Paradigmas y Lenguajes

Funcional Parte 4

Msc. Ricardo Monzón

FUNCIONES DE ASIGNACION. (1)

La función **SETF**, es utilizada como las SETQ para realizar asignación de valores a variables. Su sintaxis es

(SETF {LUGAR FORM}+)

SETF utiliza la forma lugar para, a diferencia de SETQ, determinar una localización o dirección en la que hay que realizar la asignación. El lugar queda restringido a variables o alguna forma de función aceptable que normalmente evalúa un lugar al que asignar un valor. Tal y como la hacía SETQ, realiza las evaluaciones y asignaciones secuencialmente, devolviendo el valor de la última evaluación realizada. Ejemplos,

> (SETF NOMBRE "MARIA") → **"MARIA"**

> (SETF X ´(A B C)) → **(A B C)**

> (SETF (CAR X) (CAR ´(1 2 3))) → **1**

; acá se ve la utilidad de SETF, en el cual se reemplaza el contenido de una posición por el resultado de una función.

> X → **(1 B C)**

FUNCIONES DE ASIGNACION. (2)

La Funcion **Let** permite crear variables locales interiores a una función. Su forma general consta de dos partes ***asignación de variables*** y ***cuerpo*** , y lo podemos expresar:

(LET ({VARIABLE}* / {VARIABLE FORMA}*) {DECLARATIONS}* CUERPO)

La *asignación de variables* es una lista de listas donde cada lista consta de un nombre de variable y una forma a evaluar. Cuando un LET es llamado las formas se evalúan asignando paralelamente valores a todas las variables. Si quisiéramos que las variables se asignen secuencialemente, se usa **LET*** en lugar de **LET**.

Por su parte el *cuerpo* del LET es como el cuerpo de un DEFUN. Cualquier valor previo que tuvieran variables con el mismo nombre se guardarán y se restaurarán al finalizar el LET.

Las operaciones incluidas en el LET son evaluadas en el entorno del LET.

Las variables creadas en el LET son locales (al LET) y por tanto no pueden ser accedidas desde el exterior (tal y como sucede con DEFUN).

FUNCIONES DE ASIGNACION. LET vs SET. (3)

Los parámetros del LET se ligan sólo dentro del cuerpo del LET. Las asignaciones de SET dentro de un parámetro LET no crean ni modifican variables globales. Las asignaciones realizadas por el SET en variables libres crearán o cambiarán valores globales. Ejemplos:

```
> (SETF X 5)
> (SETF E 0)
> (SETF TRIPLE 0)
> (LET ((D 2) (E 3) (CUADRUPLE))
                    (SETF DOBLE (* D X))
                    (SETF TRIPLE (* E X))
                    (SETF CUADRUPLE (* 4 X)) )
 20
> D
 ERROR - UNBOUND VARIABLE ; variable no ligada
> E
0
```

FUNCIONES DE ASIGNACION. LET vs SET. (4)

Observar que las variables D y E, están definidas en el ámbito del LET, pero además E, es una variable definida por SETF afuera del LET, por eso se devuelve su resultado.

> DOBLE
10
> TRIPLE
15
> CUADRUPLE
ERROR - UNBOUND VARIABLE

Observar que la variable CUADRUPLE esta definida en el ámbito del LET, por lo tanto es una variable local.
Los valores de las variables libres pueden modificarse en la ejecución del LET. Tal es el caso de la variable DOBLE.

FUNCIONES ESPECIALES SIN NOMBRE O FUNCIONES LAMBDA.

En algunas ocasiones, es deseable crear función es sin nombre. Estas funciones se conocen con el nombre de lambda-expresiones y tienen una estructura semejante a las de las funciones.

((lambda (p1 ... pn) cuerpo) ‘(a1 … an))

Además, el comportamiento es semejante al de la función, con la diferencia que los a1…an ya son los parámetros pasados a la función. Ejemplos,

> ((lambda (x y) (cons (car x) (cdr y))) '(a b) '(c d))

; ‘(a b) ‘(c d) son los parámetros passados a la función.

(a d)

> ((lambda (a b) (* a b)) 4 8) ; 4 y 8 son los parámetros passados a la función

32

FUNCIONES DE ASIGNACION. (5)

Así como para evitar que en el LET se asignen las variables en paralelo se usa el LET*, en el caso del DO que también se asignan las variables en paralelo, se puede utilizar DO*.

