

**INSTITUTO TECNOLÓGICO DE COSTA RICA**

**ESCUELA DE INGENIERÍA COMPUTACIÓN**

IC-4700 Lenguajes de programación

Derivadores simbólicos - programas escritos en el paradigma lógico-relacional

Valeria Chinchilla Mejías | 2020024186

Gabriel Isaías Fallas López | 2020027738

Nikholas Ocampo Fuentes | 2020061243

Profesor: Ignacio Trejos Zelaya

I Semestre, 2022

# Table de Contenido

<b>Table de Contenido</b>	<b>2</b>
<b>Introducción</b>	<b>3</b>
<b>Payne</b>	<b>4</b>
Explicación General	4
Explicación de la entrada de expresiones simbólicas	4
Explicación de cada parte del programa	5
Ejemplos	7
Simplificación	10
Análisis de los resultados	11
<b>Jurczyk</b>	<b>11</b>
Explicación General	11
Explicación de la entrada de expresiones simbólicas	12
Explicación de cada parte del programa	12
Ejemplos	17
Simplificación	19
Análisis de los resultados	20
<b>Washington</b>	<b>20</b>
Explicación General	20
Explicación de la entrada de expresiones simbólicas	21
Explicación de cada parte del programa	21
Ejemplos	26
Simplificación	28
Análisis de los resultados	29
<b>Conclusiones</b>	<b>30</b>
<b>Reflexión</b>	<b>31</b>
<b>Reparto del Trabajo</b>	<b>32</b>
<b>Referencias</b>	<b>33</b>
<b>Apéndices</b>	<b>34</b>
Apéndice A: Payne	34
Apéndice B: Jurczyk	40
Apéndice C: Washington	45

# 1. Introducción

Siempre que surge una nueva tecnología, surgen un mar de problemas que se desea resolver con ella. Muchas de las cosas que conocemos hoy en día han sido obtenidas gracias a la resolución de problemas, un ejemplo de ello sería la creación de las computadoras. En el pasado, cuando venía empezando la computación, las grandes mentes de la época buscaban resolver problemas que los afectaban en su día a día. Uno de los grandes problemas de la computación en esta época fue la diferenciación simbólica. Lo que se buscaba era utilizar la computación junto con el manejo de símbolos matemáticos, sin embargo no se sabía como hacerlo.

En 1958 John McCarthy decidió poner en uso sus conocimientos y se propuso construir un programa capaz de realizar una diferenciación simbólica de expresiones algebraicas de manera eficaz. Durante el proceso, desarrolló el lenguaje LISP debido a que ningún lenguaje de la época le facilitaba la programación de su problema. Sin duda alguna McCarthy logró crear un lenguaje de programación que iba en los próximos años logro a impactar fuertemente el mundo de la computación, y al mismo tiempo logró cumplir su objetivo inicial.

En este proyecto, se presentan tres algoritmos escritos en Prolog (Payne, Juzreczky y Washington) para la resolución diferenciación simbólica de expresiones algebraicas. En esta caso se toma como diferenciación simbólica obtener la primera derivada de una expresión algebraica. Para cada uno de los programas se explica cada una de sus partes, cómo ejecutar el programa y algunas pruebas del funcionamiento del mismo. Adicionalmente, se explica como funciona el simplificador de cada uno (si es que cuenta con uno) y se presenta un análisis de los resultados obtenidos.

## **2. Payne**

### **2.1. Explicación General**

El programa Symdiff brinda soluciones al problema de la diferenciación simbólica mediante el lenguaje de programación lógico Prolog, dentro del mismo se encuentra una serie de cláusulas que constituyen las reglas de diferenciación, donde algunas de ellas se ejecutan de manera recursiva. Por otro lado, dentro del programa se pueden encontrar reglas para la simplificación del resultado del procedimiento de diferenciación.

El programa en primer lugar recibe la expresión simbólica, la cual le pide al usuario por medio de la consola y después aplica las reglas de diferenciación simbólica expuestas anteriormente. Seguidamente implementa la simplificación en la expresión, al finalizar este proceso se muestra en la pantalla el resultado obtenido y se le indica al usuario que ingrese otra expresión simbólica o coloque “stop” para detener el programa. El código de este programa se puede observar en el **apéndice A**.

### **2.2. Explicación de la entrada de expresiones simbólicas**

Las expresiones simbólicas o términos deben ser suministrados al programa mediante parámetros para que este pueda realizar correctamente el procesamiento. Para correr el programa es necesario invocar a la cláusula `simdiff1` con un punto. Después el programa le va a pedir al usuario la expresión simbólica que se busca resolver, es importante recordar que al final de esta se debe colocar un punto. Al final la llamada a la función debería quedar de la siguiente manera (lo subrayado en amarillo es lo que debe ser digitado por el usuario):

```

?- symdiff1 .
Enter a symbolic expression, or "stop" x.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 1

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

```

## 2.3. Explicación de cada parte del programa

El programa consta en cuatro bloques en el primero está la cláusula `symdiff1` que espera que el usuario ingrese una expresión simbólica o coloque “stop”. En el segundo, está la cláusula `process` que es la encargada de llevar a cabo todo el proceso de diferenciación simbólica debido a que se enarga de llamar a las cláusulas de las reglas de diferenciación simbólica y por último las reglas para la simplificación que se encuentran en el tercer y cuarto bloque respectivamente. A continuación se explica detalladamente cada una de estas secciones.

### I. Symdiff

Esta es la función principal del programa, se podría decir que es como el `main`. Esta función le pide al usuario que digite la expresión simbólica que desea derivar y una vez digitada llama a la función `process` y le manda como argumento la expresión digitada por el usuario.

### II. Process

Esta parte del programa le pide al usuario que digite la variable sobre la cual desea realizar la derivación. Después de esto, llama a la función `d` pasandole como argumento la expresión. Luego, el resultado obtenido de la derivación es mandado como parámetro a la función `simp`. Por último, el resultado de la expresión derivada y simplificada es mostrada al usuario y se vuelve a llamar a la función `Symdiff`.

### III. D

Esta función es la encargada de realizar todo el procedimiento de derivación. Para ello cuenta con varios casos distintos donde aplica las reglas. La función se va a ejecutar de manera recursiva en las derivaciones más complejas como la suma, resta, la multiplicación, la división y las potencias. En algunos de estos casos se utiliza el operado cut (!) para indicarle a prolog que no debe de buscar más respuesta.

Además, es importante denotar que la función debe recibir tres parámetros indispensables. El primer argumento debe ser la expresión a derivar, el segundo debe ser la variable sobre la cual se va a derivar y el tercer parámetro es la variable en donde se va almacenar el resultado de la derivación. Por otro lado, a continuación se explican los casos de esta función.

- El primer caso es para indicar que la derivada de una constante es 1.
- El segundo caso es para indicar que si se recibe un número entonces su derivada es 0.
- El tercer caso es para indicar que la derivada de una suma es la suma de la derivación de ambas partes que la conforman.
- El cuarto caso es para indicar que la derivada de una resta es la resta de la derivación de ambas partes que la conforman.
- El quinto caso es para indicar que la derivada de una multiplicación que involucra una expresión y una constante o variable va a ser la constante o variable multiplicada por la derivada de la expresión.
- El sexto caso y el séptimo caso son para aplicar la regla de la multiplicación y la regla del cociente.
- El octavo y el noveno caso son para la aplicación de la regla de la cadena pero con parámetros que pueden variar.

- Los últimos 4 casos que sobran son para la aplicación de reglas sobre expresiones trigonométricas.

#### IV. Simp

Esta función sirve para simplificar expresiones algebraicas, para ello aplica una serie de casos distintos en los cuales va aplicando las reglas de simplificación. A continuación se explican estos casos.

- El primer caso es para resolver que la simplificación de una constante es la misma constante.
- El segundo, tercer, cuarto y quinto caso son para resolver que la simplificación de una variable sumada o restada con cero da la misma variable.
- El sexto caso es para resolver que la simplificación de dos números sería la suma de ambos.
- El séptimo y octavo caso son para resolver que la simplificación de una resta es la resta de ambas expresiones.
- A partir del noveno caso hasta el caso número 18 se aplican reglas básicas de la multiplicación, división y potencias.
- A partir del caso número 19 se aplican otras leyes de simplificación un poco más complejas, a las cuales no se entrará en detalle debido a que son cálculos más matemáticos.

