    - Si la referencia no es nula, la asigna y se retorna a si mismo.
- **Multiplicación:** Invoca el método sobre su hijo izquierdo:
  - Verifica la contenedora:
    - Si la contenedora no regresa vacia, crea una nueva multiplicación entre su hijo derecho y el contenido de la contenedora:
      - Transforma la multiplicación recién creada:

```
if(esDivisor()) contenedora.peek().invertirNodo();
```

      - La agrega a la contenedora.
    - Si la contenedora regresa vacia, invoca el método sobre su hijo derecho y repite el proceso.
  - Verifica la referencia:
    - Si la referencia es nula, se elimina retornando null (observe que en este caso el hijo derecho ya subió en la contenedora, por lo cual sería un error retornarlo).
    - Si la referencia no es nula, se retorna a si mismo.
- **Igualdad:** Crea un nueva contenedora e invoca el método sobre su hijo derecho:
  - Verifica la contenedora:
    - Si la contenedora no regresa vacía:
      - Crea una nueva suma colocando como hijos el contenido de la contenedora y su hijo derecho.
      - Asigna la suma recién creada a su hijo izquierdo.
      - Vuelve a invocar el método sobre su hijo derecho.
    - Si la contenedora regresa vacía suspende la ejecución del método.
- **OperadorUnario:** Mira si alguno de sus descendientes contiene la incógnita invocando la subrutina contieneIncognita():
  - Si sí la contienen, se adiciona a la contenedora y retorna null.
  - Si no la contienen se retorna a si mismo.

Para la entrada:

```
a x b * x c + 2^ b x / + + =
```

El resultado obtenido es:

```
[0] a = x*b + 2^(x + c) + b*1/x
= <Atracción: a = b + c <> a - c = b>
[1] a + -x*b = 2^(x + c) + b*1/x
= <Atracción: a = b + c <> a - c = b>
[2] a + -x*b + -2^(x + c) = b*1/x
= <Atracción: a = b + c <> a - c = b>
[3] a + -x*b + -2^(x + c) + -b*1/x = 0.0
```

Transformación Colección:

El objetivo de este proceso es agrupar variables a partir de reglas de factorización para la suma y la multiplicación, y propiedades de los logaritmos para logaritmos y exponenciaciones. La firma del método principal es:

```
public abstract Nodo coleccionar(Ecuacion ecuacion)
```

Donde ecuación es una referencia a la clase ecuación utilizada para registrar las transformaciones realizadas y devuelve el Nodo a asignar en la referencia sobre la cual se realizó el llamado.

Todas las clases que implementan este método se apoyan en la subrutina coleccionable(), cuya firma es:

```
public boolean coleccionable(Nodo candidato, Nodo[] aColeccionar,
                      Nodo[] coleccionado, Stack<Nodo> restantes)
```

En los operadores unarios el método principal solo consiste en la transmisión del llamado:

```
Nodo tempNodo = hijo.coleccionar(ecuacion);
```

Para los operadores binarios la lógica del método es la misma:

- Crear las contenedoras coleccionado, aColeccionar y restantes.
- Preguntarle al hijo izquierdo si él o alguno de sus descendientes es coleccionable con el hijo derecho:

```
if (izquierda.coleccionable(derecha, aColeccionar, coleccionado,  restantes))
```

- En caso de encontrar algún nodo coleccionable:
  - Realizo la colección[1].
  - Recupero los Nodos restantes.
  - Devuelvo el subárbol recién generado.

La recursión del algoritmo se encuentra en la subrutina coleccionable(), la lógica de esta rutina es la siguiente:

1 La forma de crear la colección depende tanto del tipo del nodo que se encuentra ejecutando el método como de los nodos a coleccionar. Por ejemplo una Suma que está tratando de coleccionar una Multiplicación retornará un nodo tipo Multiplicación donde uno de sus hijos es el nodo coleccionado y el otro es la suma de los nodos a coleccionar.

