QUANTUM THEORY PROJECT
FOR RESEARCH IN ATOMIC, MOLECULAR, AND SOLID STATE
CHEMISTRY AND PHYSICS
UNIVERSITY OF FLORIDA, GAINESVILLE, FLORIDA

LISP FUNCTIONS TO CALCULATE DERIVATIVES AND POISSON BRACKETS

By

Jose Barberan*

PROGRAM NOTE # 3

28 February 1963

* Student, University of Mexico

ACKNOWLEDGEMENTS

This paper has been prepared as a portion of the exercises in an informal winter institute in symbolic programming held by Mr. H.V. McIntosh and The Quantum Theory Project at the University of Florida. The author wishes particularly to express his gratitude to the Physics Department for its kind hospitality during his stay, and the University Computer Center for the time which was used to verify the functions described in this paper. This time was made available by a grant from the National Science Foundation, and the cooperation of the IBM corporation.

José Barberán

Gainesville, Fla.
28 February, 1963.

ABSTRACT

LISP, since it is a symbol manipulation language, may be used to calculate the derivatives of symbolic algebraic expressions by systematically applying the well - known rules for differentiating sums, products, powers, composite functions and so on. In this paper a detailed description of such a function is given, as well as of its necessary satellites, and certain closely related functions, such as the gradient, divergence and curl. LISP function whose purpose is the calculation of Poisson Brackets is similarly related, but given a separate treatment since a considerable saving may be realized by working directly from the axioms satisfied by the Poisson Bracket, which state that it is an alternating bilinear form which is a derivative in each of its arguments.

L I S P

(List Processor)

El LISP es un lenguaje de programación simbólica que basado en un conjunto de cinco funciones primitivas permite la manipulación de expresiones en forma de listas para plantear y resolver cualquier problema calculable. Es un sistema mínimo ya que usa la menor cantidad posible de funciones primitivas para ser completo. Es lógicamente completo en tanto que no tiene restringido su campo de acción a determinado tipo de problemas, sino que puede ser usado para resolver las más desconectadas cuestiones; con LISP se pueden resolver simbólicamente desde acertijos matemáticos, problemas de teoría de grupos, desarrollo simbólico de determinantes, etc. hasta problemas de cálculo diferencial.

En este reporte serán descritos más adelante los programas para calcular la derivada de una función y para resolver los Paréntesis de Poisson de dos polinomios, no con manipulaciones numéricas, sino simbólicamente.

Toda la notación de LISP esta constituida por listas, se define técnicamente a una lista como una serie de elementos cerrada entre paréntesis. Los elementos contenidos en una lista pueden ser a su vez listas o símbolos atómicos. Se llama símbolo atómico a una sucesión ininterrumpida por paréntesis o espacios

de caracteres disponibles en una máquina perforadora de tarjetas.

Un ejemplo de lista es

((CA) CHARRO (PE MEX))

Ejemplo de símbolos atómicos son:

54321BANG DERIVADA

Para representar una función en LISP se usa notación polaca con modificaciones, en la que una función es una lista cuyo primer elemento es un símbolo atómico que es el nombre de la función y los restantes elementos son los argumentos de la función. Por ejemplo

3 + A+ 3X

en notación de LISP es

(+ 3 A (* 3 X))

donde el asterisco es la función producto. En este ejemplo se puede observar la posibilidad de combinar funciones, es decir, de usar como argumentos de una función listas que a su vez son funciones de otros argumentos. Esta posibilidad de construir funciones de funciones es la que permite elaborar cualquier programa en LISP por medio de combinaciones de las llamadas funciones primitivas.

LAS FUNCIONES PRIMITIVAS

Las funciones primitivas son cinco, dos de las cuales son predicados, es decir, funciones que solo pueden tomar dos

valores (T) o (F) que se hacen corresponder a Verdadero (True) y Falso (False), las otras tres se usan para rearreglar listas.

El primer predicado, (ATOM X) tiene un solo argumento que puede ser cualquier expresión: toma el valor (T) cuando su argumento es un símbolo atómico y (F) cuando es una lista. Ejemplos:

(ATOM CU) = (T)

(ATOM (A B C)) = (F)

El segundo, (EQ X Y), tiene dos argumentos y toma el valor (T) si son símbolos atómicos idénticos, en otros casos (F). Ejemplos:

(EQ NI NA) = (F)

(EQ NA NA) = (T)

(EQ (NA) (NA)) = (F)

Las otras tres funciones son CAR, CDR y CONS; las dos primeras deben tener como argumento a una lista, CAR da el primer elemento de la lista y CDR da la lista de los restantes. Ejemplos:

(CAR L) = A
(CDR L) = (B C D)
(CAR (CDR L)) = B

en donde la lista L es

(A B C D)

Si una lista tiene un solo elemento, su CDR es una lista sin elementos, la llamada lista vacia. Ejemplo:

(CDR (BR)) = ( )

El CDR de una lista vacia no esta definido.

Por comodidad introducimos una abreviación para escribir las combinaciones de CAR y CDR que consiste en escribir (CADR L) en lugar de (CAR (CDR L)) , (CADDR L) en lugar de (CAR (CDR (CDR L))), etc., es decir, las combinaciones con A y D de cuatro o menos elementos puestos entre una C y una R indican los calculos sucesivos de CAR y CDR.

Estas dos funciones se usan para disectar una lista, y la tercera función, CONS, sirve para sintetizar una lista; tiene dos argumentos, el primero puede ser cualquier expresión, el segundo debe ser una lista. (CONS A B) forma una lista cuyo primer elemento es A y los restantes son los elementos de B. Ejemplos:

(CONS X ( )) = (X)

Si L = (A B C)
(CONS X L) = (X A B C)
(CONS (CAR L) (CDR L)) = L

La última igualdad es una identidad, es decir, que se cumple para toda lista.

Es sorprendente como solo con combinaciones de estas 5 funciones primitivas y con ayuda de las llamadas funciones

de programación se pueden construir funciones de gran complejidad.

LAS FUNCIONES DE PROGRAMACION

Las funciones de programación son LAMBDA, COND, DEFINE, AND, OR, NOT, NULL, LIST y QUOTE y serán descritas a continuación:

Al definir o construir una función compuesta que tenga varias variables, surge cieta ambigüedad respecto a la correspondencia entre las variables de la función y los símbolos empleados en la definición, así por ejemplo, si $f(x, y) = {}^{y}/_{x+y^2}$, f (a , b) puede ser ${}^{b}/_{a+b^2}$ o bien ${}^{a}/_{b+a^2}$. Para evitar esta confusión empleamos una forma de definición tomada de la lógica simbólica que por medio de la expresión LAMBDA da un apareamiento entre las variables y los símbolos usados en la definición. Para escribir una definición escribimos una lista cuyo primer elemento es el símbolo atómico LAMBDA, a continuación está la lista de variables mudas que aparecen en la definición, y despues la expresión que por medio de combinaciones de funciones define a la nueva función, esto es:

(LAMBDA (x y z . . .) (Expresión))

si a continuación de esta lista ponemos la lista de los argumentos (A_1 A_2 A_3 . . .) para los cuales va a ser calculada la función, la variable muda x que aparece en la definición será substituida por el valor A_1 , la y por A_2 y así sucesivamente

evitando de esta manera cualquier ambigüedad.

En la definición de una función, el proceso a seguir con los argumentos, dependerá la mayoría de las veces de la naturaleza de estos, por lo cual es necesario probar de que tipo son; para hacer esto usamos predicados y dividimos con ellos el conjunto de todos los posibles argumentos de la función en subconjuntos de argumentos para los cuales un predicado toma el valor (T) (verdadero); para cada uno de estos subconjuntos corresponderá una función a aplicar; así definimos la función condicional COND que prueba si un predicado, digamos P_1, toma el valor (T) para los argumentos, en caso afirmativo aplica la función correspondiente, digamos F_1, si no, procede a probar otro predicado P_2 para ver la posibilidad de aplicar la siguiente función F_2 y así sucesivamente. Esta función se escribe de la siguiente manera:

(COND $(P_1\ F_1)$ $(P_2\ F_2)$ $(P_3\ F_3)$ . . .)

y tiene como valor el correspondiente a la función apareada con el primer predicado verdadero.

Un programa en LISP consta de dos partes: Una serie de definiciones puestas como argumentos de la función DEFINE y la instrucción para la aplicación de las funciones definidas a valores particulares de los argumentos por medio de la función universal APPLY, ambas funciones DEFINE y APPLY serán descritas a continuación.

En LISP se usa un artificio para asociar un nombre que es un símbolo atómico a una función construida con combinaciones de otras, de manera que cuando se quiera hacer uso de ella y ya se haya definido solo sea necesario poner su nombre seguido de los argumentos, en lugar de poner toda la definición seguida de los argumentos. Este artificio se logra con la función DEFINE que asocia un símbolo atómico (el nombre de la función) a una definición. DEFINE tiene un número indefinido de argumentos, cada argumento es una lista cuyo primer elemento es el nombre de la función y el segundo la definición.