DO* A diferencia del anterior realiza la asignación de variables secuencialmente.

```
> (defun exponente (m n)
        (do*    ((resultado m (* m resultado))
                 (exponente n (1- exponente))
                 (contador (1- exponente) (1- exponente)) ) ;fin-pa
                 ((zerop contador) resultado) ) ;fin-do ) ;fin-defun
```

EXPONENTE ; (ejemplo de la función exponente con DO*)

```
(defun exponencial-con-times (m n)
        (let ((resultado 1))
        (dotimes (cuenta n resultado)
        (setf resultado (* m resultado) ) ) ) )
```

exponencial-con-times

> (exponencial-con-times 4 3) ➔ **64**

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO. (1)

Una **función** es un objeto Lisp que puede invocarse como un procedimiento. Una función puede tomar una serie de argumentos y devolver uno o más valores, pero al menos uno.

Lisp tiene un gran número de funciones predefinidas, a menudo llamadas *primitivas*, aunque también permite la definición de funciones propias.

Lisp es un lenguaje funcional en el que todos los programas se escriben como colecciones de funciones y procedimientos. La definición de la función se almacena en memoria como un atributo/objeto del símbolo.

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO. (2)

Por su parte una función se referenciará por el nombre imprimible de un símbolo cuando se realice una llamada a dicha función.

Por ejemplo la llamada a la función de suma **(+ 2 3)**.

La llamada a la función es una lista cuyo primer elemento es el nombre de la función y el resto formarán los parámetros de dicha función.

Lisp comprobará que efectivamente el primer elemento de la lista se corresponde con alguna función predefinida o del usuario. Los argumentos de la llamada a la función serán igualmente evaluados antes de pasárselos como información a la función.

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO. (3)

Si el argumento de una función es a su vez otra llamada a función, entonces se evalúa primero la función del argumento y el resultado de la misma se pasa a la función principal.

Por ejemplo:

ENTRADA:	**(+ 3 (* 2 4))**
EVALUACION DE ARGUMENTOS:	**(+ 3 8)**
SALIDA:	**11**

DEFINICION DE FUNCIONES. (1)

DEFUN (DEfine FUNction), permite al usuario definir sus propias funciones. En general una función consta de tres partes:

nombre de la función, argumentos y ***cuerpo*** (lo que se suele llamar lambda expresión)**.**

(DEFUN nombre-función lambda-lista {declaration: string-doc} cuerpo)

Donde nombre-función debe ser el nombre de un símbolo atómico, como para una variable, la lambda-lista es el conjunto de argumentos que se pueden pasar (hay varias opciones en las lambda-listas), de las declaraciones y documentación hablaremos un poco más adelante, y cuerpo es cualquier forma a evaluarse en secuencia cuando se invoca al nombre de la función.

DEFINICION DE FUNCIONES. (2)

Ejemplo,

```
> (defun PRIMERO (L)
                (CAR L) )                ==> PRIMERO

> (PRIMERO ´(A B C))                     ==> A

> (defun RESTO (L)
                (CDR L) )                ==> RESTO
> (RESTO ´(A B C))                       ==> (B C)
```

El parámetro L de la definición se corresponde con una variable local, es decir, sólo es accesible desde el interior de la función. Si hay una variable externa con igual nombre, la L que hace referencia dentro del cuerpo se refiere a la del argumento.

DEFINICION DE FUNCIONES. (3)

Las **declaraciones** se utilizan para proporcionar información extra al sistema. A menudo son mensajes utilizados por el compilador. Las declaraciones también pueden aparecer al principio del cuerpo de ciertas formas como en las funciones. Los **string de documentación** se utilizan para describir comandos y pueden imprimirse al invocar la función de documentación de la función. Ejemplo:

```
> (defun CUADRADO (x)
        "Esta función devuelve el cuadrado de un número"
        (* x x) ; fin de CUADRADO
              )                          ==> CUADRADO
```

Esta función utiliza un string de documentación y un comentario justo antes del último paréntesis de la función.

```
> (CUADRADO 3)
 9
```

DEFINICION DE FUNCIONES. (4)

¿Qué sucede al definir la función CUADRADO?

- Lisp crea un símbolo CUADRADO
- Incorpora en la *definición de función* el valor correspondiente a dicha función.

Y ¿cómo se evalúa y activa una función?

