UNIDAD 4 – PROGRAMACIÓN FUNCIONAL

Historia de la Programación Funcional.

El primer lenguaje de programación funcional fue Lisp, que también fue el segundo lenguaje de alto nivel que se implementó, precedido por Fortran que estaba orientado a cálculos numéricos. Lisp fue creado por John McCarthy a finales de los 50's en el laboratorio de Inteligencia Artificial del MIT, para un proyecto llamado "Advice Taker" cuyo propósito era crear 'un sistema capaz de manejar oraciones tanto declarativas como imperativas y exhibir un "sentido común" al llevar a cabo las instrucciones'.

Las listas habían probado ser una estructura de datos adecuada para el procesamiento simbólico, así que McCarthy creó LISt Processor, basándose en 'un esquema para la representación de funciones recursivas parciales de una clase de expresiones simbólicas' que llamó Expresiones-S y las funciones Funciones-S. Las funciones eran evaluadas por 'la función apply, una Función-S que juega el papel de una máquina universal de Turing'.

Lisp es un lenguaje muy pequeño, se construye con cinco operaciones básicas (cons, car, cdr, eq, y atom) y las expresiones condicionales y lambda (cond, y lambda) lo que se conoce como "Lisp puro", pero también existen instrucciones que permiten la asignación, e iteración. Originalmente Lisp fue programado en la computadora IBM 704 que tenían en el MIT, pero después surgieron varios dialectos, lo único que en ese entonces se podría considerar un standard fue la publicación de Lisp 1.5, sin embargo Lisp es un lenguaje (matemáticamente) muy elegante, y es fácil entender los programas escritos en otros dialectos distintos al que uno domine.

Esta misma elegancia permite hacer demostraciones acerca de los programas con mucha facilidad. Ante estas características del lenguaje cabría preguntarse, si se creó un lenguaje tan bonito, ¿porque no se popularizó como Fortran, COBOL, o Basic? esta pregunta está formulada de una manera truculenta, porque en realidad se popularizó, pero solamente en los laboratorios de Inteligencia Artificial, para los demás mortales, era algo oscuro, ineficiente, un juguete de los científicos, y otros mitos (que alguna vez fueron ciertos).

De hecho, el camino abierto por McCarthy fue seguido por otros investigadores que crearon lenguajes funcionales más poderosos, y aunque suene paradójico el auge actual de la programación funcional se debe en gran parte a uno de los creadores de Fortran y Algol, John Backus, quien en 1978 al recibir el premio Turing (un premio otorgado por una asociación muy prestigiada la ACM) escribió su Lectura "¿Puede la Programación Liberarse del Estilo Von Neumann? Un Estilo Funcional y su Álgebra de Programas", en donde expone los problemas de la asignación en los lenguajes de programación imperativa, proponiendo un lenguaje funcional más avanzado que Lisp. Este evento tuvo un gran impacto y ahora mucha gente se ha interesado en estilos de programación mejor fundamentados, matemáticamente, que los lenguajes convencionales.

La programación funcional, no solamente se ha aplicado en Inteligencia Artificial, sino a una vasta gama de aplicaciones como la ingeniería de software, cálculos ingenieriles, diseño asistido por computadora, manejadores de bases de datos, entre otras. Otros lenguajes funcionales que son más conocidos son: FP, FFP, KRC, Miranda, ML, SML, Haskell.

LISP

El lenguaje de Programación Funcional que vamos a ver es el LISP, el siguiente apunte ha sido desarrollado por Marcelo Luis Errecalde del Departamento de Informática de la Universidad Nacional de San Luis y es el que usaremos de guía en esta cátedra.

1 Introducción

Lisp es uno de los lenguajes de programación más antiguos, el cual fue diseñado e implementado por John McCarthy y un grupo del MIT (Massachusetts Institute of Technology) en el

año 1960.

Una característica distintiva de Lisp es su poderoso conjunto de primitivas para el procesamiento de listas, lo cual está á en reflejado en el origen mismo de su nombre: List Processing.

Desde la creación de Lisp surgieron muchos dialectos; nosotros usaremos el estándar definido a mediados de los años 80 denominado Common Lisp. Este estándar fue completamente definido en 1992, está á hoy en día disponible para la mayoría de las plataformas de computación y es de hecho el estándar comercial.

Desde sus orígenes y hasta el presente fue ampliamente utilizado en aplicaciones de Inteligencia Artificial: robótica, procesamiento de lenguaje natural, demostración de teoremas, sistemas expertos, etc. En este tipo de aplicaciones es importante que el lenguaje provea facilidades para la manipulación de expresiones simbólicas formadas por palabras (símbolos) y agrupamiento de símbolos (listas) que representan conceptos del mundo real. Lisp, al igual que Prolog, proveen este tipo de facilidades y se han convertido en los lenguajes predominantes en Inteligencia Artificial.

Otras características que suelen distinguir a Lisp de otros lenguajes de programación tradicionales son:

- Equivalencia del formato usado para los datos y los programas; esto permite que las estructuras de datos puedan ser ejecutadas como programas y que los programas puedan ser modificados como datos.
- Los programas no suelen estar estructurados como una secuencia de sentencias, sino como expresiones anidadas en notación prefijo.
- En lugar de la iteración clásica, la estructura de control por excelencia es la recursión.

2 Expresiones simbólicas.

2.1 Tipos de expresiones

Las expresiones válidas en Lisp se denominan s-expressions o expresiones simbólicas, y de ahora en más las referenciaremos con el término general de expresiones. La figura 1 muestra los tipos de expresiones básicas de Lisp.

