Instituto Tecnológico de Costa Rica

Escuela de computación

Profesor

Ignacio Trejos Zelaya

Lenguajes de programación

Derivadores simbólicos - programas escritos en el paradigma lógico-relacional

Integrantes

Kelvin Núñez Barrantes 2019046494

Kenny Vega Obando 2019162050

Fecha: 20/06/2022

Introducción

Hay problemas de la vida real que pueden llegar a ser muy complejos, pero sí se logra que estos problemas se puedan representar mediante una expresión simbólica esto puede resultar de gran valor, ya que se podría intentar procesarlos utilizando procesos computacionales, y esto sería de gran ventaja ya que se facilita su resolución.

En el siguiente informe se va a una explicación de dos programas de diferenciación simbólica en Prolog utilizando el compilador SWI-Prolog , uno escrito por Jocelyn Paine y otro por Woljtek Jurczyk. Los programas intentarán procesar una expresión simbólica dada y derivarla. Se explicará la solución y la estructura que ambos utilizan y se hará una comparación utilizando los ocho mismos ejemplos para ambos programas, esto con el fin de ver así los fuertes de cada uno de estos y si son capaces de resolverlos.

Por último se darán las conclusiones y los resultados obtenidos basadas en la comparación de los ocho ejemplos para así saber si los programas cumplen con su objetivo. Además, se hará una reflexión sobre las ventajas encontradas en la investigación sobre la utilización de la programación lógica y de los programas en Prolog.

Solución 1: Symdiff float by Jocelyn Paine

- **Explicación general de la solución: capacidades observadas del programa (tipo de expresiones que puede diferenciar simbólicamente), la estrategia de solución (según la interpretación que hagan ustedes) y la estructura del programa.**

La solución se basa en el backtracking. Recibe una expresión simbólica y la va descomponiendo recursivamente basadas en las reglas algebraicas y de derivación planteadas. Luego de derivar la expresión utilizando la función *d,* con el resultado de la función se va a llamar a la función *Simp* para simplificar y así poder omitir obviedades algebraicas tales como multiplicaciones o sumas por cero o realizar operaciones como sumas, restas, divisiones o multiplicaciones.

Para la estructura del programa se empieza declarando la función principal *symdiff* y es la que se utiliza para llamar al programa. Esta solo imprime el nombre del programa y la versión que se utiliza. Luego se pide la expresión a derivar y se escribe la variable respecto a la que se va a derivar o si se quiere salir de la aplicación se escribe "stop" y se concluye el programa.

A continuación se encuentra la función *process* que es la que procesa la expresión por la función de derivación *d* y luego por la función de simplificación *Simp* y por último imprime la respuesta. Por último están declaradas las cláusulas que hacen referencia a las reglas de derivación y luego están los procesos que se utilizan para simplificar las funciones

- **Explicación de la forma que deben tomar las expresiones simbólicas, o términos, de manera que el programa sea capaz de procesarlas (suministradas como archivos de entrada o parámetros [términos] para las cláusulas que hacen el procesamiento).**

Para la forma de las expresiones es muy importante siempre tomar en cuenta los paréntesis porque puede ser que si la expresión es muy larga genere confusión. Ejemplo de una expresión simbólica utilizada por el programa: **12*x^4+7*x^3+9*x^2+1000**.

Para representar las sumas se utiliza "+", para las restas se utiliza "**-**", para las multiplicaciones se va a utilizar el símbolo "*", para las divisiones se va a utilizar el "/" y para poner exponentes en expresiones algebraicas se utiliza el símbolo (^).

- **Explicación de cada parte del programa. Si hay uso del 'cut' (!), explique por qué es pertinente en cada lugar donde se lo usa.**

Reglas de derivación

Las cláusulas usadas en el programa se refieren a las reglas de derivación.

```
d( X, X, 1 ):- !.
```

En la primera se hace uso del "cut" para DETENER la ejecución del programa porque esta responde a que la derivada de una variable con respecto a sí misma es igual a 1 por lo que el uso del "!" asegura que el programa no intente encontrar una solución alterna.

```
d( C, X, 0 ):- atomic(C).
```

La siguiente regla hace referencia a que la derivada de una constante es cero. Acá se utiliza la función *atomic* para así ver que el dato ingresado es un átomo y no un compuesto y dar como resultado 0.

```
d( U+V, X, A+B ):-
    d( U, X, A ),
    d( V, X, B ).
```

```
d( U-V, X, A-B ):-
    d( U, X, A ),
    d( V, X, B ).
```

Luego, estas dos reglas hacen referencia a la suma y resta de derivadas. Se deriva cada expresión por aparte utilizando backtracking para así dar la expresión simple derivada y luego se suman.

```
d( C*U, X, C*A ):-
    atomic(C),
    C \= X,
    d( U, X, A ), !.
```

Esta regla hace referencia a cuando hay una constante multiplicando a una variable, Se consigue el átomo de la constante, se revisa que sea diferente a la variable, se deriva la variable y luego se multiplica el resultado de la derivación con la constante. Acá se utiliza el "!" para detener el programa, que no siga derivando la variable y que proceda a realizar la multiplicación.

```
d( U*V, X, B*U+A*V ):-
    d( U, X, A ),
    d( V, X, B ).
```

```
d( U/V, X, (A*V-B*U)/(V*V) )
    d( U, X, A),
    d( V, X, B).
```

