5.-PROGRAMACIÓN FUNCIONAL.

Historia

Sus orígenes provienen del Cálculo Lambda (o λ-cálculo), una teoría matemática elaborada por Alonzo Church como apoyo a sus estudios sobre computabilidad.

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

¿Qué es la Programación Funcional?

C, Java, Pascal, Ada, etc.. son lenguajes imperativos. Son "imperativos" en el sentido de que consisten en una secuencia de comandos que son ejecutados uno tras otro estrictamente.

Un programa funcional es una expresión simple que es ejecutada por evaluación de la expresión. La cuestión está en QUÉ va a ser computado, no en CÓMO va a serlo.

Otro lenguaje muy conocido, casi funcional es el lenguaje de consultas estándar de bases de datos, SQL. Una consulta SQL es una expresión con proyecciones, selecciones y uniones. Una consulta dice qué rela-ción se debe computar sin decir cómo debe computarse. Además, la consulta puede ser evaluada en cualquier orden que sea conveniente

Modelo Funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos.

El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre sistema y usuario: el usuario introduce una expresión inicial y el sistema la evalúa mediante un proceso de reducción. En este procesos se utilizan las definiciones de funciones realizadas por el progra- mador hasta obtener un valor no reducible.

El valor que devuelve una función está únicamente determinado por el valor de sus argumentos consi- guiendo que una misma expresión tenga siempre el mismo valor.

Es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad conmutativa, asociativa, etc.

El programador se encarga de definir un conjunto de funciones sin preocuparse de los métodos de eva-luación que posteriormente utilice el sistema. Estas funciones no tienen efectos laterales y no dependen de una arquitectura concreta.

Este modelo promueve la utilización de una serie de características como las funciones de orden superior, los sistemas de inferencia de tipos, el polimorfismo, la evaluación pere- zosa, etc.

Funciones de orden superior

Un lenguaje utiliza funciones de orden superior cuando permite que las funciones sean tratadas como valores de 1ª clase, permitiendo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados.

La utilización de funciones de orden superior proporciona una mayor flexibilidad al programador, siendo una de las características mas sobresalientes de los lenguajes funcionales.

```
quad :: Int -> Int
quad x = x * x

impar :: Int -> Bool
impar x
   | (x `mod` 2) == 1          = True
   | otherwise                 = False

map :: (t -> u) -> [t] -> [u]
map f [] = []
map f (a:x) = f a : map f x


map quad [1,2,3,4] = [1,4,9,16]
map impar [1,2,3,4] = [True, False, True, False]
```

Sistemas de Inferencia de Tipos y Polimorfismo

Muchos lenguajes funcionales han adoptado un sistema de inferencia de tipos que consiste en:

El programador no está obligado a declarar el tipo de las expresiones.
El compilador contiene un algoritmo que infiere el tipo de las expresiones.
Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido.

Los sistemas de inferencia de tipos permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de polimorfismo.

El polimorfismo permite que el tipo de una función dependa de un parámetro. Por ejemplo, si se define una función que calcule la longitud de una lista, una posible definición sería:

```
long ls = if vacia(L) then 0
                  else 1 + long(cola (L))
long :: [x] → Integer
```

El sistema de inferencia de tipos inferiría el tipo: long::[x] → Integer, indicando que tiene como argu- mento una lista de elementos de un tipo a cualquiera y que devuelve un entero.

En un lenguaje sin polimorfismo sería necesario definir una función long para cada tipo de lista que nece- sitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir algorit- mos para estructuras similares.

Evaluación Perezosa

Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si estos serán utilizados.
Por ejemplo:

```
f (x:Integer; y:Integer):Integer
begin
   return (x+3);
end;

g (x:Integer):Integer
begin
(* bucle infinito *)
while true do
   x:=x
   end;


-- Programa Principal
begin
      write
   (f(4,g(5)));
end;
```

Con el sistema de evaluación tradicional, el programador no devolvería nada, puesto que al intentar eva- luar g(5) el sistema entraría en un bucle infinito. Dicha técnica de evaluación se conoce como evaluación impaciente porque evalúa todos los argumentos de una función antes de conocer si son necesarios.

Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa que consiste en no evaluar un argumento hasta que no se necesita. En el ejemplo anterior, si se utilizase evaluación perezosa, el sistema escribiría 7.

¿Qué tienen de bueno los lenguajes funcionales?

Estos son algunas de las características más importantes de los lenguajes funcionales.

Examinemos algunos de los beneficios de la programación funcional:

1. Programas cortos. La brevedad de los programas funcionales hace que sean mucho más con- cisos que su copia imperativa.
2. Facilidad de comprensión de los programas funcionales. Deberíamos ser capaces de en- tender el programa sin ningún conocimiento previo del Haskell. No podemos decir lo mismo de un programa en C. Nos lleva bastante tiempo comprenderlo y, cuando lo hemos entendido, es muy fácil cometer un pequeño fallo y tener un programa incorrecto.
3. No hay ficheros "core". La mayoría de los lenguajes funcionales y Haskell en particular son fuertemente tipados y eliminan una gran cantidad de clases que se crean en tiempo de compila- ción con las que se pueden cometer errores. O sea, fuertemente tipados significa que no hay fi- cheros "core". No hay posibilidad de tratar un puntero como un entero o un entero como un pun-tero nulo.
4. Reutilización de código. Los tipos fuertes están, por supuesto, disponibles en muchos lengua- jes imperativos como Ada o Pascal. Sin embargo el sistema de tipos de Haskell es mucho menos restrictivo que, por ejemplo el de Pascal porque usa polimorfismo. Por ejemplo, el algoritmo Quicksort se puede implementar de la misma manera en Haskell para listas de enteros, de carac- teres, listas de listas... mientras que la versión en C es sólo para arrays de enteros.
5. Plegado. Los lenguajes funcionales no estrictos tienen otra característica, la evaluación perezo- sa. Los lenguajes funcionales no estrictos llevan exactamente esta clase de evaluación. Las es-tructuras de datos son evaluadas justo en el momento en el que se necesita una respuesta y puede que haya parte de estas estructuras que no se evalúen
6. Abstracciones potentes. Generalmente, los lenguajes funcionales ofrecen nuevas formas para encapsular abstracciones. Una abstracción permite definir un objeto cuyo trabajo interno está oculto. Por ejemplo, un procedimiento en C es una abstracción. Las abstracciones son la clave para construir programas con módulos y de fácil mantenimiento. Son tan importantes que la pre- gunta para todo nuevo lenguaje es: "¿De qué mecanismos de abstracción dispone?".Un mecanismo de abstracción muy potente que está disponible en los lenguajes funcionales son las funciones de alto orden. En Haskell una función es un "ciudadano de primera clase": pueden pasarse tranquilamente a otras funciones, ser devueltas como el resultado de otra función, ser incluidas en una estructura de datos, etc. Esto nos quiere decir que el buen uso de estas funcio- nes de alto orden puede mejorar sustancialmente la estructura y modularidad de muchos pro- gramas.
7. Manejo de direcciones de memoria. Muchos programas sofisticados necesitan asignar me- moria dinámica desde una pila. En C, esto se hace con una llamada a "malloc", seguida de un código para inicializar la memoria. El programador es el responsable de liberar memoria cuando ya no se necesite más. Esto produce, muchas veces, errores del tipo "dangling-pointer" (punteros colgados).

   Cada lenguaje funcional libera al programador del manejo de este almacenamiento. La memoria es asignado e inicializado implícitamente y es recogido por el recolector de basura.

   La tecnología de asignación del store y la recolección de basura está muy bien desarrollada y los costes son bastante insignificantes.

Cálculo lambda

El cálculo lambda es un sistema formal diseñado para investigar la definición de función, la no- ción de aplicación de funciones y la recursión. Fue introducido por Alonzo Church y Stephen Kleene en la década de 1930; Church usó el cálculo lambda en 1936 para resolver el Entscheidungsproblem[1]. Puede ser usado para definir de manera limpia y precisa qué es una "función computable".

Church resolvió negativamente el Entscheidungsproblem: probó que no había algoritmo que pudiese ser considerado como una "solución" al Entscheidungsproblem.

El cálculo lambda ha influenciado enormemente el diseño de lenguajes de programación funcionales, especialmente LISP.

Se puede considerar al cálculo lambda como el más pequeño lenguaje universal de programación. Con- siste de una regla de transformación simple (substitución de variables) y un esquema simple para definir funciones.

El cálculo lambda es universal porque cualquier función computable puede ser expresada y evaluada a través de él.

Por lo tanto, es equivalente a las máquinas de Turing. Sin embargo, el cálculo lambda no hace énfasis en el uso de reglas de transformación y no considera las máquinas reales que pueden im- plementarlo. Se trata de una propuesta más cercana al software que el hardware.

Church redujo todas las nociones del cálculo de sustitución. Normalmente, un matemático debe definir una función mediante una ecuación.

Por ejemplo, si una función f es definida por la ecuación f(x)=t, donde t es algún término que contiene a x, entonces la aplicación f(u) devuelve el valor t[u/x], donde t[u/x] es el término que resulta de sustituir u en cada aparición de x en t.
Por ejemplo, si f(x)=x*x, entonces f(3)=3*3=9.