La evaluación de una función, cuando esta se define, consiste en el nombre de la propia función. Por ejemplo, al definir en LISP la función CUADRADO, devuelve como valor de la misma **CUADRADO**.

```
>> (defun CUADRADO (x)
        (* x x) )                    ==> CUADRADO
```

DEFINICION DE FUNCIONES. (5)

Por otro lado, cuando se llama a una función con ciertos argumentos, el valor que devuelve dicha función se corresponde con la última forma ejecutada en el cuerpo de DEFUN. En la función CUADRADO, corresponde con la forma (* x x)

Ejemplos

```
> (cuadrado 3)
 9
> (cuadrado 2)
 4
> (cuadrado (cuadrado 3))
 81
> (cuadrado (+ -4 5 (/ 3 1)))
 16
```

DEFINICION DE FUNCIONES. (6)

La definición de una función puede incluir tantos argumentos como se estimen necesarios. Por ejemplo:

```
> (defun SUMA-CUADRADO (x y)
                    (+ (cuadrado x) (cuadrado y)) )
 SUMA-CUADRADO
> (suma-cuadrado 2 3)
13
```

Se pueden definir también funciones sin parámetros:

```
> (defun SALUDO ()      (print "hola a todos") )
 SALUDO
> (saludo)                    ==> "hola a todos"
          "hola a todos"
```

La respuesta de LISP al llamar a la función SALUDOS, es mediante dos strings que corresponderán: el primer string a la acción del comando PRINT, el segundo es el valor devuelto por la función saludo.

LIGADURA DE PARAMETROS EN LA FUNCION. (1)

Las variables creadas en la lista de argumentos de una DEFUN (parámetros formales) son normalmente locales para la función. La lista de parámetros organiza un espacio en memoria para el ámbito de definición de la función, que será utilizado por los valores que se pasen a dicha función en la llamada.

```
> (defun mi-funcion (varx)
        varx )                    ==> MI-FUNCION
> (mi-funcion "vale")             ==> "vale"
> varx                            ==> ERROR, UNBOUND VARIABLE
```

*Esta variable varx sólo tiene sentido dentro de la función mi-funcion, no tiene acceso desde fuera y por tanto si pretendemos tener información de dicha variable nos dará un **ERROR.***

LIGADURA DE PARAMETROS EN LA FUNCION. (2)

- La lista de parámetros apunta a los mismos valores que los correspondientes argumentos en la expresión de llamada.
- Cuando se termina la ejecución de la función se desligan los valores de la lista de parámetros.
- Las variables de la lista de parámetros no afectan a las variables que se encuentran fuera del procedimiento.
- Las variables ligadas, de la lista de argumentos de una función, no pueden ser accedidas por otras funciones llamadas en ejecución desde la misma (a no ser que estos valores se pasen como argumentos de la nueva función llamada).

DECLARACIONES. (1)

Las variables, a menos que sean declaradas de otra forma, son de ámbito léxico. Lo que se ha visto hasta ahora con respecto a la ligadura para los argumentos de una función es que sólo pueden verse dentro del texto de dicha función (tienen ámbito léxico).

Las variables pueden declararse como especiales y en estos casos se dice que son de ámbito dinámico. Es decir son variables libres que pueden ser accedidas desde cualquier parte del programa a partir del punto en el que han sido definidas.
Para declarar variables especiales para todas las funciones, sin necesidad de declararlas separadamente en cada una de ellas, usamos:

- DEFVAR y DEFPARAMETER
- DEFCONSTANT

DECLARACIONES. (2)

Esta es la idea de **variable global** utilizada en otros lenguajes de programación. La forma sintácticamente correcta de las mismas es la siguiente:

- **(DEFVAR nom-var [valor])**, declara una variable global de nombre nom-var accesible desde cualquier parte del programa (desde cualquier función).

- **(DEFPARAMETER nom-var [valor])**, tiene las mismas características que DEFVAR, excepto que será usado como parámetro y no como variable.

- **(DEFCONSTANT nom-var valor)**, el valor permanecerá constante en nom-var, si se intenta cambiar su valor dará error.

DECLARACIONES. (3)

Ejemplo.

```
> (defconstant c 36)
 c
> c
 36
> (setf c 40)
Error: C has constant value 36 and cannot be changed to 40
```

RECURSIVIDAD EN LISP. (1)

Se dice que un proceso es recursivo cuando esta basado en su propia definición. En programación, la recursividad, no es una estructura de datos, sino una técnica de programación que permite que un bloque de instrucciones se ejecute una n cantidad de veces. En muchas ocasiones reemplaza a las estructuras repetitivas.
En LISP, es común que las funciones se llamen a si mismas en el cuerpo de la función, esto es lo que se llama una función recursiva.