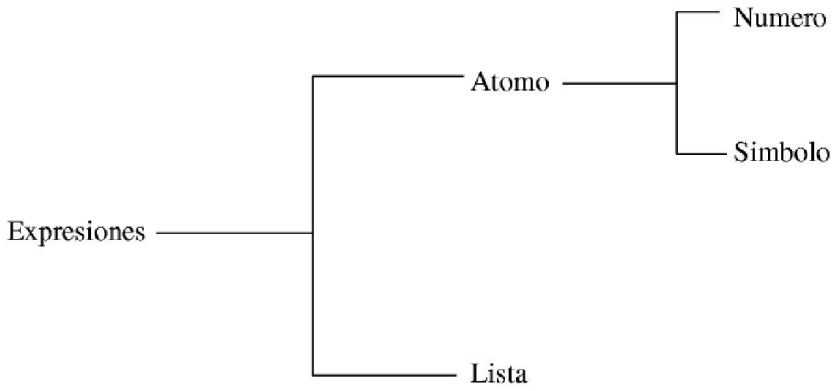

Figura 1: Tipos de expresiones básicos de Lisp

Un átomo es un objeto indivisible, como por ejemplo **PEPE**, **B27**, **18**, **+**, **CUENTA-CORRIENTE**, etc.

Los átomos como **4** y **3.14** se denominan átomos numéricos o simplemente números.

Los átomos como **PEPE** o **B27** se denominan átomos simbólicos o simplemente símbolos.

Una lista es una expresión formada por la secuencia de 0 o más expresiones entre paréntesis. Por ejemplo, las expresiones **(2 3 PEPE HOLA)** y **(+ 1 2 3 4)** serán 2 listas válidas en Lisp de 4 y 5 elementos respectivamente. La definición de lista, permite tener listas cuyos elementos sean otras listas, posibilitando de esta manera múltiples niveles de anidamiento. Así, por ejemplo, la expresión **(1 3 (A B) 4)** es una lista de 4 elementos, cuyo tercer elemento es una lista de 2 elementos. La lista vacía se denota ( ) o con el átomo NIL.

2.2 Evaluación

Para entender como evalúa Lisp estas expresiones, es necesario conocer su modelo de ejecución general.

Lisp ejecuta normalmente bajo control de un intérprete. Los programas Lisp se ejecutan en un ambiente interactivo lo que resulta en un cambio en la concepción de lo que entendemos como "programa principal". En este caso, es el usuario quien ingresa el "programa principal' como una secuencia de expresiones a ser evaluadas. El sistema Lisp evalúa cada expresión ingresada y muestra el resultado inmediatamente, trabajando en forma análoga a una poderosa calculadora.

Cuando el sistema Lisp está á listo para recibir una orden desde el usuario, informa esta situación mostrando un símbolo especial (como por ejemplo un asterisco * o en el caso de la versión LispStudio el símbolo >). Esto habilita al usuario a ingresar una expresión para su evaluación.

Para evaluar una operación o función en Lisp, la operación debe ser ingresada en forma de lista respetando la notación prefija, es decir, el primer elemento de la lista es la función a ser evaluada y los elementos restantes son sus argumentos.

```
> (+ 2 3)
5

> (/ 8 4)
2
```

Este tipo de notación, permite extender operaciones que normalmente están restringidas a 2 argumentos, y trabajar de manera uniforme con funciones de cualquier anidad:

```
> (+ 2 3 8)
13

> (/ 8 4 2)
1

> (1+  5)
6

> (MAX 3 8 9 5)
9
```

Los argumentos de una función u operación, pueden ser no solo átomos sino también otras operaciones, permitiendo así múltiples niveles de anidamiento.

```
> (+ 3 (* 4 2))
11

> (+ (* 2 2) (/ 4 2))
6
```

```
> (* (+ 1 2) (+ 3 (* 2 2)))
21
```

Cuando Lisp evalúa una lista (llamada a función, como en los ejemplos anteriores, realiza normalmente los siguientes pasos:

1. Asume que el primer elemento de la lista es la función a aplicar.
2. Evalúa (recursivamente) cada argumento, de izquierda a derecha.
3. Pasa los valores de los argumentos a la función a aplicar.
4. Ejecuta la función con los argumentos recibidos.
5. Devuelve el valor calculado por la función.

Hasta ahora hemos visto como se evalúa una lista; dado que Lisp prueba de evaluar cualquier tipo de expresión ingresada, una pregunta obvia es como evalúa los átomos. Uno esperaría que así como Lisp evalúa (+ 2 3) realizando un cálculo y retornando el resultado de la operación, si tipeamos

```
> A
```

me devuelva el resultado de la evaluación de A. En este caso sólo obtendremos un resultado de error, ya que Lisp al evaluar un átomo retorna el valor asociado con el átomo y en este caso el átomo A no tiene ningún valor asociado. Una forma de asignar valores a átomos es utilizando la función primitiva **SETF** (Lisp no diferencia entre mayúsculas y minúsculas en los nombres utilizados para los nombres de funciones y símbolos en general. Sin embargo, dado que al mostrar los resultados siempre utiliza mayúsculas, nosotros también usaremos la convención de escribir todos los símbolos en mayúscula). Esta función es similar a una sentencia de asignación en los lenguajes convencionales, y si quisiéramos hacer algo equivalente a la asignación a = 5 deberíamos tipear

```
> (SETF A 5)
5

> A
5
```