- Trato de realizar un Cast del nodoCandidato a mi propio tipo para garantizar que los tipos coinciden (una suma solo se puede coleccionar con otra suma, una multiplicación con otra multiplicación, etc):
  - Si el Cast tiene éxito:
    - Realizo la comparación de mis nodos hijos y los nodos hijos del nodoCandidato.
      - Si encuentro algún nodo coleccionable realizo las asignaciones de las variables pasadas como parámetro y retorno 'true'.
      - Si no encuentro ningún nodo coleccionable invoco el método coleccionable() sobre mis hijos.
  - Si el Cast falla, invoco el método coleccionable() sobre mis hijos.
    - Si alguno de mis hijos retorna 'true', agrego mi otro hijo a la pila de restantes y devuelvo 'true'.
    - Si ninguno de mis hijos retorna 'true', retorno 'false'.

Si bien la lógica general para todos los operadores es la misma, la implementación para cada clase debe realizarse con sumo cuidado, considerando detenidamente todas las combinaciones posibles. Hasta el momento se ha logrado la implementación exitosa del método coleccionar() para la clase Suma y coleccionable() para Multiplicación, lo cual permite obtener factorizaciones de sumas correctas.

Por ejemplo, para la entrada:

```
x a x * + x b / + a b * a c * x 3 * x 5 * + + + =
```

El resultado obtenido es:

```
[0] x + a*x + x*1/b = a*b + a*c + x*3.0 + x*5.0
= <Colección: (a*u) + (b*u) = (a + b)*u>
[1] x*(1.0 + a) + x*1/b = a*b + a*c + x*3.0 + x*5.0
= <Colección: (a*u) + (b*u) = (a + b)*u>
[2] x*(1.0 + a + 1/b) = a*b + a*c + x*3.0 + x*5.0
= <Colección: (a*u) + (b*u) = (a + b)*u>
[3] x*(1.0 + a + 1/b) = a*b + a*c + x*(3.0 + 5.0)
= <Aritmética: a + b = c, a,b,c : R>
[4] x*(1.0 + a + 1/b) = a*b + a*c + x*8.0
= <Atracción: a = b + c <> a - c = b>
[5] x*(1.0 + a + 1/b) + -x*8.0 = a*b + a*c
= <Colección: (a*u) + (b*u) = (a + b)*u>
[6] x*(1.0 + a + 1/b + -8.0) = a*b + a*c
= <Colección: (a*u) + (b*u) = (a + b)*u>
[7] x*(1.0 + a + 1/b + -8.0) = a*(b + c)
```

Transformación Aislamiento:

El objetivo de este proceso es pasar a la derecha todos los parámetros que no contienen a la incógnita. Es un proceso análogo a la transformación atracción, salvo porque los parámetros objetivo son aquellos que no contienen a la incógnita.

La lógica del algoritmo utilizado es la misma del proceso atracción salvo para las clases Parámetro, Multiplicación e Igualdad.

- Parámetro: Un parámetro se agrega a la contenedora en caso de NO ser la

incógnita.
- Multiplicación: Si se solicita una operación de aislamiento, la clase Multiplicación siempre se retorna a si misma, sin realizar el llamado sobre sus hijos.
- Igualdad: Durante un proceso de aislamiento, la igualdad debe reconocer el tipo de operador que representa su hijo izquierdo:
  - Si su hijo izquierdo es una suma el proceso a realizar es análogo al de la operación de atracción.
  - Si su hijo izquierdo es una multiplicación debe:
    - Identificar el nodo hijo de la multiplicación que no contienen a la incógnita.
    - Invertir dicho nodo y crear una nueva multiplicación entre él y su hijo derecho.
    - Asignar el nodo recién creado a su referencia derecha y el nodo restante a su referencia izquierda.
  - Repetir el proceso hasta que todos sus descendientes por la izquierda contengan a la incógnita.