Lambda Expresiones

Church propuso una forma especial (más compacta) de escribir estas funciones. En vez de decir
"la función f donde f(x)=t", él simplemente
escribió \x.t. Para el ejemplo anterior: \x.x*x.

Un término de la forma \x.t se llama "lambda expresión".

La principal característica de lambda cálculo es su simplicidad ya que permite efectuar solo dos operacio- nes:

- Definir funciones de un solo argumento y con un cuerpo especifico, denotado por la siguiente terminología: λx.B, en donde x determina el parámetro o argumento formal y B representa el cuerpo de la función, es decir f(x) = B.

Ejemplo 1:

Para la función

f: A → B

---

[1] El Entscheidungsproblem (en castellano: problema de decisión) es el reto en lógica simbólica de encontrar un algoritmo general que decida si una fórmula del cálculo de primer orden es un teorema. (Un teorema es una afirma- ción que puede ser demostrada como verdadera dentro de un marco lógico.)

$f(x) \rightarrow 2x + 1$

Podemos escribirla como una λ-expresión de la forma

$\lambda x . 2x+1$

Ejemplo 2:

$f(x, y) = (x + y)*2$

$\lambda x \rightarrow \lambda y \rightarrow (x + y)* 2$

Se expresaría en λ-Calculo como

$\lambda x. \lambda y. * (+ x y) 2$

• Aplicar alguna de las funciones definidas sobre un argumento real (A); lo que es conocido tam- bién con el nombre de reducción, y que no es otra cosa que sustituir las ocurrencias del argumento for- mal (x), que aparezcan en el cuerpo (B) de la función, con el argumento real(A), es decir: (λx.B) A.

Ejemplo 3:

(λ x. (x+5)) 3

lo que indica que en la expresión x+5, se debe sustituir el valor de x por 3.

Los dos mecanismos básicos presentados anteriormente se corresponden con los conceptos de abstrac- ción funcional y aplicación de función; si le agregamos un conjunto de identificadores para repre- sentar variables se obtiene lo mínimo necesario para tener un lenguaje de programación funcional. Lambda calculo tiene el mismo poder computacional que cualquier lenguaje imperativo tradicional.

La sintaxis BNF para las λE es:

exp	::=	cons	constante predefinida
		var	identificador de variable
		(λ var . exp)	abstracción
		(exp exp)	aplicación

Reducción de Expresiones

La labor de un evaluador es calcular el resultado que se obtiene al simplificar una expresión utilizando las definiciones de las funciones involucradas.

Ej: doble :: Integer →
Integer doble x = x
+ x

5 * doble 3
5 * (3 + 3) { por el operador + }
5 * 6 { por el operador * }
30

Una expresión se reduce sustituyendo, en la parte derecha de la ecuación de la función, los Parámetros Formales por los que aparecen en la llamada (también llamados Parámetros Actuales o Parámetros).

Cada paso es una reducción.
Un redex es cada parte de la expresión que pueda reducirse.
Cuando una expresión no puede ser reducida mas se dice que esta en forma normal.

¿Qué ocurre si hay más de un redes? por ejemplo en doble (doble 3)

Se puede reducir la expresión desde dentro hacia fuera (primero los redex internos)

Otra estrategia consiste en reducir desde fuera hacia dentro (primero los redex externos)

El valor obtenido de la función siempre dependerá únicamente de los argumentos y siempre tendrá la consistencia de retornar el mismo valor para los mismos argumentos. Por lo tanto se tiene **transparencia referencial**.

Ordenes de evaluación

Es importante el orden en el que se aplican las reducciones, y dos de los mas interesantes son: Aplicativo y Normal.

Orden Aplicativo

Se reduce siempre el término MAS INTERNO (el más anidado en la expresión). En caso de que existan varios términos a reducir (con la misma profundidad) se selecciona el que aparece más a la izquierda de la expresión.

Esto también se llama paso de parámetros por valor (call by value), ya que ante una aplicación de una función, se reducen primero los parámetros de la función.

A los evaluadores que utilizan este tipo de orden, se les llama estrictos o impacientes

```
doble (doble 3)          por la definición de doble
doble (3 + 3)            por el operador +
doble (6)                por la definición de doble
6 + 6                    por el operador +
12
```

Orden Normal

Consiste en seleccionar el término MÁS EXTERNO (el menos anidado), y en caso de conflicto el que apa- rezca más a la izquierda de la expresión.

Esta estrategia se conoce como "paso de parámetro por nombre o referencia" (call by name), ya que se pasan como parámetros de las funciones expresiones en vez de valores.

A los evaluadores que utilizan el orden normal se les llama "no estrictos".
Una de las características más interesantes es que este orden es normalizante.

```
doble (doble 3)          por la definición de doble
(doble 3) + (doble 3)    por la definición de doble
(3 + 3) + (doble 3)      por la definición de +
6 + (doble 3)            por la definición de doble
6 + (3 + 3)              por la definición de +
6 + 6                    por la definición de +
12
```

Evaluación PEREZOSA o LENTA (Lazy)

No se evalúa ningún elemento en ninguna función hasta que no sea necesario. Las listas se almacenan internamente en un formato no evaluado. La evaluación perezosa consiste en utilizar paso por nombre y recordar los valores de los argumentos ya calculados para evitar recalcularlos.

También se denomina estrategia de pasos de parámetros por necesidad (call by need).

Con una estrategia no estricta de la expresión doble (doble 3), la expresión (3 + 3) se calcula dos veces.

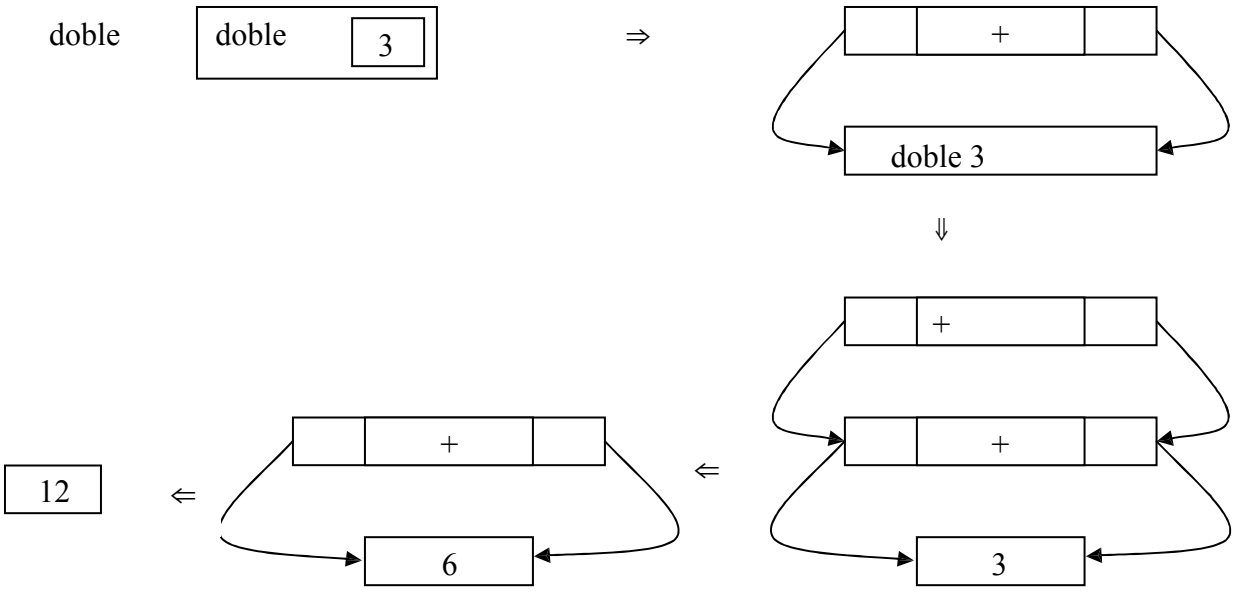
Este tipo de evaluación es útil para trabajar con listas infinitas

Ejemplo:

ones = 1 : ones – Lista con un número infinito de 1s

El problema que tiene este tipo de evaluación es que algunas expresiones se reducen varias veces como por ejemplo, en la expresión doble (doble 3); la expresión (3 + 3) se reduce dos veces, lo que no ocurre en el caso de evaluación impaciente.

```
doble (doble 3)                                  por la definición de doble
a + a            donde a = doble 3               por la definición de doble
a + a donde a = b + b        donde b = 3         por el operador +
a + a donde a = 6                                por el operador +
12
```



Estructuras de datos infinitas y Evaluación Perezosa.

Un beneficio particular de la evaluación perezosa es que hace posible manipular estructura de datos infi- nitas. Por supuesto que no podemos construir o almacenar una estructura infinita completamente. La ventaja de la evaluación perezosa es que permite construir objetos infinitos pieza por pieza según sea necesario.

Como un ejemplo simple, consideremos la siguiente función que puede usarse para construir listas infini- tas de valores enteros.

countFrom n = n : countFrom (n+1)

Si evaluamos la expresión “countFrom 1”, podremos ver que se forma una lista de valores enteros desde 1 hasta que se intrrumpe la ejecución del programa. Evaluar esta expresión equivale a ejecutar un loop infinito que imprima la lista de valores en un lenguaje imperativo.

```
? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,^C{Interrupted!}
(53 reductions, 160 cells)
?
```

Para aplicaciones prácticas, solamente nos interesará construir la lista hasta la posición que nos interese. Por ejemplo al usar “countFrom”junto con la función “take”podemos tomar solo los primeros 10 enteros y encontrar su suma.

```
? sum (take 10 (countFrom 1))
55
(62 reductions, 119 cells)
?
```

Transparencia Referencial

Principio de Transparencia Referencial

"Si en una expresión sintácticamente correcta se cambia una subexpresión por otra, también correcta, que denote el mismo objeto, la expresión resultante denotará el mismo objeto que la expresión inicial."