SETF no sólo asigna un valor a un átomo sino que también retorna un valor (como todas las funciones de Lisp). En este caso el valor retornado es el valor asignado, pero por convención y de ahora en más no lo escribiremos ya que en general sólo nos interesa el efecto colateral del SETF (la asignación). Con SETF puedo asignar a un átomo el resultado de la evaluación de cualquier expresión.

```
>(SETF A (* (+ 1 2) (+ 3 (* 2 2))))

> A
21
```

Una vez que un átomo tiene un valor asignado puede ser usado como argumento en otras expresiones.

```
> (SETF A 8)

> (SETF B 4)

> (/ A B)
2
```

Existen átomos que no necesitan (y en realidad tampoco se puede) asociarles ningún valor, ya que el resultado de su evaluación es el mismo átomo. Entre estos átomos podemos citar a dos

símbolos de propósito especial, NIL y T, y a los números en general.

```
>T
T

> NIL
NIL

> 2
2

> 3.14
3.14
```

2.3 Deteniendo la evaluación

Hasta ahora hemos asumido que Lisp evalúa todas la expresiones que ingresamos. Así, cuando ingresamos (+ 2 3) se realiza la operación de suma y Lisp retorna su resultado (un 5). La lista (+ 2 3) en este caso fue utilizado para darle una directiva al intérprete para realizar una operación, pero uno también lo podría querer ver como un dato, en este caso una lista de 3 elementos, que no requiere evaluación sino que debe ser tratado como una constante. Lisp tiene un símbolo especial: **' (quote)**, que detiene la evaluación estándar, y su efecto es que la expresión contigua al ', sea retornada textualmente como si fuera una constante. El ', detiene la evaluación de cualquier tipo de expresión, ya sea lista o símbolo.

```
> (+ 2 3)
5

> '(+23)
(+ 2 3)

> (SETF A (+ 2 3))
> (SETF B '(+ 2 3))

> A
5

> 'A
A

> B
(+ 2 3)

> 'B
B

> (+ A 8)
13

> (+ 'A 8)
ERROR
```

Es importante observar que la función SETF maneja sus argumentos en forma no estándar, ya que no evalúa su primer argumento.

Existe en Lisp una función equivalente al SETF, denominada **SET**, pero que trabaja en forma normal.

```
> (SETF X 'Y)
> (SET X 'Z)

> X
Y

> Y
Z
```

Notar que (SETF X 'Z) es equivalente a (SET 'X 'Z).

2.4 Evaluación adicional

Así como existe una forma de detener la evaluación usando el ', existe en Lisp una función, denominada **EVAL**, que cumple el objetivo inverso, ya que realiza una evaluación adicional.

```
> (SETF A 'B)
> (SETF B 'C)

> 'A
A

> A
B

> (EVAL A)
C

> (EVAL 'A)
B

> (SETF X '(+ 2 3))

> X
(+ 2 3)

>(EVAL X)
5
```

3 Funciones para la manipulación de listas

Si bien hasta ahora, la mayoría de los ejemplos se han centrado en funciones aritméticas simples, la verdadera potencia de Lisp reside en su variedad de operaciones para la manipulación de listas. Entre las funciones más comunes para trabajar con listas, encontramos las primitiva **FIRST** y **REST**. FIRST retorna el primer elemento de la lista pasada como argumento, y REST devuelve el resto o cola de la lista, es decir, una lista con todos los elementos de la lista argumento a excepción del primero.

```
> (FIRST '(A B C))
A

> (REST '(A B C))
(B C)

> (FIRST '((A B) (C D)))
(A B)

> (REST '((A B) (C D)))
((C D))
```

Como todas las funciones de Lisp, pueden ser combinadas para formar expresiones más complejas.

```
> (FIRST (REST '(A B C)))
B

> (REST (FIRST (REST '((A B) (C D)))))
(D)
```

Observe la importancia del ' en la evaluación de una expresión como (FIRST (REST ' (A B C»»). En este caso Lisp asume que el primer elemento de la lista, el FIRST, es una función, pero antes de su invocación debe evaluar recursivamente a su argumento (REST ' (A B C)). Nuevamente Lisp asume que REST es una función pero antes de su ejecución debe evaluar el argumento del REST o sea ' (A B C). Dado que el ' detiene la evaluación, sólo se limita a devolver la expresión contigua, (A B C), la cual se convertirá en el parámetro actual para REST, quien devolverá á (B C) y esta expresión será el parámetro actual para FIRST quien retornará a B. En este proceso, si no hubiéramos colocado un ' , Lisp asumiría que (A B C) es una expresión a ser evaluada y daría un mensaje de error informando que no existe ninguna función de nombre A. Como vemos, la presencia o no del ' podrá á afectar significativamente el resultado de la evaluación de una expresión, al igual que su ubicación dentro de la expresión.

```
> (FIRST (REST (REST '(A B C D))))
C

> (FIRST (REST '(REST (A B C D))))
(A B C D)

> (FIRST '(REST (REST (A B C D))))
REST

> '(FIRST (REST (REST (A B C D))))
(FIRST (REST (REST (A B C D))))

> (SETF A '(A B C))

> (REST A)
(B C)

> (REST 'A)
ERROR
```

En el último ejemplo, el error surge de haber intentado hacer un REST directamente sobre el átomo simbólico A y no sobre el valor asociado a él (la lista (A C)).

Cuando intentamos hacer un REST sobre una lista de un elemento, el resultado será á un lista sin elementos, es decir, una lista vacía.