Este artificio permite la construcción de definiciones recurrentes esto es, de definiciones en las que aparece el nombre de la función que se esta definiendo. Dicha técnica implica un condicional en el que podemos distinguir 2 partes: una, la llamada condición terminal que define la función para cierto argumento conocido, y la otra, la parte iterativa que puede reaplicar la definición a nuevos argumentos o a los mismos en una nueva combinación. Por ejemplo podemos definir recurrentemente la función factorial diciendo que si $n = 0$, $n! = 1$ (condición terminal), pero en otros casos se define $n!$ como $n(n-1)!$ (parte iterativa). Obviamente para que una definición recurrente tenga sentido es necesario que después de un número finito de pasos se llegue a la condición terminal.

La función APPLY es la instrucción para calcular el valor de una función para determinados argumentos dados. Es una función de 2 argumentos, el 1o es un símbolo atómico y es el nombre de la función que se va a calcular y que ya debe estar definida, el 2o es la lista de los argumentos para los que se va a calcular dicha función.

En LISP hay una serie de metafunciones incluidas a veces entre las funciones de programación, que completan los cimientos para construir la totalidad de un programa en LISP; estas son AND, OR, NOT, NULL, LIST, QUOTE y FUNCTION que serán descritas brevemente a continuación:

Las tres primeras son predicados tomados de la lógica y tienen el mismo significado que Y, O y NO en lógica. Los argumentos de estas funciones son predicados.

AND tiene cualquier número de argumento y es verdadero solo si todos sus argumentos lo són.

OR también tiene cualquier número de argumentos y es verdadero si alguno de sus argumentos lo es.

NOT tiene un solo argumento y toma el valor contrario al valor de su argumento.

También tomados de la lógica, en LISP, se usan los predicados (T) (verdadero) y (F) (falso) que no tienen argumentos y que siempre tienen el mismo valor: verdadero y falso respectivamente.

NULL es otro predicado que solo es verdadero cuando su argumento es una lista vacia.

(LIST A B . . .) es una función de un número indefinido de argumentos y da la lista de los valores de sus argumentos. (LIST) sin argumentos representa a la lista vacia.

La función (QUOTE L) siempre tiene un argumento y da como resultado su mismo argumento.

La diferencia entre estas 2 funciones es que mientras la primera evalúa cada uno de los argumentos, la segunda los reproduce tal y como son, como se ve sobre los ejemplos a continuación:

(LIST (CAR (A B)) (ATOM ABC))=(A (T))

(QUOTE (CAR (A B))=(CAR (A B))

(QUOTE (ATOM (ABC))=(ATOM (ABC))

Cuando en una definición aparece como argumento una función o combinación de funciones de varios argumentos, es necesario poner dicho argumento de la siguiente manera:

(FUNCTION (LAMBDA (argumentos) (función)))

Donde el átomo FUNCTION indica que dicho argumento es una función y que es descrita a continuación de él.

Hasta aquí han sido descritas las funciones necesarias y suficientes para construir cualquier programa en LISP por simple combinación de ellas.

EL PROGRAMA PARA CALCULAR SIMBOLICAMENTE LA DERIVADA DE UNA FUNCION

Supongamos que queremos calcular la derivada de una función y le damos el problema a un calculista sin inventiva. El, por lo general, no tratará de resolverlo por medio de la definición de derivada en términos del concepto de límite, sino que buscará en su memoria o entre sus libros una tabla de derivadas, localizará en ella la fórmula que corresponda a la función que se le encomendó, la aplicará y tratará de simplificar el resultado en la medida de sus posibilidades; si por desgracia la función que le dimos no estuviera en su tabla, nos devolvería el problema diciendonos: "la derivada de la función f es df/dx" dándonos asi un resultado correcto aunque no muy satisfactorio pues la derivada quedaría solamente indicada.

Nosotros podemos hacer con LISP un programa que haga exactamente el mismo trabajo que dicho calculista, con la diferencia de que será más rápido y nunca tendrá errores. Dicho programa consistirá en una tabla condicional en la que a cada tipo de función prevista corresponderá la instrucción de la fórmula correspondiente que da el cálculo diferencial. Si la función a derivar no esta prevista entre las condiciones, por medio de una condición final lograremos que el programa nos deje indicada la derivada de la misma forma que la haría nuestro calculista sin inventiva.

Pasemos a describir con cierto detalle el programa de LISP para derivar funciones.

En LISP la función que calcula y simplifica la derivada de una función se llama (DIFFERENTIATE X L), tiene 2 argumentos, la varible con respecto a la que se va a derivar y la función que se va a derivar puesta explicitamente y en notación polaca (en forma de lista). DIFFERENTIATE consta de 2 partes perfectamente separadas, la primera ejecuta la derivada propiamente dicha y la segunda efectua simplificaciones elementales sobre el resultado de la primera. Dichas funciones se llaman respectivamente (DERIVATIVE X L) y (SIMPLIFY A). Por lo tanto la función DIFFERENTIATE se escribe como combinación de estas dos:

```
(DIFFERENTIATE (LAMBDA (X L) (SIMPLIFY (DERIVATIVE X L))))
```

La función DERIVATIVE está formada por una tabla en la que a cada tipo de función (suma, producto, cociente, potencia, etc.) corresponde una diferente instrucción.

Esta tabla es construida por medio de la función condicional; en cada condición el predicado prueba si la función es de tal o cual tipo y la expresión siguiente da la instrucción para aplicar la formula correspondiente de la derivada. En dicha instrucción se supondrá siempre que la función a derivar es una función de función, es decir, la definición será recurrente y

despues de derivar la función original se derivarán los argumentos que a su vez pueden ser funciones; el proceso de penetración termina en la expresión mínima, es decir, en los simbolos atómicos, que como la función esta en forma explícita, podemos decir que o son constantes y entonces su derivada será cero o que son iguales a la variable con respecto a la cual se esta derivando, entonces su derivada será uno. Por lo tanto la condición terminal de esta definición recurrente es:

```
((ATOM  L)  (COND  ((EQ  X  L)  (QUOTE  1))
                   ((T)  (QUOTE  O))
```

donde (T) equivale a: en otros casos, ya que siempre es verdadero.

Las demás condiciones contendrán las fórmulas para derivar diferentes tipos de funciones. La definición de máquina de la función DERIVATIVE esta en el apéndice, pero su estructura puede aclararse por medio de un esquema que se dá a continuación.

PREDICADO	RESULTADO
(A) (ATOM L)	CONDICIONAL: PREDICADO: L = X → RESULTADO: 1; En otros casos → 0

Si L es del tipo

	NOTACION POLACA	NOTACION ORDINARIA	NOTACION ORDINARIA	NOTACION POLACA
(B)	(+ A B)	A+B	A' + B'	(+ A' B')
(C)	(+ A B C ...)	A+B+C+...	A'+B'+C'+...	(+ A' B' C' ...)
(D)	(- A B)	A-B	A' - B'	(- A' B')
(E)	(* A B)	AB	A'B + AB'	(+ (* A' B) (* A B'))
(F)	(* A B C ...)	ABC...	A'BC...+AB'C...	(+ (* A' BC...) (...)
(G)	(/ A B)	A/B	(A'B - AB')/B²	(/ (- (* A' B) (* AB')) (** B 2))
(H)	(** A N)	A^N	N A^{N-1} A'	(* N (** A (- N 1)) A')
(I)	(- A)	- A	- A'	(- A')
(J)	En otros	casos	$\partial L/\partial X$	(DERIVATIVE X L)

Donde L' es abreviación de (DERIVATIVE X L) = $\partial L/\partial X$;

A, B y C son expresiones que pueden o no depender de X y N es independiente de X.

Notese que la función producto se representa por un asterisco y la función potencia por dos asteriscos.

A este conjunto de condiciones podemos agregar otros para completar la tabla, evitando así que las derivadas de algunas funciones queden tan solo indicadas. Como ejemplos de condiciones que se pueden agregar tenemos:

PREDICADO		RESULTADO	
NOTACION POLACA	NOTACION ORDINARIA	NOTACION ORDINARIA	NOTACION POLACA
(sen A)	sen A	A' cos A	(* A' (cos A))
(cos A)	cos A	-A' sen A	(-(* A' (sen A)
(arc sen A)	arc sen A	$A'/\sqrt{1-A^2}$	(/A'(**(-1(**A 2)) (/ 1 2)))
(arc cos A)	arc cos A	$-A'/\sqrt{1-A^2}$	(-(/ A'(**(-1(**A 2)) (/ 1 2))))
(exp A)	e^A	e^A A'	(* (exp A) A')
(log N A)	$\log_N A$	$A'/A \log_e N$	(/ A'(* A(log E N)))

Obviamente estas condiciones agregadas deben ser puestas antes de la condición final que tiene la instrucción para dejar indicada la derivada.

En los predicados para probar si L es de tal o cual tipo, se usa por lo general el predicado similar que determina tipos de listas de acuerdo con listas modelos en las que aparecen guiones que son similares a cualquier cosa; por ejemplo (+ - -) es similar a (+ A B), (log - -) es similar a (log 10 6), etc.

La definición rigurosa de la función DERIVATIVE y la descripción de las funciones elementales, como SIMILAR y otras, que se usan en ella, aparecen en el apéndice.