Según esto, el orden de reducción de las subexpresiones reducibles de una expresión no debe alterar el resultado.

Sea cual fuere la estrategia seguida, el resultado final en ambos tipos de evaluación será el mismo valor
(en el ejemplo: 12).
Si aparecen varios redex podemos elegir cualquiera, sin embargo, la reducción de un redex equivocado, puede que no conduzca a la forma normal de una expresión:

```
infinito :: Integer
infinito = 1 + infinito
cero :: Integer → Integer
cero x = 0
```

Si en cada momento se elige el redex más interno ocurriría lo siguiente:

```
cero infinito                por la definición de infinito
cero (1 + infinito)          por la definición de infinito
cero (1 + (1 + infinito)     por la definición de infinito
...
```

la evaluación no terminaría nunca.

Si elegimos el redex más externo:

```
cero infinito                por la definición de cero
0
```

La selección de la estrategia utilizada para seleccionar el redex es crucial.

Con la evaluación impaciente podríamos efectuar reducciones que no son necesarias

```
cero (10 * 4)          por la definición de *
cero 40                por la definición de cero
0
```

5. PROGRAMACIÓN FUNCIONAL EN LENGUAJE LISP

LISP es un lenguaje diseñado para la manipulación de fórmulas simbólicas. Más adelante, nació su aplicación a la inteligencia artificial. La principal característica de LISPes su habilidad de expresar algoritmos recursivos que manipulen estructuras de datos dinámicos.

En LISP existen dos tipos básicos de palabras, los átomos y las listas. Todas las estructuras definidas posteriormente son basadas en estas palabras.

Atomos

Los átomos pueden ser palabras, tal como CASA, SACA, ATOMO, etc. o cualquier disparate como EDSDS, DFKM454, etc. En general, un átomo en LISP puede ser cualquier combinación de las 26 letras del alfabeto (excluyendo obviamente la "ñ") en conjunto con los 10 dígitos. Al igual que en otros sistemas, no son átomos aquellas combinaciones que comienzan con dígitos.

Ejemplos de átomos

- Hola
- Casa
- Mientras
- Uno34
- F4fg5

Ejemplos de No átomos

5456dgfv	Comienza con dígito.
Ab cd	Incluye un espacio entre medio.
%bc	No comienza con una letra.
A5.)	Incluye caracteres que no son ni letras ni dígitos.

LISTAS

El segundo tipo de palabras con las que trabaja LISP son las listas. Una lista es puede ser una secuencia de átomos separados por un espacio y encerrados por paréntesis redondos, incluyendo la posibilidad de que una lista contenga una sublista que cumple con las mismas características.

EJEMPLOS

- (ESTA ES UNA LISTA)
- (ESTALISTAESDISTINTAALAANTERIOR)
- (ESTA LISTA (TAMBIEN) ES DISTINTA)
- ((ESTA ES OTRA) (POSIBILIDAD DE LISTA))

En adelante, definiremos **TÉRMINO** de una lista como un elemento de una lista, ya sea un átomo o una sublista.

Así, lista quedaría definida como la secuencia:

$$(término_1\ término_2\ .....\ término_k)$$

Donde K es el número de elementos de la lista.

EJEMPLOS

LISTA	NÚMERO DE TÉRMINOS	TÉRMINOS
(HOLA)	1	HOLA
(ESTA ES UNA LISTA)	4	ESTA, ES, UNA, LISTA
((AB T56) HOLA ())	3	(AB T56), HOLA, ()

En LISP, una lista se reconoce porque va entre paréntesis, en cambio, un átomo no.

- (LISTA) es una lista.
- ATOMO es un átomo.

¡ IMPORTANTE !

NO OLVIDAR NUNCA DE REVISAR QUE LOS PARENTESIS ESTEN BIEN

COMANDOS FUNDAMENTALES

QUOTE CAR CDR CONS ATOM EQ NULL

¡ IMPORTANTE !

SIEMPRE REVISAR QUE LAS FUNCIONES RECIBAN EL NÚMERO CORRECTO DE ARGUMENTOS

QUOTE

FUNCION	**:**	**QUOTE**
NUMERO DE ARGUMENTOS	:	1
ARGUMENTOS	:	Un término cualquiera.
RETORNA	:	El argumento.

EJEMPLOS

OPERACIÓN	*RESULTADO*
(QUOTE(ESTA ES UNA PRUEBA))	(ESTA ES UNA PRUEBA)
(QUOTE((ESTA)(ES UNA)PRUEBA))	((ESTA)(ES UNA)PRUEBA)
(QUOTE HOLA)	HOLA
(QUOTE(()))	()

Notar que QUOTE devuelve lo mismo que recibe; aparentemente esto no tiene mucho sentido, no obstante, la utilidad de este comando aparece cuando se utiliza, por ejemplo, el comando CAR entre paréntesis, por ejemplo:

(CAR(A B C))

En este caso, CAR buscará el primer elemento de la lista que genere la función A, pero como A no es una función (a menos que se defina como tal) generará un error. La sentencia correcta sería:

(CAR(QUOTE(A B C))

Un error común es escribir algo así:

(CAR(QUOTE ERROR))

Ya que en este caso QUOTE retorna el átomo ERROR, y CAR debe recibir como argumento una lista (ver definición siguiente).

CAR

FUNCION	**:**	**CAR**
NUMERO DE ARGUMENTOS	:	1
ARGUMENTOS	:	Lista no vacía.
RETORNA	:	El primer término de la lista.

EJEMPLOS

OPERACIÓN	***RESULTADO***
(CAR(QUOTE((ESTA) ES UNA PRUEBA)))	(ESTA)
(CAR(QUOTE((ESTA ES UNA PRUEBA))))	(ESTA ES UNA PRUEBA)
(CAR (QUOTE(()(ESTA ES UNA PRUEBA))))	()
(CAR(QUOTE (ESTA ES UNA PRUEBA)))	(ESTA ES UNA PRUEBA)

Un error común que se comete es algo como lo siguiente:

CAR ((ESTO ES)(UN ERROR))

El primer paréntesis es para indicar que se incluirá el argumento de CAR, lo que no identifica a una lista, luego, en el argumento van dos listas en vez de una, que serían ESTO ES y UN ERROR. Esto se corrige haciendo la llamada:

CAR(((ESTO NO ES)(UN ERROR)))

Generalizando tenemos que cualquier comando que trabaje con una o más listas como argumento debe encerrarlas entre paréntesis, no así con los átomos (ver la aplicación del comando QUOTE).

CDR

FUNCION	**:**	**CDR**
NUMERO DE ARGUMENTOS	:	1
ARGUMENTOS	:	Lista no vacía.
RETORNA	:	El resto de la lista que queda después de borrar el primer término.

EJEMPLOS

(CDR(QUOTE((ESTA ES)UNA PRUEBA)))	(UNA PRUEBA)
(CDR(QUOTE((ESTA ES UNA PRUEBA))))	()
(CDR(QUOTE(()(ESTA ES UNA PRUEBA))))	((ESTA ES UNA PRUEBA))

Las restricciones para CDR son iguales que para CAR.

EJEMPLOS

OPERACIÓN	***RESULTADO***
(CONS(QUOTE(ESTA ES)QUOTE(UNA	PRUEBA)))

ATOM

FUNCION	:	**ATOM**
NUMERO DE ARGUMENTOS	:	1
ARGUMENTOS	:	Cualquier término.
RETORNA	:	T si el argumento es un átomo; NIL en otro caso.

EJEMPLOS

OPERACIÓN	***RESULTADO***
(ATOM(QUOTE ABC54))	T
(ATOM(QUOTE(UN EJEMPLO)))	NIL
ATOM(ABC54)	T
ATOM(ESTO ES UN EJEMPLO)	NIL

EQ

FUNCION	:	**EQ**
NUMERO DE ARGUMENTOS	:	2
ARGUMENTOS	:	Dos términos.
RETORNA	:	T si ambos átomos son iguales; NIL en otro caso.

EJEMPLOS

OPERACIÓN	***RESULTADO***
EQ(HOLA HOLA)	T
EQ(HOLA B)	NIL
(EQ(QUOTE HOLA)(QUOTE HOLA))	T
(EQ(QUOTE G)(QUOTE HOLA))	NIL

NULL

FUNCION	:	**NULL**
NUMERO DE ARGUMENTOS	:	1
ARGUMENTOS	:	Cualquier término.
RETORNA	:	T si el argumento es una lista vacía [()]; NIL en otro caso.

EJEMPLOS

OPERACIÓN	***RESULTADO***
NULL(())	T
NULL((()))	NIL
NULL(ESTA ES UNA PRUEBA)	NIL
(NULL(QUOTE()))	T

ESCRITURA DE PROGRAMAS EN LISP

Un programa en LISP se ejecuta normalmente interpretativa e interactivamente. En su forma más sencilla, un programa o una función se representa como una expresión completamente puesta entre paréntesis con todos los operadores en la forma prefija. Todas las variables tienen valores átomos o listas.