Para denotar una lista vacía, el usuario puede hacerlo utilizando la forma más intuitiva, es decir, la expresión (), o también el átomo NIL. Si bien ambas expresiones son totalmente equivalentes, cuando Lisp debe mostrar como resultado una lista vacía, siempre utiliza el átomo NIL.

```
> (REST '(A))
NIL
```

Si bien tanto FIRST como REST dan ERROR cuando son aplicados a átomos, por convención cuando son aplicados a la lista vacía, ambas retornan NIL.

```
> (FIRST ())
NIL
```

```
> (REST ())
NIL
```

Además de la primitiva FIRST, que retorna el primer elemento de una lista, Lisp provee una serie de funciones que permiten acceder a los elementos en otras posiciones de la lista (**SECOND**, **THIRD**, **FOURTH**, y hasta **TENTH**).

```
> (THIRD '(A B C D))
C

> (FOURTH '(A (B C) D (E F) G))
(E F)
```

FIRST y REST permiten tomar las componentes de una lista por separado. Ahora veremos algunas funciones que permiten construir listas: **CONS**, **APPEND** y **LIST**.

CONS toma como argumento una expresión y una lista y retorna una lista cuyo primer elemento es la expresión y los elementos restantes son los elementos de la lista que se pasa como segundo argumento. Dicho de otra manera, el CONS retorna el resultado de agregar el primer argumento a la lista del segundo argumento (en la primera posición).

```
> (CONS 'A '(B C))
(A B C)

> (CONS '(A G) '(B C))
((A G) B C)

> (SETF LISTA '(1 2 3))
> (CONS 'PEPE LISTA)
(PEPE 1 2 3)
```

APPEND, por su parte, realiza la concatenación de las lista pasadas como argumentos.

```
> (APPEND '(A B) '(C D))
(A B C D)

> (APPEND '(A (B)) '(C D) '((E)) () '(((F))))
(A (B) C D (E) ((F)))
```

LIST, construye una lista cuyos elementos son sus argumentos.

```
> (LIST 'A 'B 'C)
(A B C)

> (LIST 'A '(B C))
(A (B C))
```

La diferencia entre CONS, APPEND y LIST, puede observarse en el siguiente ejemplo:

```
> (SETF LISAB '(A B))
> (SETF LISCD '(C D))

> (CONS LISAB LISCD)
((A B) C D)

> (APPEND LISAB LISCD)
(A B C D)
```

```
> (LIST LISAB LISCD)
((A B) (C D))

> (APPEND 'LISAB LISCD)
ERROR

> (CONS 'LISAB LISCD)
(LISAB C D)

> (LIST 'LISAB LISCD)
(LISAB (C D))
```

Es importante observar que CONS, APPEND y LIST no alteran los valores de sus argumentos. Si deseamos modificar una lista asignada a un símbolo, estas funciones deberán ser combinadas con el uso del SETF.

```
> (SETF L1 '(1 2 3))

> (CONS 0 L1)
(0 1 2 3)

> L1
(1 2 3)

> (SETF L1 (CONS 0 L1))
> L1
(0 1 2 3)
```

Si bien existen muchas más funciones para el manejo de listas, finalizaremos esta sección mencionando 3 funciones frecuentemente usadas en Lisp: **LAST**, **LENGTH** y **REVERSE**.

Uno intuitivamente podría pensar que LAST retorna el último elemento de la lista argumento, pero en realidad devuelve una lista con el último elemento. LENGTH y REVERSE por su parte realizan las funciones que sus nombres indican, pero teniendo en mente que cuando la lista argumento tiene múltiples niveles, siempre se trabaja sobre el nivel superior de la lista.

```
> (SETF LISTA '(A (B C) (D E) F G))

> (LAST LISTA)
(G)

> (LENGTH LISTA)
5

> (REVERSE LISTA)
(G F (D E) (B C) A)
```

4 Predicados

Un predicado es una función que retorna un valor que representa verdadero o falso. El **NIL** siempre indica falso. Para indicar que algo es verdadero se suele utilizar el símbolo especial **T**, pero en realidad cualquier cosa distinta de NIL es considerada como verdadero.

El predicado **EQUAL** chequea sí los valores de sus argumentos son la misma expresión.

```
> (EQUAL 4 (+ 2 2))
T
```

```
> (EQUAL 4 '(+ 2 2))
NIL

> (EQUAL '(+ 2 2) '(+ 2 2))
T
```

El predicado **MEMBER** chequea la pertenencia de un elemento a una lista. Sí el primer argumento no pertenece al segundo argumento retorna NIL, pero sí pertenece no retorna T, sino la lista formada por los elementos del segundo argumento a partir de donde se encontró ó el primer argumento. Hay que tener en cuenta que sí tenemos listas de múltiples niveles, el MEMBER sólo trabaja sobre los elementos del nivel superior.

```
> (SETF LISTA '(PEPE (TITO TOTA) PEDRO CARLOS RAUL))

> (MEMBER 'JOSE LISTA)
NIL

> (MEMBER 'CARLOS LISTA)
(CARLOS RAUL)

> (MEMBER 'TITO LISTA)
NIL
```

4.1 Predicados de control de tipo de expresión

Lisp cuenta con un conjunto de primitivas para controlar si un objeto pertenece a un tipo particular de dato (o tipo de expresión). Las más importantes se detallan en la siguiente figura:

Nombre	Propósito
ATOM	¿El argumento es un átomo?
NUMBERP	¿El argumento es un número?
SYMBOLP	¿El argumento es un símbolo?
LISTP	¿El argumento es una lista?

```
>(SETF A '(PEPE 1 (TITO)))

> (ATOM A)
NIL

> (ATOM 'A)
T

> (NUMBERP (SECOND A))
T

> (LISTP A)
T

>(LISTP 'A)
NIL
```

Un caso muy particular surge de la total equivalencia entre las dos expresiones utilizadas para representar la lista vacía: la lista () y el átomo simbólico NIL. Ambas expresiones son listas y átomos (simbólicos) al mismo tiempo, por lo que tanto ATOM, como LISTP como SYMBOLP responderán T, cuando son aplicados a cualquiera de las dos expresiones.