Estas reglas se encargan de descomponer las expresiones que va encontrando utilizando las reglas de derivación de la multiplicación y de la división. Al llamar la función *d* con U y V se cumple que los resultados de cada una, siendo A y B sean acomodados como se requiere en las reglas.

```
d( U^C, X, C*A*U^(C-1) ):-
    atomic(C),
    C\=X,
    d( U, X, A ).
```

Esta regla hace referencia a cómo derivar una expresión con un exponente pero que este exponente sea una constante. Se llama a atomic y se revisa que la constante no sea igual a la variable respecto a la que se está derivando y se vuelve a llamar a *d* con el resto de la expresión. Al derivarse se multiplica la constante por la variable y se le resta 1 al exponente de la variable.

```
d( U^C, X, C*A*U^(C-1) ):-
    C = -(C1), atomic(C1),
    C1\=X,
    d( U, X, A ).
```

Esta regla es muy similar a la descrita anteriormente pero es cuándo la constante que se encuentra en exponente es negativa y acá hace la verificación de que esta constante sea diferente a la variable respecto a la que se está derivando. Luego continúan los pasos de la regla anterior.

```
d( exp(W), X, Z*exp(W) ):-
   d( W, X, Z).
```

La siguiente regla hace referencia a cómo derivar una expresión con un exponente pero que este exponente también es una expresión.. Esta manda a la función *d, lo* que se está utilizando como exponente y cuando lo obtiene lo multiplica por lo que haya enfrente de esta.

```
d( sin(W), X, Z*cos(W) ):-
   d( W, X, Z).
```

```
d( cos(W), X, -(Z*sin(W)) ):-
   d( W, X, Z).
```

Estas reglas hacen referencia a cómo derivar una expresión que contenga reglas trigonométricas como SIN(x) o COS(x). Esta dice que la derivada de seno es coseno y la derivada de coseno es -seno por lo que simplemente utilizando backtracking, descompone la expresión y multiplica *z* por el resultado de cada una.

```
d( log(W), X, Z/W ):-
   d( W, X, Z).
```

La última regla de derivación hace referencia a cómo derivar una expresión logarítmica. Esta manda a la función *d,* el argumento de la expresión y luego lo que retorna, lo pone como denominador en una división.

Reglas para la simplificación

```
simp( X, X ):-
   atomic(X), !.
```

Esta regla es para cuando se encuentra una variable y no se puede simplificar por lo que este es el resultado. Se utiliza “Cut” para detener el proceso y retornar la variable.

```
simp( X+0, Y ):-
    simp( X, Y ).

simp( 0+X, Y ):-
    simp( X, Y ).

simp( X-0, Y ):-
    simp( X, Y ).

simp( 0-X, -(Y) ):-
    simp( X, Y ).

simp( X*0, 0 ).

simp( 0*X, 0 ).

simp( 0/X, 0 ).
```

Las siguientes reglas hacen referencia a las expresiones en conjunto con el cero. La suma y resta dependiendo dónde se encuentra el cero, la división, la multiplicación.

```
simp( A+B, C ):-
    numeric(A),
    numeric(B),
    C is A+B.

simp( A-A, 0 ).

simp( A-B, C ):-
    numeric(A),
    numeric(B),
    C is A-B.
```

Luego se dan las sumas y restas de expresiones.

```
simp( X*1, X ).

simp( 1*X, X ).

simp( X/1, X ).

simp( X/X, 1 ) :- !.

simp( X^1, X ) :- !.

simp( X^0, 1 ) :- !.

simp( X*X, X^2 ) :- !.
```

Las siguientes reglas hacen su función respecto al uno. Al multiplicarse por 1 da lo mismo y al multiplicarse por sí mismo da su identidad elevada al cuadrado. Con la división si se divide por uno da la misma expresión y si se divide por sí mismo da uno. Se puede notar que las últimas cuatro reglas utilizan el "cut" ya que no se necesita que el procedimiento siga.

```
simp( A*B, X ) :-        /*
   numeric( A ),
   numeric( B ),
   X is A*B.

simp( A*X+B*X, Z ):-     /*
   A\=X, B\=X,           /*
   simp( (A+B)*X, Z ).

simp( (A+B)*(A-B), X^2-Y^2
   simp( A, X ),
   simp( B, Y ).

simp( X^A/X^B, X^C ):-
   numeric(A), numeric(B),
   C is A-B.
```

La primera regla hace referencia a la multiplicación de dos números, la segunda se basa en la factorización y se vuelve a llamar para así tratar de descomponer la expresión. La tercera hace referencia a la simplificación de una expresión basada en la diferencia de cuadrados y la cuarta hace referencia a la división de una misma variable en el numerador y en el denominador

```
simp( A/B, X ) :-
    numeric( A ),
    numeric( B ),
    X is A/B.

simp( A^B, X ) :-
    numeric( A ),
    numeric( B ),
    X is A^B.
```

Estas dos reglas hacen la evaluación si ambos A y B son números, la primera realiza la división y la segunda eleva A a B.

```
simp( W+X, Q ):-
    simp( W, Y ),
    simp( X, Z ),
    ( W \== Y ; X \== Z )
    simp( Y+Z, Q ).

simp( W-X, Q ):-
    simp( W, Y ),
    simp( X, Z ),
    ( W \== Y ; X \== Z )
    simp( Y-Z, Q ).

simp( W*X, Q ):-
    simp( W, Y ),
    simp( X, Z ),
    ( W \== Y  ; X \== Z
    simp( Y*Z, Q ).

simp( A/B, C ) :-
    simp( A, X ),
    simp( B, Y ),
    ( A \== X ; B \== Y )
    simp( X/Y, C ).

simp( X^A, C ) :-
    simp( A, B ),
    A \== B,
    simp( X^B, C).
```

Estas últimas revisa que todas las posibles maneras de simplificar se hayan considerado.