El programa que se muestra a continuación es un programa en LISP que calcula y visualiza la media de una lista de números de entrada. (Aunque esteproblema concreto es la antítesis de los problemas a los que se aplica normalmente el LISP, lo usaremos para ilustrar la sintaxis y desarrollar una basepara la enseñanza del lenguaje.) Por ejemplo, si la entrada es la lista:

(85.5 87.5 89.5 91.5)

Entonces el resultado presentado será el valor 88.5. La variable x se utiliza para almacenar la lista de entrada y la variable n se utiliza para determinar cuántos valores hay. La variable med contiene al final la media calculada.

```
(defun sum (x) ;calcula la suma de una lista x
(cond ((null x) O)
      ((atom x) x)
      (t (+ (car x) (sum (cdr x)]

(defun cont (x) ; cuenta el número de valores de x
      (cond ((null x) O)
            ((atom x) 1)
            (t (addl (cont (cdr x)]

(defun media () ; el programa principal comienza aquí
      (print 'introducir la lista a promediar')
      (setq x (read))
      (setq n (count x))
      (setq med (/ (sum x) n))
      (princ "la media es = ' ')
      (print med)]
```

El programa está compuesto de tres funciones del LISP, cada una indicada por la cabecera "defun" (abreviación de "define function"). La primera función "sum" calcula la suma aritmética de los elementos de la lista x. La segunda, "cont", calcula el *número* de valores de la lista. La tercera función, "media", es el programa principal y controla la entrada (usando "read"), el cálculo de la media "med" y la salida (usando "print").

Los comentarios en LISP comienzan con el delimitador especial punto y coma (;) y continúan hasta el final de la línea. Las variables están normalmente sin declarar y tienen un ámbito global. También pueden especificarse variables acotadas, las cuales son locales a una función.

Los bucles en LISP se dan normalmente mediante la recursividad en vez de mediante la iteración. Entonces, par ejemplo, el cálculo de *sum* se hace mediante la siguiente definición recursiva:

- Si la lista está vacía (es decir, "nula"), la suma es 0.
- Si la lista tiene una entrada, la suma es esa entrada.
- Si la lista tiene más de una entrada, la suma es el resultado de añadir la primera entrada (es decir, el "car") y la suma de la lista compuesta de las restantes entradas (es decir, la "cdr").

Por tanto, si la lista es (1 2 3), entonces su suma es 1 más la suma de la lista (2 3), y así sucesivamente.

La estructura sintáctica del LISP es muy sencilla. El programa es una expresión completamente puesta entre paréntesis, en la cual todas las funciones aparecen como operadores prefijos. En algunas implementaciones del LISP hay dos clases de paréntesis, () y []. Los corchetes se utilizan para especificar cierres múltiples. El corchete derecho,], puede usarse al final de una definición de función para cerrar efectivamente ***todos*** *los* paréntesis izquierdos, (, que le precedan. Esto evita la necesidad de contar y explícitamente equilibrar los paréntesis derechos en muchos casos. Usaremos más adelante este criterio.

Finalmente, observe que no hay una diferencia fundamental entre la estructura de los programas en LISP y sus datos. Esta característica conduce a una fuerte facilidad inherente para que los programas manipulen a otros programas, como veremos. También esto permite una uniformidad básica de las expresiones no encontradas en otros lenguajes. Por tanto, este breve ejemplo contiene los condimentos básicos de la programación en LISP.

DATOS ELEMENTALES: VALORES Y TIPOS

Los tipos de datos elementales del LISP son los "números" y "símbolos". Un número es un valor que es un entero o un real (decimal). Los siguientes son ejemplos de números:

0	-17
234	49.5
10.5E-5	-7E4

Los últimos dos ejemplos son abreviaciones de la notación científica, donde E significa "potencias de 10" como en otros lenguajes.

Un símbolo comprende cualquier cadena de caracteres que no representa un número decimal. Los siguientes son símbolos válidos en LISP:

med	naranja
NOMBRE	alfa
A1	2+3

FUNCIONES EN LISP

Las siguientes sonlas funciones que conformanel cuerpo de las funciones de LISP, sólo se incluyen en la tabla las funciones que se presentan en la mayoríade las versiones de LISP (incluyendo las vistas anteriormente).

abs	and	append	apply	assoc	map
car	cdr	close	cond	mapc	difference (-)
defun (de,df)	dm(macro)	eq	equal	eval	explode
function	gensym	get	getd	go	greaterp(gt)
intern	lambda	length	list	mapcan	mapcar
lessp (It)	expt	fix	fixp	float	floatp
mapcon	maplist	max	min	nconc	not
numberp	null	open	or	princ	print
prin1	prog	progn	put	quote (')	quotient (/)
read	remainder	remob	remprop	return	reverse
rplaca	rplacd	set	setq	subst	plus (+)
terpri	times (*)	cons	atom		

Una función en LISP se escribe siempre de la siguiente forma general:

(nombre argl arg2...)

"Nombre" identifica a la función y "argl", "arg2", ... son los argumentos a los que se aplica la función. Por ejemplo: (+ 2 3) denota la suma 2 + 3, mientras que (list 2 3) denota la construcción de una lista cuyos elementos son 2 y 3, lo que se presenta como: (2 3)

Las funciones pueden anidarse arbitrariamente, en cuyo caso se evalúan de "dentro a fuera". Por tanto,

(+ (* 2 3) 4)

denota la suma de (2*3) y 4. Además, una lista de funciones se evalúa de izquierda a derecha. Por ejemplo,

(-(+23)(*34))

denota la diferencia de la suma 2 + 3 seguida del producto 3 * 4.

NOMBRES, VARIABLES Y DECLARACIONES

Una variable en LISP tiene un nombre, el cual puede ser cualquier símbolo y un valor que puede ser un átomo o una lista. Los siguientes son ejemplos de nombres de variables en LISP:

X	med
ALFA	comida

(Algunas implementaciones no permiten las letras minúsculas). Los nombres también se utilizan para identificar funciones, tales como las mostradas en el programa ejemplo anterior: ***sum***, ***cont***, ***media***.

Normalmente, en LISP no se declaran Las variables; la activación de una variable ocurre dinámicamente cuando se hace la primera referencia a ella durante la ejecución del programa. Además, el *tipo* del valor almacenado en una variable puede variar durante la ejecución. Por tanto, cuando una variable se utiliza en un contexto aritmético, el programa debe asegurar que su valor actual es numérico. Si no, se producirá un error en tiempo de ejecución.

Debe evitarse el uso de nombres de funciones del cuerpo del LISP para los nombres de variables, aunque no sean, estrictamente hablando "palabras reservadas". Por ejemplo, la legibilidad de un programa se hace más difícil cuando se utiliza "and" y "or" como nombres de variable en la siguiente expresión:

(and and or)

Debido a que las variables no se declaran, no puede suponerse que tengan ningún valor inicial preasignado. En algunos sistemas LISP, Las variables se inicializan todas automáticamente a "nil" (nada), pero depender de esto no es normalmente una buena práctica de programación.

ARRAYS Y OTRAS ESTRUCTURAS DE DATOS

ARRAYS

Un array puede declararse explícitamente en algunos dialectos del LISP usando la función "array", la cual tiene la siguiente forma:

(ARRAY NOMBRE T TAMAÑO)

"Nombre" identifica al array y "tamaño" es una secuencia de enteros que identifica al número de elementos de cada dimensión. Por ejemplo, supongamos que A es un array de cinco elementos y B es un array de 5 x 4. Estos arrays pueden declararse como sigue:

(array A t 5) (array B t 5 4)

Una entrada a un array se referencia mediante una lista que contiene el nombre del array y una serie de subíndices que identifican la posición de la entrada en el array. Para este propósito, Las filas y las columnas se prenumeran desde 0 (como en el lenguaje C), en vez de desde 1 (como en la mayoría de los demás lenguajes). Por tanto, la tercera entrada de A se referencia por (A 2) y el elemento de la cuarta fila y tercera columna de B se referencia por (B 3 2).

Para asignar un valor a una entrada de un array, se utiliza la siguiente función:

(store (nombre subíndices) valor)

Por ejemplo, para almacenar el valor 0 en la entrada de la cuarta fila y tercera columna de B escribimos:

(store (B 3 2) 0)

En general, el valor almacenado puede especificar cualquier expresión en LISP y cada uno de los subíndices pueden también ser cualquier expresión en LISP cuyo valor sea un entero en el rango adecuado. Por tanto, los arrays en LISP generalmente no tienen tipo, puesto que cada entrada puede ser diferente en estructura y tipo del resto.

LISTA DE PROPIEDADES

Un método mucho más útil de estructurar datos en una lista es mediante la llamada "lista de propiedades". Esta es una lista que tiene la siguiente forma general:

(p1 V1 p2 V2 .. pn Vn)

donde los p son átomos que denotan propiedades y las V son valores asociados a estas propiedades. Para ilustrar esto, supongamos que tenemos la información para un individuo clasificada de la siguiente forma:

```
(NOMBRE (ALLEN B TUCKER)
SS# 275407437
SUELDQ BRUTO 25400
DIRECCION ((1800 BULL RUN) ALEXANDRIA VA 22200)
      )
```

Aquí hay cuatro propiedades, un nombre, un número de la Seguridad Social, un sueldo bruto y una dirección. Lo anterior corresponde a la definición de una lista de propiedades (denominada por ejemplo PERSONA).