### 2.4. Ejemplos

Para probar el funcionamiento de este derivador se va a hacer uso del libro libro de Domínguez (s.f) y otras fuentes. Para las 8 pruebas se escogieron ejemplos que demuestren el uso de las distintas reglas de derivación. A continuación se expondrán cada una de ellas.

- Prueba 1: Se utiliza el ejemplo 83 del libro de Borbón (2018, p.59) para probar el funcionamiento de la regla de la suma y la resta junto con potencias y divisiones. El programa no logra realizar la derivación, posiblemente por la potencia de 1/2.

$$\frac{d}{du} (3xu^5 - 2u^3 + x^2\sqrt{u} + x)$$

```
Enter a symbolic expression, or "stop" 3*x*u^5 - 2*u^3 + x^2*(u^(1/2)).
Differentiate w.r.t. :|: x.

false.
```

- Prueba 2: Se utiliza el ejemplo 6 del libro de Domínguez (s.f, p.14) para probar el funcionamiento de la regla de la suma, la resta y las potencias de una manera más simple. El resultado es bueno, logra ejecutarse la operación.

$$f(x) = 2x^7 - 3x^6 + 3x^3 - 4x^2 - 7$$

```
Enter a symbolic expression, or "stop" |: (2*x^7 - 3*x^6 + 3*x^3 - 4*x^2 - 7).
Differentiate w.r.t. :|: x.

Differential w.r.t x is 2*(7*x^6)-3*(6*x^5)+3*(3*x^2)-4*(2*x)
```

- Prueba 3: Se utiliza el ejemplo 7 del libro de Domínguez (s.f, p.14) para probar el funcionamiento de la regla del cociente. El resultado es bueno, la respuesta coincide con la del libro.

$$f(x) = \frac{x-3}{2}$$

```
Enter a symbolic expression, or "stop" |: (x-3)/2.
Differentiate w.r.t. :|: x.

Differential w.r.t x is 0.5
```



- Prueba 4: Se utiliza el ejemplo 24 del libro de Domínguez (s.f, p.18) para probar el funcionamiento de la regla de la multiplicación. El resultado es bueno, la respuesta coincide con la del libro.

$$f(x) = (x - 1) \cdot (x + 1)^2$$

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

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

- Prueba 5: Se utiliza el ejemplo 61 del libro de Domínguez (s.f, p.24) para probar el funcionamiento de la reglas con logaritmos. El resultado es bueno, logra ejecutar la operación y coincide con la respuesta del libro.

$$f(x) = \log(x - 3)^2$$

```
Enter a symbolic expression, or "stop" |: log(x-3)^2.
Differentiate w.r.t. :|: x.

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

- Prueba 6: Se utiliza el ejemplo 66 del libro de Domínguez (s.f, p.25) para probar el funcionamiento de la reglas con coseno. El resultado es bueno, logra ejecutar la operación y coincide con la respuesta del libro.

$$f(x) = \cos(3x + 3)$$

```
Enter a symbolic expression, or "stop" |: cos(3*x + 3).
Differentiate w.r.t. :|: x.

Differential w.r.t x is - ((3*1+0)*sin(3*x+3))
```

- Prueba 7: Se utiliza el ejemplo de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de x. El programa logra realizar la derivación y coincide con la respuesta de la página.

$$f(x, y) = 4x^3y^2 + 3x^3 + 2y - 6$$

```
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)
```

- Prueba 8: Se utiliza el ejemplo de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de y. El programa logra realizar la derivación y coincide con la respuesta de la página.

$$f(x, y) = 4x^3y^2 + 3x^3 + 2y - 6$$

```
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
```

## 2.5. Simplificación

El programa si cuenta con un simplificador que funciona bastante bien. Sin embargo, en los resultados obtenidos de las pruebas anteriores las respuestas podían ser aún más simplificadas. Incluso hubo que simplificarlas a mano para poder determinar si coincidían con

las respuestas dadas en los libros. Esto es una gran limitación para el simplificador ya que significa que no está realizando del todo bien su trabajo. A pesar de esta limitación mencionada, no se encontró ninguna otra que afecte.

## **2.6. Análisis de los resultados**

Los resultados obtenidos de este programa fueron muy buenos, básicamente pasó todas las pruebas a las que fue sometido y solo en una ocasión falló pero se cree que fue por elevar la variable con un número decimal, lo cual posiblemente no soporta el algoritmo. Lo único que representó un problema en este programa fue el simplificador, como se mencionaba anteriormente. No obstante, se tiene un buen nivel de satisfacción con los resultados obtenidos en este programa.

## **3. Jurczyk**

### **3.1. Explicación General**

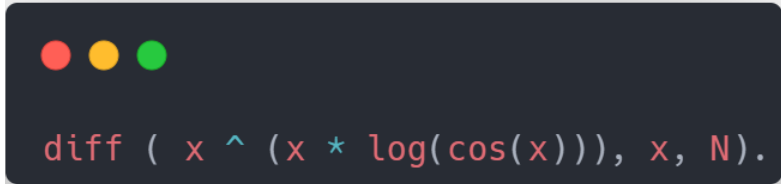
Este programa está conformado por una función principal llamada diff. Esta función recibe tres parámetros. El primer parámetro va a ser la expresión algebraica que se desea derivar. El segundo parámetro va a definir la variable sobre la cual se desea realizar la derivación. Por último, el tercer parámetro es la variable en la que se va a almacenar el resultado de la primera derivada. En el programa se tienen distintas versiones de diff, esto es para poder aplicar las distintas reglas de derivación. El código de este programa se puede observar en el **apéndice B**.

### 3.2. Explicación de la entrada de expresiones simbólicas

Las entradas o parámetros que se deben recibir para poder ejecutar el programa se deben realizar de la siguiente manera. En primer lugar se digita el nombre de la función `diff` y entre paréntesis se colocan los 3 argumentos seguidos. La expresión algebraica se debe de escribir utilizando los siguientes símbolos:

- `*` : para multiplicación. Ejemplo: Si se desea multiplicar  $x$  y  $m$  entonces se escribe  $x * m$ .
- `^` : para elevar un número. Ejemplo: Si se desea escribir  $x$  cubico entonces se deberá escribir  $x^3$ .
- `+` : para sumas. Ejemplo: Si se desea sumar  $x$  y  $m$  entonces se escribe  $x + m$ .
- `-` : para restas. Ejemplo: Si se desea restar  $x$  y  $m$  entonces se escribe  $x - m$ .
- `/` : para dividir. Ejemplo: Si se desea dividir  $x$  entre  $m$  entonces se escribe  $x / m$ .
- `()` : para agrupar operaciones

Utilizando los simbolos listados anteriormente se debería tener listo el primer argumento de la función. Para los otros dos nada más se coloca la variable sobre la cual se va a derivar y la variable donde se va a almacenar el resultado. Al final la llamada a la función debería quedar así:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The text `diff ( x ^ (x * log(cos(x))), x, N).` is displayed in a light blue monospace font.

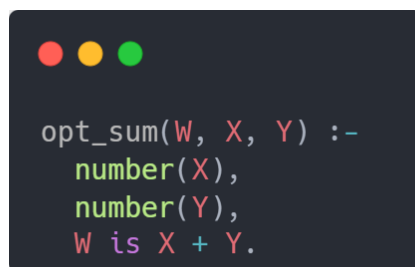
```
diff ( x ^ (x * log(cos(x))), x, N).
```

### 3.3. Explicación de cada parte del programa

#### I. Funciones Opt

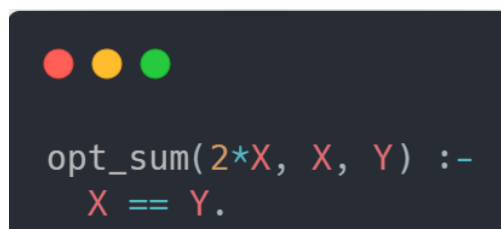
En esta sección del programa se crean las operaciones básicas para la derivación, como lo son la suma, la resta, la multiplicación y la división. A continuación se explica detalladamente cada una de estas funciones.

- **Opt\_sum:** Esta parte va a contar con varios subcasos posibles, esto se debe a que se deben establecer todas las propiedades de la suma que se conocen.
  - La primera operación es `opt_sum(X, X, 0)`. Además, se tiene la operación `opt_sum(Y, 0, Y)`. Estas operaciones sirven para establecer que cualquier número  $X$  sumado con  $0$  va a dar cero.
  - La tercera operación determina si las variables  $X$  e  $Y$  son números, de ser así los suma y los almacena en  $W$ .



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

- La cuarta operación se utiliza cuando al sumar  $X$  e  $Y$  se obtiene  $2X$ . En este caso, la  $Y$  va a ser igual a la  $X$ .



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

- La quinta y última operación es `opt_sum(X+Y, X, Y)`. Esta operación se utiliza para establecer la conmutatividad de la operación suma. Entonces,  $X$  sumado con  $Y$  es lo mismo que  $Y$  sumado con  $X$ .

- **Opt\_sub:** Esta parte se divide en varios subcasos en los cuales se establecen las distintas propiedades de la resta.
  - La primera operación es para establecer que cualquier número menos 0 es igual al mismo número.
  - La segunda operación es para establecer que cero menos cualquier número es ese número negado
  - La tercera operación es para establecer que si dos variables son números entonces su resta va a ser  $X - Y$ .
  - La cuarta operación es para establecer que si  $X - Y$  es igual a cero, entonces los números son iguales.
  - La quinta operación es para establecer la función default de la resta, si se reciben dos variables entonces su resta va a ser  $X - Y$ .

