Paradigmas y Lenguajes

Funcional 3

Lic. Ricardo Monzón

ESTRUCTURAS CONDICIONALES (1)

Los condicionales son formas que permiten tomar decisiones.

Verifica y ramifica: verifica devolviendo NIL o NO-NIL (como cualquier función), y ramifica es el comando a ejecutarse posteriormente basándose en el resultado de la verificación.

Existen diferentes condicionales en Lisp, igual que en otros lenguajes: IF, CASE, COND, WHEN, UNLESS, TYPECASE.

- **IF** TEST THEN [ELSE]. La forma condicional IF puede expresar opcionalmente su parte THEN.

Dos formas sintácticamente correctas serían:

(IF FORMA-COND FORMA-THEN)

(IF FORMA-COND FORMA-THEN FORMA-ELSE)

ESTRUCTURAS CONDICIONALES (2)

> (setq obj 5)
5
> (if (numberp obj) "Es un número" "Esto no es un número ")
“Es un número"
> (setq obj ‘a)
A
> (if (numberp obj) "Es un número" "Esto no es un número ")
"Esto no es un número"

• **COND** {(TEST {FORMA}+) }+. Permite expresar varias condiciones, ejecutándose la primera que se cumple. La forma COND acepta cualquier número de listas cuyos elementos son formas.
Estas listas están formadas normalmente por un test seguido de cero o más formas consecuentes.
Estas formas se ejecutaran si y solo si el test devuelve un valor NO-NIL.

ESTRUCTURAS CONDICIONALES (3)

La cabeza de las listas se evalúa secuencialmente hasta encontrar una que devuelva un valor NO-NIL, entonces las formas siguientes a la cabeza de esa lista se evalúan y se sale de la forma COND. Si hubiera listas más adelante cuyas condiciones (cabeza de lista) fueran también ciertas éstas no se evalúan y por tanto tampoco se ejecutan las formas incluidas en el resto. COND devolverá el resultado de la última forma evaluada de la lista con un test NO-NIL. Si todos los test devuelven NIL, entonces COND devolverá NIL. El ejemplo general para la sintaxis sería:

(COND (TEST1 FORMA11 FORMA12 … FORMA1N)

(TEST2) ; no tiene ningún consecuente y se devolverá el valor NO-NIL del test

(TEST3 FORMA31 FORMA32 … FORMA 3M)

…

(TESTR …))

ESTRUCTURAS CONDICIONALES (4)

```
> (setq a 4)
4
> (cond          ((numberp a) "Esto es un número")
                 (t "Esto no es un número") )
"Esto es un número.

> (setq x 8)
8
> (cond ((< x 0) (print "NEGATIVO"))
                 ((= x 0) (print "CERO"))
                 ((<= X 10) (print "VALOR EN RANGO"))
                 (T (print "VALOR MUY GRANDE") x)
          ) ;fin-cond
"VALOR EN RANGO"
```

ESTRUCTURAS CONDICIONALES (5)

WHEN TEST FORMA1 ... FORMAN. Es equivalente a la forma IF sin opción de incluir la forma else. Si el TEST es verdadero, se evalúan todas las formas restantes.

Ejemplo,

```
> (setq a 4)
> (when (equal a 4) (print (* a a)) (print "cierto"))
16
"cierto"
"cierto"
> (when (equal a 3) (print (* a a)) (print "cierto"))
nil
```

Al ser NIL el test, no se evalúa nada mas y devuelve NIL.

ESTRUCTURAS CONDICIONALES (6)

UNLESS TEST FORMA1 ... FORMAN. Esta forma condicional difiere de las vistas hasta ahora en que se evaluarán las formas siempre que el resultado de la evaluación del test sea NIL.
Hasta ahora la evaluación de las formas asociados a un test estaba condicionada a que éstas dieran como resultado un valor NO-NIL.