La construcción de las instrucciones para efectuar la derivada, como se puede ver en la definición, no presenta ningun problema y no amerita ser discutida. Solo hablaremos brevemente de dos de ellos, que usan funciones satélites.

Al derivar un producto de varios factores, en la condición (F), necesitamos sumar los productos que contienen a la derivada de cada factor; para esto usamos la función DIAG que aplica una función a cada uno de los elementos de una lista de la siguiente manera:

(DIAG G (A B C D))= (((G A) B C D)
(A (G B) C D)
(A B (G C) D)
(A B C (G D)))

si usamos como función G a la función DERIVATIVE y sumamos los resultados obtenemos:

(+(*A' B C D) (*A B' C D) (*A B C' D) (*A B C D'))

que es la fórmula que se necesita.

Al derivar una función de la forma (** A N) = A^n en la condición (M) basta con dar la instrucción para construir el siguiente resultado

(* N (** A (- N 1)) * (DERIVATIVE X A))= $NA^{n-1}A'$

Esto se hará asi si N no es un entero, pero si sí lo es, para dar un resultado mas simplificado, evaluamos N-1, es decir, calculamos el entero anterior a N. Para ello, como primer paso averiguamos si N es entero, esto se hace con el predicado (NUMERAL N) (ver su definición en el apéndice) que es verdadero si N esta en la lista INTEGERS, esta lista es una función elemental de l LISP que no tiene argumentos y su valor es la lista de los enteros de 0 a 60. Si N no esta en la lista, el resultado se da en la forma mencionada, en caso contrario procedemos a calcular su predecesor en los enteros por medio de la función (PREDECESOR N)

(definida en el apendice) queda el anterior a N, es decir N - 1. Con este artificio logramos un resultado ligeramente más simplificado al derivar potencias.

Analicemos ahora la segunda parte del programa para derivar, es decir la función (SIMPLIFY L) esta función al igual que DERIVATIVE esta formada por una tabla condicional en la que se preveen las funciones que se pueden simplificar y se da la instrucción para su simplificación. Es necesario que esta función efectue las simplificaciones triviales de sumas, diferencias, productos, cocientes y potencias y todas sus combinaciones por intrincadas que sean; al decir triviales nos referimos a simplificaciones del tipo de multiplicaciones por cero y por uno, sumas con cero, elevación a las potencias cero y uno, etc.

Para lograr esto, es necesario que la función SIMPLIFY penetre en los argumentos de las diferentes funciones aritméticas, esto es que si la función a simplificar es por ejemplo un producto, antes de ser simplificado sea simplificado cada uno de los factores, y si estos factores son a su vez funciones de funciones el proceso de penetración continue hasta topar con expresiones mínimas no simplificables, es decir son símbolos atómicos que seran simplemente constantes o variables. Por lo tanto la definición de esta función será recurrente (vease la definición en el apendice) y la condición terminal será:

((ATOM L) L)

es decir, los símbolos atómicos no son simplificables. Si la función L no es un átomo empieza la recursión que se logra mediante un cambio de variable:

La nueva variable X a simplificar tiene por valor el resultado de

(MAPCAR (QUOTE SIMPLIFY) L)

(vease la descripción de MAPCAR en el apéndice) es decir el resultado de simplificar cada uno de los elementos de L .

Las posibles simplificaciones hechas sobre la función cuyos argumentos o son átomos o ya estan simplificados están puestas en forma de condiciones en la definición que esta en el apendice y están explicados en el siguiente esquema.

Si X es un producto ((CAR X)=*)

CONDICIONAL

PREDICADO	RESULTADO
Si alguno de los factores es cero	0
Si alguno de los factores es uno	Se hace una nueva variable (Y) que es la lista de los factores diferentes de uno.
En otros casos	X (no hay simplificación)

CONDICIONAL

PREDICADO	RESULTADO
Si Y es vacia	1 (Todos los factores eran igual a 1)
Si Y tiene un solo elemento	Y (El único factor diferente de 1)
En otros casos	(CONS (QUOTE *) Y) (El producto de los factores diferentes de 1)

Si X es una suma ((CAR X)= +) y alguno de los sumandos es cero, se hace una nueva variable (Y) que es la lista de los sumandos diferentes de cero, y sobre ella actua el siguiente condicional:

CONDICIONAL

PREDICADO	RESULTADO
Si Y es vacia	0 (Todos los sumandos son cero)
Si tiene un solo elemento	Y (El único sumando distinto de cero)
En otros casos (más de un sumando distinto de cero)	(CONS (QUOTE +) Y) (La suma de los sumandos distintos de cero)

Las demas condiciones son:

PREDICADO Si X es de la forma		RESULTADO
NOTACION POLACA	NOTACION ORDINARIA	
(- 0 A)	0 - A	(- A)
(- A 0)	A - 0	A
(- 0)	- 0	0
(/ A 1)	A/1	A
(/ 0 A)	0/A	0
(** A 1)	A^1	A
(** A (- B))	A^{-B}	(/ 1 (** A B))
(** A 0)	A^0	1
(** (** A B) C)	$(A^B)^C$	(** A (* B C))
En otros casos		X (no se simplifica)

Esta función puede ser adaptada a cada problema agregando nuevas condiciones con instrucciones para simplificar otros tipos de funciones. Si la expresión a simplificar no es de ninguna de las formas previstas, no es modificada.

Veamos con un ejemplo sencillo, la derivada de $(X+1)^2$, como trabajan las funciones DERIVATIVE Y SIMPLIFY:

(APPLY DERIVATIVE (X (** (+ X 1) 2)))

el resultado final es:

(* 2 (** (+ X 1) (+ 1 0)))

que es igual a $2(x+1)^1\,(1+0)$ pero si aplicamos la función DIFFERENTIATE, obtendremos un resultado simplificado:

(APPLY DIFFERENTIATE (X (** (+X 1) 2)))

resultando

(* 2 (+ X 1))

que es igual a

$$2(x+1)$$

OTRAS FUNCIONES DE CALCULO DIFERENCIAL

Una vez programada la función derivada es fácil construir funciones interesantes referentes al cálculo diferencial, como son el gradiente la multiderivada de una función, así como la divergencia y el rotacional de un vector. A continuación serán descritos brevemente los programas para calcular estas 4 funciones; su definición rigurosa y la descripción de las funciones satélites que usan están en el apendice.

GRADIENT.- Es una función de dos argumentos, la lista de las variables (coordenadas) con respecto a las cuales se va a derivar y la función que se va a derivar puesta en forma de lista. Da como resultado la lista de las derivadas de la función con respecto a cada una de las variables, dicha lista es el vector gradiente de la función original.

Esto es;

∇F = (GRADIENT (X Y Z . . .) F)=

= ((DIFFERENTIATE X F) (DIFFERENTIATE Y F) . . .)=

$= (\partial F/\partial x\ ,\ \partial F/\partial y\ ,\ \cdots)$

Como se puede ver en su definición, la función GRADIENT se programa por medio de la función FOREACH que esta descrita en el apéndice, de manera que va aplicando una función de dos argumentos, en este caso la función DIFFERENTIATE a cada una de las variables (coordenadas) y a la lista por derivar, formando así una lista con los resultados. Veamos un ejemplo sencillo:

(APPLY GRADIENT ((X Y Z W) (* 3 X Y Z W)))

es decir $\nabla(3XYZW)$ en cuatro dimensiones. El resultado es:

((* 3 Y Z W) (* 3 X Z W) (* 3 X Y W) (* 3 X Y Z))=

=(3YZW, 3XZW, 3XYW , 3XYZ)

MULTI DERIVATIVE.- Con esta función se pueden calcular las derivadas múltiples de una función con respecto a las variables que se quieran. Es una función de dos argumentos: la lista de variables con respecto a las cuales se va a derivar sucesivamente y la función que se va a derivar puesta en forma de lista.

Si por ejemplo se quiere calcular la tercera derivada con respecto a X de una función F , la instrucción será:

(APPLY MULTIDERIVATIVE ((X X X) F))

y en un caso particular:

(APPLY MULTIDERIVATIVE ((X X X) (** X 3)))

que es $\frac{\partial^3 X^3}{\partial X^3}$

el resultado será:

(* 3 2) = 3 × 2 = 6

La programación de esta función se hace con la función FOREACH que es una función de tres argumentos: una función G de dos argumentos y dos listas; la manera como opera esta función se ve con claridad a través del siguiente ejemplo:

(FOREACH * G (X Y Z) L) (G X (G Y (G Z L)))

que en notación ordinaria en que G (X Y) indica G calculada para los argumentos X y Y:

G(X , G(Y , G(Z , L)))

Asi, en el caso particular de la función MULTIDERIVATIVE, se deriva la lista con respecto a una variable, el resultado se deriva con respecto a la siguiente y asi sucesivamente, obteniendo derivadas de orden mayor cada vez. La fórmula anterior para el caso de MULTIDERIVATIVE es equivalente a

(DERIVATIVE X (DERIVATIVE Y (DERIVATIVE Z L)))

que equivale a $\frac{\partial^3 L}{\partial x \partial y \partial z}$

que si x=y=z se reduce a $\frac{\partial^3 L}{\partial x^3}$

como en el ejemplo antes mencionado.