Para obtener información de una lista de propiedades usamos la función "get" como sigue:

(get nombre p)

"Nombre" identifica a la lista y p identifica la propiedad cuyo valor se desea. Por ejemplo, para obtener el SS# de una PERSONA escribimos:

(get PERSONA SS #)

y el valor devuelto, para el anterior ejemplo, será 275407437.

Para reemplazar información en una lista de propiedades, se utiliza la función "put":

(put nombre p v)

"Nombre " identifica a la lista, p la propiedad cuyo valor va a ser reemplazado y v es el nuevo valor. Por ejemplo:

(put PERSONA SUELDO_BRUTO 30000)

Altera la propiedad SUELDO_BRUTO de PERSONA, de forma que su valor se hace 30000 en vez de 25400, como había sido en el presente ejemplo.

Finalmente, la función "remprop" *quita* una propiedad y su valor asociado de la lista.

(remprop nombre p)

"Nombre" identifica la lista de propiedades afectada y p identifica la propiedad y el valor que han de quitarse.

Por tanto, las listas de propiedades son muy útiles para definir lo que se conoceen otros lenguajescomo "registros". Típicamente una lista de propiedades representa un nodo en una lista mayor, enlazada de registros, todos con el mismo conjunto de propiedades. Esta lista mayor es conocida comúnmente como un "archivo" en otros lenguajes. Por ejemplo, el registro PERSONA puesto anteriormente puede ser un nodo de una lista que contenga todas las personas empleadas en una determinada organización. Igualmente, puede definirse otra lista de propiedades especificas para una entrada en un catálogo de fechas de una librería y la lista de todas las entradas puede representar el propio catálogo.

FUNCIONES BÁSICAS

Anteriormente se dieron a conocer los comandos básicos de LISP, en esta parte se darán a conocer más a fondo las funciones clasificadas desde otro punto de vista.

Funciones aritméticas y asignación de valor

Las funciones aritméticas básicas del LISP son:

PLUS	DIFFERENCE	TIMES	QUOTIENT
SETQ	SET	ADD1	SUB1

Las cuales se abrevian en la mayoría de los sistemas mediante los familiares signos +, -, * y /. Todas estas funciones tienen cero o más operandos, o argumentos, y se escriben en la forma prefija como las otras funciones del LISP. Cuando se combinan para formar el equivalente a las expresiones aritméticas, estas funciones están completamente puestas entre paréntesis y, por tanto, no hay necesidad en LISP de definir ninguna precedencia jerárquica entre ellos.

Para ilustrar esto, supongamos que H, I, N y X son variables en LISP. Las expresiones algebraicas mostradas en la parte de la izquierda (que es como se escribirían en otros lenguajes) se escriben en LISP como se muestra a la derecha:

EXPRESION	***FORMA EN LISP***
H + 2	(+ H 2)
H + 2 + I	(+(+H 2)I)
H + 2 * I	(+H(*2 I))
SUM(X)/N	(/(SUM X)N)

Puesto que las formasenLISPestán completamente encerradas entre paréntesis, su orden de evaluación es siempre explícito. El segundo ejemplo supone que la evaluación es de izquierda a derecha para las operaciones con la misma precedencia, mientras que el tercero supone que "*" tiene precedencia sobre "+" en las expresiones de la izquierda. El cuarto ejemplo ilustra laaplicaciónde una función definida par el programador en el contexto de una expresión mayor.

La asignación del valor de una expresión a otra variable se realiza por la función "setq" o la función "set". Estas funciones tienen dos operandos:

(setq variable expresión)
(set 'variable expresión)

En cualquiera de las dos, "variable" es el nombre de una variable que es el destino de la asignación y "expresión" es una lista cuyo resultado se asignará a la variable. Por ejemplo, la función

(setq I (+ I 1))

Es equivalente a la familiar forma I := I + 1 en Pascal. Se evalúa primero la expresión de la derecha y el resultado se asigna a la variable

I. En general, este resultado puede ser un átomo (un número o un símbolo) o una lista, como veremos en posteriores ejemplos.

Las funciones unarias "addl" y "subl" se dan en LISP para simplificar la especificación de la común operación de sumar o restar 1 del valor de una variable. Esta operación se encuentra en todas Las aplicaciones de programación y es especialmente frecuente en la programación *recursiva.*

Comilla (') y Evaluación

La comilla ('), la letra q en setq y la función "quote" se utilizan en LISP para distinguir explícitamente entre expresiones evaluadas. Esta distinción se necesita debido a la naturaleza dual de un símbolo en LISP en algunos contextos: su uso como un ítem de dato y su uso como un nombre de variable. Si"E" denota cualquier expresión en LISP entonces su aparición dentro de un programa implica normalmente que va a evaluarse inmediatamente y que el valor resultante se utilizará a continuación. Sin embargo, la expresión

(quote E)

Que normalmente se abrevia como 'E, denote que E permanece *por* si misma *y* que **no** va a ser evaluada.

Por ejemplo, consideremos las siguientes dos expresiones, en donde X se va a usar como variable (en la izquierda) y como un valor literal (en la derecha):

COMO VARIABLE	*COMO LITERAL*
((SETQ X 1)	((SETQ X 1)
(SETQ Y X))	(SETQ Y 'X))

En la expresión de la izquierda, a las variables X e Y se les asigna a ambas el valor 1. En la expresión de la derecha, a X se le asigna el valor I y a Y se le asigna el valor (literal) X. El último es, en efecto, una cadena de caracteres de longitud 1 o un átomo no numérico.

Esto ayuda a explicar la diferencia entre "set" y "setq" del párrafo anterior. "Setq" es solo una forma conveniente de combinar "set" con un primer argumento con comilla. Es decir, el nombre de la variable de la izquierda de una asignación no debe ser evaluado; designa una dirección en vez de un valor. Por tanto, las tres formas siguientes son equivalentes:

(setq X 1)
(set,X 1)
(set (quote X) 1)

En la práctica se prefiere normalmente la primera forma; la tercera es el precedente histórico de las otras dos.

Funciones De Manipulación De Listas

La principal fuerza del LISP consiste en su potencia para manipular **expresiones simbólicas** en vez de expresiones numéricas. Para ilustrar esto, supongamos que tenemos una lista L compuesta de los nombres de nombres de lenguajes de programación. Esto es,

L = (pascal, fortran, cobol, pli)

El valor de L puede ser asignado mediante la siguiente sentencia:

(setq L '(pascal fortran cobol pli))

Observe aquí que la comilla (') fuerza a que se trate a la lista de lenguajes como un literal, en vez de como una colección de variables denominadas "pascal", "fortran", etc.
Recuerde que, en general, una lista puede ser nada, un átomo o una serie de elementos entre paréntesis (e1 e2 ... en).

Las dos funciones básicas vistas anteriormente, "car" y "cdr", se utilizan para dividir listas (es decir, sus representaciones subyacentes) en dos partes. "Car" define al primer elemento, e1, de una lista y "cdr" define a la lista que comprende el resto de los elementos (e2 ... en). En el caso especial en que n=1, la cdr se define como nada. Cuando la lista original es un átomo simple o nada, Las funciones car y cdr producen valores indefinidos y se producirá un error en tiempo de ejecución. (Algunas implementaciones son una excepción y definen a car y cdr como nada en este caso especial. Este compromiso simplifica algunas veces la programación de las situaciones "límites", aunque sacrifica la consistencia y transportabilidad.) Ejemplos de aplicación de estas funciones se pueden ver en la primera parte de este apunte.

Algunas de las múltiples aplicaciones de las funciones car y cdr pueden abreviarse en LISP con los siguientes criterios:

FUNCION	*RESULTADO*
(CAR (CDR L))	(CADR L)
(CDR(CAR L))	(CDAR L)
(CAR(CAR .. (CAR L)..)))	(CAA .. AR L)
(CDR(CDR .. (CDR L)..)))	(CDD .. DR L)

Recuerden que todas estas aplicaciones son en el contexto de la construcción de funciones.

Algunas implementaciones del LISP limitan el grado de anidamiento que puede abreviarse de esta forma. Sin embargo, para la mayoría de las implementaciones pueden asumirse con seguridad al menos tres niveles.

En contraste con la división de una lista en sus constituyentes está la función de construir una lista a partir de otras listas. Esto se realiza en LISP mediante las siguientes funciones. En esta descripción, e1, e2, ..., son términos cualesquiera.

FUNCION	SIGNIFICADO
(CONS e1 e2)	YA VISTA.
(LIST e1 e2 .. en)	Construye una lista de la forma (e1 e2 .. en.
(APPEND e1 e2 .. en)	Construye una lista de la forma (f1 f2 .. fn), donde cada f es el resultado de quitar los paréntesis exteriores de cada e. Aquí los términos e no pueden ser átomos.