```

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:** Esta parte se divide en varios subcasos en los cuales se establecen las distintas propiedades de la multiplicación.

- La primera operacion es para establecer que cero multiplicado por cualquier parámetro recibido, va a ser igual a cero.
- La segunda operacion es para establecer que cualquier parámetro recibido multiplicado por cero, va a ser igual a cero.
- La tercera operacion es para establecer que si una variable es multiplicada por 1 entonces el resultado va a ser la misma variable.
- La cuarta operacion es para establecer que si 1 es multiplicado por cualquier variable entonces el resultado va a ser la misma variable.
- La quinta operacion es para establecer que si X multiplicado por Y es igual a Y, entonces X es igual a 1.
- La sexta operacion es para establecer la función default de la multiplicación, si se reciben dos variables entonces su multiplicación va a ser  $X * Y$ .

```
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: Esta parte se divide en varios subcasos en los cuales se establecen las distintas propiedades de la división.
  - La primera operacion es para establecer que cualquier cosa dividido por cero va a dar un error.
  - La segunda operacion es para establecer que cualquier número dividido por si mismo va a dar 1.
  - La tercera operacion es para establecer que si la división de dos números X e Y da 1 entonces X va a ser igual a Y.
  - La cuarta operacion es para establecer que dos numeros divididos es  $X / Y$ .
  - La quinta operacion es para establecer la operación default de la división, si se reciben dos variables entonces su división va a ser  $X / Y$ .

```

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).

```

## II. Funciones Diff



En esta sección del programa, se van a definir todos los subcasos de la operación diff. En cada una de estas funciones se va a establecer alguna regla de derivación. A continuación se listan cada uno de los subcasos de esta parte del programa.

- Constantes: La derivada de una constante va a ser 0.
- Variables: La derivada de una variable que es igual a la variable sobre la cual se está derivando es 1.
- Variables: La derivada de una variable que es diferente a la variable sobre la cual se está derivando es 0.
- Suma: La derivada de una suma es la suma de las derivadas de las variables.
- Resta: La derivada de una resta es la resta de las derivadas de las variables.
- Regla del producto: Este diff lleva a cabo lo siguiente  $fg' = f'g + f'g$ .
- Regla del cociente: Este diff lleva a cabo lo siguiente  $f/g = (f'g - g'f) / g^2$
- Regla de la cadena:  $(h(g))' = h'(g) * g'$ . Esta regla se define para cada el coseno, tangente y seno.
- Potencia: Este diff lleva a cabo lo siguiente  $(f^g)' = f^{(g-1)}(gf' + g'f \log f)$ . Se le resta al exponente 1 y se le multiplica por  $(gf' + g'f \log f)$ .

### 3.4. Ejemplos

Para probar el funcionamiento de este derivador se va a hacer uso del libro de Alexander Borboón Alpízar (2018) utilizado durante el curso de Cálculo Diferencial e Integral y otras fuentes. Para las 8 pruebas se escogieron ejemplos que demuestren el uso de las distintas reglas de derivación. A continuación se expondrán cada una de ellas.

- Prueba 1: Se utiliza el ejemplo 87 del libro de Borbón (2018, p.59) para probar el funcionamiento de la regla del producto. El programa logra realizar la derivación.

$$f(x) = 2^x \cdot x$$

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

- Prueba 2: Se utiliza el ejemplo 88 del libro de Borbón (2018, p.59) para probar el funcionamiento de la regla del cociente. El programa logra realizar la derivación.

$$g(x) = \frac{\ln x}{x^2}$$

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

- Prueba 3: Para probar la regla de la cadena se utiliza el ejemplo 2.32 de libro de Hernández (2013). El programa logra realizar la derivación.

$$D_x(5x + 3)^4$$

```
?- diff((5*x + 3)^4, x, N).
N = (5*x+3)^3*20 .
```

- Prueba 4: Se utiliza el ejemplo 82 del libro de Borbón (2018, p.57) para probar el funcionamiento de la regla de la suma. El programa logra realizar la derivación.

```
?- diff(x^7+5*x^3-x^e+1, x, N).
N = x^6*7+5*(x^2*3)-x^(e-1)*e .
```

- Prueba 5: Se utiliza el ejemplo de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de x. El programa logra realizar la derivación.

$$f(x, y) = 4x^3 + 5y^2$$

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

- Prueba 6: Se utiliza el ejemplo de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de y. El programa logra realizar la derivación.

$$f(x, y) = 4x^3 + 5y^2$$

```
?- diff(4*x^3 + 5*y^2, y, N).
N = 5*(y^1*2) .
```

- Prueba 7: Se utiliza el ejemplo dado por el autor del código para probar las operaciones trigonométricas, logaritmos y las exponenciales. Para resolver este ejemplo se utilizan varias reglas derivación. El programa logra realizar la derivación.

```
?- 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)) .
```

- Prueba 8: Se utiliza el ejemplo de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar el funcionamiento de varias leyes con dos variables. El programa logra realizar la derivación.

$$f_x(x, y) = \frac{10x}{5x^2 - 3y}$$

