Trabajo práctico N ° 3. Árboles binarios.

Algoritmos y Programación II. Cátedra Carolo

23 de mayo de 2009

1. Objetivos

El objetivo de este trabajo práctico es que el alumno adquiera destreza en el manejo de árboles binarios, al tiempo que se presenta una de sus más interesantes aplicaciones.

Se deberá implementar un Tipo de Dato Abstracto (TDA) que permita la manipulación de expresiones algebraicas. Básicamente, el TDA debe permitir cargar una expresión en una estructura en memoria, obtener la expresión de su derivada, simplificarla y devolverla en un string.

2. Conceptos y algoritmos

2.1. Notación infija y postfija

Existen distintas formas de escribir una misma expresión matemática. La **notación** más común (pues se utiliza en la mayoría de los lenguajes de programación y al hacer cuentas en papel) es la **notación infija**. Por ejemplo:

$$x^2 - 3x + \sin(\pi * x)$$

En este caso, toda expresión puede reducirse a una forma genérica del tipo

$$[\textit{operando izq}]\ \textit{operador}\ [\textit{operando der}]$$

donde *operando izq* y *operando der* son expresiones. Para el caso de operadores unarios (los que reciben un solo operando, por ejemplo *sin*, *log*, etc) uno de los dos operandos debe omitirse. Para los fines de este trabajo práctico, supondremos que los operadores unarios poseen **solo** operando derecho.

Esta notación es muy cómoda a la hora de ser leída por un humano, pero el algoritmo para evaluarla con una computadora es ciertamente complejo. Por esta razón, se utiliza la **notación polaca**, en sus formas **prefija** y **postfija**. El ejemplo anterior expresado en estas notaciones es:

Notación	Ejemplo
polaca prefija	$+\ -\ \hat{}\ x\ 2\ *\ 3\ x\ \sin\ *\ \pi\ x$
polaca postfija	$x\ 2\ \hat{}\ 3\ x\ *\ -\ \pi\ x\ *\ \sin\ +$

En la notación prefija se escribe primero el operador y a continuación los dos operandos, mientras que en la notación postfija se escriben primero los dos operandos y luego el operador.

La ventaja de este tipo de notaciones, es que no es necesario utilizar paréntesis para marcar la precedencia de las operaciones, pues los operandos se encuentran **siempre** próximos al operador. En este trabajo práctico vamos a ocuparnos solamente de la notación **postfija**.

Para evaluar expresiones en notación postfija, se recorre la misma de izquierda a derecha guardando los operandos en una pila hasta encontrar el primer operador. Luego, se recuperan los operandos necesarios de la pila (uno o dos según si el operador es binario o unario), se ejecuta la operación y se guarda el resultado en la pila. Se repite este proceso hasta alcanzar el final de la expresión, punto en el cual la pila contendrá el resultado de la evaluación.

La figura 1 muestra los pasos realizados para evaluar la expresión del ejemplo anterior.

2.2. Árboles de expresión

Para representar expresiones resultan muy convenientes los llamados **árboles de expresión**. Se trata de un árbol binario en el cual cada uno de sus nodos internos es un operador, y cada uno de sus nodos hoja es un operando. Por ejemplo, la expresión anterior puede representarse en forma de árbol de expresión como muestra la figura 2.

Esta estructura de datos será el corazón del presente trabajo práctico pues facilitará enormemente

Pila	Expresión	Operación
(vacía)	1 2ˆ3 1 $*$ $-$ π 1 $*$ sin $+$	Comienzo
1, 2	ˆ3 1 $*$ $-$ π 1 $*$ sin $+$	Leídos 1 y 2
1	3 1 $*$ $-$ π 1 $*$ sin $+$	Evaluado 1^2
1, 3, 1	$*$ $-$ π 1 $*$ sin $+$	Leídos 3 y 1
1, 3	$-$ π 1 $*$ sin $+$	Evaluado $3*1$
-2	π 1 $*$ sin $+$	Evaluado $1-3$
-2, π, 1	$*$ sin $+$	Leídos π y 1
-2, π	sin $+$	Evaluado $\pi*1$
-2, 0	$+$	Evaluado $\sin(\pi)$
-2	(vacía)	Evaluado $(-2)+0$