DIVERGENCE.- Es una función de dos argumentos, uno es la lista de las coordenadas y la otra es la función vectorial puesta en forma de lista, cada uno de sus elementos es una componente de la función vectorial; necesariamente el número de elementos de la primera lista (coordenadas) debe ser igual al número de elementos de la segunda lista (componentes de la función vectorial). Esta función da como resultado la divergencia de la función vectorial representada por la segunda lista:

$\nabla\cdot$(F1, F2, . . .)=(DIVERGENCE (X1 X2 . . .) (F1 F2 . . .))=

=(+(DERIVATIVE X1 F1) (DERIVATIVE X2 F2) . . .)=

$= \frac{\partial F1}{\partial X1} + \frac{\partial F2}{\partial X2} + \cdots$

Para obtener dicho resultado, inicialmente se construye una lista de la forma

((X1 F1) (X2 F2) . . .)

Esto se logra con la función PAIRLIST (ver su descripción en el apéndice) aplicada a la lista de coordenadas y al vector. Después se aplica la función DERIVATIVE a cada uno de los elementos de esta nueva lista (esto se hace con MAPCAR) y finalmente se suman los resultados y se simplifica. Veamos un

ejemplo secillo de DIVERGENCE:

La instrucción

```
(APPLY   DIVERGENCE
((X Y Z)  ( ( (* * X 2) (** Y 2) (** Z 2) (* X Y Z ) 4)))
```

equivale a

$$\nabla \cdot (x^2 + y^2 + z^2 , xyz , 4)$$

se obtiene:

```
(+(* 2 X)  ( * X  Z ))
```

que equivale a

$$2X + XZ$$

que es la divergencia del vector dado.

ROTACIONAL. - Es una función de dos argumentos, el primero es una lista de tres elementos (las coordenadas) y el segundo la función vectorial puesta en forma de lista con tres elementos, las componentes:

```
(ROTACIONAL ( X Y Z) ( F1  F2 F3))
```

Esta función a través de la primera instrucción, forma una lista del tipo:

```
(( - (G  Y  F3) (G Z F2))  (- (G Z F1)  (G X F3))
( - ( G X F2)  ( G Y F1)))
```

donde G representa a la función DERIVATIVE y por medio de la segunda instrucción simplifica este resultado.

Dicha lista equivale a:

$$\left(\frac{\partial F3}{\partial Y} - \frac{\partial F2}{\partial Z} , \frac{\partial F1}{\partial Z} - \frac{\partial F3}{\partial X} , \frac{\partial F2}{\partial X} - \frac{\partial F1}{\partial Y} \right)$$

PROGRAMACION DE LOS PARENTESIS DE POISSON EN LISP

Los Paréntesis de Pisson. - Considérense dos funciones f y g que son funciones continuas y diferenciables, de las coordenadas y de los momentos, esto es:

$$f = f(q_1, q_2, \cdots, p_1, p_2, \cdots)$$
$$g = g(q_1, q_2, \cdots, p_1, p_2, \cdots)$$

Se llama Paréntesis de Pisson de dichas funciones a la siguiente expresión:

$$\{ f, g \} = \sum_{i=1}^{n} \left(\frac{\partial f}{\partial p_i} \frac{\partial g}{\partial q_i} - \frac{\partial f}{\partial q_i} \frac{\partial g}{\partial p_i} \right) \qquad (1)$$

Existe una notable analogía entre el Paréntesis de Poisson $\{ f, g \}$ tal como se acaba de definir y el conmutador $[f, g]$ definido como

$$[f, g] = fg - gf$$

como puede verse de las propiedades del Paréntesis de Poisson que se enumeran a continuación:

i) $\{f, g\} = -\{g, f\}$

ii) $\{cf, g\} = c\{f, g\}$

iii) $\{f, c\} = 0$

iv) $\{f, g+h\} = \{f, g\} + \{f, h\}$

v) $\{f, gh\} = \{f, g\}h + g\{f, h\}$

vi) $\{p_j, f\} = \partial f / \partial q_j$

vii) $\{q_i, f\} = -\partial f / \partial p_i$

(c = constante)

La propiedad vi) se sigue directamente de la definición:

$$\{p_j, f\} = \sum_{i=1}^{n} \left(\frac{\partial p_j}{\partial p_i} \frac{\partial f}{\partial q_i} - \frac{\partial p_j}{\partial q_i} \frac{\partial f}{\partial p_i} \right)$$

ya que el segundo término de cada sumando siempre se anula por ser

$$\frac{\partial p_j}{\partial q_i} = 0$$

para toda i ; y los primeros términos se anulan para $i = j$, pero para $i \neq j$ toman el valor

$$\frac{\partial f}{\partial q_j}$$

La propiedad vii) se concluye de un razonamiento semejante.

El empleo sistemático de estas propiedades facilita la evaluación de los Paréntesis de Poisson de ciertas expresiones, en especial de expresiones polinomiales, sin hacer uso de la definición por medio de derivadas parciales.

El proceso de la evaluación de los Paréntesis de Poisson de dos expresiones polinomiales de la forma

$$\{u_1 + u_2 + \cdots, v_1 + v_2 + \cdots\}$$

donde u_i y v_i son monomios es como sigue:

Haciendo uso de la propiedad iv) de la distributividad frente a la suma convetimos el Paréntesis de Poisson inicial en una suma:

$$\{u_1, v_1\} + \{u_1, v_2\} + \cdots + \{u_2, v_1\} + \cdots + \{u_n, v_n\}$$

reduciendose el caso a la solución de Paréntesis de Poisson de expresiones monomiales que son del tipo

$$\{u_i, v_i\} = \{p_1^a q_1^b \pi, p_1^a q_1^b \rho\}$$

donde π y ρ son productos que no contienen a la coordenada q_1 ni al momento p_1 como factores.

A continuación, el empleo de la propiedad v) de distributividad frente al producto nos da como un primer resultado

$$p_1 \{p_1^{a-1} q_1^b \pi, p_1^c q_1^d \rho\} + \{p_1, p_1^c q_1^d \rho\} p_1^{a-1} q_1^b \pi$$

y esta operación repetida a veces con p_1 y b veces con q_1 y analogamente con p_1^c y q_1^d nos da como resultado:

$$p_1^a q_1^b \left[p_1^c q_1^d \{\pi, \rho\} + c \{\pi, p_1\} p_1^{c-1} q_1^d \rho + d \{\pi, q_1\} p_1^c q_1^d \pi \rho \right] + a \{p_1, p_1^c q_1^d \rho\} p_1^{a-1} q_1^b \pi + b \{q_1, p_1^c q_1^d \rho\} p_1^a q_1^{b-1} \pi$$

donde de acuerdo con las propiedades vi) y vii) podemos ver que $\{\pi, p_1\} = 0$ y $\{\pi, q_1\} = 0$ ya que π no depende de p_1 ni de q_1 ; y también,

$$\{p_1, p_1^c q_1^d \rho\} = d\, p_1^c q_1^{d-1} \rho$$

$$\{q_1, p_1^c q_1^d \rho\} = -c\, p_1^{c-1} q_1^d \rho$$

Resultando asi

$$p_1^{a+c} q_1^{b+d} \{\pi, \rho\} + ad\, p_1^{a+c-1} q_1^{b+d-1} \pi \rho - bc\, p_1^{a+c-1} q_1^{b+d-1} \pi \rho$$

que equivale a:

$$p_1^{a+c} q_1^{b+d} \{\pi, \rho\} + u_i v_i p_1^{-1} q_1^{-1} (ad - bc) \qquad (2)$$

Si continuamos el proceso eliminando de los factores π y ρ a todos los demás pares de variables conjugadas, obtendremos una suma de monomios y un término de la forma:

$$\frac{u_i v_i}{\alpha\beta} \{\alpha, \beta\}$$

donde α y β son los factores constantes contenidos en u_i y v_i respectivamente; éste término es cero ya que el Paréntesis de Poisson de una constante con cualquier función es cero (propiedad iii)), por lo tanto el resultado final será:

$$\{u_i, v_i\} = \{p_1^{a_1} p_2^{a_2} \cdots p_n^{a_n} q_1^{b_1} \cdots q_n^{b_n} \alpha, p_1^{c_1} \cdots p_n^{c_n} q_1^{d_1} \cdots q_n^{d_n} \rho\} =$$
$$= u_i v_i p_1^{-1} q_1^{-1} (a_1 d_1 - b_1 c_1) + u_i v_i p_2^{-1} q_2^{-1} (a_2 d_2 - b_2 c_2) + \cdots + u_i v_i p_n^{-1} q_n^{-1} (a_n d_n - b_n c_n) \qquad (3)$$

Que nos da la fórmula para los Paréntesis de Poisson de dos polinomios. Nótese que esta fórmula sirve también para cualquier operación que cumpla con las propiedades de los Paréntesis de Poisson antes mencionadas.

Con estos puntos teóricos como base pasemos a explicar la programación en LISP.

PROGRAMACION DE LOS PARENTESIS DE POISSON A PARTIR DE SU DEFINICION.