```
?- diff(10*x/(5*x^2 - 3*y), x, N).
N = (10*(5*x^2-3*y)-10*x*(5*(x^1*2)))/((5*x^2-3*y)*(5*x^2-3*y)) .
```

### 3.5. Simplificación

Este programa no presente un simplificador para la expresiones algebraicas.

### **3.6. Análisis de los resultados**

Los resultados para este derivador son bastante positivos, se logró probar su funcionamiento con distintas pruebas y en ningún caso falló la respuesta. Lo único negativo sobre este derivador es que no simplifica las respuestas que da, por lo que hay que simplificarlas uno mismo para que concuerden con las respuestas dadas en los libros o páginas consultadas. Sin embargo, el derivador funciona perfecto y es capaz de resolver derivadas con un nivel de complejidad alto. Por lo que, el nivel de satisfacción con este programa es muy elevado.

## **4. Washington**

### **4.1. Explicación General**

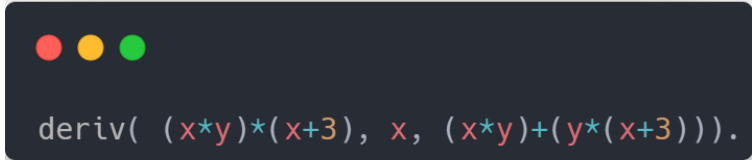
Este programa cuenta con 3 partes importantes: derivador, derivador de derivadas básicas y simplificador. El derivador es como la función principal del programa, esto se debe a que desde el se llama a las otras dos partes del código. Esta función va a recibir 3 argumentos importantes. El primero es la expresión algebraica, el segundo es la variable sobre la cual se va derivar y el tercero es el resultado con el cual se quiere comparar o también se le puede pasar una variable para que almacene el resultado. Una vez que se llama a deriv, automáticamente el programa aplica una derivación básica y simplifica la expresión. Adicionalmente, se hace uso del operador cut ! Este le indica a prolog que no obtenga más combinaciones ya que solo es de interés la expresión derivada actual. El código de este programa se puede observar en el **apéndice C**.

## 4.2. Explicación de la entrada de expresiones simbólicas

Las entradas o parámetros que se deben recibir para poder ejecutar el programa se deben realizar de la siguiente manera. En primer lugar se digita el nombre de la función deriv y entre paréntesis se colocan los 3 argumentos seguidos. La expresión algebraica se debe de escribir utilizando los siguientes símbolos:

- $*$  : para multiplicación.
- $^$  : para elevar un número.
- $+$  : para sumas.
- $-$  : para restas.
- $/$  : para dividir.
- $()$  : para agrupar operaciones

Utilizando los símbolos listados anteriormente el primer argumento de la función estaría listo. Para los otros dos nada más se coloca la variable sobre la cual se va a derivar y el resultado sobre el cual se desea determinar si son iguales o la variable donde se va a almacenar el resultado. Al final la llamada a la función debería quedar así:

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. The text inside the terminal is the function call: `deriv( (x*y)*(x+3), x, (x*y)+(y*(x+3)) )`.

```
deriv( (x*y)*(x+3), x, (x*y)+(y*(x+3)) )
```

## 4.3. Explicación de cada parte del programa

El programa se divide en 3 tres partes importantes, cada una de estas partes se va a explicar a continuación. Sin embargo, antes de continuar es importante denotar que al llamar a la función principal deriv, se llama automáticamente a las derivaciones básicas y al simplificador.

En esta llamada también se hace uso del cut (!) el cual le dice al programa que solo obtenga la primera respuesta y que no dé más resultados.

## I. Basic\_deriv

En esta sección del programa se van a aplicar las reglas de derivación básicas, como lo son la suma, la multiplicación y otros. Seguidamente se explican cada una de los casos del programa.

- El primer caso es para la reglas de las constantes. Si el primer parámetro que se recibe es un número entonces no va a importar sobre la variable que se va a derivar y automáticamente se devuelve como respuesta 0 debido a que la derivada de una constante es cero.
- El segundo caso se utiliza cuando la expresión algebraica es igual a la variable sobre la cual se desea derivar. En esta ocasión lo que se retorna es 1, ya que, por regla, la derivada de una constante es 1.
- El tercer caso se utiliza cuando se recibe una solvariable diferente a la que se desea derivar. En este caso lo que se indica es que la expresión algebraica es distinta de la variable a derivar.
- El cuarto caso se utiliza para la suma de dos variables como expresión algebraica. Para este caso se recibe las dos variables sumadas, la variable sobre la cual se va a derivar y como resultado se obtendrá la suma de la derivada de cada una de las partes. Entonces por ejemplo, si se llama a la función de la siguiente forma  $\text{deriv}(y+x, x, N)$  el programa va a determinar que es una suma y va a sacar la derivada de “y” y la de “x” y las va a sumar, formando así el resultado N. En las pruebas se puede observar y entender con mayor facilidad.

- El quinto caso se utiliza para la multiplicación, para este se recibe la multiplicación de dos variables y se va a retornar como resultado la suma de la multiplicación de cada variable por su derivada.

```

deriv(A,X,C) :- basic_deriv(A,X,B), simplify(B,C), !.

basic_deriv(N,_,0) :- number(N).
basic_deriv(X,X,1).
basic_deriv(Y,X,0) :- atom(Y), Y\==X.

basic_deriv(A+B,X,A1+B1) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).
basic_deriv(A*B,X,A*B1+A1*B) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).

```

## II. Simplify

En esta sección del programa lo que se realiza es simplificar la expresión algebraica obtenida como resultado de la derivación. Se subdivide en varios casos, los cuales se van a explicar a continuación.

- El primer caso es la simplificación de una variable. La simplificación da como resultado la misma variable.
- El segundo caso es la simplificación de un número. La simplificación da como resultado el mismo número pasado como parámetro.
- El tercer caso es la simplificación de una suma. En este caso lo que se realiza es llamar recursivamente a simplify para que simplifica la primera parte de la suma y luego la segunda parte. Una vez obtenido esto se llama a la función llamada simplify\_sum. Esta función se explicará en la sección III, pero básicamente lo que realiza es sumar ambas simplificaciones y da como resultado la suma.

- El cuarto caso es la simplificación de una multiplicación. Para este caso se simplifica cada una de las partes de la multiplicación recursivamente y después se multiplica el resultado de ambas.

```
simplify(X,X) :- atom(X).
simplify(N,N) :- number(N).

simplify(A+B,C) :-
    simplify(A,A1),
    simplify(B,B1),
    simplify_sum(A1+B1,C).

simplify(A*B,C) :-
    simplify(A,A1),
    simplify(B,B1),
    simplify_product(A1*B1,C).
```

### III. Simplify\_X

En esta sección se explicarán las funciones de apoyo del simplificador. Estas funciones son fundamentales para el funcionamiento de simplify. A continuación se explican las dos funciones que se presentan en el código.

- Simplify\_sum: Esta función tiene como objetivo simplificar sumas y tiene varios subcasos que se presentan a continuación.
  - En el primer caso, se indica que la suma de cero más una variable es la misma variable.
  - En el segundo caso se indica que la suma de una variable más cero es la misma variable.
  - En el tercer caso, si dos variables son números entonces la simplificación va a ser la suma de ambas.



- En el cuarto caso es la suma de variables, el resultado será lo mismo recibido como parámetro.
- **Simplify\_product:** Esta función tiene como objetivo simplificar multiplicaciones y tiene varios subcasos que se presentan a continuación.
  - El primer y segundo caso es para la multiplicación con cero. Cualquier cosa multiplicada por cero, dará como resultado cero.
  - El tercer y cuarto caso es para la multiplicación con uno. Si una variable o constante es multiplicada por uno, dará como resultado la misma variable o constante.
  - El quinto caso es para la multiplicación de dos números. Se determina si los parámetros recibidos son números y de ser así se multiplican.
  - El sexto caso es para la multiplicación de dos variables. Es la operación default, se da como resultado lo mismo obtenido como parámetros.

```
simplify_sum(0+A,A).
simplify_sum(A+0,A).
simplify_sum(A+B,C) :- number(A), number(B), C is A+B.
simplify_sum(A+B,A+B).

simplify_product(0*_,0).
simplify_product(_*0,0).
simplify_product(1*A,A).
simplify_product(A*1,A).
simplify_product(A*B,C) :- number(A), number(B), C is A*B.
simplify_product(A*B,A*B).
```

## 4.4. Ejemplos

Para probar el funcionamiento de este derivdaor se va a hacer uso del libro de Alexander Borboón Alpízar (2018) utilizado durante el curso de Cálculo Diferencial e Integral y otras fuentes. Para las 8 pruebas se escogieron ejemplos que demuestren el uso de las distintas reglas de derivación. A continuación se expondrán cada una de ellas.

- Prueba 1: Se utiliza el ejemplo propuesto por el autor de Juzrezky para probar el uso de expresiones trigonométricas y la mayoría de reglas de derivación. El programa no logra realizarlo, no cuenta con soporte de potencias ni expresiones trigonométricas.

```
?- deriv(x^(x*log(cos(x))), x, N).  
false.
```

- Prueba 2: Se utiliza el ejemplo 5 del libro de Domínguez (s.f, p.14) para probar el funcionamiento de la regla de la suma.