- **Ejemplos del procesamiento que realiza el programa sobre al menos 8 funciones no triviales que ejerciten todas las capacidades del programa. Al menos una de esas funciones debe tener más de una variable y ustedes deben determinar si el programa es capaz de obtener (o no) la derivada respecto de cada una de las variables.**

Ej 1

```
?- symdiff.

SYMDIFF : Symbolic Differentiation and Algebraic Simplification. Version 1.0
Enter a symbolic expression, or "stop" 12*x^4+7*x^3+9*x^2+1000.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 12*(4*x^3)+7*(3*x^2)+9*(2*x)

Enter a symbolic expression, or "stop" |:
```

Ej 2

```
Enter a symbolic expression, or "stop" |: (1+4*x^3)/(1+2*x^2).
Differentiate w.r.t. :|: x.

Differential w.r.t x is (4*(3*x^2)*(1+2*x^2)-2*(2*x)*(1+4*x^3))/(1+2*x^2)^2
```

Ej 3 Importante destacar este ejemplo ya que se da la utilización de dos variables y podemos ver que el programa da el resultado respecto a cuál variable uno escribe que se quiere derivar.

```
Differential w.r.t x is (4*(3*x^2)*(1+2*x^2)-2*(2*x)*(1+4*x^3))/(1+2*x^2)^2

Enter a symbolic expression, or "stop" |: 4*(x^3)*(y^2)+3*x^3+2*y-6.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 4*(3*x^2)*y^2+3*(3*x^2)

Enter a symbolic expression, or "stop" |: 4*(x^3)*(y^2)+3*x^3+2*y-6.
Differentiate w.r.t. :|: y.

Differential w.r.t y is 2*y*(4*x^3)+2
```

Ej 4 Se destaca este ejemplo ya que NO logra ser funcional

```
Enter a symbolic expression, or "stop" x^(x*log(cos(x))).
Differentiate w.r.t. :|: x.

false.
```

Ej 5 Se destaca este ejemplo ya que NO logra ser funcional, el problema es sí hay un operador de esta manera al principio de la variable, NO cambia el resultado cómo se pusieran los paréntesis.

```
?- symdiff.

SYMDIFF : Symbolic Differentiation and Algebraic Simplification. Version 1.0
Enter a symbolic expression, or "stop" -(x^3+x-1)/2.
Differentiate w.r.t. :|: x.

false.

?- symdiff.

SYMDIFF : Symbolic Differentiation and Algebraic Simplification. Version 1.0
Enter a symbolic expression, or "stop" (-(x^3+x-1)/2).
Differentiate w.r.t. :|: x.

false.
```

Ej 6 Se resalta que el programa tiene problemas con el manejo de las raíces al usar exponentes fraccionales

```
?- symdiff.

SYMDIFF : Symbolic Differentiation and Algebraic Simplification. Version 1.0
Enter a symbolic expression, or "stop" x^(3/2).
Differentiate w.r.t. :|: x.

false.

?- symdiff.

SYMDIFF : Symbolic Differentiation and Algebraic Simplification. Version 1.0
Enter a symbolic expression, or "stop" x^3/2.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 3*x^2*2/4
```

Ej 7

```
Enter a symbolic expression, or "stop" |: ((x^2)-2)^2.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 2*(2*x)*(x^2-2)
```

Ej 8

```
Enter a symbolic expression, or "stop" |: (x^5-x^3+3)^4.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 4*(5*x^4-3*x^2)*(x^5-x^3+3)^3
```

- **Si el programa tiene componentes para la simplificación de expresiones algebraicas, explorar las capacidades y limitaciones de simplificación de expresiones algebraicas que posee el programa.**

El programa SÍ cuenta con la capacidad de dar la simplificación de expresiones. Cuenta con manejo de errores, simplificaciones basadas en reglas del cero y del uno, diferencias de cuadrados y una manera para factorizar la expresión si fuera el caso de que sea posible. Se encontró el limitante que el programa no es capaz de realizar una multiplicación si hay paréntesis en medio por lo que las soluciones se encuentran sin esa manera de simplificar.

- **Una sección con los análisis de los resultados obtenidos con el programa estudiado; nos interesa que determinen qué es capaz de hacer bien y cuáles son sus limitaciones.**

Se encontró que este programa es bastante útil para realizar derivadas sencillas tanto como derivadas complejas. Uno de los resultados principales es que se destaca que el uso de los paréntesis es bastante tedioso y confuso por lo que puede llegar a generar muchos inconvenientes. Basándose en los ejemplos, podemos ver que en el ejemplo #7 se nos da un resultado de **(x^2-2)**, por lo que a primera vista, la resta a la par del exponente puede generar confusión.

El ejemplo #3 nos da la demostración de la capacidad del programa para poder derivar dependiendo de la variable que se escribe, esto aumenta mucho la capacidad del programa para así resolver mayor cantidad de problemas.

El ejemplo #4 es destacable ya que el sistema no es capaz de resolverlo. La expresión cuenta con un logaritmo de exponente, pero lo que más llama la atención es que comparado al otro programa utilizado en la investigación, aún así que está escrito de manera equivalente para comparar la eficacia en la resolución de problemas, este sistema no logra darle solución. Es igual al por qué no puede resolver el ejemplo #6.