Basandonos en su definición y usando la función

DERIVATIVE antes descrita, es fácil hacer una función que los ejecute.

Esta función que se llama PB (Poisson Brackets) tiene tres argumentos, los dos primeros son las funciones de las que se quiere obtener el Paréntesis de Poisson, puestas en forma de lista en notación polaca y el tercero es la lista de las variables conjugadas puestas ordenadamente en dos sublistas separadas; es decir, los argumentos de PB son de la forma:

f , g , ((X Y Z . . .) (Px Py Pz))

Como primer paso, esta función construye con la lista de variables una nueva lista de la forma

((X Px) (Y Py) (Z Pz) . . .) (4)

donde cada par de variables conjugadas están juntas. Este primer paso se logra con la función PAIRLIST (vease en el apéndice) aplicada a las dos sublistas iniciales de variables conjugadas.

El segundo paso es ejecutar la fórmula

$$\frac{\partial f}{\partial p_i}\frac{\partial g}{\partial q_i} - \frac{\partial f}{\partial q_i}\frac{\partial g}{\partial p_i} \qquad (5)$$

para cada par de variables conjugadas, es decir, para cada elemento de (4) esto se logra con la instrucción para construir una lista de la forma

(- (* DERIVATIVE p_i f) (DERIVATIVE q_i g))
(* DERIVATIVE q_i f) (DERIVATIVE p_i g))) (6)

Una lista de esta forma se construirá para cada par de variables por medio de la función FOREACH. Finalmente se suman todas las listas de la forma (6) y se simplifica el resultado con la función SIMPLIFY.

Esta definición de Paréntesis de Poisson a partir de derivada tiene 2 graves inconvenientes:

En primer lugar es bastante lenta ya que aplica la función DERIVATIVE un número muy grande de veces: cuatro por cada par de variables conjugados (12 veces en un problema de 3 dimensiones).

El segundo inconveniente es la complejidad con que aparece el resultado, o en otras palabras, la necesidad de aplicar al resultado procesos largos de simplificación como sería la multiplicación de potencias de la misma base sumando los exponentes, etc.

Sin embargo es la única definición que nos garantiza poder resolver los Paréntesis de Poisson de funciones no polinomiales.

Para resolver los Paréntesis de Poisson de dos plinomios, y en menos tiempo obtener un resultado bastante simplificado, es conveniente hacer un programa que haga uso de las propiedades del Paréntesis de Poisson antes mencionadas. Dicho programa tendrá además la ventaja de que servirá para resolver operaciones que cumplan con las mismas propiedades.

PROGRAMACION DE LOS PARENTESIS DE POISSON A PARTIR DE SUS PROPIEDADES

El primer paso para resolver los Paréntesis de Poisson de dos polinomios es distribuir frente a la suma, reduciendo el caso a Paréntesis de Poisson de monomios en los que hay potencias de las variables conjugadas y factores constantes. La resolución de estas expresiones se puede hacer recurrentemente usando la formula en la que al sacar un par de variables conjugadas se genera un monomio. Este método tiene el inconveniente de que da un resultado al que todavía hay que hacerle simplificaciones difíciles para ponerlo en forma abreviada.

Es preferible haciendo uso de la fórmula final (3) encontrar una manera sistemática de obtener cada uno de los monomios en su forma más abreviada, lo cual evita grandes complicaciones al simplificar, como la de multiplicar potencias de la misma base, ya que podemos lograr un resultado en el que los

monomios aparezcan en su forma mas compacta.

A continuación describiremos paso a paso el programa haciendo uso de la fórmula (3) que da un resultado bastante simplificado.

La función que efectua todo el proceso también se llama PB (ver en el apéndice su definición) y sus argumentos se especifican de la misma manera que en el caso anterior. Es una combinación de varias funciones que se pueden dividir en dos grandes grupos:

El primer grupo es el que ejecuta la operación Paréntesis de Poisson propiamente dicha y esta condensado inicialmente en la función PB

El segundo grupo se podría llamar de presentación externa y es el que traduce el resultado final de una notación abreviada que se usa en el proceso interno y que luego se describirá a notación ordinaria polaca, y además efectúa las simplificaciones necesarias sobre el resultado final, como son aplicar la ley asociativa, multiplicar por cero y por uno, elevar a la cero y a la uno, etc.

Describamos cada uno de estos grupos de funciones:

En el primero, la operación inicial es transformar la lista de variables conjugadas [redacted] en otra lista de la forma

((X Px) (Y Py) . . .)

este paso se hace con PAIRLIST al igual que se hizo en la programación por medio de derivada; sirve para facilitar el proceso de construcción de la expresión final.

El segundo paso lo hace específicamente la función PB y consiste en distribuir a la operación Paréntesis de Poisson frente a la suma, transformando así el Paréntesis de Poisson de dos polinomios en la suma de varios Paréntesis de Poisson de dos monomios.

El Parentesis de Poisson de dos monomios es procesado por la función PB*, pero antes de que ella entre en acción, los argumentos son transformados a otra notación que facilita el proceso. La transformación a dicha notación la efectua la función PREPB

Esta función inicialmente examina si alguno de los dos monomios es negativo, en tal caso invierte el orden de los monomios y quita el signo haciendo uso de las propiedades i) y ii):

$$\{-f, g\} = \{g, f\}$$

Si los dos monomios son negativos hace la inversión dos veces: $\{-f, -g\} = \{f, g\}$

Una vez eliminados los posibles signos, los monomios son simples productos donde los factores pueden ser o bien constantes o bien potencias de las variables del problema.

Cada monomio es transformado en un producto cuyos últimos factores son los factores constantes del monomio si los hay y cuyo primer factor es una lista ordenada de los exponentes a los cuales aparecen elevadas las variables en el monomio, esto es, por ejemplo, si la lista de variables es:

((X PX) (Y PY))

y el monomio

(* 3(** X 2) (** PY 2) X Y)) $= 3X^2 Py^2 X Y$

será transformado a

(* ((3 0) (1 2)) 3)

donde como se puede ver el primer factor es el resultado de sustituir, en la lista de variables, a cada variable por su multiplicidad.

Una vez puestos los monomios en dicha notación, la función PB* procede a ejecutar la fórmula (3) multiplicando los monomios en la manera que ella índica.

Para multiplicar los monomios entre sí y por P_j^{-1} y q_j^{-1} simplemente se suman vectorialmente las listas de exponentes, restando 1 a la componente j correspondiente. La

función PB1 construye los factores del tipo (a_j d_j $-b_j$ c_j) en cada uno de los polinomios. Para sumar vectorialmente las listas de exponentes se usa la función VECSUM que simplemente suma los vectores componente a componente.

En toda esta formulación se usan ciertas funciones satélites que son importantes de mencionar:

En primer lugar, como se trabaja con exponentes enteros, sumandolos y restandolos, se construye un equipo de funciones que efectuan la suma y diferencia de enteros para no dejarla indicada. Dicho equipo esta condensado en la función (BISUMA A B) que efectua la suma de dos enteros positivos o negativos, usando las funciones MAS Y MENOS que trabajando sobre la lista INTEGERS, ya mencionada, suman o restan enteros positivos.

La función (MULTIPLICIDAD X L) da la potencia a que se encuentra elevado el factor X en el monomio L sumando los exponentes correspondientes. La función (EXPORDER U W) construye el vector cuyas componentes son los exponentes a que aparecen elevadas las variables.

Por ejemplo, el monomio

(* 3 (** X 2) (** PY 2) X Y)

es arreglado con respecto a la lista de variables

((X PX) (Y PY))

dando

((3 0) (1 2))

Esta función trabaja con ayuda de la función MULTIPLICIDAD para calcular los exponentes.

El segundo grupo de funciones de la función PB como ya se dijo, traduce el resultado a notación ordinaria y lo simplifica. En la primera parte se aplica la ley asociativa, ya que el resultado vendrá dado como una suma de sumas. Esto se hace con la función (PERASSOCIATE L) que aplica la ley asociativa para productos y sumas penetrando en los argumentos por medio de la función(PERCOLATE G L) (ver su descripción en el apéndice). Por ejemplo si se aplica la función PERASSOCIATE a la expresión

(+ (+ a b) (* c d) (+ f (g h)))

que equivale a

(a+b) + cd+ (f+(g+h))

se obtiene

(+ a b (* c d) f g h)

que es igual a

a + b + c d + f + g + h

Una vez asociado el resultado, tenemos una simple suma de monomios puestos en notación abreviada (como una lista de exponentes) . El siguiente paso es traducir estos monomios a notación ordinaria de LISP, lo cual se logra con la función

(TRANSLATE L W) aplicada acada uno de los monomios; esta función pasa de un monomio de la forma

(* ((a b) (c d) . . .) r s)

(donde r y s son factores constantes) que esta referido a la lista de variables

((X PX) (Y PY) . . .)

a uno de la forma

(* (** X a) (** PX b) (** Y c)

(** PY d) . . . r s)

que equivale a:

$X^a * PX^b * Y^c * PY^d * \ldots * r * s$

Una vez puesto el resultado en forma de polinomio solo falta hacerle simplificaciones elementales. Para ello se puede usar la función (SIMPLIFY L) o bien una combinación de funciones que hacen casi lo mismo y por lo tante no merecen ser explicadas con mucho detalle. Estas funciones que penetran en las subexpresiones por medio de la función PERCOLATE, antes mencionada, son:

(ZEROBEGONE L).- Hace las simplificaciones referentes a los ceros: multiplicación, suma y diferencia con cero.