```
?- deriv(-5*x + 2, x, N).  
N = -5 .
```

- Prueba 3: Se utiliza el ejemplo 1 del “Regla del Producto – Fórmula, Demostración y Ejemplos” (2022) para probar el funcionamiento de la regla de la multiplicación.

$$f(x) = x(x + 5)$$

```
?- deriv(x *(x+5), x, N).  
N = x+(x+5) .
```

- Prueba 4: Se utiliza el ejemplo modificado de la página “Función de dos variables independientes - Derivadas” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de x.

$$f(x, y) = 4x + 5y$$

```
?- deriv(4*x + 5*y, x, N).
N = 4 .
```

- Prueba 5: Se utiliza el ejemplo modificado de la página “*Función de dos variables independientes - Derivadas*” (2016) para probar la función con dos variables distintas. En este caso se encuentra la derivada de y.

$$f(x, y) = 4x + 5y$$

```
?- deriv(4*x + 5*y, y, N).
N = 5 .
```

- Prueba 6: Se utiliza el ejemplo 6 del libro de Domínguez (s.f, p.14) para probar el funcionamiento de la regla de la suma y resta. Como se puede observar, el programa falla debido a que no soporta las restas ni las potencias.

$$f(x) = 2x^7 - 3x^6 + 3x^3 - 4x^2 - 7$$

```
?- deriv(2*x^7-3*x^6+3*x^3-4*x^2-7, x, N).
false.
```

- Prueba 7: Se utiliza el ejemplo 6 del libro de Domínguez (s.f, p.14) pero se modifica para que no tenga restas ni potencias y se puede observar que si funciona correctamente. Se prueba el uso de la regla de suma y multiplicación juntas.

```
?- deriv(2*x+3*x+3*x+4*x+7, x, N).
N = 12 .
```

- Prueba 8: Se utiliza el ejemplo 7 del libro de Domínguez (s.f, p.14) para probar la ley del cociente. Pero también se observa que el programa no logra realizarlo debido a la falta de casos de divisiones.

$$f(x) = \frac{x-3}{2}$$

```
?- deriv(x-3/2, x, N).
false.
```

#### 4.5. Simplificación

El programa sí cuenta con un simplificador de expresiones algebraicas. Este es capaz de simplificar multiplicaciones, sumas, constantes y variables. Sin embargo, no presenta ningún tipo de reglas de simplificación para la resta y la división. Estas operaciones son igual de importantes que el resto de las presentadas, por lo que limita bastante el funcionamiento del simplificador.

Por otro lado, al probar el funcionamiento del simplificador que se tiene, se observó que en la suma y en la multiplicación no se detecta si las variables son iguales o diferentes. Esto es importante ya que si se tiene  $x + x$  entonces el resultado debería poder ser  $2x$ . O con otro ejemplo, si se tiene  $y * y$  el resultado debería ser  $y^2$ . No obstante, se omitieron estos casos y nada más se dejaron iguales. Esto también representa una limitación, no tan grande como la falta de restas y divisiones, pero agregar estas simplificaciones ayudaría a la comprensión de los resultados dados por el simplificador.

#### **4.6. Análisis de los resultados**

Este programa cuenta con una solución al derivador distinta a las que se vieron anteriormente, es bastante interesante y es muy fácil de entender lo que se está realizando. A pesar de contar con un buen código, el programa presenta muchas limitaciones las cuales perjudican el funcionamiento del mismo. Principalmente, la falta de restas, divisiones y potencias es lo que más afectó. De hecho, no se conseguían pruebas de fuentes confiables debido a que normalmente las expresiones algebraicas que se busca derivar son mucho más complejas. Se puede concluir que este derivador funciona solo para expresiones algebraicas muy sencillas, por lo tanto no es muy útil.

## 5. Conclusiones

Durante la realización de este informe se pusieron a prueba tres algoritmos importantes que brindan distintas soluciones para el problema de diferenciación simbólica. La primera solución que se puso a prueba fue el conocido como Payne. Este presentaba una solución más amigable con el usuario, ya que se pide digitar los datos y el programa se encarga de llamar a las funciones correspondientes para la derivación. Además, la comprensión del mismo fue relativamente fácil. Por último, los resultados obtenidos fueron muy positivos. El programa logró ejecutar casi todas las pruebas que se le dieron y el único problema que presentó fue la simplificación de las expresiones algebraicas.

Para la segunda solución se utilizó Juzrczky. Esta solución no era tan amigable con el usuario pero tampoco fue difícil ejecutarlo. La comprensión del código tampoco fue difícil y los resultados obtenidos fueron muy buenos. Todos los casos que se pusieron a prueba fueron resueltos correctamente. El único problema que presentó esta solución es que no presentaba un simplificador, por lo que las expresiones hay que simplificarlas a mano.

Por último, para la tercera solución se usó Washington. Este programa es relativamente corto y no fue difícil de entender. Los resultados obtenidos con esta solución no fueron buenos. Se presentaron errores en muchos casos debido a que no se presentan reglas de derivación para la resta, la división y para las potencias. Además, el simplificador también era muy simple. No contaba con varias reglas de simplificación, afectando así el resultado obtenido al derivar.

## **6. Reflexión**

Durante la realización del presente proyecto se logro trabajar con un nuevo lenguaje de programación llamado Prolog. Este lenguaje, al igual que Standard ML y Haskell, es un poco diferente a lo que se conocía. No obstante, gracia a los conocimientos obtenidos durante las asignaciones pasada ayudaron a que no fuera tan desafiante aprender prolog. Aunque los tres lenguajes sean semejantes en algunas cosas, tienen diferencias muy marcadas.

Gracias al estudio del lenguaje durante las clases y a las programas estudiados se logró obtener un mejor conocimiento acerca de como funciona este. Trabajar en la resolución de esta asignación fue muy interesante ya que normalmente se acostumbra a programar lo que se pide de una vez pero en esta ocasión se empezó por analizar con calma como funcionaba cada programa y como funciona Prolog en general. Esto facilita mucho el aprendizaje. En general, trabajar con Prolog fue sencillo y muy enriquecedor. Además, trabajar con el paradigma de programación en lógica fue fácil y se aprendió mucho sobre este.

## 7. Reparto del Trabajo

**Tabla 1.** Reparto del trabajo

Sección del trabajo	Persona que lo realizó
Programa Payne	Gabriel Fallas
Programa Washington	Nikholas Ocampo
Programa Juzrczky	Valeria Chinchilla
Introducción	Gabriel Fallas
Conclusiones	Nikholas Ocampo
Reflexión	Valeria Chinchilla
Referencias	Gabriel Fallas, Nikholas Ocampo y Valeria Chinchilla
Apéndices	Gabriel Fallas, Nikholas Ocampo y Valeria Chinchilla



## 8. Referencias

Borbon Alpizar, A. (2018). *Calculo Diferencial e Integral*. Instituto Tecnológico de Costa Rica.

*Función de dos variables independientes - Derivadas*. (2016, 11 enero). AulaFacil.

<https://www.aulafacil.com/cursos/maticas/derivadas/funcion-de-dos-variables-independientes-l30898#:~:text=En%20una%20funci%C3%B3n%20con%20,se%20compar%C3%ADa%20como%20una%20constante>.

Hernandez, E. (2013). *Calculo Diferencial e Integral con aplicaciones*. Revista Digital Matematica, Educacion e Internet.

*Regla del Producto – Fórmula, Demostración y Ejemplos*. (2022, 3 junio). Neurochispas.

<https://www.neurochispas.com/wiki/regla-del-producto-formula-demostracion-y-ejemplos/>

Ruiz Domínguez, M. (s. f.). *100 derivadas resueltas*. Yo soy tu profe.

<https://yosoytuprofe.20minutos.es/wp-content/uploads/2018/03/100derivadasresueltasyosoytuprofe.pdf>

Jurczyk, W. Symbolic Differentiation Implementation in Prolog

Paine, J. 'SYMDIFF'. Symbolic Differentiation and Algebraic Simplification.

<https://www.j-paine.org/prolog/mathnotes/files/symdiff.pl>

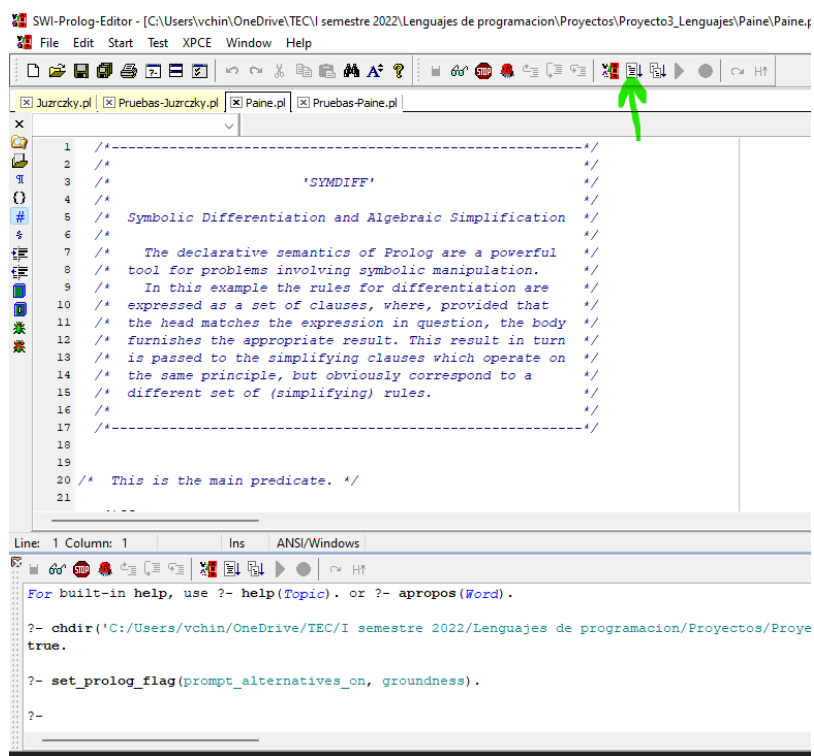
Washington Symbolic differentiation in Prolog.

<https://courses.cs.washington.edu/courses/cse341/18sp/prolog/deriv.txt>

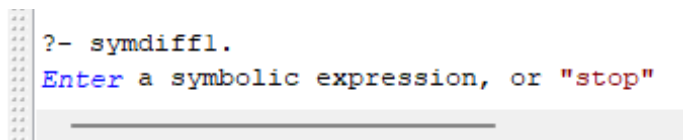
## 9. Apéndices

### Apéndice A: Payne

- Instrucciones para correr el programa
1. Abrir SWI-Prolog-Editor
  2. Abrir el archivo Pruebas-Paine y Paine
  3. Ir al archivo paine y cargarlo en la línea de comandos



4. Colocar symdiff1. Y dar enter.



5. Ir al archivo de pruebas y copiar la prueba que se desea usar. Luego pegarla en la línea de comandos y dar enter. Después, se la da la variable sobre la cual se desea derivar y se da enter de nuevo. Luego ya se podrá ver el resultado.