También, el ejemplo #5 nos da como resultado que no importa de qué forma se utilizan los paréntesis en este problema, no haya manera de darle resultado. Se dirige el causante a que en el problema, la variable que se mantuvo sin cambios, fue el operador de resta al principio de la expresión.

Por último, los demás ejemplos nos demuestran la gran capacidad del sistema para resolver problemas tanto con raíces, como con exponentes y con expresiones complejas y extensas. Tomando en cuenta su sistema para la simplificación, este también es de gran ayuda para dar una solución más exacta a la expresión derivada.

Solución 2: Sym-diff-prolog by Woljet Jurczyk

- **Explicación general de la solución: capacidades observadas del programa (tipo de expresiones que puede diferenciar simbólicamente), la estrategia de solución (según la interpretación que hagan ustedes) y la estructura del programa.**

La solución se va a basar en la recursividad y específicamente en el backtracking. Se va a introducir una expresión simbólica y se va a descomponer para ir así derivando de lo más simple a lo más complejo. Lo que principalmente destaca de esta solución es que las funciones algebraicas se encuentran optimizadas por lo que su capacidad de resolución de problemas es más amplia.

La primera parte del programa se encuentran las funciones algebraicas detalladas: sumas, restas, multiplicaciones con el cero y con el uno, divisiones con el cero y con el uno, y manejo de errores. Este sería el apartado que se utiliza para realizar las operaciones y para simplificar ciertas expresiones.

- **Explicación de la forma que deben tomar las expresiones simbólicas, o términos, de manera que el programa sea capaz de procesarlas (suministradas como archivos de entrada o parámetros [términos] para las cláusulas que hacen el procesamiento).**

La forma de los parámetros es utilizando la función *diff*. Como primer argumento se ingresa la expresión, como segundo argumento es la variable respecto con la que se quiere derivar y el último argumento se le asigna al resultado a retornar.

De igual forma que en el programa #1 para representar las sumas se utiliza "+", para las restas se utiliza "**-**", para las multiplicaciones se va a utilizar el símbolo "*", para las divisiones se va a utilizar el "/" y para poner exponentes en expresiones algebraicas se utiliza el símbolo (^). Ej: **diff(12*x^4+7*x^3+9*x^2+1000,x,N).**

- **Explicación de cada parte del programa. Si hay uso del 'cut' (!), explique por qué es pertinente en cada lugar donde se lo usa.**

Importante destacar que en el programa no se utiliza el comando 'cut' (!), por lo que llama la atención para el manejo del backtracking.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
opt_sum(X, X, 0).
opt_sum(Y, 0, Y).

opt_sum(W, X, Y) :-
    number(X),
    number(Y),
    W is X + Y.

opt_sum(2*X, X, Y) :-
    X == Y.

opt_sum(X+Y, X, Y).
```

Estas serían las funciones que se utilizan para las operaciones matemáticas, específicamente para la sumas. En estos casos el valor resultante es el primer argumento y los valores sumatorios son el segundo y tercer argumento.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
opt_sub(X, X, 0).
opt_sub(-Y, 0, Y).

opt_sub(W, X, Y) :-
    number(X),
    number(Y),
    W is X - Y.

opt_sub(0, X, Y) :-
    X == Y.

opt_sub(X-Y, X, Y).
```

Del mismo modo que las funciones anteriores, estas serían las que se utilizan para las operaciones algebraicas y específicamente para las restas. El valor resultante es el primer argumento y los valores resultantes el segundo y el tercer argumento.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
opt_mul(0, 0, _).
opt_mul(0, _, 0).

opt_mul(Y, 1, Y).
opt_mul(X, X, 1).

opt_mul(Y, X, Y) :-
    X == 1.

opt_mul(W, X, Y) :-
    number(X),
    number(Y),
    W is X * Y.

opt_mul(X*Y, X, Y).
```

Estas serían las funciones para poder realizar las multiplicaciones. Vemos cómo en estos casos se toman en cuenta las multiplicaciones por uno y por cero. De igual forma que el estándar utilizado en las funciones anteriores, el primer valor es el valor resultante y los demás los operandos.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
opt_div(_,_,0) :-
    throw(error(evaluation_error(zero_divisor),(is)/2)).

opt_div(0, 0, N).

opt_div(X, X, 1).

opt_div(1, X, Y) :-
    X == Y.

opt_div(W, X, Y) :-
    number(X),
    number(Y),
    W is X / Y.

opt_div(X/Y, X, Y).
```

En este caso se encuentran las funciones referentes a la divisiones que se realizan en la aplicación. Se puede notar cómo se utiliza el manejo de errores para la división entre cero. Además, es interesante destacar que cuando se hacen operaciones donde se da una obviedad del resultado como se tiende a utilizar una variable anónima representando la variable como "_".

Las siguientes funciones van a representar a las funciones que se encargan directamente del backtracking y de ir manejando la derivación de las expresiones.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% diff
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constant
diff(E, _, 0) :- number(E).

% the same variable
diff(E, V, 0) :-
    atom(E),
    E \== V.

% other variables
diff(E, V, 1) :-
    atom(E),
    E == V.
```

En este primer grupo se encuentran las funciones más simples donde se manejan el uso de derivaciones con constantes, y cuando la expresión solicitada es la misma que la variable con respecto a derivar.