Ejemplos,

```
> (unless (equal a 4) (print (* a a)) (print "cierto"))
NIL
> (unless (equal a 3) (print (* a a)) (print "cierto"))
16
"cierto"
"cierto"
```

ESTRUCTURAS CONDICIONALES (7)

CASE KEYFORM {(LIST { FORM}+)}+
Permite realizar ciertas acciones (FORM) cuando una determinada forma KEYFORM toma ciertos valores (LIST), expresados a través de una lista. Por ejemplo,

```
> (setq mes ‘jun)
> jun
> (case mes
        ((ene mar may jul ag oct dic) 31)
        ((abr jun sep nov) 30)
        (feb (if (zerop (mod año 4)) 29 28) ) )
30
```

La clave mes, se compara con el contenido de los elementos de cada lista y si alguno coincide, muestra el resultado.

ESTRUCTURAS CONDICIONALES (8)

TYPECASE KEYFORM {(TYPE {FORMA}+)}+
Permite como en el caso anterior realizar una serie de acciones siempre que un determinado objeto sea de la clase indicada.

Ejemplo,

```
> (SETQ X 2) ==> 2
> (TYPECASE X
        (string "es un string")
        (integer "es un entero")
        (symbol (print '(es un símbolo)))
        (otherwise (print "operador")(print "desconocido") ) )
"es un entero"
"es un entero "
```

ESTRUCTURAS ITERATIVAS (1)

LOOP {FORMA}*. Expresa una construcción iterativa que cicla continuamente, a menos que se le indique explícitamente que pare, evaluando las formas en secuencia. La forma de indicar la terminación del ciclo es mediante RETURN. En general la evaluación de la sentencia RETURN provocará la salida de cualquier sentencia iterativa. Ejemplo,

```
> (setq x 1)
1
> (loop
        (print x)
        (setq x (+ x 1))
        (if (= x 20) (return))
   ) ; end loop
```

Imprime del 1 al 19 y sale con un NIL.

ESTRUCTURAS ITERATIVAS (2)

DO ({PARAM}* / {(PARAM VALOR)}* / {(PARAM VALOR INCRE)}*) (TEST-EVALUACION {FORMAS}*) {FORMAS}*

La estructura DO tiene tres partes: ***lista de parámetros, test de final*** y ***cuerpo***. La lista de parámetros liga las variables con sus valores iniciales y en cada ciclo se asignan nuevos valores dependiendo de la función de incremento indicada. En la estructura DO, primero se activa la ligadura de los parámetros y después se evalúa el test de final, y si no se cumple se ejecuta el cuerpo.
Esta estructura podemos encontrarle semejanza con la de while ... do utilizada en otros lenguajes.

```
(DO ((var1 inic1 [paso1])...
(varn inicn [pason]))          ; Asignación de variables en paralelo
(test-final resultado)         ; Test-final se evalúa antes que cuerpo
        declaraciones
        cuerpo)
```

ESTRUCTURAS ITERATIVAS (3)

Ejemplos: Función exponente $\mathbf{m^n}$

```
> (setq m 3 n 4)
> (do   ((resultado 1) (exponente n))  ; asignación de parámetros
                                       ; asigna 1 a resultado y n a exponente
        ((zerop exponente) resultado)  ; test de final
                                       ; si exponente es 0, muestra resultado,
                                       ; sino sigue con la próxima sentencia.
        (setq resultado (* m resultado))  ; asigna el producto a resultado
        (setq exponente (- exponente 1)) )  ; resta 1 a exponente, vuelve al
81                                          ; test.

> (do   ((resultado 1 (* m resultado))
                (exponente n (1- exponente)))
                ((zerop exponente) resultado) )
81
```

Otra forma con el test al final.

ESTRUCTURAS ITERATIVAS (4)

Ejemplos:

```
> (setq lista '(1 2 3 4))
> (do    ((l lista (cdr l))
              (resultado nil (cons (car l) resultado)))
              ((null l) resultado) )
(4 3 2 1)
```

Iteración	l	resultado
1	(1 2 3 4)	nil
2	(2 3 4)	(1)
3	(3 4)	(2 1)
4	(4)	(3 2 1)
5	()	(4 3 2 1)

ESTRUCTURAS ITERATIVAS (5)

DOTIMES permite escribir procedimientos sencillos con iteración controlada por un contador. Ejecuta el cuerpo un número de veces determinado.