```
?- symdiff1.  
Enter a symbolic expression, or "stop" 2*x^7 - 3*x^6 + 3*x^3 - 4*x^2 - 7.  
Differentiate w.r.t. :|: x.  
  
Differential w.r.t x is 2*(7*x^6)-3*(6*x^5)+3*(3*x^2)-4*(2*x)
```

- Código Fuente

```

/*-----*/
/*
/*          'SYMDIFF'
/*
/* Symbolic Differentiation and Algebraic Simplification */
/*
/* The declarative semantics of Prolog are a powerful */
/* tool for problems involving symbolic manipulation. */
/* In this example the rules for differentiation are */
/* expressed as a set of clauses, where, provided that */
/* the head matches the expression in question, the body */
/* furnishes the appropriate result. This result in turn */
/* is passed to the simplifying clauses which operate on */
/* the same principle, but obviously correspond to a */
/* different set of (simplifying) rules.
/*
/*-----*/

/* 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 */

```

```

/* The following clauses are rules for the simplification */
/* of the result of the differentiation process. For */
/* example, multiples of zero are dropped, terms gathered */
/* together where possible and factorisation is attempted. */
/* (It should be noted that in this program only adjacent */
/* terms are candidates for simplification.) */

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 simplification 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... */

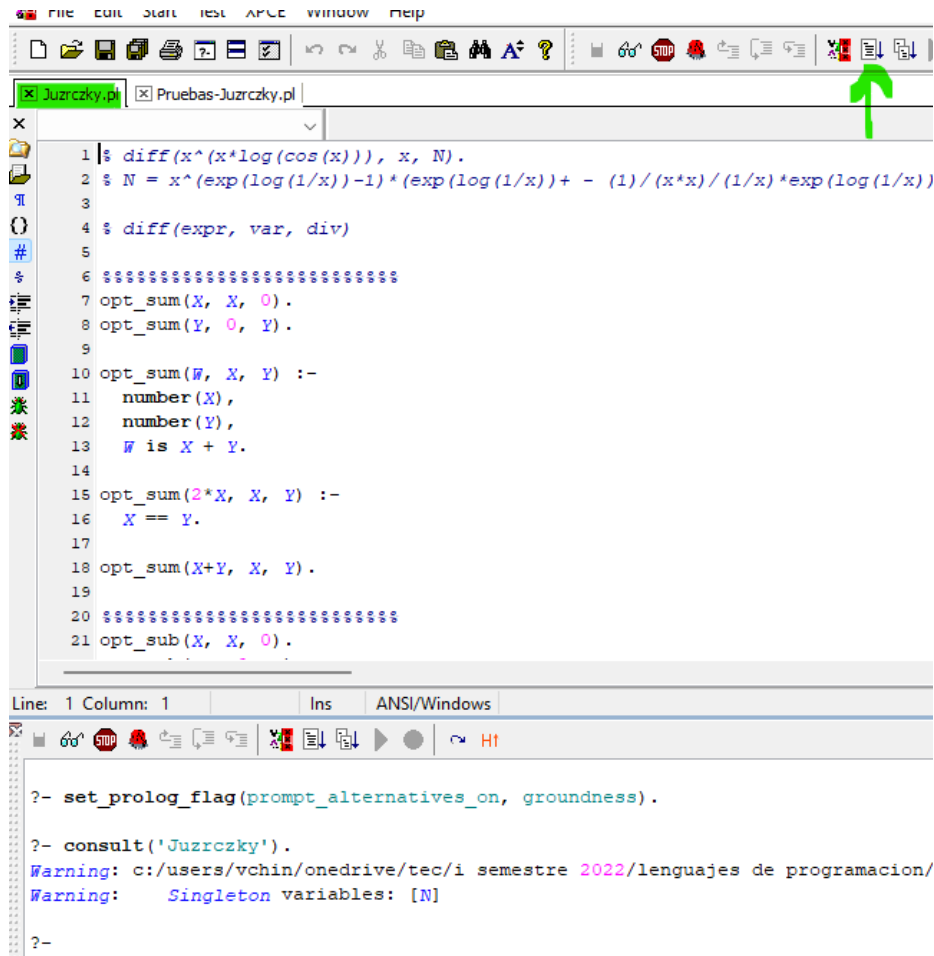
numeric(A) :- integer(A).