CARACTERISTICAS DE LA PROGRAMACION RECURSIVA

- Implementación intuitiva y elegante.
- La traducción de la solución recursiva de un problema (caso base y caso recursivo) a código Lisp es prácticamente inmediata.
- Útil cuando hay varios niveles de anidamiento. La solución para un nivel es válida para el resto.
- La interpretación y comprensión del código puede ser compleja.

RECURSIVIDAD EN LISP. (2)

Ejemplos.

1. **Función Factorial (n!)**

```
(defun factorial (n)
        (if (= n 0) 1
                (* n (factorial (- n 1)))
        )
)  ; existe una llamada a función factorial en el cuerpo de la función factorial.
```

2. **Función Potencia (m^n)**

```
(defun potencia (x m)
        (if (= m 0) 1
                (* x (potencia x (- m 1)))
        )
) ; existe una llamada a la función potencia desde el cuerpo de la funcipon potencia.
```

RECURSIVIDAD EN LISP. (3)

3. Obtener el valor correspondiente a la posición n de una serie de Fibonacci.

Método Iterativo

```
(defun fibonacci (n)
        (do     ((i 1 (1+ i))
                 (fib1 0 fib)
                 (fib   1 (+ fib fib1)))
                ((= i n) fib)))
```

Método Recursivo

```
(defun fibo-recur (x)
    (if (< x 2)
        x
        (+ (fibo-recur (1- x)) (fibo-recur (- x 2)))))
```

SEGUIMIENTO DE FUNCIONES (Trace) (1)

Cuando se necesita saber exactamente *cómo* una función está trabajando en un punto particular en el código, hacer Break y usar el depurador de Lisp son herramientas indispensables. Pero es trabajoso y lento (por lo menos en relación con la ejecución normal del programa).
A veces, es suficiente saber que una determinada función se ha llamado y devolvió un valor. TRACE nos brinda esta capacidad imprimiendo el nombre de la función y sus argumentos a la entrada, y el nombre de la función y de sus valores a la salida. Todo esto ocurre sin cambiar el código fuente de la función.

(trace [<sym>...])

Ejemplo
> (trace factorial fibo potencia)
(FACTORIAL FIBO POTENCIA)

La ejecución del trace permanecerá activa hasta que se ejecute la macro UNTRACE. Si se ejecuta sin argumentos, detendrá el traceo de todas las funciones.

(untrace [<sym>...])

SEGUIMIENTO DE FUNCIONES (Trace) (2)

Si se ejecuta la macro trace sin argumentos, mostrara una lista con todas las funciones que estan siendo trazeadas en su ejecución.

(trace)
(FACTORIAL FIBO POTENCIA)

> (fibo 4)

```
Entering: FIBO, Argument list: (4)
  Entering: FIBO, Argument list: (3)
    Entering: FIBO, Argument list: (2)
      Exiting: FIBO, Value: 1
        Entering: FIBO, Argument list: (1)
      Exiting: FIBO, Value: 1
     Exiting: FIBO, Value: 2
    Entering: FIBO, Argument list: (2)
  Exiting: FIBO, Value: 1
Exiting: FIBO, Value: 3
3
```

STREAMS (1)

Es un tipo de datos que mantiene la información sobre el dispositivo (fichero) al que está asociado. Hará que las E/S sean independientes del dispositivo.
Un stream puede estar conectado a un fichero o a un terminal interactivo.

OPEN Esta función devuelve un stream.
(OPEN nomfich &key :direction)

donde nomfich será un string/ pathname/ stream;

:direction permitirá señalar si el dispositivo va a ser de lectura-entrada, :input, salida-escritura, :output, o ambos, :io.

STREAMS (2)

Ejemplo.