Figura 1: Evaluación de x 2ˆ3 x $*$ $-$ π x $*$ sin $+$ (notación polaca inversa) para $x = 1$.

la tarea de manipular la expresión en memoria, sobre todo teniendo en cuenta la definición recursiva de expresión vista en 2.1.

2.3. Evaluación

Evaluar una expresión representada de esta forma es tan fácil como hacer un recorrido post-orden del árbol. Por otro lado, para escribir la expresión en cualquiera de las notaciones vistas en la sección 2.1 solo hay que hacer un recorrido en orden, pre-orden o post-orden según el tipo de notación que se quiera obtener en la salida.

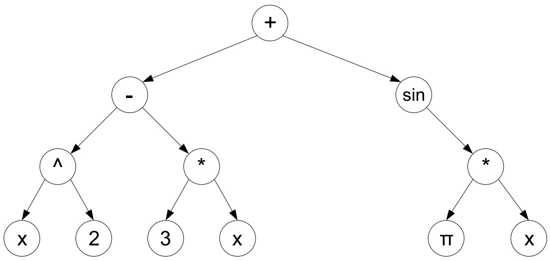


Figura 2: Árbol de expresión de $x^2 - 3*x + \sin(\pi * x)$

2.4. Derivación

Dado un árbol de expresión como el de la figura 2, obtener un nuevo árbol con la expresión de su derivada es sumamente sencillo. Debe inspeccionarse el nodo raíz y, según el operador que sea, generar el nuevo árbol.

Por ejemplo, en la figura 2, el nodo raíz es un operador de suma. Esto nos indica que en el nuevo árbol debe ir como nodo raíz también un operador de suma (pues la derivada de una suma es la suma de las derivadas) y en sus subárboles izquierdo y derecho deben ir las derivadas de los subárboles izquierdo y derecho (respectivamente) del árbol original. Como puede notarse de la descripción, este algoritmo es recursivo, y su condición de corte es cuando se llega a alguna de las hojas del árbol, pues las derivadas de estos nodos son constantes.

Para aclarar las ideas, vamos a ver cómo derivar la expresión anterior paso a paso. En primer lugar, vamos a nombrar cada nodo como si fuera una expresión distinta (ver figura 3).

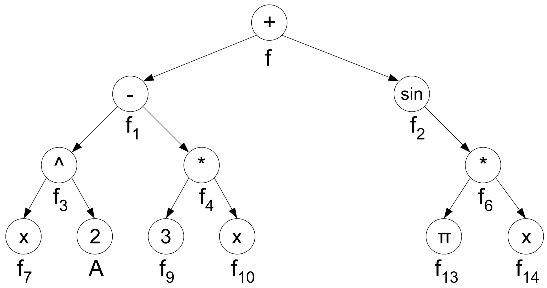


Figura 3: Nombres asignados a los distintos nodos del árbol de expresión.

Ahora, teniendo en cuenta que la raíz del árbol es un operador de suma, podemos empezar a construir el árbol de la derivada, cuya raíz será también un operador de suma, y sus subárboles serán las derivadas de f_1 y f_2, como se ve en la figura 4(a).

A continuación vamos a derivar la expresión de f_1. Como es una resta, el procedimiento es muy similar al de la suma, lo que puede verse en la figura 4(b).

La derivación de f_3 es una de las más complejas del ejemplo. Para este TP vamos a suponer que la potenciación tiene como segundo operando una constante (en este caso llamada A). En la figura 4(c) vemos cómo debe quedar el árbol de la derivada para esta expresión, que en este caso incluye un subárbol derivado, y luego el mismo sin derivar.