Existen dos predicados para chequear si el argumento es una lista vacía: **NULL** y **ENDP**. La

diferencia es que ENDP es más restrictiva, ya que NULL controla si el argumento es una lista vacía, mientras que ENDP controla si el argumento, el cual debe ser una lista, está á vacía (no funciona con átomos).

```
> (NULL '(ESTO ES UNA LISTA))
NIL

> (ENDP '(ESTO ES UNA LISTA))
NIL

> (NULL ())
T

> (ENDP ())
T

> (NULL 'A)
NIL

> (ENDP 'A)
ERROR
```

4.2 Predicados con argumentos numéricos

Lisp cuenta con una amplia variedad de predicados que trabajan sobre números:

Nombre	Propósito
ZEROP	¿El argumento es 0?
PLUSP	¿El argumento es positivo?
MINUSP	¿El argumento es negativo?
EVENP	¿El argumento es par?
ODDP	¿El argumento es impar?
>	¿Los argumentos están en orden descendente estricto?
<	¿Los argumentos están en orden ascendente estricto?

Todos estos predicados esperan argumentos numéricos, produciéndose un error en otro caso. En la mayoría de los casos el significado de estos predicados es obvio y no los analizaremos en detalle. No obstante esto, es importante observar como funcionan los predicados > y <. Estos predicados verifican si dos o más argumentos están en orden descendente estricto, o en orden ascendente estricto.

Para el caso particular en que tenemos dos argumentos, trabajan como el mayor y el menor usuales.

```
> (> 4 2)
T

> (> 4 4)
NIL

> (> 2 4)
NIL

> (> 8 4 2)
T
```

```
> (> 8 2 4)
NIL
```

4.3 Combinando predicados usando AND y OR

Lisp permite combinar predicados usando las conectivas **AND** y **OR**. La función AND devuelve NIL si alguno de sus argumentos evalúa como NIL, y devuelve un valor distinto de NIL en caso contrario. La función OR devuelve NIL si todos sus argumentos evalúan como NIL y devuelve un valor distinto de NIL en caso contrario.

El AND, evalúa sus argumentos de izquierda a derecha y en cuanto uno de sus argumentos evalúa como NIL, retorna NIL sin evaluar los argumentos restantes. Si todos los argumentos evalúan valores distintos de NIL, el resultado del AND es la evaluación del último argumento.

El OR, también evalúa sus argumentos de izquierda a derecha y retorna el resultado de la evaluación del primer argumento cuya evaluación es distinta de NIL, sin evaluar los argumentos restantes. En caso que la evaluación de todos los argumentos sea NIL, el OR retorna NIL.

```
> (SETF LISTA '(A B C))

> (AND (EQUAL (LENGTH LISTA) 3) (MEMBER 'B LISTA))
(B C)

> (AND (> (LENGTH LISTA) 3) (MEMBER 'B LISTA))
NIL

> (OR (EQUAL (LENGTH LISTA) 3) (MEMBER 'B LISTA))
T

> (OR (> (LENGTH LISTA) 3) (MEMBER 'B LISTA))
(B C)

> (OR (> (LENGTH LISTA) 3) (MEMBER 'D LISTA))
NIL
```

La negación en Lisp, es implementada mediante la función **NOT**. NOT devuelve T si su argumento es NIL, y NIL si su argumento es distinto de NIL.

```
> (NOT NIL)
T

> (NOT T)
NIL

> (NOT (MEMBER 'B '(A B C)))
NIL

> (NOT (MEMBER 'D '(A B C)))
T
```

Observe la total equivalencia entre las funciones NULL y NOT. Si bien ambas trabajan de la misma manera el uso de una u otra servirá á para indicar si uno está á pensando en valores lógicos o en listas.

5 Condicionales

La forma de evaluación estándar de Lisp, establece que todos los argumentos de una función

deben ser evaluados. Los condicionales son una excepción a esta regla, permitiendo que la evaluación de los argumentos dependa de la evaluación de uno o más predicados. El condicional más elemental es la función **IF**, que implementa la selección tradicional. El formato general del IF es:

(IF <test> <expresión-verdadero> <expresión-falso>)

Cuando se ejecuta el IF, primero se ejecuta el test (un predicado); si el resultado es distinto de NIL se evalúa <expresión-verdadero> y en caso contrario se evalúa <expresión-falso>. El resultado retornado por la expresión evaluada, será á el valor de retorno del IF.

```
> (SETF A 5)

> (IF (AND (> A 3) (< A 8)) A (+ A 1))
5

>(SETF A 9)

> (IF (AND (> A 3) (< A 8)) A (+ A 1))
10
```

Otro condicional muy utilizado en Lisp es el **COND**. Esta función es más general que el IF, ya que permite una evaluación condicional basada sobre múltiples tests. El formato general del COND es:

```
(       COND  (<condición 1> <consecuente 1-1> .....)
              (<condición 2> <consecuente 2-1> .....)
                    ...
                    ...
              (<condición m> <consecuente m-1> .....)
)
```