```
% sum
diff(E1 + E2, V, W) :-
    diff(E1, V, Ed1),
    diff(E2, V, Ed2),
    opt_sum(W, Ed1, Ed2).

% subtraction
diff(E1 - E2, V, W) :-
    diff(E1, V, Ed1),
    diff(E2, V, Ed2),
    opt_sub(W, Ed1, Ed2).
```

```
% (fg)' = f'g + fg'
diff(E1 * E2, V, W) :-
    diff(E1, V, Ed1),
    diff(E2, V, Ed2),
    opt_mul(L, Ed1, E2),
    opt_mul(P, E1, Ed2),
    opt_sum(W, L, P).


% (f/g)' = (f'g - fg') / (g*g)
diff(E1 / E2, V, E) :-
    diff(E1, V, Ed1),
    diff(E2, V, Ed2),
    opt_mul(A, Ed1,E2),
    opt_mul(B, E1,Ed2),
    opt_mul(C, E2,E2),
    opt_sub(D, A, B),
    opt_div(E, D,C).
```

Estas reglas van a hacer referencia a las reglas que se utilizarán para completar las sumas y restas de las derivaciones. En el lado izquierdo se encuentran las reglas con las que se realizan las sumatorias y restas simples de las derivadas y del lado derecho cuando se realizan las multiplicaciones y divisiones de las derivadas. Acá podemos ver cómo se utilizan de una manera más directa las operaciones que con el programa anterior visto. De esta manera el resultado se da de una manera simplificada.

```
% functions
% (h(g))' = h'(g) * g'
diff(sin(E), V, M) :-
    diff(E,V, Ed),
    opt_mul(M, Ed, cos(E)).

diff(cos(E), V, K) :-
    diff(E,V, Ed),
    opt_mul(M, Ed,sin(E)),
    opt_sub(K, 0, M).

diff(tan(E), V, F) :-
    diff(E,V, Ed),
    opt_mul(A, Ed, 2),
    opt_mul(B, 2, E),
    opt_mul(C, 2, cos(B)),
    opt_sum(D, C, 1),
    opt_div(F, A, D).
```

Las siguientes funciones van a hacer la referencia para el apartado de las reglas para la derivación de las funciones trigonométricas. Podemos ver que de igual forma trata de completar el mismo principio de ir descomponiendo las expresiones y deriva el argumento de las funciones trigonométricas y luego las acomoda dependiendo de las reglas.

```
diff(exp(E), V, F) :-
    diff(E,V, Ed),
    opt_mul(F, Ed, exp(E)).

diff(log(E), V, F) :-
    diff(E,V, Ed),
    opt_div(F, Ed, E).

% (f^g)' = f^(g-1)*(gf' + g'f logf)
diff(F^G, V, FF) :-
    diff(F, V, Fd),
    diff(G, V, Gd),
    opt_sub(AA, G, 1),
    opt_mul(BB, G,Fd),
    opt_mul(CC, Gd, F),
    opt_mul(DD, CC, log(F)),
    opt_sum(EE, BB, DD),
    opt_mul(FF, F^(AA), EE).
```

En el último grupo se encuentran las funciones que se encargan de completar las derivaciones con respecto a las funciones exponenciales y logarítmicas y también para resolver la posibilidad de derivar cuando una función presenta otra función como su exponente.

- **Ejemplos del procesamiento que realiza el programa sobre al menos 8 funciones no triviales que ejerciten todas las capacidades del programa. Al menos una de esas funciones debe tener más de una variable y ustedes deben determinar si el programa es capaz de obtener (o no) la derivada respecto de cada una de las variables.**

Ej 1

```
?- diff(12*x^4+7*x^3+9*x^2+1000,x,N).
N = 12*(x^3*4)+7*(x^2*3)+9*(x^1*2) .
```

Ej 2

```
?- diff((1+4*x^3)/(1+2*x^2),x,N).
N =  (4*(x^2*3)*(1+2*x^2)-(1+4*x^3)*(2*(x^1*2)))/((1+2*x^2)*(1+2*x^2)) .
```

Ej 3 Importante este ejemplo ya que muestra cómo el programa es capaz de dar respuesta según la variable respecto a la cual se quiere derivar

```
?- diff(4*(x^3)*(y^2)+3*x^3+2*y-6,x,N).
N = 4*(x^2*3)*y^2+3*(x^2*3) .

?- diff(4*(x^3)*(y^2)+3*x^3+2*y-6,y,N).
N = 4*x^3*(y^1*2)+2
```

Ej 4 Este ejemplo es destacable, ya que es un ejemplo complejo y aún así logra el resultado correcto.

```
?- diff(x^(x*log(cos(x))), x, N).
N = x^(x*log(cos(x))-1)*(x*log(cos(x))+(log(cos(x))+x*(-sin(x)/cos(x)))*x*log(x))
```

Ej 5 Ejemplo que llama la atención ya que no logra funcionar no importa el orden de los paréntesis.,

```
?- diff(-x^3,x,N).
false.