(DOTIMES (var forma-limite-superior [forma-resultado])
cuerpo)

Cuando comienza el ciclo se evalúa la forma límite superior, produciendo un valor n. Entonces desde el valor 0 incrementándose en uno hasta n-1, se asigna a la variable var.

Para cada valor de la variable se ejecuta el cuerpo y al final, la ligadura con la variable se elimina y se ejecuta la forma resultado dando valor al DOTIMES. Si no tiene forma-resultado, finaliza con NIL.

ESTRUCTURAS ITERATIVAS (6)

```
> (dotimes (cont 4) (print (* cont cont))
0
1
4
9
NIL

> (dotimes (x 5 '(fin)) (print (list x x))
(0 0)
(1 1)
(2 2)
(3 3)
(4 4)
fin
```

ESTRUCTURAS ITERATIVAS (7)

DOLIST. asigna a VAR el primer elemento de la lista L; evalúa S1,…, SN para cada valor de VAR; si L no tiene más elementos, devuelve resultado; en otro caso, le asigna a VAR el siguiente elemento de L e itera el proceso. Si no tiene el valor de resultado, finaliza con NIL.

```
(DOLIST (VAR L [resultado])
                    S1, …, SN)
> (dolist (x '(a b c))
        (print x) )
A
B
C
NIL
```

FUNCIONES ITERATIVAS PREDEFINIDAS (1)

(MAPCAR fn lista1 ... listan)

Va aplicando la función fn a los sucesivos car's de las listas.
Devuelve una lista con los resultados de las llamadas a la función.
Utiliza LIST para construir el resultado.

> (mapcar '+ '(7 8 9) '(1 2 3)) → **(8 10 12)**
> (mapcar 'oddp '(7 8 9)) → **(T NIL T)**
> (mapcar 'atom '(A (B) 3)) → **(T NIL T)**

FUNCIONES ITERATIVAS PREDEFINIDAS (2)

(MAPLIST fn lista1 ... listan)

Se aplica la función fn a los CDR´s sucesivos de las listas.
Devuelve una lista con los resultados de las llamadas a la función.
Utiliza LIST para construir el resultado.

> (maplist '+ ' (7 8 9) '(1 2 3)) → ((8 10 12) (10 12) (12))

> (maplist 'cons '(1 2 3) '(a b c)) → (((1 2 3) A B C) ((2 3) B C) ((3) C))

ENTRADA/SALIDA SIMPLE (1)

Lisp tiene primitivas de funciones de entrada/salida. Las más comúnmente utilizadas son: READ, PRINT y FORMAT, aunque también veremos otras.

Los dispositivos por defecto son, el de entrada el teclado y salida la pantalla, pero se puede redireccionar a otros dispositivos. Es decir, todas estas funciones toman un argumento opcional llamado input-stream o output-stream.

Si este valor no es suministrado o es nil, se tomará por defecto el valor contenido en la variable *standard-input* que será teclado o pantalla respectivamente. Si el valor es t se usará el valor de la variable *terminal-io*.

ENTRADA/SALIDA SIMPLE (2)

Función de entrada.

- **READ**, lee un objeto Lisp del teclado y devuelve dicho objeto. Esta función detiene la ejecución del programa mientras no se termine con un RETURN. Sintácticamente tendríamos

(READ &optional stream)

Ejemplos:

```
> (read)
 35
 35
> (setq a (read))
HOLA
HOLA
> a
Hola
```

ENTRADA/SALIDA SIMPLE (3)

Funciones de salida.

PRINT, toma un objeto Lisp como argumento y lo escribe en una nueva línea con un blanco por detrás (introduce <Return> y blanco después del objeto). Escribe los objetos por su tipo, tal y como serían aceptados por un READ.

(PRINT objeto &optional stream)

Ejemplos:

```
> (print 'a) ;          imprime A con #\Newline
> (print '(a b)) ;            imprime (A B) con #\Newline
> (print 99) ;          imprime 99 con #\Newline
> (print "hi") ;        imprime "hi" con #\Newline
```

ENTRADA/SALIDA SIMPLE (4)

Funciones de salida.