/* numeric(A) :- real(A). José Daniel Jiménez Barahona 2020.08.18 */
numeric(A) :- float(A).

```

## Apéndice B: Juzrczky

- Instrucciones para correr el programa
1. Abrir SWI-Prolog-Editor
  2. Abrir el archivo Pruebas-Juzrczky y Juzrczky
  3. Ir al archivo Juzrczky y cargarlo en la línea de comandos



The screenshot shows the SWI-Prolog-Editor interface. The top menu bar includes FILE, EDIT, STATE, TEST, APPLE, WINDOW, and HELP. Below the menu is a toolbar with various icons. The main window displays the file 'Juzrczky.pl' with the following Prolog code:

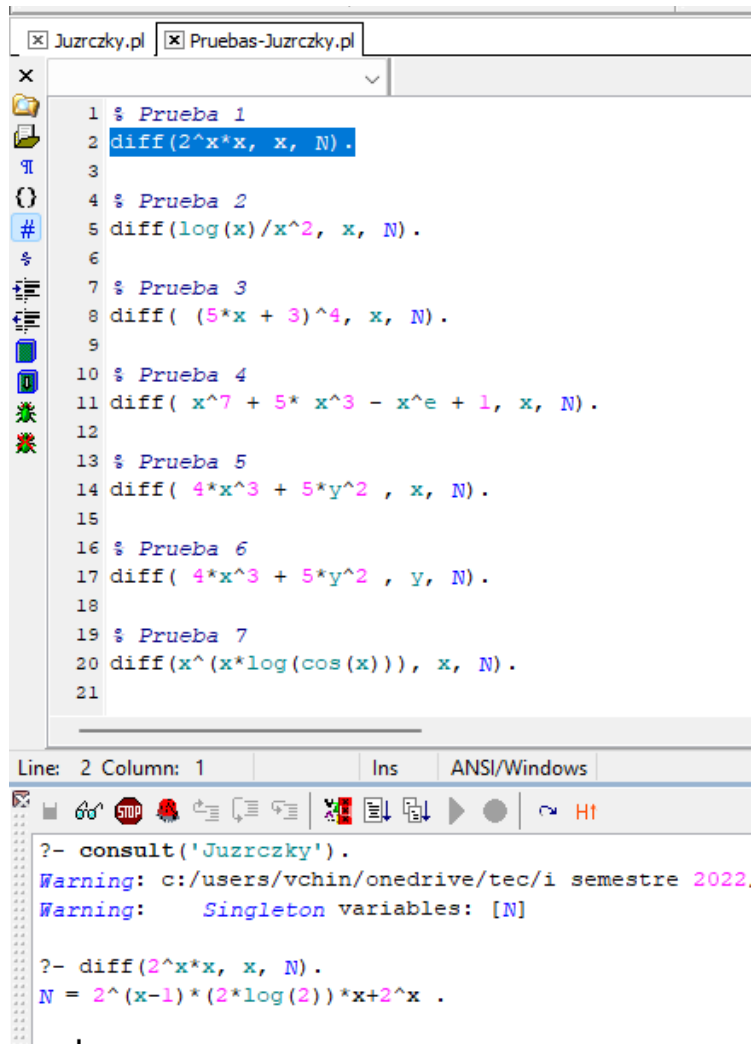
```
1 | % diff(x^(x*log(cos(x))), x, N).
2 | % N = x^(exp(log(1/x))-1)*(exp(log(1/x))+ - (1)/(x*x)/(1/x)*exp(log(1/x))
3 |
4 | % diff(expr, var, div)
5 |
6 | %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7 | opt_sum(X, X, 0).
8 | opt_sum(Y, 0, Y).
9 |
10 | opt_sum(W, X, Y) :-
11 |     number(X),
12 |     number(Y),
13 |     W is X + Y.
14 |
15 | opt_sum(2*X, X, Y) :-
16 |     X == Y.
17 |
18 | opt_sum(X+Y, X, Y).
19 |
20 | %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
21 | opt_sub(X, X, 0).
```

The bottom window shows the command line interface with the following text:

```
?- set_prolog_flag(prompt_alternatives_on, groundness).
?- consult('Juzrczky').
Warning: c:/users/vchin/onedrive/tec/i semestre 2022/lenguajes de programacion/
Warning: Singleton variables: [N]
?-
```

4. Copiar la prueba y pegarla en la línea de comandos, dar enter y listo.





```
1 % Prueba 1
2 diff(2^x*x, x, N).
3
4 % Prueba 2
5 diff(log(x)/x^2, x, N).
6
7 % Prueba 3
8 diff((5*x + 3)^4, x, N).
9
10 % Prueba 4
11 diff(x^7 + 5*x^3 - x^e + 1, x, N).
12
13 % Prueba 5
14 diff(4*x^3 + 5*y^2, x, N).
15
16 % Prueba 6
17 diff(4*x^3 + 5*y^2, y, N).
18
19 % Prueba 7
20 diff(x^(x*log(cos(x))), x, N).
21
```

Line: 2 Column: 1    Ins    ANSI/Windows

```
?- consult('Juzrczky').
Warning: c:/users/vchin/onedrive/tec/i semestre 2022.
Warning: Singleton variables: [N]

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

- Código Fuente

```

% 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, Ed2),
    opt_mul(B, E1, Ed2),
    opt_mul(C, E2, Ed2),
    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)).

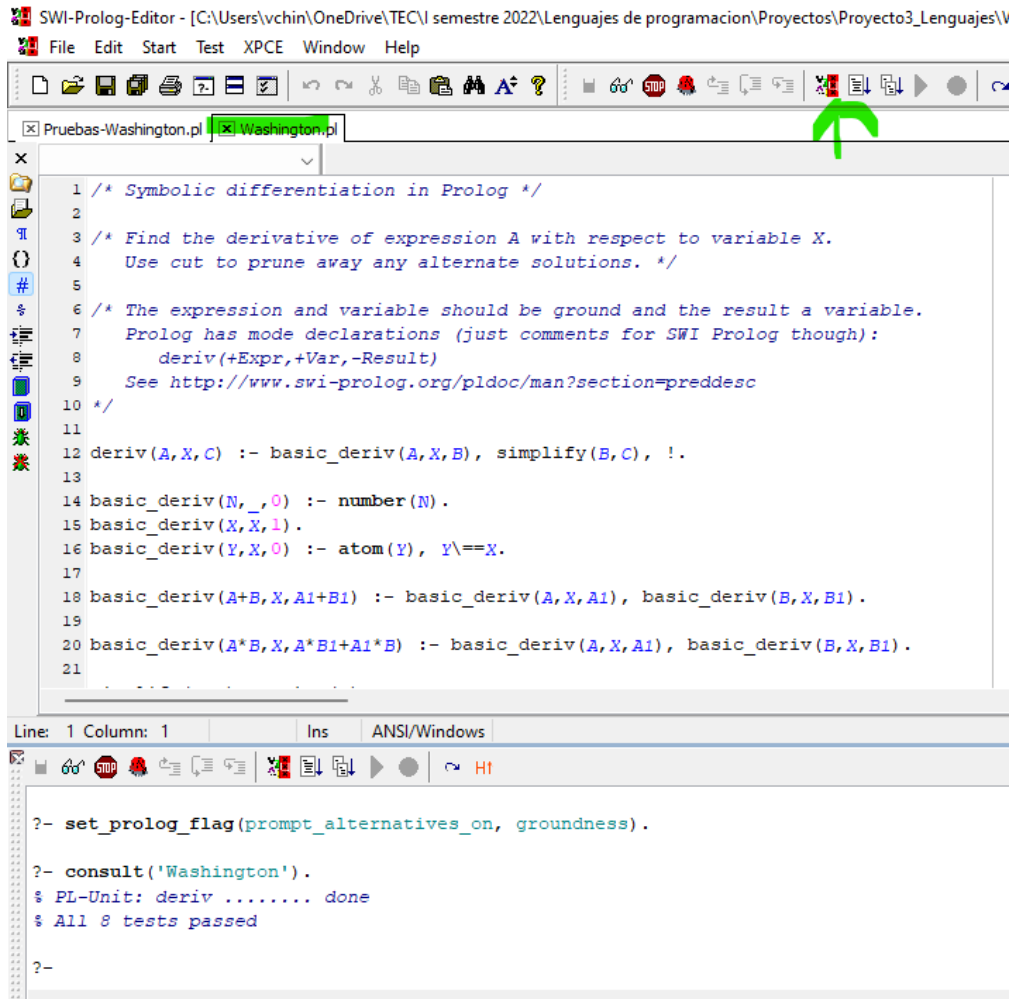
diff(log(E), V, F) :-
    diff(E, V, Ed),
    opt_div(F, Ed, E).

% (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(EF, BB, DD),
    opt_mul(FF, F^(AA), EF).