?- diff((-x^3),x,N).
false.
```

Ej 6 Se resalta la capacidad del programa para poder resolver problemas con raíces

```
?- diff(x^(3/2),x,N)
N = x^(3/2-1)*(3/2)
```

Ej 7

```
?- diff(((x^2)-2)^2,x,N).
N =  (x^2-2)^1*(2*(x^1*2))
```

Ej 8

```
?- diff((x^5-x^3+3)^4, x, N).
N =  (x^5-x^3+3)^3*(4*(x^4*5-x^2*3))
```

- **Si el programa tiene componentes para la simplificación de expresiones algebraicas, explorar las capacidades y limitaciones de simplificación de expresiones algebraicas que posee el programa.**

El programa SÍ presenta una manera para realizar simplificaciones. Esta es una solución interesante y optimizada ya que va a llamar a solo las funciones necesarias para simplificar las operaciones que se saben que estarán. Esto permite al programa ser eficaz para solo procesar las expresiones con la función de simplificación en procesos en donde se sabe que van a estar.

Una sección con los análisis de los resultados obtenidos con el programa estudiado; nos interesa que determinen qué es capaz de hacer bien y cuáles son sus limitaciones.

Es un programa bastante completo y posee una gran capacidad para resolver derivadas. Esto se puede notar principalmente con el ejemplo #3 es donde se ve que el programa es capaz de dar solución a una derivada con dos variables y da la respuesta a la variable respectiva.

El principal problema en el código es su falta de documentación que lo hace un poco complejo entender ya que se realizan las operaciones directamente. Esto también es de ayuda ya que no tiene la necesidad de cada vez que revisa una expresión corre todas las maneras para simplificar.

Con respecto a los ejemplos, se encontró que este programa tiene la capacidad de resolver problemas complejos, tomando de base el ejemplo #4 en dónde se encuentra una expresión con una función logarítmica como su exponente y este programa es capaz de darle solución de una manera eficaz. Además, se resalta específicamente el ejemplo #6 en donde nos dice que el programa es capaz de derivar funciones en donde cuente con exponentes fraccionales

También se encontró que este programa tiene un problema a la hora de intentar reconocer funciones que tiene como inicio una variable con un operando como lo es el ejemplo #5. Se especificaron los paréntesis pero el resultado no mostró cambio.

Conclusiones

Se concluye que los sistemas de procesamiento simbólicos tienen la capacidad de ser altamente eficientes pero su principal desventaja es que para ingresar una expresión, la expresión tiene que estar adaptada a los parámetros de recibimiento del programa.

También, realizando una comparación de ambos se puede ver cómo el programa de Paine, tiene más capacidad de simplificación con sus funciones para la realización de diferencias de cuadrados, la factorización y para el manejo de errores que aseguran que la expresión final pueda quedar lo más simplificada posible, aparte se da la utilización del comando "cut", que facilita la terminación de recursiones de una manera eficiente y eficaz. Además la claridad y especificación del código hace que sea una manera más sencilla y rápida de entender y optimizar.

Sin embargo, el programa realizado por Jurczyk también demostró una gran capacidad de optimización de sus funciones, Asimismo, este presentaba una función para poder resolver derivadas de funciones exponenciales donde su exponente fuera otra función por lo que eso aumenta su capacidad de resolución de problemas todavía más complejos. En general, ambos programas son capaces de lograr su objetivo que es el de resolver derivadas simples y complejas, pero también hay que tomar en cuenta que no tienen manera fácil para poder adaptar funciones matemáticas como expresiones simbólicas..

- **Reflexión sobre la experiencia de estudiar programas escritos en Prolog y su adquisición de conocimientos sobre el paradigma de programación (en) lógica.**

  En la experiencia del equipo se encontró que los programas en Prolog para el manejo simbólico de este tipo son extremadamente útiles. Desde su corto tiempo de ejecución hasta su manera sencilla de lograr validaciones lógicas, son ventajas de estos programas y de este tipo de programación. Además, los programas escritos en Prolog son diferentes a los usualmente vistos como lo son los programas orientados a objetos, por lo que también es importante ver diferentes puntos de vista para dar distintos tipos de soluciones.

  Como estudiantes de ingeniería en computación para nosotros es de gran valor el estudio de diferentes tipos de paradigmas, más que todo, porque estos nos dan herramientas para el futuro para así poder resolver diferentes tipos de problemas según el caso y el mercado lo requiera.

- **Referencias. Los libros, artículos, revistas y sitios Web que utilizaron durante la investigación y el desarrollo de su trabajo. Citar toda fuente consultada.**

Jurczyk, W. Symbolic Differentiation Implementation in Prolog.
https://github.com/wjur/sym-diff-prolog

Paine, J. 'SYMDIFF'. Symbolic Differentiation and Algebraic Simplification.
https://www.j-paine.org/prolog/mathnotes/files/symdiff.pl

- **Apéndice: Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.**

Ambos integrantes trabajaron conjuntamente en las funciones para poder lograr su resultado.

- **Apéndices: el código de cada programa estudiado, o con copia de la sección del documento donde aparece el código sujeto de su estudio (pueden ser imágenes 'pegadas' a su informe).**
  - **Código de Paine**

```
/*  This is the main predicate. */

symdiff :-
   nl,
   write('SYMDIFF : Symbolic Differentiation and '),
   write('Algebraic Simplification. Version 1.0'), nl,
   symdiff1.


  /*  symdiff1 controls the main processing; It prompts  */
  /*  the user for data, differentiates the input,       */
  /*  simplifies the result and then displays it on      */
  /*  the terminal.                                      */

symdiff1  :-
   write('Enter a symbolic expression, or "stop" '),
   read(Exp),
   process( Exp ).


process( stop ) :- !.
```

```
process( Exp ) :-
   write('Differentiate w.r.t. :'),
   read(Wrt),
   d( Exp, Wrt, Diff ),
   simp( Diff, Sdiff ), nl,
   write('Differential w.r.t '), write(Wrt),
   write(' is '),
   write(Sdiff), nl, nl,
   symdiff1.