Para ilustrar el uso de estas funciones, reconsideremos la lista L cuyo valor es (pascal fortran cobol pli). Si necesitamos construir una nueva lista M, cuyos elementos sean los de la lista L y los de la nueva lista (snobol apl lisp), podemos escribir lo siguiente:

(setq M (list L '(snobol apl lisp)))

El valor resultante de M es:

((pascal fortran cobol pli) (snobol apl lisp))

la cual es una lista de **dos** elementos, no de **siete.**

Por otra parte, como ya vimos, "cons" tiene siempre dos argumentos. Por tanto,

(cons 'snobol (cons 'apl (cons lisp nil)))

construye la lista

(snobol apl lisp)

Por tanto, esta lista podría haberse construido de forma equivalente mediante la siguiente función:

(list 'snobol 'apl 'lisp)

"Append" es alga diferente de "list", en el sentido de que necesita que sus argumentos sean listas encerradas entre paréntesis y ella quite dichos paréntesis para construir el resultado. Por tanto,

(append L '(snobol apl lisp))

da la lista de siete elementos:

(pascal fortran cobol pli snobol apl lisp)

la cual es estructuralmente distinta de la lista M formada en el anterior ejemplo .

Finalmente, se añaden a este grupo de funciones las resumidas en la tabla siguiente.(Todas ellas usan como parámetros términos, se debe verificar que parámetros son restringidos a solamente listas.)

FUNCION	SIGNIFICADO
(RPLACA e1 e2)	Reemplaza el car de e1 por e2.
(RPLACD e1 e2)	Reemplaza el cdr de e1 por e2.
(SUBST e1 e2 e3)	Reemplaza cada instancia de e2 en e3 por (el sustituto) e1.
(REVERSE (e1 e2 .. en))	Invierte los elementos de la lista formando (en .. e2 e1).
(LENGTH (e1 e2 .. en))	El número de términos de la lista n.

En las funciones "rplaca" y "replacd", el argumento e1 debe denotar una lista puesta entre paréntesis, para que las correspondientes car y cdr queden bien definidas. Igualmente, el argumento e3 de la función "subts" debe ser también una lista encerrada entre paréntesis.

Para ilustrar esto, supongamos de nuevo que tenemos las listas L y M, con los valores respectivos (pascal fortran cobol pli) y ((pascal fortran cobol pli)(snobol apl lisp)). Las siguientes expresiones conducen a los resultados mostrados a la derecha:

EXPRESION	RESULTADO
(RPLACA L 'MODULA)	(MODULA FORTRAN COBOL PLI)
(RPLACD M 'PROLOG)	((PASCAL FORTRAN COBOL PLI)PROLOG)
(REVERSE (CARD M))	(LISP APL SNOBOL)
(SUBST 'ADA 'PLI L)	(PASCAL FORTRAN COBOL ADA)
(LENGTH L)	4
(LENGTH M)	2

ESTRUCTURAS DE CONTROL

Las funciones en LISP pueden evaluarse en serie, condicional, iterativa o recursivamente. La recursividad se estudiará más adelante, mientras que la evaluación condicional e iterativa se tratan a continuación.

Detrás de la noción de evaluación condicional existe una colección de funciones en LISP, las cuales se clasifican como "predicados". Un predicado es cualquier función que cuando es evaluada devuelve el valor t (significando true) o nil (significando false). En otros lenguajes, estos predicados básicos se definen normalmente vía operadores "relacionales" y "booleanos". A continuación se da una lista de los principales predicados del LISP, junto con sus significados. Aquí e, e1 y e2 son listas, x, x1 y x2 son expresiones aritméticas *y* p, p1, p2, ..., son predicados.

PREDICADO	**SIGNIFICADO**
(PLUSP X)	Devuelve t si x>0 y ni si no lo es.
(MINUSP X)	Devuelve t sí x<0 y nil en los demás casos.
(ZEROP X)	Devuelve t si x=0 y nil en los otros casos.
(LESSP x1 x2)	Devuelve t si x1<x2 y nil en los demás casos.
(GREATERP x1 x2)	Devuelve t si x1>x2 y nil en los demás casos.
(AND p1 p2 .. pn)	Devuelve t si todos los p1, p2, .., pn son t y nil en los otros casos.
(OR p1 p2 .. pn)	Devuelve t si uno o más de los p1, p2, .., pn es t y nil en los demás casos.
(NOT p)	Devuelve t si p es nil y nil en los demás casos.
(FIXP x) - (FLOATP x)	Devuelve t si x es entero - punto flotante, respectivamente, y nil en los demás casos.
(EQUAL e1 e2)	Devuelve t si el valor de e1 es el mismo que el de e2 y nil en los otros casos.
(NUMBERP e)	Devuelve t si e es un átomo numérico y nil en los otros casos.
(ATOM e)	YA VISTA.
(NULL e)	YA VISTA.

Para ilustrar estas funciones, supongamos que la lista PUNTUACIONES tiene el valor (87.5, 89.5. 91.5) y la lista L tiene de nuevo el valor (pascal fortran cobol pli). La mayoría de los ejemplos de la tabla siguiente utilizan estos valores para ilustrar los predicados del LISP.

Algunos de estos ejemplos ilustran las situaciones "límites" que surgen en el procesamiento de listas y la importancia de aplicar funciones a unos argumentos con el tipo correcto. Por ejemplo, los predicados "zerop", "plusp", "minusp", "lessp" y "greaterp" se aplican principalmente a expresiones que tienen valores numéricos.

PREDICADO	RESULTADO
(PLUSP (CAR SCORES))	t
(MINUSP 3)	Nil
(ZEROP(CAR L))	Indefinido
(LESSP (CAR SCORES) (CADR SCORES))	t
(GREATERP (CAR SCORES) 90)	Nil
(AND (PLUSP(CAR SCORES)) (LESSP(CAR SCORES) 90)	T
(EQUAL (CAR L) 'LISP)	Nil
(OR (GREATERP(CAR SCORES)90) (GREATERP(CADR SCORES)90))	Nil
(NUMBERP(CAR L))	Nil
(NOT(NUMBERP(CAR L)))	T

Expresiones condicionales

Una expresión condicional en LISP es equivalente a una serie de sentencias if anidadas en lenguajes tipo Pascal. Tiene la siguiente forma general:

```
(cond (p1 e1)
(p2 e2)
.
.
.
(pn en))
```

Cada uno de Los p1, ..., pn, denota un predicado y cada una de las e1, ..., en, la expresión que le corresponde, devolviendo la función como resultado la primera e de la secuencia, para la cual el correspondiente p es verdad y saltándose el resto.(Si ninguno de los p es verdad, la condición cond devuelve nil.)

Por tanto, en efecto, esto es lo mismo que el siguiente código en Pascal:

```
if p1 then e1
else if p2 then e2
else
.
.
.
else if pn then en
```

En el caso de que queramos que la última alternativa en se evalúe en "todos Los demás casos" —es decir, siempre que todos los p sean nil— entonces ponemos pn a t en esta expresión condicional. Por ejemplo, supongamos que queremos calcular el IMPUESTO de una persona como

el 25% de sus ingresos BRUTOS siempre que este último exceda de 18.000 dólares y como el 22% de BRUTO en los demás casos. La siguiente expresión condicional realiza esto:

```
(cond ((greaterp BRUTO18000) (setq IMPUESTO (* 0.25BRUTO)))
(t (setq IMPUESTO (* 0.22 BRUTO))))
```

Por tanto, esto es equivalente a la siguiente expresión en Pascal:

```
if BRUTO >18000 then IMPUESTO := .25 * BRUTO else IMPUESTO := .22 * BRUTO
```

Surge una confusión cuando alcanzamos el final de una expresión compleja en LISP, que se refiere sobre cuántos paréntesis derechos deben aparecer al final para mantener el equilibrio necesario. Esta es una buena situación para usar el corchete,], para forzar un cierre múltiple sin tener que contar los paréntesis izquierdos existentes. Por tanto, podemos escribir lo anterior como:

```
(cond ((greaterp BRUTO 18000) (setq IMPUESTO (* 0.25 BRUTO)))
(t (setq IMPUESTO (* 0.22 BRUTO]
```

y todos los paréntesis abiertos y no cerrados se cerrarán hasta el comienzo de la expresión condicional.

Iteración

Aunque la *recursividad* es la forma primaria de expresar los procesos repetitivos en LISP, en algunas situaciones se prefiere la especificación de bucles "iterativos". Para servir a esta necesidad, LISP contiene la "característica prog", la cual, combinada con la función "go" (sí, ¡es la vieja sentencia goto!) permite que se especifique un tipo primitivo de bucle. Además, algunas implementaciones del LISP contienen otras estructuras de control comparables a las sentencias while o for encontradas en lenguajes como el Pascal .