En la figura 4(d) se ve cómo debe derivarse la expresión f_4, cuyo operador principal es un producto. Nuevamente vuelven a aparecer subárboles derivados y otros que se conservan iguales. En el caso de la figura 4(f), que corresponde a la expresión de f_6, el procedimiento es muy similar.

Por último, tenemos la derivación de la expresión f_2, que está constituída por un operador unario sin, y que puede verse en la figura 4(e).

Para el caso de las expresiones f_7, f_9, f_{10}, f_{13} y f_{14}, sus derivadas son constantes que pueden valer cero o uno dependiendo de si el nodo en cuestión es una constante o la variable, respectivamente.

Juntando todos estos resultados parciales se obtiene el árbol de la expresión derivada, que puede apreciarse en la figura 5.

2.5. Simplificación

El objetivo de la simplificación es eliminar de la expresión términos superfluos (e.g: $3 * x * \sin(0)$).

El algoritmo es sumamente sencillo, y consta en recorrer el árbol en forma post-fija eliminando los subárboles que cumplan ciertas condiciones (por ejemplo, raíz '*' y uno de sus hijos '0').

En la figura 6 podemos ver los pasos seguidos para simplificar la expresión de la figura 5, que finalmente queda como muestra la figura 7.

3. Desarrollo

3.1. Árbol binario "Cut & Paste" (ABC&P)

Como primer punto en el desarrollo del trabajo práctico, se pide la implementación de un árbol binario "Cut & Paste" [1]. Este consiste en un árbol binario convencional con dos primitivas adicionales que se describen a continuación:

3.1.1. AB_CopiarSubarbol

```
int AB_CopiarSubarbol(const TAB* origen,
                      TAB* destino, int movim)
```

Toma el subárbol de `origen` que comienza en el hijo del corriente indicado por `movim` y lo copia a `destino`. `Origen` debe ser un árbol creado y no vacío, y `destino` debe ser un árbol no creado. `Movim` acepta `RAIZ`, `IZQ` y `DER`. Devuelve 1 si pudo realizar la copia, 0 en caso contrario.

[1] La denominación que se le ha dado a este árbol fue creada ad-hoc por la cátedra.

3.1.2. AB_PegarSubarbol

```
int AB_PegarSubarbol(const TAB* origen,
                     TAB* destino, int movim)
```

Toma el árbol `origen` y lo copia al hijo del corriente de `destino` indicado por `movim`. La variable `origen` debe ser un árbol creado, y `destino` debe ser un árbol creado y con el corriente en un nodo hoja. `Movim` acepta `IZQ` y `DER`. Devuelve 1 si pudo realizar la copia, 0 en caso contrario.

3.2. TDA Literal

A excepción de las funciones de simplificación y derivación, este TDA será provisto por la cátedra y abstraerá al alumno de la complicación que surge a la hora de parsear expresiones matemáticas. El mecanismo utilizado para distribuir esta biblioteca será brindarle a los alumnos un archivo ".lib" con el código fuente compilado y los archivos ".h" con las declaraciones de las funciones, estructuras y enumerados.

Es importante que los alumnos no asuman ningún tipo de hipótesis que surga de inspeccionar los archivos ".h", sino que para utilizar el TDA solo se basen en la información provista en este enunciado.

3.2.1. Literal_Crear

```
int Literal_Crear(TLiteral* lit,
                  const char* string,
                  const char* var)
```

Obtiene el próximo literal de `string` y lo guarda en `lit`, asumiendo al parsear que `var` es la variable independiente. Devuelve la longitud del literal en carácteres. Si no se encontró ningún literal devuelve 0. Si se encontró un operador desconocido devuelve -1.