```

## Apéndice C: Washington

- Instrucciones para correr el programa
1. Abrir SWI-Prolog-Editor
  2. Abrir el archivo Pruebas-Washington y Washington
  3. Ir al archivo Washington y cargarlo en la línea de comandos



SWI-Prolog-Editor - [C:\Users\vchin\OneDrive\TEC\I semestre 2022\Lenguajes de programacion\Proyectos\Proyecto3\_Lenguajes\W

File Edit Start Test XPCE Window Help

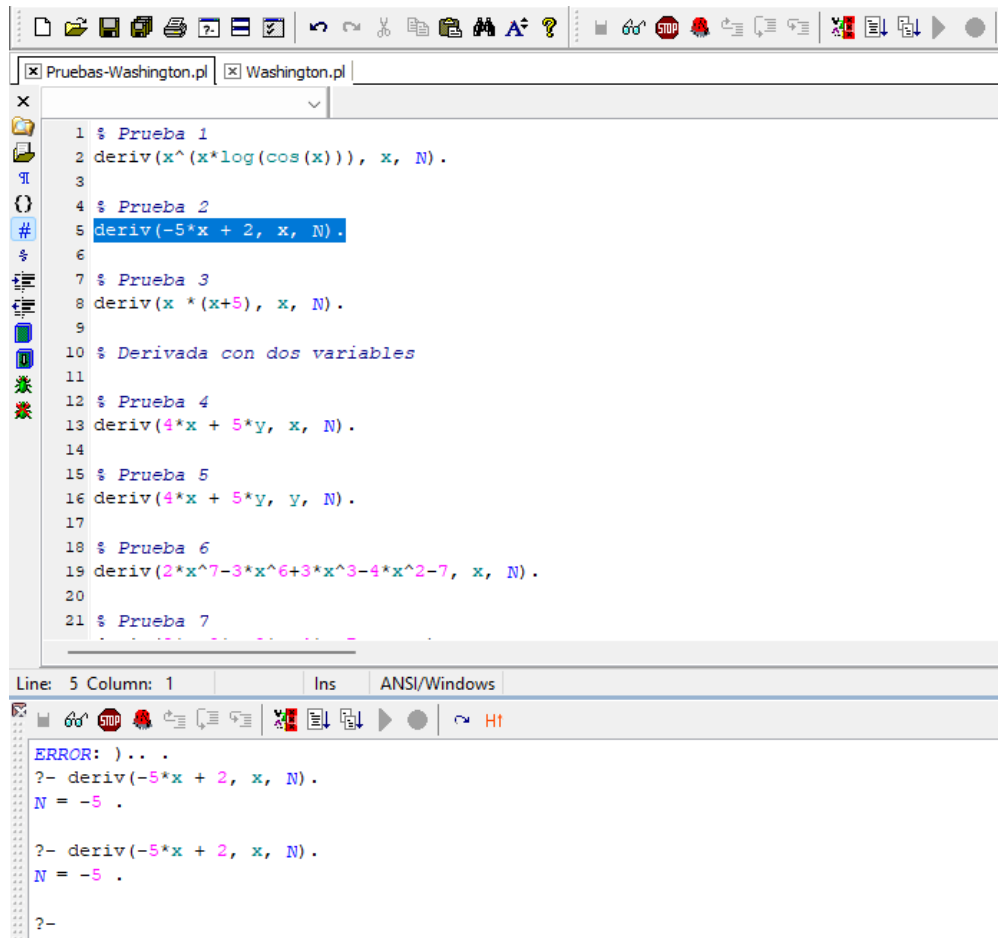
Pruebas-Washington.pl Washington.pl

```
1 /* Symbolic differentiation in Prolog */
2
3 /* Find the derivative of expression A with respect to variable X.
4    Use cut to prune away any alternate solutions. */
5
6 /* The expression and variable should be ground and the result a variable.
7    Prolog has mode declarations (just comments for SWI Prolog though):
8    deriv(+Expr,+Var,-Result)
9    See http://www.swi-prolog.org/pldoc/man?section=preddesc
10 */
11
12 deriv(A,X,C) :- basic_deriv(A,X,B), simplify(B,C), !.
13
14 basic_deriv(N,_0) :- number(N).
15 basic_deriv(X,X,1).
16 basic_deriv(Y,X,0) :- atom(Y), Y\==X.
17
18 basic_deriv(A+B,X,A1+B1) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).
19
20 basic_deriv(A*B,X,A*B1+A1*B) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).
21
```

Line: 1 Column: 1 Ins ANSI/Windows

```
?- set_prolog_flag(prompt_alternatives_on, groundness).
?- consult('Washington').
% PL-Unit: deriv ..... done
% All 8 tests passed
?-
```

4. Copiar la prueba y pegarla en la línea de comandos. Dar enter y se obtiene el resultado.



The screenshot shows a MATLAB editor window with a script file named 'Pruebas-Washington.pl'. The script contains seven tests, each starting with a comment line '% Prueba X' followed by a 'deriv' function call. The second test, 'Prueba 2', is highlighted in blue and shows the expression 'deriv(-5\*x + 2, x, N)'. The command window at the bottom shows the output of this test, which is 'N = -5'.

```
1 % Prueba 1
2 deriv(x^(x*log(cos(x))), x, N).
3
4 % Prueba 2
5 deriv(-5*x + 2, x, N).
6
7 % Prueba 3
8 deriv(x*(x+5), x, N).
9
10 % Derivada con dos variables
11
12 % Prueba 4
13 deriv(4*x + 5*y, x, N).
14
15 % Prueba 5
16 deriv(4*x + 5*y, y, N).
17
18 % Prueba 6
19 deriv(2*x^7-3*x^6+3*x^3-4*x^2-7, x, N).
20
21 % Prueba 7
```

Line: 5 Column: 1    Ins    ANSI/Windows

```
ERROR: )..
?- deriv(-5*x + 2, x, N).
N = -5 .

?- deriv(-5*x + 2, x, N).
N = -5 .

?-
```

- Código Fuente

```

/* Symbolic differentiation in Prolog */

/* Find the derivative of expression A with respect to variable X.
   Use cut to prune away any alternate solutions. */

/* The expression and variable should be ground and the result a variable.
   Prolog has mode declarations (just comments for SWI Prolog though):
   deriv(+Expr,+Var,-Result)
   See http://www.swi-prolog.org/pldoc/man?section=preddesc
*/

deriv(A,X,C) :- basic_deriv(A,X,B), simplify(B,C), !.

basic_deriv(N,_,0) :- number(N).
basic_deriv(X,X,1).
basic_deriv(Y,X,0) :- atom(Y), Y\==X.

basic_deriv(A+B,X,A1+B1) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).

basic_deriv(A*B,X,A*B1+A1*B) :- basic_deriv(A,X,A1), basic_deriv(B,X,B1).

simplify(X,X) :- atom(X).
simplify(N,N) :- number(N).

simplify(A+B,C) :-
    simplify(A,A1),
    simplify(B,B1),
    simplify_sum(A1+B1,C).

simplify(A*B,C) :-
    simplify(A,A1),
    simplify(B,B1),
    simplify_product(A1*B1,C).

simplify_sum(0+A,A).
simplify_sum(A+0,A).
simplify_sum(A+B,C) :- number(A), number(B), C is A+B.
simplify_sum(A+B,A+B).

simplify_product(0*_,0).
simplify_product(_*0,0).
simplify_product(1*A,A).
simplify_product(A*1,A).
simplify_product(A*B,C) :- number(A), number(B), C is A*B.
simplify_product(A*B,A*B).

:- begin_tests(deriv).

test(const) :- deriv(3,x,0).
test(x) :- deriv(x,x,1).
test(y) :- deriv(y,x,0).
test(plus) :- deriv(x+3,x,1).
test(plus) :- deriv(x+y,x,1).
test(plus_unsimp) :- deriv((2+3)*x,x,5).
test(times) :- deriv(10*(x+3), x, 10).
test(times) :- deriv((x*y)*(x+3), x, (x*y)+(y*(x+3))).

:- end_tests(deriv).

:- run_tests.

```