El símbolo COND es seguido por un número variable de listas denominadas cláusulas. Cada cláusula contiene una condición seguida por 0 o más expresiones denominadas consecuentes. Cuando se ejecuta un COND, se evalúan las condiciones de las cláusulas una a una (y comenzando por la primera), hasta encontrar una condición cuyo valor retornado sea distinto de NIL. En ese caso, se dice que la cláusula es disparada y todos los consecuentes de la cláusula son evaluados. El valor que retorna el COND es el valor de la evaluación del último consecuente de la cláusula disparada. Si la evaluación de las condiciones de todas las cláusulas es NIL, el valor de retorno del COND es NIL.

```
>(SETF L1 '(A B C))

>(SETF L2 '(D E F))

>       (COND ((AND (ENDP L1) (ENDP L2)) 'AMBAS-VACIAS)
                    ((ENDP L1) 'VACIA1) ((ENDP L2) 'VACIA2)
                    (T (SETF SUMA (+ (LENGTH L1) (LENGTH L2))) SUMA)
)
6
```

Como se puede observar en el ejemplo, es común que la condición de la última cláusula sea el átomo T. En realidad, podría ser cualquier expresión suya evaluación siempre sea distinta de NIL, de manera tal de ser una cláusula "por defecto" que siempre será disparada cuando las cláusulas anteriores han fallado.

6 Definición de funciones

Hasta ahora hemos usado un conjunto de funciones primitivas provistas por Lisp.

En esta sección, veremos de que manera podemos definir nuestras propias funciones, las cuales se comportarán en forma análoga a las ya provistas por el sistema.

6.1 La primitiva DEFUN

La primitiva que nos permite definir nuestras propias funciones, es la función DEFUN, que tiene el siguiente formato general:

```
(DEFUN <nombre de función> (<parámetro 1>   <parámetro n>)
        <expresión 1>
        ...
        ...
        <expresión m>
)
```

El nombre de la función debe ser un átomo simbólico. Al igual que cualquier otra función, DEFUN también retorna un valor, que en esta caso será el nombre de la nueva función definida.

Si por ejemplo, quisiéramos definir una función que, dado un argumento numérico, devuelva el doble del argumento, podríamos hacer:

```
> (DEFUN DOBLE (N)
        (* 2 N)
)
DOBLE
```

Esta función podrá ser usada posteriormente como cualquier otra primitiva de Lisp.

```
> (DOBLE 4)
8

> (SETF A (DOBLE 9))

> A
18
```

Cuando una función es usada (invocada), toma lugar el siguiente proceso:

1. Se reserva memoria para contener el resultado de la evaluación de los parámetros reales.

2. Se ligan los parámetros formales a éstos lugares de memoria (salvando previamente los viejos valores si fuera necesario).

3. Se evalúan las expresiones del cuerpo de la función y se retorna como resultado la evaluación de la última expresión (las expresiones previas a la última sólo suelen tener la finalidad de obtener efectos colaterales (p. ej. SETF).

4. Liberar la memoria reservada para los parámetros.

5. Restaurar los viejos valores de los parámetros formales si fuera necesario.

Es importante destacar, que Lisp crea una especie de "defensa virtual" de los parámetros formales definidos en una función. Esto significa que si uno de los símbolos utilizados como parámetro formal, ha tenido otro uso previo, Lisp preserva estos valores, manteniéndolos inalterados al finalizar la ejecución de la función (inclusive en el caso en que el parámetro formal hubiera sido alterado explícitamente con un SETF dentro de la función).

```
>(SETF N 6)

> (DOBLE 4)
8

> N
6
```

Para ver otro ejemplo más completo de una definición de función, consideremos la situación en que, por algún motivo, deseo tomar 2 listas y obtener una nueva lista de 2 elementos cuyo primer elemento es el primero de la primer lista incrementado en 1 y el segundo es el último de la segunda lista como está. Este efecto, lo podríamos lograr haciendo directamente:

```
>(SETF L1 '(1 2 3))

>(SETF L2 '(4 5 6 7))

> (CONS (1+ (FIRST L1)) (LAST L2))
(2 7)
```

Si este tipo de procesamiento necesitáramos hacerlo para cualquier par de listas, podríamos definir una función (a la que llamaremos FUN1) como la siguiente:

```
> (DEFUN FUN1 (LISTA1 LISTA2)
        (CONS (1+ (FIRST LISTA)) (LAST LISTA2))
)
FUN1

> (FUN1 L1 L2)
(2 7)

> (FUN1 L2 L1)
(5 3)

> (FUN1 '(8 6) L2)
(9 7)
```

Una función puede también tener varias expresiones y la podría haber escrito:

```
> (DEFUN FUN1 (LISTA1 LISTA2)
        (SETF TEMP1 (1+ (FIRST LISTA)))
        (SETF TEMP2 (LAST LISTA2))
        (CONS TEMP1 TEMP2)
)
FUN1
```

Debemos destacar sin embargo, que este último ejemplo en el cual se realiza una especie de "secuencia" de expresiones, utilizando valores temporarios guardados con SETF, no constituye una buena práctica de programación funcional y debería ser evitado en lo posible.

6.2 Recursión

El término recursión, es usado normalmente en computación para referenciar un tipo de estructura de control, generada a partir de la definición de una función en términos de sí misma.

Una definición de función recursiva consta normalmente de:

- La utilización de una o más copias de la función que está siendo definida, que toman como argumentos una versión simplificada del problema original.

- Uno o más "puntos de parada", para aquellos argumentos en los que el valor de la función es conocido, y el resultado es directo y no implica nuevas invocaciones recursivas de la función.