(EXPBEGONE L).- Hace las simplificaciones referentes a exponentes: Potencia cero y potencia unidad.

(- GO L) .- Es una función que cuando encuentra un producto con factores con signos negativos les quita el signo a todos y le pone el signo correcto al producto. Por ejemplo:

(* A (+ B (* C (- D))) (- E) (- F))

es transformada a

(* A (+B (-(* CD))) E F)

(ONEGEONE L) .- Hace las simplificaciones referentes a la unidad (productos con algun 1 como factor).

El proceso descrito hasta ahora, de ejecución y simplificación de los Paréntesis de Poisson se puede comprender mejor con un ejemplo sencillo en el serán descritos los pasos intermedios:

Calculemos el Paréntesis de Poisson del momento angular con la coordenada Y para el caso de dos dimensiones (X y Y):

$$\{Y , XPY - YPX\}$$

la instrucción correspondiente será:

(APPLY PB (Y (+ (* X PY) (- (* Y PX)))

((X Y) (PX PY))))

Nótese que los polinomios solo se pueden poner como sumas, no como diferencias, por eso se le pone signo negativo a un monomio.

El primer paso será transformar el tercer argumento. (la lista de variables) a una nueva lista en la que aparezcan juntas las variables conjugadas, por lo tanto los nuevos argumentos serán:

```
Y
(+(* X PY) (- (* Y PX)))
(( X PX) (Y PY))
```

El siguiente paso será bifurcar el problema al distribuir al Paréntesis de Poisson frente a la suma; los argumentos de los dos nuevos problemas serán ahora:

```
Y
(* X PY)
(( X PX) (Y PY))
```

y

```
Y
(- (* Y PX))
(( X PX) (Y PY))
```

Ahora se efectua la traducción a la notación abreviada;

En el primero problema queda

```
Y = (* (( 0 0) (1 0)))
(* X PY) = (* (( 1 0) (0 1)))
```

y en el segundo, como hay un monomio negativo se invierte el orden y se hace la traducción:

```
(* Y PX) = (* (( 0 1) (1 0)))
Y = ((0 0) (1 0)))
```

observese que como en ningún monomio había factores constantes, los productos quedaron con un sólo factor que es la lista o vector de los exponentes ordenados.

El siguiente paso es aplicar la fórmula (3) obteniendo así las siguientes sumas para el primer problema:

```
(+(* ((0 (-1)) (1 1)) (- (*0 1) (* 0 0)))
   (* ((1 0) (0 0)) (- (* 0 0) (6 1 1))))
```

que se traduce con la función TRANSLATE a

```
(+ (*(** X 0)(** PX (-1))
     (** Y 1)(** PY 1)
     (-(* 0 1)(* 0 0)))
   (*(** X 1)(** PX 0)
     (** Y 0)(** PY 0)
     (-(* 0 0)(* 1 1))))
```

que en notación ordinaria es

$$X^{0}\,PX^{-1}\,Y^{1}\,PY^{1}\,(0\times 1-0\times 0)+$$
$$+\,X^{1}\,PX^{0}\,Y^{0}\,PY^{0}\,(0\times 0-1\times 1)$$

y para el segundo:

```
(+(*((0 (-1)) (2 0)) (-(* 1 0)(* 0 0)))
  (* ((0 1) (1 (-1))) (-(* 0 1)(* 1 0))))
```

que se traduce a

```
(+ (* (**X 0) (** PX (-1))
      (** Y 2)(** PY 0)
      (-(* 1 0)(* 0 0)))
   (* (** X 0)(** PX 1)
      (** Y 1)(** PY (-1))
      (-(* 0 1)(* 1 0))
```

que en notación ordinaria es:

$$X^0\ PX^{-1}\ Y^2\ PY^0\ (1\times 0 - 0\times 0) +$$
$$+\ X^0\ PX^1\ Y^1\ PY^{-1}\ (0\times 1 - 1\times 0)$$

Estas expresiones se suman y se les aplica ZEROBEGONE y se reducen a:

(* (** X 1) (** PX O) (** Y O) (** PY O) (- (* 1 1)))

que es lo mismo que

$$X^1\ PX^0\ Y^0\ PY^0\ (-\ 1\ 1)$$

Después se aplica EXPBEGONE (simplificaciones relativas a los exponentes) y se obtiene:

(* X 1 1 1 (- (* 1 1)))

A lo cual se aplica - GO dando

(- (* X 1 1 1 (* 1 1))

A lo cual se aplica ONEGO (multiplicaciones por uno) y se obtiene el resultado final

(- X)