La "característica prog" también contiene una facilidad para definir "variables locales" dentro de la definición de una función. A menos que se especifique así, todas las variables de un programa en LISP son (por defecto) de ámbito global. La forma general de la característica prog es la siguiente:

```
(prog (locales) e~ e2... en)
```

"Locales" es una lista que identifica a las variables locales de esta función. Cada una de Las e denote un término arbitrario en LISP y puede estar precedido opcionalmente por una "etiqueta". La etiqueta puede ser, a su vez, cualquier símbolo único y sirve para dar un punto de bifurcación a la función "go" que aparece en cualquier lugar entre estas expresiones. La función go tiene la siguiente forma general:

```
(go etiqueta)
```

"Etiqueta" es la etiqueta simbólica que precede a alguna otra expresión dentro de la lista de expresiones. Por ejemplo, si queremos especificar

la ejecución repetida de una expresión e 10 veces, controlada por la variable (local) I, podemos escribir el siguiente segmento de programa:

```
(prog (I)
(setq I 1) bucle
(setq I (addl b)
(cond ((lessp I 11) (go bucle))) )
```

Esto es equivalente al siguiente bucle en un lenguaje como el Pascal:

```
I:= 1;
bucle:
I := I + 1;
if I < 11 then goto bucle
```

CRITERIOS DE ENTRADA-SALIDA

LISP es un lenguaje interactivo, por lo que las funciones de entrada-salida se realizan principalmente sobre el terminal. La mayoría de las implementaciones permiten también el almacenamiento de archivos en memoria secundaria, pero esto es muy dependiente de la implementación. En esta parte, trataremos sólo con las funciones de entrada-salida orientadas a terminal.

La función "read" no tiene argumentos y hace que se introduzca una lista por el terminal. Tiene la siguiente forma:

(read)

Cuando se encuentra esta función, el programa espera que el usuario introduzca una lista, la cual se convertirá en el valor devuelto por esta función. Para asignar ese valor a una variable del programa, read puede combinarse dentro de una función "setq", como sigue:

(setq X (read))

En efecto, esto dice "asignar a X el siguiente valor de entrada". Por tanto, si escribimos 14 en respuesta a la función read, 14 será el valor de X.

La forma más directa de visualizar la salida sobre la pantalla de un terminal es usar la función "print", la cual tiene la siguiente forma:

(print e)

Aquí e puede ser cualquier expresión en LISP, y su valor se presentará sobre la pantalla como resultado de la ejecución de esta función.

En LISP existen tres variaciones disponibles de "print", las cuales se llaman "terpri", "prin1" y "princ". La expresión

(**terpri)**

se utiliza para saltar al comienzo de una nueva línea. La expresión

(prin1 e)

es como (print e), excepto que *no* comienza en una nueva línea. La expresión

(princ e)

se utiliza para suprimir las barras verticales, las cuales se utilizan para encerrar los átomos que contienen caracteres especiales (como "$" y "blanco", los cuales no se permiten normalmente dentro de un átomo). Por ejemplo, si quisiéramos que un átomo tuviera el valor ¡HURRA! tendríamos que encerrarlo dentro de barras verticales, como las siguientes:

|¡ HURRA !|

Si visualizáramos esto usando print o **prin1**, Las barras verticales también aparecerían.

SUBPROGRAMAS, FUNCIONES Y BIBLIOTECAS

Cada implementación del LISP contiene una extensa biblioteca de funciones predefinidas, para el procesamiento de listas y cadenas. Los siguientes son ejemplos encontrados en la mayoría de las implementaciones. x, xl, .., xn son expresiones numéricas.

FUNCION	*SIGNIFICADO*
(ABS x)	Valor absoluto de x.
(MAX x1 x2 .. xn)	Valor máximo de x1, x2, .. , xn.
(MIN x1 x2 .. xn)	Valor mínimo de x1, x2, .. , xn.
(EXPT x)	Función exponencial de x, e^x .

Además, LISP contiene poderosas facilidades para que el programador extienda el lenguaje definiendo funciones adicionales. Esta característica tiene la siguiente forma general:

(defun nombre (parámetros) e1 e2 ... en)

"Nombre" identifica a la función, "parámetros" es una lista de símbolos atómicos que son los parámetros de la función y e1, e2, ..., en son las expresiones que definen la función. Tres de tales funciones, llamadas "sum", "cont" y "media", se han definido en el programa anterior. Las primeras dos tienen el parámetro x, mientras que la tercera *no* tiene parámetros y sirve como el programa principal.

Las diferentes implementaciones del LISP contienen algunas diferencias sintácticas para la definición de funciones. Algunas de estas variaciones respecto a la forma dada son las siguientes:

- **(def nombre) (lambda (parámetros) e, e2 ... en))**
- **(de nombre (parámetros) e, e2 ... en)**
- **nombre: (lambda (parámetros) el e2 ... en)**

La palabra "lambda" proviene de las primeras versiones del LISP, las cuales tenían una sintaxis más parecida a la notación de Church, de la que proviene el LISP. Pero todas estas versiones sirven al mismo propósito y ninguna es intrínsecamente superior a las otras. Usamos la forma original a lo largo de este manual, suponiendo que el lector podrá asimilar las otras variaciones si es necesario.

En el corazón de la definición de función está la idea de la recursividad. Esto es al LISP como la "sentencia WHILE" al C. La definición de funciones recursivas proviene directamente de las matemáticas, como se ilustra en la siguiente definición de la función factorial.

factorial (n) = 1 si n <= 1
= n * factorial(n-1) si n>1

Como es evidente, la definición del factorial se basa en si misma para poder calcular el factorial de cualquier valor particular de n > 1.

Por ejemplo, el factorial de 4 depende de los anteriores cálculos del factorial de 3, y así sucesivamente. El proceso termina cuando se llega al factorial de 1, que es cuando puede completarse cada uno de losotros cálculos dependientes de los anteriores. Esta definición de función puede escribirse directamente en LISP, aprovechándose del hecho de que una función puede llamarse a si misma.

```
(defun fact (n)
(cond ((lessp n 2) 1)
(t (* n (fact (sub I n)]
```

Hemos identificado al parámetro n y definido "fact" mediante una expresión condicional que es equivalente a la siguiente expresión en un lenguaje tipo Pascal:

```
if n<2then 1
else n * fact (n - 1)
```

En un caso, el resultado devuelto será 1, mientras que en los otros el resultado será el cálculo (* n (fact (subl n))), el cual llamará a la propia función.

Para llamar así a una función, se utiliza la misma forma que para las funciones dadas par el LISP:

(nombre argl arg2...)

"Nombre" identifica a la función y "argl arg2..." es una serie de expresiones, correspondiente a los parámetros de la definición de la función.

Por tanto, para calcular el factorial de 4 escribimos

"(fact 4)"

El resultado de la evaluación dará la siguiente expresión

(* 4 (fact 3))

que, una vez evaluada, dará lugar a la expresión

(* 3 (fact 2))

y finalmente se llegará a lo siguiente:

(* 2 (fact 1))

Ahora, cuatro llamadas diferentes de "fact" están activas y la última devuelve finalmente el valor 1. Esto permite que se evalúe la expresión (*2 1) y que el resultado 2 se pose a la expresión (* 3 2), y así sucesivamente hasta que se completan todas las activaciones.

Otras dos formas sencillas de definiciones recursivas aparecen en el programa del comienzo, una para la función "sum" y la otra para la función "cont". Cada una de ellas combina las funciones primitivas de procesamiento de listas "car" y "cdr" con los cálculos numéricos, para llegar al resultado deseado. Aquí la recursividad es esencial porque las *longitudes* de las listas varían normalmente de una ejecución a la siguiente.

Un examen de la función sum con el argumento (87.5, 89.5, 91.5, 93.5) muestra que, puesto que este argumento ni es "nulo" ni es un átomo, la llamada recursiva

(+ (car x) (sum (cdr x)))

se evalúa a continuación.

Funciones, Variables Locales Y La Característica Program

Aunque la recursividad es el dispositivo primario para definir funciones en LISP, en algunas ocasiones es necesaria la iteración. Además, la mayoría de las funciones requieren el uso de variables locales para disponer de un almacenamiento temporal mientras realizan sus tareas. Cualquiera de estos dos requerimientos fuerzan al uso de la "característica prog" dentro de la definición de una función como sigue:

(defun nombre (parámetros) (prog locales) e1 e2 ... en))

La secuencia de expresiones e1, e2, ..., en puede incluir ahora a la función go, mientras que "parámetros" y "locales" tienen el mismo significado que se dio originalmente.

En muchos casos, la naturaleza de la función fuerza la denotación explícita del resultado que ha de devolverse en la llamada. Esto se activa mediante la función "return", la cual tiene la siguiente forma dentro de una definición de función:

(return expresión)

"Expresión" es el valor que ha de devolverse a la expresión llamadora y la devolución del control se produce en cuanto se encuentra esta función. Para ilustrar esto, se muestra a continuación una interpretación iterativa de la función factorial:

```
(defun fact (n)
(prog (i f)
(setq f 1)
(setq i 1)
bucle (cond ((greaterp i n) (return f))
(t ((setq f (* f i))
(setq i (addl i))
(go bucle)]
```

Aquí tenemos las variables locales f e i inicializadas ambas a 1. La sentencia condicional etiquetada con "bucle" se repite hasta que i > n, devolviéndose en ese momento el valor resultante de f. Si no, se realizan los cálculos

```
f:=f*i
yi:=i+ I
```

y se repite el test para i > n.

Aunque este ejemplo ilustra el uso de la función "return", la característica prog y la iteración sirve también para subrayar la superioridad expresiva de la recursividad como dispositivo descriptivo del LISP.

OTRAS CARACTERISTICAS

Entre las restantes características del LISP, la facilidad de definición de macro y las funciones "eval", "mapcar", "mapcan" y "apply" son, quizá, las más importantes.

