LENGUAJES DE PROGRAMACIÓN

(Sesión 10)

5. PROGRAMACIÓN FUNCIONAL

5.3. Subprogramas y gestión de almacenamiento

5.4. Evaluación del lenguaje

Objetivo: Manejar la estructura de un programa bajo LISP y evaluarlo

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" denotacualquier 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.

(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.)

(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.

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 si 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
(OR p1 p2 .. pn)	Devuelve t si uno o más de los p1, p2, .., pn es t
(NOT p)	Devuelve t si p es nil y nil en los demás casos
(FIXP x) y (FLOATP x)	Devuelve t si x es entero - punto flotante,
(EQUAL e1 e2)	Devuelve t si el valor de e1 es el mismo que el de
(NUMBERP e)	Devuelve t si e es un átomo numérico y nil en los
(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))
]
```

Bibliografía Recomendada:

Richard Bird; *Introducción a la Programación Funcional con Haskell;* Prentice Hall, 2000. (Segunda edición);

Simon Thompson; *Haskell: The Craft of Functional Programming;* Addison-Wesley, 1999. (Segunda edición);

Paul Hudak; *The Haskell School of Expression: Learning Functional Programming through Multim;* Cambridge University Press, 2000;

Blas Ruiz, Francisco Gutiérrez, Pablo Guerrero, José Gallardo; *Razonando con Haskell. Una introducción a la programación funcional ;* Universidad de Málaga, Manuales, 2000. (Segunda edición);

Bibliografía complementaria

- SEIBEL, PETER.(2005); *Practical Common Lisp*. Apress, 2005.
- GRAHAM, Paul. *Ansi Common Lisp*. New Jersey: Prentice Hall, 1996. ISBN 0-13-370875-6
- GRAHAM, PAUL.(1993); *On Lisp*. Prentice Hall, 1993. (Descreve técnicas avançadas para uso de macros)
- STEELE, Guy L.. *Common Lisp - The Language*. Lexington: Digital Press, 1990. ISBN 1-55558-041-6
- TOURETZKY, DAVID S.(1990); *Common Lisp - A Gentle Introduction to Symbolic Computation*. Benjamin Cummings, Redwood City, 1990. ISBN 0-8053-0492-4

Sitios consultados

http://www.fing.edu.uy/inco/cursos/progfunc/clases/uno.html

http://www.cs.us.es/~jalonso/publicaciones/2013-Temas_de_PF_con_Haskell.pdf

http://www.ida.liu.se/~ulfni/lpp/bok/bok.pdf

http://www.ia.uned.es/ia/regladas/prog-ia/libros/CL01JGB.pdf
http://www.dccia.ua.es/dccia/inf/asignaturas/LPP/2010-2011/teoria/tema1.html
http://www.ecured.cu/index.php/Emacs_Lisp
http://www.academia.edu/2400724/Introduccion_al_Lenguaje_Common_Lisp
http://labsys.frc.utn.edu.ar/ppr-2011/Unidad%20IV%20-%20Paradigma%20funcional/Unidad%20V%20-%20Paradigma%20Funcional.pdf

http://cursos.aiu.edu/Lenguages%20de%20Programacion/PDF/Tema%201.pdf