La definición de funciones recursivas permite resolver problemas en forma clara y directa, especialmente cuando el problema a resolver es inherentemente recursivo, o puede ser planteado en forma recursiva sin dificultad. Un ejemplo clásico de este tipo de problemas, es el cálculo del factorial (!) de un número, que se puede definir recursivamente de la siguiente manera:

factorial(0) = 1
factorial(N) = N * (factorial (N - 1)); si N > 0

El factorial puede ser implementado sencillamente en Lisp, mediante la siguiente definición de función recursiva:

```
> (DEFUN FACTORIAL (N)
        (IF (ZEROP N)
                1
                (* N (FACTORIAL (1- N)))
)
FACTORIAL

> (FACTORIAL 0)
1

> (FACTORIAL 3)
6
```

Otro problema de resolución directa usando recursión, es la implementación de una función que trabaje como la primitiva MEMBER de Lisp (y que nosotros llamaremos MIEMBRO). Esta función tendrá á 2 puntos de parada: el primero controlará á si el segundo argumento es la lista vacía, lo cual acontecerá á cuando la función ha sido aplicada directamente a NIL, o se ha terminado de recorrer completamente la lista y no se ha encontrado al primer argumento. En este caso la función debe devolver NIL. El segundo punto de parada, es cuando el primer argumento es igual al primer elemento del segundo argumento, en cuyo caso la función debe devolver el segundo argumento como está.

Cuando ninguno de estos 2 casos ocurra, la función MIEMBRO retornará á el mismo valor que MIEMBRO aplicado a la cola de la lista del segundo argumento.

```
> (DEFUN MIEMBRO (ELEMENTO LISTA)
        (COND ((ENDP LISTA) NIL)
                        ((EQUAL ELEMENTO (FIRST LISTA)) LISTA)
                        (T (MIEMBRO ELEMENTO (REST LISTA)))
        )
)
MIEMBRO

> (MIEMBRO 3 ())
NIL

> (MIEMBRO 3 '(1 2 4))
NIL

> (MIEMBRO 3 '(1 2 3 4))
(3 4)
```

El hecho de que la estructura de control predominante en Lisp sea la recursión no es casual, si consideramos que la estructura de datos predominante en Lisp, la lista, también es de naturaleza recursiva. Uno puede ver a una lista como una estructura formada por su primer elemento, y la cola, la cual a su vez es una lista. Esta idea puede ser utilizada cuando debemos definir una función que

realice algún tipo de procesamiento sobre todos los elementos de una lista. En este caso, es común definir la función recursivamente de manera tal de procesar el primer elemento, e invocar la función recursivamente sobre la lista restante, y así í sucesivamente hasta llegar al final de la cola (NIL).

Teniendo estas ideas en mente, uno podría definir una función que incrementa en 1 todas las componentes de una lista de números (que llamaremos INCREMENTAR1) de la siguiente manera:

```
>(DEFUN INCREMENTAR1 (LISTA)
        (IF (ENDP LISTA)
                NIL
                (CONS (1+ (FIRST LISTA)) (INCREMENTAR1 (REST LISTA));
        )
)
INCREMENTAR1

> (INCREMENTAR1 '(1 3 6 8))
(2 4 7 9)
```

En el ejemplo anterior, la lista tenía sólo un nivel de profundidad y para procesar un elemento de la lista sólo necesitábamos accederlo mediante el FIRST para posteriormente aplicar el tipo de procesamiento requerido.

Cuando se trabaja con expresiones anidadas (listas de múltiples niveles), la situación se complica ya que al tomar el FIRST de la lista, este elemento puede ser una lista que debe a su vez ser procesada recursivamente. Es por este motivo, que cuando se deben procesar expresiones anidadas en Lisp, usualmente la función es doblemente recursiva ya que la función se debe llamar a sí í mismo dos veces: una vez para analizar el primer elemento y otra para analizar los elementos restantes. Como se puede ver, esta forma de descomponer y procesar la lista conducirá á eventualmente a que la función sea aplicada a la lista vacía o a un átomo, siendo éstos los puntos de parada más usuales en estas situaciones.

A modo de ejemplo de lo discutido previamente, veamos como se podría definir una función que cuente la cantidad de 0' s en una expresión cualquiera en Lisp:

```
> (DEFUN CONTAR-CEROS (EXPRESION)
        (COND ((NULL EXPRESION) 0)
                        ((EQUAL EXPRESION 0) 1) ((ATOM EXPRESION) 0)
                        (T (+   (CONTAR-CEROS (FIRST EXPRESION))
                                (CONTAR-CEROS (REST EXPRESION))))
        )
)
CONTAR-CEROS

> (CONTAR-CEROS 0)
1

> (CONTAR-CEROS 'A)
0

> (CONTAR-CEROS '((0) 3 0 (((0 A)))))
3
```

6.3 Iteración

La recursión es una estructura de control que permite realizar alguna acción en forma repetida. Usualmente en Lisp, esta acción se repetía para cada elemento de una lista. Otra estructura de control que permite repetir una acción sin utilizar la recursión es la iteración. Lisp provee una primitiva denominada **MAPCAR** que permite repetir una acción sobre todos los elementos de una lista en forma iterativa (sin usar la recursión).