  /*  The following clauses constitute the differentiation   */
  /*  rules, some of which are recursive. Note too the use   */
  /*  of the cut which ensures that once a special case has  */
  /*  been identified, backtracking will not attempt to find */
  /*  an alternative solution.                               */

d( X, X, 1 ):- !.                  /* d(X) w.r.t. X is 1      */

d( C, X, 0 ):- atomic(C).          /* If C is a constant then */
                                   /* d(C)/dX is 0            */
```

```
d( U+V, X, A+B ):-                 /* d(U+V)/dX = A+B where   */
   d( U, X, A ),                   /* A = d(U)/dX and         */
   d( V, X, B ).                   /* B = d(V)/dX             */

d( U-V, X, A-B ):-                 /* d(U-V)/dX = A-B where   */
   d( U, X, A ),                   /* A = d(U)/dX and         */
   d( V, X, B ).                   /* B = d(V)/dX             */

d( C*U, X, C*A ):-                /* d(C*U)/dX = C*A where     */
   atomic(C),                     /* C is a number or variable */
   C \= X,                        /* not equal to X and        */
   d( U, X, A ), !.               /* A = d(U)/dX               */

d( U*V, X, B*U+A*V ):-            /* d(U*V)/dX = B*U+A*V where */
   d( U, X, A ),                  /* A = d(U)/dX and           */
   d( V, X, B ).                  /* B = d(V)/dX               */

d( U/V, X, (A*V-B*U)/(V*V) ):- /* d(U/V)/dX = (A*V-B*U)/(V*V) */
   d( U, X, A),                /* where A = d(U)/dX and       */
   d( V, X, B).                /*       B = d(V)/dX           */

d( U^C, X, C*A*U^(C-1) ):-        /* d(U^C)/dX = C*A*U^(C-1)   */
   atomic(C),                     /* where C is a number or    */
   C\=X,                          /* variable not equal to X   */
   d( U, X, A ).                  /* and d(U)/dX = A           */
```

```
d( U^C, X, C*A*U^(C-1) ):-       /* d(U^C)/dX = C*A*U^(C-1)   */
   C = -(C1), atomic(C1),        /* where C is a negated number or  */
   C1\=X,                        /* variable not equal to X   */
   d( U, X, A ).                 /* and d(U)/dX = A           */

d( sin(W), X, Z*cos(W) ):-       /* d(sin(W))/dX = Z*cos(W)   */
   d( W, X, Z).                  /* where Z = d(W)/dX         */

d( exp(W), X, Z*exp(W) ):-       /* d(exp(W))/dX = Z*exp(W)   */
   d( W, X, Z).                  /* where Z = d(W)/dX         */

d( log(W), X, Z/W ):-            /* d(log(W))/dX = Z/W        */
   d( W, X, Z).                  /* where Z = d(W)/dX         */

d( cos(W), X, -(Z*sin(W)) ):-    /* d(cos(W))/dX = Z*sin(W)   */
   d( W, X, Z).                  /* where Z = d(W)/dX         */
```

```
simp( X, X ):-          /* an atom or number is simplified */
   atomic(X), !.

simp( X+0, Y ):-        /* terms of value zero are dropped  */
   simp( X, Y ).

simp( 0+X, Y ):-        /* terms of value zero are dropped  */
   simp( X, Y ).

simp( X-0, Y ):-        /* terms of value zero are dropped  */
   simp( X, Y ).

simp( 0-X, -(Y) ):-     /* terms of value zero are dropped  */
   simp( X, Y ).

simp( A+B, C ):-        /* sum numbers */
   numeric(A),
   numeric(B),
   C is A+B.

simp( A-A, 0 ).         /* evaluate differences */
```

```
simp( A-B, C ):-        /* evaluate differences */
   numeric(A),
   numeric(B),
   C is A-B.

simp( X*0, 0 ).         /* multiples of zero are zero */

simp( 0*X, 0 ).         /* multiples of zero are zero */

simp( 0/X, 0 ).         /* numerators evaluating to zero are zero */

simp( X*1, X ).         /* one is the identity for multiplication */

simp( 1*X, X ).         /* one is the identity for multiplication */

simp( X/1, X ).         /* divisors of one evaluate to the numerator */

simp( X/X, 1 ) :- !.

simp( X^1, X ) :- !.

simp( X^0, 1 ) :- !.

simp( X*X, X^2 ) :- !.
```

```
simp( X*X^A, Y ) :-
   simp( X^(A+1), Y ), !.

simp( X^A*X, Y ) :-
   simp( X^(A+1), Y ), !.

simp( A*B, X ) :-       /* do evaluation if both are numbers  */
   numeric( A ),
   numeric( B ),
   X is A*B.

simp( A*X+B*X, Z ):-    /* factorisation and recursive */
   A\=X, B\=X,          /* simplification              */
   simp( (A+B)*X, Z ).

simp( (A+B)*(A-B), X^2-Y^2 ):-   /* difference of two squares */
   simp( A, X ),
   simp( B, Y ).

simp( X^A/X^B, X^C ):-     /* quotient of numeric powers of X */
   numeric(A), numeric(B),
   C is A-B.
```

```
simp( A/B, X ) :-       /* do evaluation if both are numbers  */
   numeric( A ),
   numeric( B ),
   X is A/B.