3.2.2. Literal_Derivar

```
int Literal_Derivar(const TLiteral* lit,
                    const TAB* entrada,
                    TAB* salida)
```

Sean `entrada` y `salida` dos árboles de expresión que contienen objetos de tipo `TLiteral`, deriva la expresión `entrada` a partir del corriente, guardando el resultado en `salida`. `Lit` debe ser un literal creado, `entrada` debe ser un árbol creado y no vacío y `salida` debe ser un árbol creado y vacío. Devuelve 1 si pudo derivar la expresión, 0 en cualquier otro caso.

3.2.3. Literal_Simplificar

```
int Literal_Simplificar(const TLiteral* lit,
                        const TAB* entrada,
                        TAB* salida)
```

Sean `entrada` y `salida` dos árboles de expresión que contienen objetos de tipo `TLiteral`, simplifica la expresión `entrada` a partir del corriente, guardando el resultado en `salida`. `Lit` debe ser un literal creado, `entrada` debe ser un árbol creado y no vacío y `salida` debe ser un árbol creado y vacío. Devuelve 1 si pudo simplificar la expresión, 0 en cualquier otro caso.

3.2.4. Literal_EsOperador

```
int Literal_EsOperador(const TLiteral* lit)
```

Devuelve 0 si el literal no es un operador (es una constante o una variable), 1 si es un operador unario y 2 si es un operador binario.

3.2.5. Literal_AString

```
int Literal_AString(const TLiteral* lit,
                    char** salida)
```

Copia a `string` una representación del literal `lit` en notación polaca inversa. `String` debe apuntar a `NULL` y `lit` debe apuntar a un literal creado. Devuelve 1 si pudo crear la representación, 0 en cualquier otro caso.

3.2.6. Literal_Destruir

```
void Literal_Destruir(TLiteral* lit)
```

Destruye `lit`, que debe apuntar a un `TLiteral` creado.

3.2.7. infijo_a_sufijo

```
int infijo_a_sufijo(const char* infijo,
                    const char* var,
                    TPila* sufijo)
```

Función auxiliar. Sea `infijo` una expresión algebráica en notación infija, con variable independiente `var`, entonces guarda en `sufijo` una pila de objetos `TLiteral` que representa la misma expresión en formato sufijo. Los literales son sacados de la pila como si la expresión en notación sufija fuera leída de derecha a izquierda. Devuelve 1 si pudo parsear la expresión, 0 en caso contrario.

3.3. Funciones de derivación y simplificación

Se proveerá a los alumnos de un archivo ".h" con las declaraciones de las funciones utilizadas para derivar y simplificar cada literal. Esto es, al llamar a `Literal_Derivar` o `Literal_Simplificar`, la biblioteca se encarga de redireccionar la invocación a la función de derivación o simplificación correspondiente, según el tipo de literal del que se trate.

Estas funciones serán implementadas por los alumnos y es donde se debe realizar la manipulación de los árboles de expresión utilizando las capacidades del árbol ABC&P.

3.4. TDA Expresión

Este TDA es el centro del trabajo práctico y deberá ser implementado por los alumnos utilizando como herramienta el TDA Literal y el árbol ABC&P. El diseño de la estructura del TDA queda a cargo de los alumnos, con la salvedad de que se obliga a que la expresión sea guardada en memoria en un árbol ABC&P.

3.4.1. Expresion_Crear

```
int Expresion_Crear(TExpresion* expr)
```

Crea una expresión vacía. `Expr` debe apuntar a una variable de tipo `TExpresion` sin crear. Devuelve 1 si pudo crear la expresión, 0 en cualquier otro caso.

3.4.2. Expresion_Parsear

```
int Expresion_Parsear(TExpresion* expr,
                      const char* string,
                      const char* var)
```

Parsea la expresión dada en `string` y la guarda en `expr`. Asume que `var` es la variable independiente. `Expr` debe ser una variable de tipo `TExpresion` creada, `string` debe ser una cadena de carácteres no vacía y `var` debe ser una cadena de carácteres de longitud 1 o vacía. Devuelve 1 si pudo parsear la expresión, -1 si hubo error de sintaxis y -2 en cualquier otro caso.