La función MAPCAR, tiene el siguiente formato general:

(MAPCAR #'<nombre-de-función> <lista>)

La función MAPCAR toma como primer argumento un nombre de función (o para ser más precisos un objeto función y una lista, y retorna como resultado una lista cuyos elementos son los resultados de aplicar la función con cada elemento de la lista como argumento.

```
> (MAPCAR #'ODDP '(1 2 3))
(T NIL T)
```

Usando MAPCAR, la función INCREMENTAR1 de la sección 6.2, se podría haber definido en forma iterativa de la siguiente manera:

```
> (DEFUN INCREMENTAR1 (LISTA)
        (MAPCAR #'1+ LISTA)
)
INCREMENTAR1
```

En los dos ejemplos previos, las funciones aplicadas a cada elemento de la lista fueron funciones de un sólo argumento (ODDP y 1+), sin embargo, MAPCAR puede tomar como argumento una función de más de un argumento. En este caso, el MAPCAR deberá á ser invocado con tantas listas como argumentos requiera la función, y el MAPCAR será á el encargado de aplicar la función del primer argumento, suministrándole los argumentos requeridos, tomando uno de cada lista argumento.

```
> (MAPCAR #'EQUAL '(1 2 3) '(3 2 1))
(NIL T NIL)
```

La función MAPCAR no es la única función de Lisp que toma como uno de sus argumentos a otra función. Otra dos primitivas importantes que también usan una función como argumento son **FUNCALL** y **APPLY**.

FUNCALL aplica su primer argumento (una función) a los argumentos restantes.

El formato general del FUNCALL, se presenta a continuación:

(FUNCALL #'<nombre de función> <argumento 1> ... <argumento n>)

Algunos ejemplos del uso del FUNCALL son los siguientes:

```
> (FUNCALL #'LENGTH '(1 2 3))
3

> (FUNCALL #'APPEND '(1 2 3) '(A B))
(1 2 3 A B)
```

Una forma fácil de entender el efecto del uso del FUNCALL es considerar que el resultado es el mismo que si tapáramos la palabra FUNCALL con el dedo.

Como vemos, hacer un (FUNCALL 'APPEND ' (1 2 3) ' (A B)) es totalmente equivalente a hacer (APPEND ' (1 2 3) ' (A B)) pero esto no nos dice nada acerca de la utilidad del FUNCALL. En realidad, el FUNCALL es realmente útil cuando debemos definir nuestros propios procedimientos que toman funciones como argumentos. Consideremos la siguiente función:

```
> (DEFUN POSICION (FUNCION LISTA)
        (FUNCALL FUNCION LISTA)
)
POSICION
```

```
> (POSICION #'FIRST '(A B C))
A

> (POSICION #'THIRD '(A B C))
C
```

La función POSICION retorna el elemento del segundo argumento que se obtiene aplicando la función del primer argumento. Si en la definición de POSICION, hubiéramos intentado llamar directamente a (FUNCION LISTA), Lisp hubiera indicado un error, dado que la función FUNCION no está á definida.

La primitiva APPLY por su parte, también toma una función como primer argumento pero su segundo argumento es una lista. Su efecto es el mismo que si invocáramos la función no con la lista, sino con los elementos de la lista.

El formato general del APPLY, se presenta a continuación:

```
(APPLY #'<nombre de función> <lista de argumentos>)
```

Para observar la necesidad del APPLY consideremos el siguiente ejemplo:

```
> (+ '(3 5 7))
ERROR

> (APPLY #'+ '(3 5 7))
15
```

En el primer caso, el error surge de haber intentado hacer una suma con una lista, siendo que la suma solo acepta números como argumentos. En el segundo caso, el APPLY hace que la suma se realice sobre los elementos de la lista, logrando el efecto de ubicar a la operación al mismo nivel que sus argumentos y así trabajar en forma totalmente equivalente a si hubiéramos hecho (+ 3 5 7).

6.4 Combinando MAPCAR y recursión

Retornando al tema de la recursión e iteración es necesario aclarar que el uso del MAPCAR no es excluyente con la recursión.

Esto es particularmente cierto, en aquellas situaciones similares a las plantea das cuando hablamos del procesamiento recursivo de expresiones anidadas. Así como uno invocaba recursivamente a la función con el primer elemento de la lista por un lado, y recursivamente con el resto de los elementos de la lista, es obvio que la idea básica es invocar recursivamente sobre todos los elementos de la lista.

Esto también puede ser logrado utilizando el MAPCAR para aplicar la función recursivamente sobre todos los elementos de la lista. Si por ejemplo, quisiéramos definir la función CONTAR-CEROS utilizando el MAPCAR en la definición recursiva, una posible solución sería la siguiente:

```
> (DEFUN CONTAR-CEROS (EXPRESION)
        (COND ((NULL EXPRESION) 0)
                        ((EQUAL EXPRESION 0) 1)
                        ((ATOM EXPRESION) 0)
                        (T (APPLY #'+ (MAPCAR #'CONTAR-CEROS EXPRESION)))
        )
)
```

Si bien esta versión es muy similar a la anterior (de hecho los puntos de parada lo son), es importante observar la necesidad de incluir la primitiva APPLY para poder realizar la suma de los totales parciales retornados por las invocaciones recursivas de CONTAR-CEROS. El hecho de haber

incorporado el MAPCAR, que como sabemos devuelve una lista como resultado, incorpora un nuevo nivel de anidamiento, por lo que en este caso y en general, el uso del MAPCAR debe ser combinado con un APPLY que le permita procesar los valores retornados en forma de lista.

Bibliografía:

- Introducción a la Programación Funcional – Autor: Elías Samra Asan – Universidad Autónoma de México.

- Programación Declarativa – Autor: Manuel Lucena López – Departamento de Informática de la Escuela Politécnica Superior – Universidad de Jaén, España.

- Teoría de LISP – Autor: Marcelo Luis Errecalde – Departamento de Informática – Universidad Nacional de San Luis, Argentina.