que es el valor de

$$\{Y,\ X\ PY - Y\ PX\}$$

A P E N D I C E

A continuación aparecen las programas de Derivada y de Paréntesis de Poisson. Posteriormente se describen algunas funciones elementales que se emplean en estos programas y cuya definición no esta incluida.

```
(DEFINE
(SIMPLIFY (LAMBDA (L) (COND
((ATOM L) L)
((T) ((LAMBDA (X) (COND

((EQ (CAR X) (QUOTE *))
(COND ((ELEMENT (QUOTE 0) (CDR X)) (QUOTE 0))
((ELEMENT (QUOTE 1) (CDR X))
((LAMBDA (Y) (COND
((NULL Y) (QUOTE 1)) ((NULL (CDR Y)) (CAR Y))
((T) (CONS (QUOTE *) Y))))
(EXPUNGE (QUOTE 1) (CDR X))))
((T) X)))

((EQ (CAR X) (QUOTE +)) (COND
((ELEMENT (QUOTE 0) (CDR X)) ((LAMBDA (Y) (COND
((NULL Y) (QUOTE 0)) ((NULL (CDR Y)) (CAR Y))
((T) (CONS (QUOTE +) Y))))
(EXPUNGE (QUOTE 0) (CDR X))))
((T) X)))

((SIMILAR (QUOTE (- 0 -)) X) (LIST (QUOTE -) (CADDR X)))
((SIMILAR (QUOTE (- - 0)) X) (CADR X))
((EQUAL (QUOTE (- 0)) X) (QUOTE 0))

((SIMILAR (QUOTE (/ - 1)) X) (CADR X))
((SIMILAR (QUOTE (/ 0 -)) X) (QUOTE 0))

((SIMILAR (QUOTE (** - 1)) X) (CADR X))
((SIMILAR (QUOTE (** - (- -))) X) (LIST
(QUOTE /) (QUOTE 1) (LIST (QUOTE **) (CADR X) (CADR (CADDR X)))))
((SIMILAR (QUOTE (** - 0)) X) (QUOTE 1))
((SIMILAR (QUOTE (** (** - -) -)) X)
(LIST (QUOTE **) (CADR (CADR X)) (LIST (QUOTE *)
(CADDR (CADR X)) (CADDR X))))

((T) X)))
(MAPCAR (QUOTE SIMPLIFY) L))))))

(DIFFERENTIATE (LAMBDA (X L)
(SIMPLIFY (DERIVATIVE X L))))

(DERIVATIVE (LAMBDA (X L) (COND

((ATOM L) (COND ((EQ X L) (QUOTE 1)) ((T) (QUOTE 0))))                    (A)

((SIMILAR (QUOTE (+ - -)) L)
(LIST (QUOTE +) (DERIVATIVE X (CADR L)) (DERIVATIVE X (CADDR L))))       (B)

((SIMILAR (QUOTE (+ ---)) L)
(CONS (QUOTE +)                                                          (C)
(MAPCAR (FUNCTION (LAMBDA (L) (DERIVATIVE X L))) (CDR L))))

((SIMILAR (QUOTE (- - -)) L) (LIST (QUOTE -)                             (D)
```



```
(DERIVATIVE X (CADR L)) (DERIVATIVE X (CADDR L))))

((SIMILAR (QUOTE (* - -)) L) (LIST (QUOTE +)
(LIST (QUOTE *) (CADR L) (DERIVATIVE X (CADDR L)))                       (E)
(LIST (QUOTE *) (DERIVATIVE X (CADR L)) (CADDR L))))

((SIMILAR (QUOTE (* ---)) L)
(CONS (QUOTE +) (MAPCAR (QUOTE (LAMBDA (L) (CONS (QUOTE *) L)))          (F)
(DIAG (FUNCTION (LAMBDA (L) (DERIVATIVE X L))) (CDR L)))))

((SIMILAR (QUOTE (/ - -)) L) (LIST (QUOTE /)
(LIST (QUOTE -) (LIST (QUOTE *) (DERIVATIVE X (CADR L)) (CADDR L))       (G)
(LIST (QUOTE *) (CADR L) (DERIVATIVE X (CADDR L))))
(LIST (QUOTE **) (CADDR L) (QUOTE 2))))

((SIMILAR (QUOTE(** - -)) L)
(LIST (QUOTE *) (CADDR L))                                               (H)
(LIST (QUOTE **) (CADR L)
(COND
((NUMERAL (CADDR L)) (PREDECESOR (CADDR L)))
((T) (LIST (QUOTE -) (CADDR L) (QUOTE 1)))))
(DERIVATIVE X (CADR L))))

((SIMILAR (QUOTE (- -)) L)
(LIST (QUOTE -) (DERIVATIVE X (CADR L))))                                (I)

((EQ(QUOTE SIN) (CAR L))
(LIST (QUOTE *)(DERIVATIVE X (CADR L)) (LIST (QUOTE COS) (CADR L))))

((EQ (QUOTE COS) (CAR L))
(LIST (QUOTE -) (LIST (QUOTE *)
(LIST (QUOTE SIN) (CADR L))
(DERIVATIVE X (CADR L)))))

((SIMILAR (QUOTE (ARC SIN -)) L)
(LIST (QUOTE /) (DERIVATIVE X (CADDR L))
(LIST (QUOTE **) (LIST (QUOTE -) (QUOTE 1)
(LIST (QUOTE **) (CADDR L) (QUOTE 2)))
(LIST (QUOTE /) (QUOTE 1) (QUOTE 2)))))

((SIMILAR (QUOTE (ARC COS -)) L)
(LIST (QUOTE -) (LIST (QUOTE /) (DERIVATIVE X (CADDR L))
(LIST (QUOTE **) (LIST (QUOTE -) (QUOTE 1)
(LIST (QUOTE **) (CADDR L) (QUOTE 2)))
(LIST (QUOTE /) (QUOTE 1) (QUOTE 2))))))

((SIMILAR (QUOTE (EXP -)) L)
(LIST (QUOTE *) L (DERIVATIVE X (CADR L))))

((SIMILAR (QUOTE (LOG - -)) L)
(LIST (QUOTE /) (DERIVATIVE X (CADDR L))
(LIST (QUOTE *) (CADDR L)
(LIST (QUOTE LOG) (QUOTE E) (CADR L)))))
```

```
((T) (LIST (QUOTE DERIVATIVE) X L)))))

(DIAG (LAMBDA (G L) (DIAG* G L L (LIST))))

(DIAG* (LAMBDA (G N L R) (COND
((NULL N) R)
((T) (DIAG* G (CDR N) L (CONS (DIAG** G (CDR N) L) R))))))

(DIAG** (LAMBDA (G N L) (COND
((NULL N) (CONS (G (CAR L)) (CDR L)))
((T) (CONS (CAR L) (DIAG** G (CDR N) (CDR L)))))))

(PREDECESOR (LAMBDA (X) (PREDECESOR* X (INTEGERS))))

(PREDECESOR* (LAMBDA (X I) (COND
((EQ X (CADR I)) (CAR I))
((T) (PREDECESOR* X (CDR I))))))

(NUMERAL(LAMBDA (X) (ELEMENT X (INTEGERS))))

(GRADIENT (LAMBDA (U L)
(FOREACH
(FUNCTION (LAMBDA (X L) (DIFFERENTIATE X L))) U L)))

(DIVERGENCE (LAMBDA (U L) (SIMPLIFY
(CONS (QUOTE +) (MAPCAR
(FUNCTION (LAMBDA (X) (DERIVATIVE (CAR X) (CADR X))))
(PAIRLIST U L))))))

(CURL (LAMBDA (U L) (LIST
(SIMPLIFY (LIST (QUOTE -)
(DERIVATIVE (CADDR U) (CADDR L))
(DERIVATIVE (CADDR U) (CADR L))))
(SIMPLIFY (LIST (QUOTE -)
(DERIVATIVE (CADDR U) (CAR L))
(DERIVATIVE (CAR U) (CADDR L))))
(SIMPLIFY (LIST (QUOTE -)
(DERIVATIVE (CAR U) (CADR L))
(DERIVATIVE (CADR U) (CAR L)))))))

(MULTIDERIVATIVE (LAMBDA (U L) (FOREACH*
(FUNCTION (LAMBDA (X L) (DIFFERENTIATE X L))) U L)))

(PB (LAMBDA (U V W)
(SIMPLIFY
(CONS (QUOTE +) (FOREACH (FUNCTION (LAMBDA (X L)
(LIST (QUOTE -) (LIST (QUOTE *)
(DERIVATIVE (CADR X) (CAR L))
(DERIVATIVE (CAR X) (CADR L)))
(LIST (QUOTE *)
(DERIVATIVE (CAR X) (CAR L))
(DERIVATIVE (CADR X) (CADR L))))))
(PAIRLIST (CAR W) (CADR W)) (LIST U V))
))))
```



```
(PAIRLIST (LAMBDA (U V) (COND
((NULL U) (LIST))
((T) (CONS (LIST (CAR U) (CAR V)) (PAIRLIST (CDR U) (CDR V)))))))

)
```