Nota: para implementar esta primitiva debe utilizarse la función `infijo_a_sufijo` del TDA Literal.

3.4.3. Expresion_Derivar

```
int Expresion_Derivar(const TExpresion* expr,
                      TExpresion* deriv)
```

Guarda en `deriv` la derivada de `expr`. `Expr` debe apuntar a una expresión creada y con al menos un string parseado (no vacía) y `deriv` debe apuntar a una variable de tipo `TExpresion` vacía. Devuelve 1 si pudo derivar la expresión, 0 en cualquier otro caso.

3.4.4. Expresion_Simplificar

```
int Expresion_Simplificar(const TExpresion* expr,
                          TExpresion* simpl)
```

Guarda en `simpl` una copia simplificada de `expr`. `Expr` debe apuntar a una expresión creada y con al menos un string parseado (no vacía) y `simpl` debe apuntar a una variable de tipo `TExpresion` vacía. Devuelve 1 si pudo simplificar la expresión, 0 en cualquier otro caso.

3.4.5. Expresion_AString

```
int Expresion_AString(const TExpresion* expr,
                      char** string)
```

Devuelve en `string` un puntero a una cadena de carácteres con la expresión `expr` en formato infijo. `Expr` debe ser una expresión creada y con al menos un string parseado, y `string` debe apuntar a NULL. Devuelve 1 si pudo convertir la expresión, 0 en cualquier otro caso.

Nota: Deberá utilizarse la función `realloc` de la biblioteca standard de C para redimensionar la cadena `string`.

3.4.6. Expresion_Destruir

```
void Expresion_Destruir(TExpresion* expr)
```

Destruye `expr`, que debe ser una expresión creada.

3.5. Operadores soportados por el TDA Expresión

- Suma (+)
- Resta (-)
- Producto (*)
- División (/)
- Potenciación por una constante (ˆ)
- Seno (sin)
- Coseno (cos)
- Logaritmo natural (ln)

3.6. Reglas de simplificación

La siguiente tabla define las operaciones de simplificación que deberán ser utilizadas. En los primeros tres casos –en que las operaciones son conmutativas– deben ser validadas ambas conmutaciones. Se define *exp* como una expresión –un valor simple o bien una combinacion de valores– y A como una constante numérica (no representada por letras).

Expresión	→ Simpl.	Condición
$exp + 0$	$\rightarrow exp$	
$exp - 0$	$\rightarrow exp$	
$exp * 0$	$\rightarrow 0$	
$exp * 1$	$\rightarrow exp$	
$exp/1$	$\rightarrow exp$	
A/A	$\rightarrow 1$	A cte $\wedge A \neq 0$
$0\hat{}A$	$\rightarrow 0$	$A \notin \{-1, 0\}$
$exp\hat{}0$	$\rightarrow 1$	$exp \neq 0$
$exp\hat{}1$	$\rightarrow exp$	
$\sin(0)$	$\rightarrow 0$	
$\cos(0)$	$\rightarrow 1$	
$\ln(1)$	$\rightarrow 0$	

3.7. Aplicación

Deberá desarrollarse una aplicación que utilice el TDA Expresión descripto más arriba. El objetivo de la aplicación es que, dadas una o varias expresiones algebráicas, el programa debe derivarlas y simplificarlas. Si se quiere derivar una única expresión, la sintaxis será:

```
$ tp3 --single <expresión>
```

expresión es la expresión a derivar.