Para la entrada:

```
a x * b + c x * d + =
```

El resultado obtenido es:

```
[0] a*x + b = c*x + d
= <Atracción: a = b + c <> a - c = b>
[1] a*x + b + -c*x = d
= <Colección: (a*u) + (b*u) = (a + b)*u>
[2] b + x*(a + -c) = d
= <Aislamiento: a + b = c <> a = c - b>
[3] x*(a + -c) = d + -b
= <Aislamiento: a*b = c <> a = c/b>
[4] x = (d + -b)*(1/(a + -c))
```

Reconocimiento de Casos Especiales

En este momento el sistema no cuenta con ningún tipo de algoritmo heurístico y basa toda su operación en el método de resolución *descentralizado* mencionado anteriormente. Actualmente el método despejarEcuacion() de la clase 'Ecuacion' simplemente llama los métodos simplificar(), coleccionar(), atraccion() y aislamiento() de forma secuencial sobre la raíz del árbol sintáctico.

Los métodos heurísticos para el reconocimiento y procesamiento de casos especiales debían implementarse en la clase 'Igualdad', ya que ésta representa la raíz del árbol sintáctico y como tal puede analizar sus descendientes.

Algunos casos especiales a reconocer, después de finalizar las operaciones de atracción y aislamiento, podrían ser:

- Reconocimiento de expresiones cuadráticas:
  Si a la izquierda se encuentra una suma entre la incógnita y la incógnita al cuadrado, aplicar la fórmula de raíz cuadrada.
- Reconocimiento de expresiones factorizadas:
  Si a la izquierda se encuentra el producto dos sumas que contienen a la incógnita y a la derecha se encuentra el número cero, deducir los posibles

valores de la incógnita.

- Reglas de inferencia para operadores unarios:
  Actualmente el proceso de aislamiento detiene su ejecución al encontrar cualquier operador unario en su hijo izquierdo. Resulta relativamente simple programar las reglas de inferencia para los inversos de los operadores unarios:
  - Aplicar el inverso del operador detectado sobre los dos Nodos hijos
  - Llamar el proceso de simplificacion() sobre el hijo izquierdo.

Comentarios Finales

Hasta este punto, la aplicación es capaz de resolver de forma relativamente satisfactoria aquellas ecuaciones lineales simples de tamaño arbitrario que no contienen incógnitas factorizadas. Para las ecuaciones que incluyen otro tipo de operadores, el programa es capaz de transformar la ecuación dejando todos los operadores con incógnitas a la izquierda y los demás a la derecha. Adicionalmente es capaz de realizar algunas simplificaciones evidentes sobre todos los operadores.

Se hizo un gran énfasis en lograr que el sistema no realice transformaciones matemáticamente incorrectas. Actualmente todo operador que no encuentra una forma correcta de operar a su hijos se retorna a sí mismo, dejando el árbol sintáctico de la ecuación inalterado. Finalmente se ha garantizado la finalización de todo proceso a través de un diseño detallado de los algoritmos recursivos y el manejo de excepciones.

Se sospecha que los procedimientos y el modo de operación diseñado constituyen una base suficiente para la implementación del sistema completo propuesto como ejercicio. Resta realizar la implementación de los métodos coleccion() y coleccionable() para los operadores unarios y los algoritmos de reconocimiento en la clase 'Igualdad'. También puede ser necesaria la implementación de algunos algoritmos heurísticos que operen localmente, para determinar por ejemplo si ciertos subárboles se deben expandir o contraer[2].

La mayoría de los algoritmos implementados se encuentran totalmente documentados en su cuerpo, sin embargo la documentación tipo API con precondiciones, parámetros de entrada, salida y poscondiciones no se encuentra finalizada. La aplicación en total contiene 593 instrucciones[3].

2 Un ejemplo de esto para los operadores binarios sería determinar si es conveniente realizar una distribución */+ o +/* para extraer un aparición de la incógnita.

3 Medida obtenida contando el número de ';' en el código fuente.