> (setq f (open 'mine :direction :output)) ; crea un archivo llamado MINE

```
> (print “Primer HolaMundo" f)          ; retorna “Primer Hola Mundo"
> (print "Segundo Hola Mundo" f)        ; retorna "Segundo Hola Mundo"
> (close f)                             ; archiva "hola" <newline>
> (setq f (open 'mine :direction :input))   ; abre el archive como input
> (read f)                              ; lee y devuelve " Primer Hola
                                                Mundo "
> (read f)                              ; lee y devuelve " Segundo Hola
                                                Mundo "
> (close f)                             ; cierra el archivo
```

STREAMS (3)

Algunos predicados con streams serían:

(**STREAMP** Objeto), dará T si objeto es un stream
(**OUTPUT-STREAM-P** stream), dará T si stream es de salida.
(**INPUT-STREAM-P** stream), será T si stream es de entrada.

CLOSE, Cierra un stream, desaparece la ligadura entre el fichero y el stream (en caso de estar ligado a él).

(CLOSE stream)

Algunos streams estandar
standar-input
standar-output
error-output
terminal-io

Funciones de Entrada de STREAMS

READ-CHAR lee un carácter y lo devuelve como un objeto de tipo carácter. Su forma sintácticamente correcta es,
(READ-CHAR &optional stream)

READ-LINE lee caracteres hasta encontrar un carácter #\newline. Y devuelve la línea como un string de caracteres sin #\newline -> (READ-LINE &optional input-stream)

PEEK-CHAR lee un carácter del stream de entrada sin eliminarlo del stream. La próxima vez que se acceda para leer se leerá el mismo carácter.
(PEEK-CHAR &optional peek-type input-stream) Si peek-type es **t** se saltarán los caracteres en blanco. Si su valor es **nil** no.

Funciones de Manipulación de STREAMS (1)

(**RENAME-FILE** fichero nuevo-nombre), cambia de nombre un fichero. Si fichero es un nombre de stream, el stream y el fichero a él asociado resultan afectados.

(**DELETE-FILE** fichero), borra un fichero. Si fichero es un nombre de stream, el stream y el fichero a él asociado resultan afectados.

(**FILE-LENGTH** stream-de-fichero), devuelve la longitud de un fichero. Si esta no puede ser determinada devolverá nil.

(**LOAD** fichero &key :verbose :print :if-does-not-exist), carga un fichero al interprete Lisp. A partir de ese momento el entorno Lisp contará con los objetos Lisp definidos en ese fichero.

Funciones de Manipulación de STREAMS (2)

Las claves incluidas en la forma LOAD corresponden con:

:verbose Si es t permitirá al cargar el fichero, sacar un comentario sobre el *standar-output* indicando las características del fichero que está siendo cargado.

:print Si es t se saca un valor de cada expresión cargada sobre *standar-output*.

:if-does-not-exist Si es nil devuelve nil; si es t el error correspondiente en caso de que lo haya.

DOCUMENTACION (1)

Se puede hacer comentarios en cualquier lugar del programa siempre que se utilice el ";", de tal forma que después de la coma comienza el comentario. Por convenio se utilizará:

;;;; 4 ";" y un blanco, al comienzo del fichero para indicar su contenido.

;;; 3 ";" y un blanco, para realizar comentarios sobre una función y aparecerá en la línea posterior a la de definición de función (cabecera).

;; 2 ";" y un blanco, se utiliza para comentar una línea de código. En la línea inmediatamente seguida a esta.

; sólo un ; y sin blanco, se utiliza para explicar una línea de código en la misma línea.

DOCUMENTACION (2)

Ejemplo de un fichero comentado:

```
;;;;
************************************************************
;;;; Fichero: Ejemplo-comentario.lisp
;;;; Fecha-de-creación: 20/02/2014
;;;; Modulo: Apuntes de lisp.
;;;; Comentarios: Este fichero ha sido creado con fines didácticos.
;;;; Autores: Los profesores de Paradigmas
;;;; UNNE
;;;;
************************************************************
```

DOCUMENTACION (3)

Ejemplo de un fichero comentado:

```
(defun pareja (x)
;;; Esta función devuelve la pareja del personaje que se pasa como valor del parámetro x.
(case x
;; se realiza la selección.
((Peter-pan) "Campanilla") ;Campanilla es una chica que vuela.
((Mortadelo) "Filemón") ;Ambos son agentes de la T.I.A.
( t "no tiene pareja")
) ;Fin de selección
) ;Fin de la función.
;;; Esperamos que haya encontrado su pareja con esta función. Si no modifíquela siempre que lo crea necesario.
;;;; Fin del fichero.
```

Paradigmas y Lenguajes

Fin Paradigma Funcional

Msc. Ricardo Monzón