PRIN1, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás. Escribe los objetos por su tipo, tal y como serían aceptados por un READ, sin salto de línea. La próxima impresión será al lado.

(PRIN1 objeto &optional stream)

Ejemplos:

```
> (prin1 'a) ;          imprime A sin #\Newline
> (prin1 '(a b)) ;      imprime (A B) sin #\Newline
> (prin1 2.5) ;         imprime 2.5 sin #\Newline
> (prin1 "hi") ;        imprime "hi" sin #\Newline
```

ENTRADA/SALIDA SIMPLE (5)

Funciones de salida.

PRINC, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás, y en el caso de los string, imprime sin las comillas. No escribe los objetos por su tipo.

(PRINC objeto &optional stream)

Ejemplos:

```
> (princ 'a) ;          imprime A sin #\Newline
> (princ '(a b)) ;      imprime (A B) sin #\Newline
> (princ 99) ;          imprime 99 sin #\Newline
> (princ "hi") ;        imprime hi sin #\Newline
```

ENTRADA/SALIDA SIMPLE (6)

Funciones de salida.

PPRINT, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás, introduce solo un Return. No realiza la segunda evaluacion.

(PPRINT objeto &optional stream)

Ejemplos:

```
> (pprint 'a) ;          imprime A returns NIL
> (pprint "abcd") ;      imprime "abcd" returns NIL
```

ENTRADA/SALIDA SIMPLE (7)

Funciones de salida.

FORMAT, aparece como un mecanismo más generalizado de dar la salida. Se indican directivas que son como variables situadas entre el string y comienzan con ~. Muchas directivas miran el siguiente argumento de la lista del format y lo procesan de acuerdo a sus propias reglas. Su forma es

(FORMAT destino control-string &rest argumentos)

- destino = nil/ t/ stream. Si se indica el nil, se creará un string con las características indicadas y éste será lo que devuelva el format. Si se indica t, será la salida por defecto y sino será otro dispositivo indicado en stream.

- control-string = contiene el texto a escribir y "directivas"

- argumentos = lista con los parámetros para las "directivas"

ENTRADA/SALIDA SIMPLE (7)

Funciones de salida.

Se usan diferentes directrices para procesar diferentes tipos de datos e incluirlos en el string:

~A - Imprime un objeto cualquiera.,

~D - Imprime un número en base 10

~S - Imprime una expresión simbólica.

Además se puede utilizar el símbolo @ junto con la directiva ~A, para justificar a la derecha valores numéricos. No todas la directrices utilizan argumentos: ~

% - Inserta una nueva línea y

~| - Nueva página.

Ejemplos,

> (FORMAT T "STRING DE SALIDA")

STRING DE SALIDA

NIL

ENTRADA/SALIDA SIMPLE (9)

> (SETQ VAR 5)

5

> (FORMAT T "STRING QUE INCLUYE ~A" VAR)

STRING QUE INCLUYE 5

NIL

> (FORMAT NIL "STRING QUE INCLUYE ~A" VAR)

STRING QUE INCUYE 5

> (FORMAT T "ESCRIBE EL ARGUMENTO ~10@A JUSTIFICADO A LA DERECHA CON 10 ESPACIOS." 1000)

ESCRIBE EL ARGUMENTO 1000 JUSTIFICADO A LA DERECHA CON 10 ESPACIOS.

NIL

> (FORMAT T "~%LA PRIMERA RESPUESTA ES ~5@A ~%Y LA SEGUNDA RESPUESTA ES ~3@A " (* 5 3 2) (/ 3 5))

LA PRIMERA RESPUESTA ES 30

Y LA SEGUNDA RESPUESTA ES 0.6

NIL

ENTRADA/SALIDA SIMPLE (10)

TERPRI realiza un salto de línea

(TERPRI &optional stream)

Ejemplos de salida:

> **(princ "hola")** ==> **hola**

"hola"

> (print "hola") ==> "hola"

"hola"

> (prin1 "hola") ==> "hola"

"hola"

> (format t "hola") ==> hola

NIL

> (format nil "hola")

"hola"

> (format nil "hola soy ~a, ~a" "yo" "Bonn")

"hola soy yo, Bonn"