Definición y expansión de macros

La noción de "macro", en general, es la de que una función puede automáticamente reinstanciarse o "generarse" *en* línea dentro del texto de un programa siempre que se necesite. Las macros han sido una importante herramienta de los programadores de sistemas durante mucho tiempo, pero su potencia se ha utilizado principalmente a nivel del lenguaje ensamblador.

En LISP, las macros ofrecen una alternativa a las funciones para el mismo propósito. Una definición de macro en LISP tiene la siguiente forma básica:

(defun nombre macro (parámetro) e1 e2 ... en)

(Algunas implementaciones requieren una sintaxis alga diferente a ésta, usando "dm" para la definición de macro en vez de "defun" y "macro". El lector debe ser capaz de asimilar estas diferencias sintácticas.) "Nombre" identifica a la macro y "parámetro" es un valor que será sustituido en las expresiones e1, e2, ... y en cuando se expanda la macro. El resultado de la expansión se ejecuta entonces en línea.

Una macro se llama igual que si fuera una función:

(nombre argumentos)

Pero la acción que tiene lugar no es una transferencia de control, como con las funciones, sino una generación en línea de texto del programa, el cual se ejecuta a continuación.

Para ilustrar esto, supongamos que queremos definir una macro que simule una sentencia **while** de otro lenguaje; esto es,

(while X Y)

ejecutaría repetidamente Y hasta que X se hace 0. La "forma" de este bucle será como sigue:

```
(prog ()
bucle (cond ((greaterp X 0)
      (Y
      (setq X (subl X))
      (go bucle)))))
```

Esto es, si X > 0 entonces se ejecuta Y, se decrementa X en 1 y se repite el bucle.

Puede definirse entonces una macro llamada "while" con la anterior configuración como su cuerpo y representando el parámetro P toda la llamada a la macro (while X Y), como sigue:

```
(defun while macro (P)
      (subst (cadr P) 'X) (
      subst (caddr P) 'Y)
      (prog ()
      bucle (cond ((greaterp X 0)
      (Y (
      setq X (subl X)) (go bucle)))))
```

Las dos funciones "subst" reemplazan al primer y segundo argumento de la llamada para X en el cuerpo de la definición de la macro y luego se ejecuta el cuerpo. Por ejemplo, la siguiente llamada a macro repite el cálculo del factorial F de N, mientras el valor de I es mayor que 0.

(while I (setq F (* F D))

La expansión de esta llamada a macro, cuando es ejecutada, aparece a continuación

```
(prog ()
bucle (cond ((greaterp I 0)
((setq F (* F b)
(setq I (subl b)
(go bucle)))))
```

Observe que (cadr P) es I en este caso y (caddr P) es (setq F (* F D). Este código puede compararse con la versión iterativa de la función factorial presentada anteriormente.

Eval, Mapcar Y Aaply

La macro es un dispositivo para la suspensión temporal de la ejecución de un programa, mientras se generan o transforman automáticamente ciertas sentencias del programa.

Lo mismo puede hacerse de una forma más modesta usando la función "eval".

Eval es la función opuesta de "quote", en el siguiente sentido. Si una variable X tiene el valor (A B), entonces la expresión

(list X 'C)

da el resultado (A B C). Sin embargo, la expresión (list 'X 'C) fuerza a que X se trate como un literal sin valor, en vez de como una variable, por lo que el resultado es (X C).

Cuando se aplica eval a una expresión con comilla, anula el efecto de la comilla y fuerza a que se evalúe la expresión. Por tanto, continuando con el ejemplo anterior:

(list (eval 'X) 'C)

da de nuevo el resultado (A B C), puesto que (eval 'X) es equivalente a la expresión X en este contexto.

La función "mapcar" tiene la siguiente forma general:

(mapcar 'nombre 'args)

"Nombre" es el nombre de alguna función y "args" es una lista de argumentos para los cuales y en orden debe aplicarse repetidamente la función especificada.

Por ejemplo:

(mapcar 'subl '(1 2 3))

da la expresión resultante:

((subl 1) (subl 2) (subl 3))

Esto es, la función es copiada para los elementos de la lista, dando como resultado una lista de aplicaciones de la misma función a diferentes argumentos.

La función "apply" se utiliza para que se aplique repetidamente una función a una lista de argumentos, en vez de sólo una vez a los distintos argumentos requeridos por la función. Su forma general es:

(apply función (argumentos))

Por ejemplo, supongamos que queremos calcular la suma de los elementos de la lista (1 2 3). Podemos hacer esto escribiendo

(apply ' + (I 2 3))

lo cual es equivalente a la evaluación anidada (+ (+ 1 2) 3). Observe que

(+ (1 2 3))

no funcionaría, puesto que "+" es estrictamente una función de argumentos numéricos.

UN EJEMPLO DE APLICACIÓN DE LISP

"El problema de los misioneros y los caníbales"

La principal estructura de datos para este problema es una cola con todos los caminos desde el estado inicial al estado final. Un "estado" en este contexto es un registro del número de misioneros y caníbales que están en coda orilla y un conmutador que dice si la barca está en la orilla izquierda o en la derecha. Cada estado está compuesto de una lista de dos tripletas como sigue:

```
Orilla izquierda                Orilla derecha
( (M C B)                         (M C B) )
```

M y C son el número de misioneros y el número de caníbales sobre coda orilla y B es 1 ó 0, dependiendo de si la barca está o no en una orilla particular. Por tanto, el estado original del juego es el siguiente:

```
((331) (000))
```

con todos los misioneros, caníbales y la barca en la orilla izquierda. El estado final deseado se representa como:

```
((000) (331))
```

Conforme progresa el juego, crece la cola cada vez que se encuentra una transición de estados posibles, y en cada etapa se realiza una comprobación para ver que no ha tenido lugar canibalismo como resultado de la transición realizada. El nombre de esta cola en el programa es q.

Un movimiento, denotado por la variable "movimiento", se da también mediante una tripleta, la cual contiene el número de misioneros en la barca, el número de caníbales en la barca y la constante 1 denotando a la propia barca. Por tanto, por ejemplo, (111) denota a un movimiento en el que la barca contiene un misionero y un caníbal.

La variable "historia" contiene la historia de todos los estados de la orilla izquierda, Los cuales han sido ya tratados anteriormente en el juego. La variable "posible" es una lista de constantes que contiene todas las posibles alternativas para "un movimiento"; esto es, todas las posibles combinaciones de misioneros y caníbales que pueden cruzar el río de una vez.

El programa consta de una función principal "myc" y varias funciones auxiliares "comido", "expandir", "movcorrecto", "mover" y "presentar". Cada una de estas funciones se documenta brevemente en el texto del programa.

PROGRAMA

```
 (defun myc ()
(prog (q historia) ; inicializa la cola y los posibles movimientos
(setq posibles '((O 2 1)(0 1 1)(1 1 1)(1 0 1)(2 0 1))))
(setq q (list (list (list '(3 3 1) '(O O O)))))

repeat ; este bucle se repite hasta que está vacía la orilla izquierda
(cond ((equal (caaar q) '(O O O))
(return (display (reverse (car q)))))
; desecha un camino si da lugar a canibalismo
; o representa un bucle
((or (comido (caar q)) (member (casar q) historia))
(setq q (cdr q))
(go repeat))
)

; ahora añade este estado a la historia y pasa
; al siguiente estado
(setq historia (cons (caar q) historia))
(setq q (append (expandir (car q) posibles) (cdr q)))
(go repeat)
]

(defun comido (estado)
; esta función comprueba si existe canibalismo examinando
; la orilla izquierda (car estado). Si allí M es 1 0 2
; y M <>C, entonces hay canibalismo en una u otra orilla.
; Si no, no hay en ninguna.

(and (or (equal (caar estado) 1) (equal (caar estado) 2))
(not (equal (caar estado) (cadar estado)))
]

(defun expandir (caminos posibles)
; esta función desarrolla todos los posibles movimientos
; a partir del estado actual.

(cond ((null posibles) nil)
((movcorrecto (car mover) (car posibles))
(cons (cons (camino (mover (car camino) (car posibles)) camino)
(expandir camino (cdr posibles))))
(t (expandir camino (cdr posibles)))

(defun movcorrecto (estado unmovimiento)
; aquí se resta el número de misioneros y caníbales
; que hay en el bote del número que queda
; en la orilla actual, para asegurarse que no se cogen
; más de los que hay.

(cond ((zerop (caddar estado)) ; ve si bate en la derecha
(restatodo (cadr estado) unmovimiento))
(t (restatodo (car estado) unmovimiento))
```

```
(defun restatodo (triple unmovimiento)
; esta función resta loa tres números de un movimiento
; del bate del contenido de una orilla y devuelve
; nil si cualquiera de Las diferencias es <0

(not (minusp (apply 'min (mapcar ' - triple unmovimiento)
]

 (defun mover (estado unmovimiento)
; esta función realiza un movimiento restando
; los números de un movimiento del bote de una orilla
; y sumándolos a la otra.

(cond ((zerop (caddar estado))
; comprueba si bote en la derecha

(list (mapcar '+ (car estado) unmovimiento)
      (mapcar '- (cadr estado)
      unmovimiento)))
(t (list (mapcar '- (car estado) unmovimiento)
         (mapcar '+ (cadr estado)
         unmovimiento)))
]

(defun display (path)
; esta función presenta la solución resultante
(con ((null camino)
'end) (t (print (car
camino)) (terpri)
(display (cdr camino))
]
```