simp( A^B, X ) :-       /* do evaluation if both are numbers  */
   numeric( A ),
   numeric( B ),
   X is A^B.

simp( W+X, Q ):-        /* catchall */
   simp( W, Y ),
   simp( X, Z ),
   ( W \== Y ; X \== Z ), /* ensure some simplification has taken place */
   simp( Y+Z, Q ).

simp( W-X, Q ):-        /* catchall */
   simp( W, Y ),
   simp( X, Z ),
   ( W \== Y ; X \== Z ), /* ensure some simplification has taken place */
   simp( Y-Z, Q ).

simp( W*X, Q ):-        /* catchall */
   simp( W, Y ),
   simp( X, Z ),
   ( W \== Y  ; X \== Z ), /* ensure some simplificaion has taken place */
   simp( Y*Z, Q ).
```

```
simp( A/B, C ) :-
   simp( A, X ),
   simp( B, Y ),
   ( A \== X ; B \== Y ),
   simp( X/Y, C ).

simp( X^A, C ) :-
   simp( A, B ),
   A \== B,
   simp( X^B, C).

simp( X, X ).           /* if all else fails... */


numeric(A) :- integer(A).
```

- **Código de Jurczyk**

```
% diff(x^(x*log(cos(x))), x, N).
% N = x^(exp(log(1/x))-1)*(exp(log(1/x))+ - (1)/(x*x)/(1/x)*exp(log(1/x))*x*log(x))*cos(x^exp(log(1/x)))

% diff(expr, var, div)

%%%%%%%%%%%%%%%%%%%%%%%%%
opt_sum(X, X, 0).
opt_sum(Y, 0, Y).

opt_sum(W, X, Y) :-
        number(X),
        number(Y),
        W is X + Y.

opt_sum(2*X, X, Y) :-
        X == Y.

opt_sum(X+Y, X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%
opt_sub(X, X, 0).
opt_sub(-Y, 0, Y).
```

```
opt_sub(W, X, Y) :-
        number(X),
        number(Y),
        W is X - Y.

opt_sub(0, X, Y) :-
        X == Y.

opt_sub(X-Y, X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%
opt_mul(0, 0, _).
opt_mul(0, _, 0).

opt_mul(Y, 1, Y).
opt_mul(X, X, 1).

opt_mul(Y, X, Y) :-
        X == 1.

opt_mul(W, X, Y) :-
        number(X),
        number(Y),
        W is X * Y.

opt_mul(X*Y, X, Y).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
opt_div(_,_,0) :-
        throw(error(evaluation_error(zero_divisor),(is)/2)).

opt_div(0, 0, N).

opt_div(X, X, 1).

opt_div(1, X, Y) :-
        X == Y.

opt_div(W, X, Y) :-
        number(X),
        number(Y),
        W is X / Y.

opt_div(X/Y, X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%
% diff
%%%%%%%%%%%%%%%%%%%%%%%%%%%
% constant
diff(E, _, 0) :- number(E).
```

```
% the same variable
diff(E, V, 0) :-
         atom(E),
         E \== V.

% other variables
diff(E, V, 1) :-
         atom(E),
         E == V.

% sum
diff(E1 + E2, V, W) :-
         diff(E1, V, Ed1),
         diff(E2, V, Ed2),
         opt_sum(W, Ed1, Ed2).

% subtraction
diff(E1 - E2, V, W) :-
         diff(E1, V, Ed1),
         diff(E2, V, Ed2),
         opt_sub(W, Ed1, Ed2).

% (fg)' = f'g + fg'
diff(E1 * E2, V, W) :-
         diff(E1, V, Ed1),
         diff(E2, V, Ed2),
         opt_mul(L, Ed1, E2),
         opt_mul(P, E1, Ed2),
         opt_sum(W, L, P).


% (f/g)' = (f'g - fg') / (g*g)
diff(E1 / E2, V, E) :-
         diff(E1, V, Ed1),
         diff(E2, V, Ed2),
         opt_mul(A, Ed1,E2),
         opt_mul(B, E1,Ed2),
         opt_mul(C, E2,E2),
         opt_sub(D, A, B),
         opt_div(E, D,C).
```

```
% functions
% (h(g))' = h'(g) * g'
diff(sin(E), V, M) :-
    diff(E,V, Ed),
        opt_mul(M, Ed, cos(E)).

diff(cos(E), V, K) :-
    diff(E,V, Ed),
        opt_mul(M, Ed,sin(E)),
        opt_sub(K, 0, M).

diff(tan(E), V, F) :-
    diff(E,V, Ed),
        opt_mul(A, Ed, 2),
        opt_mul(B, 2, E),
        opt_mul(C, 2, cos(B)),
        opt_sum(D, C, 1),
        opt_div(F, A, D).

diff(exp(E), V, F) :-
    diff(E,V, Ed),
        opt_mul(F, Ed, exp(E)).

diff(log(E), V, F) :-
    diff(E,V, Ed),
        opt_div(F, Ed, E).

% (f^g)' = f^(g-1)*(gf' + g'f logf)
diff(F^G, V, FF) :-
    diff(F, V, Fd),
    diff(G, V, Gd),
        opt_sub(AA, G, 1),
        opt_mul(BB, G,Fd),
        opt_mul(CC, Gd, F),
        opt_mul(DD, CC, log(F)),
        opt_sum(EE, BB, DD),
        opt_mul(FF, F^(AA), EE).
```