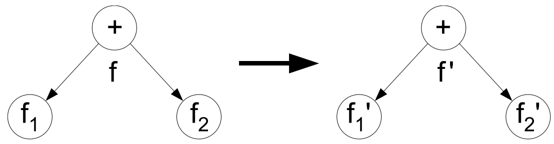
Si se quieren derivar varias expresiones, la sintaxis será:

```
$ tp3 --file <input><output><log>
```

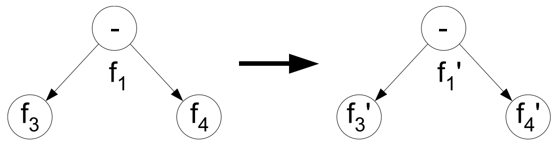
intput es el archivo del cual el programa leerá, línea por línea, las expresiones a derivar

output es el archivo al cual el programa escribirá la derivada simplificada de cada una de las expresiones incluídas en `input`

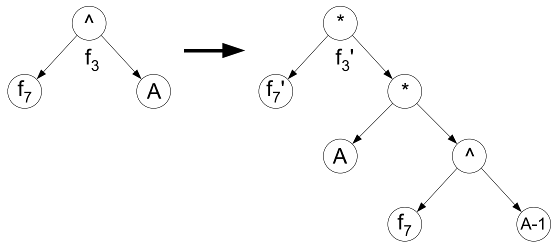
log es el archivo en el cual se escribirá cualquier mensaje de error que ocurra durante la ejecución



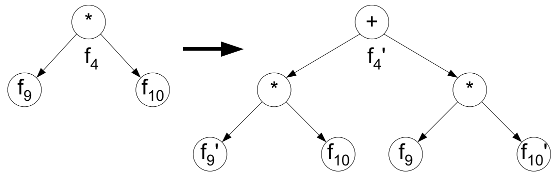
(a) Derivación de la suma



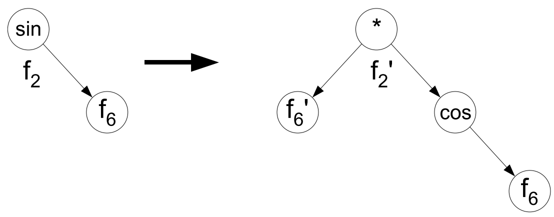
(b) Derivación de la resta



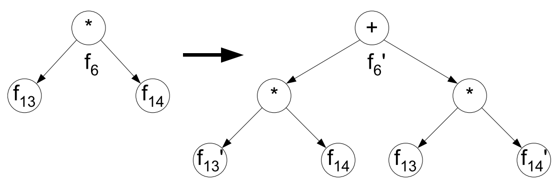
(c) Derivación de la potencia



(d) Derivación del producto



(e) Derivación del seno



(f) Derivación del producto

Figura 4: Pasos seguidos para obtener la derivada de la expresión de la imagen 3

4. Entrega

Si existiera alguna ambigüedad en el enunciado, los alumnos deberán tomar una determinación al respecto y hacerla explícita en el informe entregado junto al código fuente. La desición tomada no debe facilitar el enunciado considerablemente, y si eso sucediera, el alumno deberá consultar con su ayudante.

Se hará una pre-entrega del TP el día 26 de mayo de 2009. La fecha de entrega final es el 16 de junio de 2009.

Referencias

[1] Robert Leroy Kruse y Efrén Alatorre Miguel, "Estructura de datos y diseño de programas", Prentice-Hall Hispanoamericana, c1988

[2] Brian W. Kernighan, Dennis M. Ritchie, Néstor Gómez Muñoz, "El lenguaje de programación C", Prentice-Hall Hispanoamericana, c1991