00000169 *

```
(DEFINE
(PB (LAMBDA (F G U) (ONEBEGONE (-GO (EXPBEGONE (ZEROBEGONE
(CONS (QUOTE +) (FOREACH (QUOTE TRANSLATE)
(CDR (PERASSOCIATE (PB+ F G (PAIRLIST (CAR U) (CADR U)))))
(PAIRLIST (CAR U) (CADR U))))))))))

(PB+ (LAMBDA (F G W) (COND
((SIMILAR (QUOTE (+ ---)) F) (CONS (QUOTE +)
(MAPCAR (FUNCTION (LAMBDA (X) (PB+ X G W))) (CDR F))))
((SIMILAR (QUOTE (+ ---)) G) (CONS (QUOTE +)
(MAPCAR (FUNCTION (LAMBDA (X) (PB* (PREPB F X W)))) (CDR G))))
((T) (PB* (PREPB F G W))))))

(PREPB (LAMBDA (U V W) (COND
((SIMILAR (QUOTE (- -)) U) (PREPB V (CADR U) W))
((SIMILAR (QUOTE (- -)) V) (PREPB (CADR V) U W))
((T) (LIST (APPEND (LIST (QUOTE *) (EXPORDER U W))
(COND ((ATOM U) (COND ((FORSOME (FUNCTION (LAMBDA (X)
(ELEMENT U X))) W) (LIST)) ((T) (LIST U))))
((T) (POSSESSING
(FUNCTION (LAMBDA (X) (OR (AND (ATOM X) (NOT (FORSOME
(FUNCTION (LAMBDA (Y) (ELEMENT X Y))) W)))
(AND (NOT (ATOM X)) (NOT (SIMILAR (QUOTE (** - -)) X))))))) (CDR U)))))
(APPEND (LIST (QUOTE *) (EXPORDER V W))
(COND ((ATOM V) (COND ((FORSOME (FUNCTION (LAMBDA (X)
(ELEMENT V X))) W) (LIST)) ((T) (LIST V))))
((T) (POSSESSING (FUNCTION (LAMBDA (X) (OR
(AND (ATOM X) (NOT (FORSOME (FUNCTION (LAMBDA (Y)
(ELEMENT X Y))) W))) (AND (NOT (ATOM X))
(NOT (SIMILAR (QUOTE (** - -)) X))))))) (CDR V)))))))))))

(PB* (LAMBDA (U) (CONS (QUOTE +) (MAPCAR (FUNCTION (LAMBDA (X)
(APPEND X (APPEND (CDDAR U) (CDDADR U))))))
(FOR3 (QUOTE PB1) (FOREACH (QUOTE VECSUM/) (DIAG (FUNCTION (LAMBDA (Y)
(LIST (BISUMA (CAR Y) (QUOTE (- 1))) (BISUMA (CADR Y) (QUOTE (- 1))))))
(CADAR U)) (CADADR U))
(CADAR U) (CADADR U))))))

(PB1 (LAMBDA (Z U V) (LIST (QUOTE *) Z (LIST (QUOTE -)
(LIST (QUOTE *) (CADR U) (CAR V)) (LIST (QUOTE *) (CAR U) (CADR V))))))

(PAIRLIST (LAMBDA (U V) (COND
((NULL U) (LIST))
((T) (CONS (LIST (CAR U) (CAR V)) (PAIRLIST (CDR U) (CDR V)))))))

(EXPORDER (LAMBDA (U W) (COND ((NULL W) (LIST))
((T) (CONS (LIST (MULTIPLICIDAD (CAAR W) U)
(MULTIPLICIDAD (CADAR W) U)) (EXPORDER U (CDR W))))))))

(MULTIPLICIDAD (LAMBDA (X L) (COND
((NULL L) (QUOTE 0))
((ATOM L) (COND ((EQ X L) (QUOTE 1)) ((T) (QUOTE 0))))
((EQ (CAR L) X) (BISUMA (QUOTE 1) (MULTIPLICIDAD X (CDR L))))
```



```
((AND (SIMILAR (QUOTE (** - -)) (CAR L))
(OR (EQ X (CADAR L))
(AND (NOT (ATOM (CADAR L)))
(ELEMENT X (CADAR L)))))
(BISUMA (CADDAR L) (MULTIPLICIDAD X (CDR L))))
((T) (MULTIPLICIDAD X (CDR L))))))

(VECSUM/ (LAMBDA (A B) (COND ((NULL A) (LIST))
((T) (CONS (LIST (BISUMA (CAAR A) (CAAR B))
(BISUMA (CADAR A) (CADAR B))) (VECSUM/ (CDR A) (CDR B)))))))

(BISUMA (LAMBDA (A B) (COND
((ATOM A) (COND ((ATOM B) (MAS A B))
((T) (MENOS A (CADR B)))))
((T) (COND ((ATOM B) (MENOS B (CADR A)))
((T) (LIST (QUOTE -) (MAS (CADR A) (CADR B)))))))))

(MAS (LAMBDA (A B) (COND ((EQ A (QUOTE 0)) B)
((EQ B (QUOTE 0)) A)
((T) (CAR (TALLYCOMPLEMENT (APPEND
(FRAGMENT A (INTEGERS)) (FRAGMENT B (INTEGERS ))) (INTEGERS )))))))

(MENOS (LAMBDA (A B) (COND
((EQ A (QUOTE 0)) (LIST (QUOTE -) B))
((EQ B (QUOTE 0)) A)
((EQ A B) (QUOTE 0))
((ELEMENT A (FRAGMENT B (INTEGERS))) (LIST (QUOTE -)
(CAR (TALLYCOMPLEMENT (DIFFERENCE (FRAGMENT A (INTEGERS))
(FRAGMENT B (INTEGERS))) (INTEGERS)))))
((T) (CAR (TALLYCOMPLEMENT (DIFFERENCE (FRAGMENT A (INTEGERS))
(FRAGMENT B (INTEGERS))) (INTEGERS)))))))

(DIFFERENCE (LAMBDA (A B) (COND
((NULL A) B) ((NULL B) A) ((T) (DIFFERENCE (CDR A) (CDR B))))))

(FOR3 (LAMBDA (G Z U V) (COND ((NULL Z) (LIST))
((T) (CONS (G (CAR Z) (CAR U) (CAR V))
(FOR3 G (CDR Z) (CDR U) (CDR V)))))))

(DIAG (LAMBDA (G L) (DIAG* G L L (LIST))))

(DIAG* (LAMBDA (G N L R) (COND
((NULL N) R)
((T) (DIAG* G (CDR N) L (CONS (DIAG** G (CDR N) L) R))))))

(DIAG** (LAMBDA (G N L) (COND
((NULL N) (CONS (G (CAR L)) (CDR L)))
((T) (CONS (CAR L) (DIAG** G (CDR N) (CDR L)))))))

(TRANSLATE (LAMBDA (L W) (CONS (CAR L) (APPEND (APAIR (CADR L) W)
(CDDR L)))))

(APAIR (LAMBDA (L W) (COND ((NULL L) (LIST))
((T) (APPEND (LIST (LIST (QUOTE **) (CAAR W) (CAAR L))
```

```
(LIST (QUOTE **) (CADAR W) (CADAR L))) (APAIR (CDR L) (CDR W)))))))

(ASSOCIATE (LAMBDA (L) (COND
((EQ (QUOTE +) (CAR L))
(CONS (QUOTE +) (ASSOCIATE* (QUOTE (+ ---)) (CDR L))))
((EQ (CAR L) (QUOTE *))
(CONS (QUOTE *) (ASSOCIATE* (QUOTE (* ---)) (CDR L))))
((T) L))))

(ASSOCIATE* (LAMBDA (X L) (COND
((NULL L) (LIST))
((SIMILAR X (CAR L))
(APPEND (CDAR L) (ASSOCIATE* X (CDR L))))
((T) (CONS (CAR L) (ASSOCIATE* X (CDR L)))))))

(PERASSOCIATE (LAMBDA (L) (PERCOLATE (QUOTE ASSOCIATE) L)))

(EXPGO (LAMBDA (L) (COND
((SIMILAR (QUOTE (** - 0)) L) (QUOTE 1))
((SIMILAR (QUOTE (** - 1)) L) (CADR L)) ((T) L))))

(EXPBEGONE (LAMBDA (L) (PERCOLATE (QUOTE EXPGO) L)))

(ZEROGO (LAMBDA (L) (COND
((AND (EQ (CAR L) (QUOTE *)) (ELEMENT (QUOTE 0) (CDR L))) (QUOTE 0))
((EQ (CAR L) (QUOTE +))
((LAMBDA (X) (COND ((NULL X) (QUOTE 0))
((NULL (CDR X)) (CAR X))
((EQUAL (QUOTE (- 0)) L) (QUOTE 0))
((T) (CONS (QUOTE +) X)))) (EXPUNGE (QUOTE 0) (CDR L))))
((EQUAL (QUOTE (- 0)) L) (QUOTE 0))
((SIMILAR (QUOTE (- - 0)) L) (CADR L))
((SIMILAR (QUOTE (- 0 -)) L) (LIST (QUOTE -) (CADDR L)))
((T) L))))

(ZEROBEGONE (LAMBDA (L) (PERCOLATE (QUOTE ZEROGO) L)))

(ONEGO (LAMBDA (L) (COND
((EQ (CAR L) (QUOTE *))
((LAMBDA (X) (COND ((NULL X) (QUOTE 1))
((NULL (CDR X)) (CAR X))
((T) (CONS (QUOTE *) X))))
(EXPUNGE (QUOTE 1) (CDR L)))) ((T) L))))

(ONEBEGONE (LAMBDA (L) (PERCOLATE (QUOTE ONEGO) L)))

(-GO (LAMBDA (L) (COND ((ATOM L) L) ((EQ (CAR L) (QUOTE *))
((LAMBDA (X) (COND ((EQ (CAR X) (QUOTE -)) (LIST (QUOTE -)
(CONS (QUOTE *) (CDR X)))) ((T) (CONS (QUOTE *) (CDR X)))))
(-GO* (MAPCAR (QUOTE -GO) (CDR L)))))
((T) (MAPCAR (QUOTE -GO) L)))))

(-GO* (LAMBDA (L) (COND ((NULL L) (LIST (QUOTE +)))
((T) ((LAMBDA (X) (COND
```



```
((EQ (CAR X) (QUOTE +)) (COND ((SIMILAR (QUOTE (- -)) (CAR L))
(CONS (QUOTE -) (CONS (CADAR L) (CDR X))))
((T) (CONS (QUOTE +) (CONS (CAR L) (CDR X))))))
((T) (COND ((SIMILAR (QUOTE (- -)) (CAR L)) (CONS (QUOTE +)
(CONS (CADAR L) (CDR X)))) ((T) (CONS (QUOTE -)
(CONS (CAR L) (CDR X))))))))
(-GO* (CDR L)))))))

)

-

  00000173 *
```

BREVE DESCRIPCION DE LAS FUNCIONES ELEMENTALES QUE SE USAN EN ESTOS PROGRAMAS Y CUYA DEFINICION NO ESTA INCLUIDA.

(ELEMENT X L)
Es un predicado que es cierto solo si X es un elemento de la lista L Ejemplo:

(ELEMENT A (B C A)) = (T) (verdadero)

(EXPUNGE X L)
Es una función que quita de L todos los elementos iguales a X Ejemplo:

(EXPUNGE A (A B A C D)) = (B C D)

(FORSOME P L)
Es un predicado de dos argumentos: un predicado y una lista. Es cierto solo si P lo es para algún elemento de L. Ejemplo:

(FORSOME ATOM ((A B) C (D E))) = (T)

(FOREACH G X L)
Es una función de tres argumentos, el primero G , es una función de dos argumentos y las otras dos son listas. Forma una lista cuyos elementos son los resultados de aplicar G a cada elemento de X y a L . Ejemplo:

(FOREACH G (A B C) (D E))=

=((G A (D E)) (G B (D E)) (G C (D E)))

(INTEGERS)
Es una función sin argumentos cuyo valor es la lista de los enteros del 0 al 60

(MAPCAR G L)
Es una función cuyos argumentos son una función G y una lista L . Da como resultado la lista de los resultados de aplicar G a cada elemento de L.

(PAIRLIST L M)
Sus argumentos son dos listas de igual número de elementos. Forma una lista cuyos elementos son listas de parejas de elementos que son una de L y su correspondiente de M Ejemplo: (PAIRLIST (A B C) X Y Z)) =

=((A X) (B Y) (C Z))

(POSSESSING P L)
Tiene dos argumentos: un predicado y una lista. Da la lista de los elementos de L para los cuales P es cierto. Ejemplo:

(POSSESSING ATOM ((B C) LISP (1 2 3) A)) =

=(LISP A)

(PERCOLATE G L)
Tiene dos argumentos, una función y una lista. Aplica G a todas las subexpresiones de la lista L menos a los átomos, y a la lista total, esto es, si L=(A (B C) D)

(PERCOLATE G L) = (G (A (G(B C)) D))

(SIMILAR X L)
Es un predicado de dos argumentos que es verdadero si X es similar a L. Dos expresiones son similares si y solo sí ocurre alguna de las siguientes posiblidades:

1a. Si son dos símbolos atómicos identicos.

2a. Si son dos listas vacias

3a. Si uno de ellos es un guión (-).

4a. Si el primer elemento de X es un átomo que consta de 3 guiones seguidos (---).

5a. Si el CAR de X es similar al de L y sus CDR son similares.