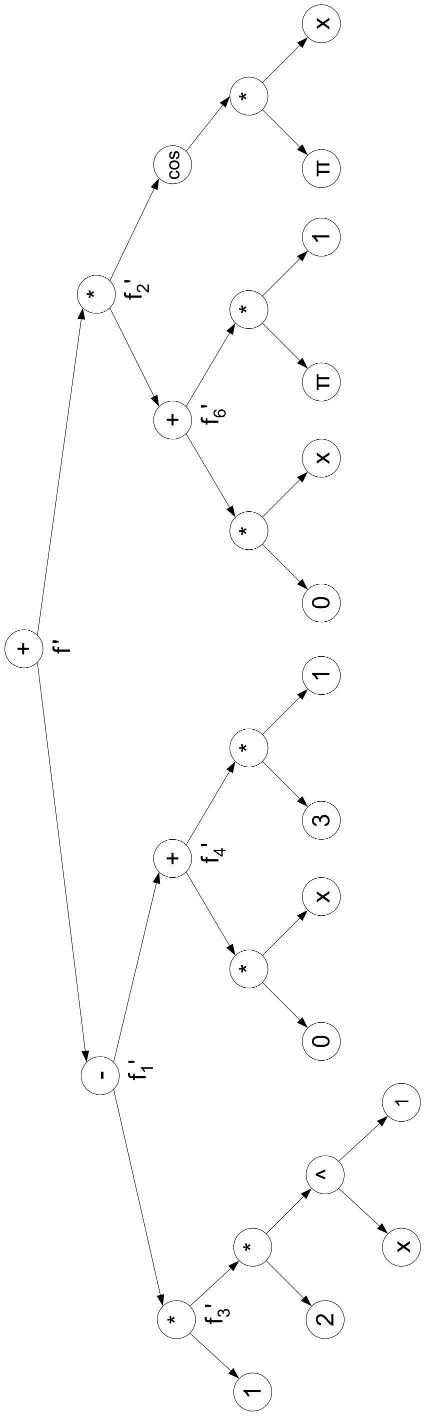


Figura 5: Árbol de expresión de $1*2*x^1-(0*x+3*1)+(0*x+\pi*1)*\cos(\pi*x)$, que es la derivada de $x^2-3x+\sin(\pi*x)$.

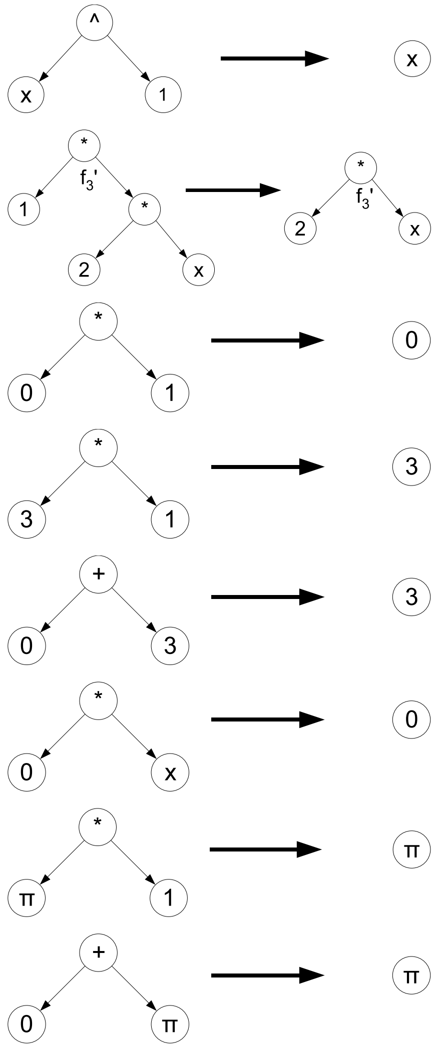


Figura 6: Pasos seguidos para simplificar la expresión de la imagen 5

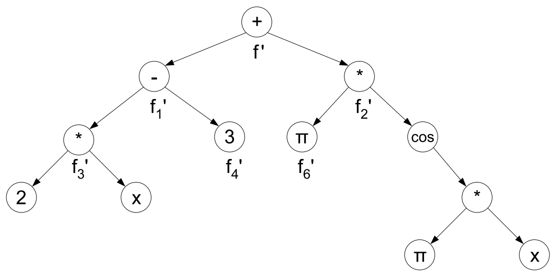


Figura 7: Árbol de expresión de $1*2*x^1-(0*x+3*1)+(0*x+\pi*1)*\cos(\pi*x)$ simplificado.