Programación para la Inteligencia Artificial

Introducción a Lisp

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097

mailto:aguerra@uv.mx
https://www.uv.mx/personal/aguerra/pia

Maestría en Inteligencia Artificial 2023

Contenido

1 Introducción

2 Conceptos básicos

3 Listas

4 Mapeos

5 Macros

Objetivo

- Poder programar en Lisp tan pronto como sea posible.

PS Y el alien de Barski es casi Halcón de la UV!

Referencias

- Graham [1] presenta un estudio a fondo de técnicas avanzadas en Common Lisp.
- Norvig [3] se orienta a aplicaciones para la Inteligencia Artificial: `https://github.com/norvig/paip-lisp`
- Seibel [5] se orienta a la práctica general del uso del lenguaje. `http://www.gigamonkeys.com/book/`

Top-Level

- Uno escribe expresiones Lisp en el top-level, y el sistema despliega sus valores, al estar en un ciclo permanente REPL (Read-Eval-Print Loop).
- El prompt > indica que Lisp está esperando que una expresión sea escrita. La expresión es evaluada al pulsar return.
- Ejemplo. Evaluando un átomo:

```
1 > 1
2 1
3 >
```

Calculadora

- Las cosas son más interesantes cuando una expresión necesita algo de trabajo para ser evaluada.
- Ejemplo. Sumar dos números:

```
1 > (+ 2 3)
2 5
3 >
```

Operador, Argumentos, Prefijo

- En la expresión `(+ 2 3)`, + es llamado el operador; los números 2 y 3 son sus operandos o argumentos.
- Como el operador viene al principio de la expresión, esta notación se conoce como prefija y aunque parezca extraña al principio, veremos que es uno de los aspectos sobresalientes de Lisp...
- Provee uniformidad.

Generalidad del prefijo

- Si queremos sumar tres números en notación "normal"(infija), necesitaríamos usar dos veces el operador +: $2 + 3 + 4$.
- En Lisp sólo necesitamos agregar otro argumento:

```
1 > (+)
2 0
3 > (+ 2)
4 2
5 > (+ 2 3)
6 5
7 > (+ 2 3 5)
8 10
```

Paréntesis

- Como los operadores pueden tomar un número variable de argumentos, es necesario utilizar los paréntesis para indicar donde inicia y donde termina una expresión.
- Las expresiones se pueden anidar:

```
> (/ (- 7 1)(- 4 2))
3
```

- En español esto corresponde a siete menos uno, dividido por cuatro menos dos.

Estética minimalista

- Esto es todo lo que hay que decir sobre la notación en Lisp.
- Toda expresión Lisp es un átomo, p. ej., `1`, `a`, `alex`;
- O bien es una lista que consiste de cero o más expresiones delimitadas por paréntesis, p. ej., `()`, `(1)`, `(1 a alex)`
- Código y datos usan la misma notación en Lisp.

Evaluación

- En Lisp, + es una función y `(+ 2 3)` es una llamada a la función. Cuando Lisp evalúa una llamada a alguna función, lo hace en dos pasos:

  1. Los argumentos de la llamada son evaluados de izquierda a derecha. En este caso, los valores de los argumentos son 2 y 3, respectivamente.
  2. Los valores de los argumentos son pasados a la función nombrada por el operador. En este caso la función + que regresa 5.

Universidad Veracruzana

Evaluación recursiva.

- Si alguno de los argumentos es a su vez una llamada a una función, será evaluado con las mismas reglas.
- Ejemplo. Al evaluar la expresión (/ (- 7 1) (- 4 2)) pasa lo siguiente:

  1. Lisp evalúa el primer argumento de izquierda a derecha (- 7 1). 7 es evaluado como 7 y 1 como 1. Estos valores son pasados a la función - que regresa 6.
  2. El siguiente argumento (- 4 2) es evaluado. 4 es evaluado como 4 y 2 como 2. Estos valores son pasados a la función - que regresa 2.
  3. Los valores 6 y 2 son pasados a la función / que regresa 3.

Operadores especiales (quote)

- Un operador Lisp que no sigue la regla de evaluación es quote (abreviado como ’).
- La regla de evaluación de quote es –No evalúes nada, despliega lo que el usuario tecleo:

```
1 > (quote (+ 2 3))
2 (+ 2 3)
3 > '(+ 2 3)
4 (+ 2 3)
```

- Lisp provee el operador quote como una forma de evitar que una expresión sea evaluada.
- Más adelante veremos porque esta protección puede ser útil.

Universidad Veracruzana

Datos

- Lisp ofrece los tipos de datos que podemos encontrar en otros lenguajes de programación y otros que no.
- Un tipo de datos que ya usamos es el entero, que se escribe como una secuencia de dígitos, p. ej., 256.
- Otro tipo de datos es la cadena de caracteres que se delimita por comillas, p. ej., "ora et labora... pos ora".
- Enteros y cadenas de caracteres evalúan a ellos mismos.

```
1  > 256
2  256
3  > "ora et labora"
4  "ora et labora"
```

Símbolos

- Los símbolos son palabras para denotar valores que nos interesan.
- Normalmente se evaluan como si estuvieran escritos en mayúsculas, independientemente de como fueron tecleados.

```
1 > 'curso
2 CURSO
```

- Por lo general, no evalúan a si mismos, así que si es necesario referirse a ellos, se debe usar `quote`, como en ejemplo anterior.
- Problema. ¿Qué sucede si no protegemos `curso` con el quote?

Listas

- Las listas se representan como cero o más elementos entre paréntesis.
- Los elementos pueden ser de cualquier tipo, incluidas las listas:

```
1  > '(La lista (a b c) tiene 3 elementos)
2  (LA LISTA (A B C) TIENE 3 ELEMENTOS)
```

- Se debe usar quote con las listas, ya que de otra forma Lisp las tomaría como una llamada a función:

```
1  > '(+ 3 2)
2  (+ 3 2)
3  > (+ 3 2)
4  5
```

- Un sólo quote protege a toda la expresión, incluidas sus sub-expresiones.

Construcción de Listas (list)

- Se puede construir listas usando el operador `list` que es una función, y por lo tanto, sus argumentos son evaluados:

```
1 > (list 'mis (+ 4 2) "colegas")
2 (MIS 6 COLEGAS)
```

- Estética minimalista y pragmática, los programas Lisp se representan como listas.
- Si el argumento estético no bastará para defender la notación de Lisp, esto debe bastar –¡Un programa Lisp puede generar código Lisp! Por eso es necesario `quote`.

Protección de listas

- Si una lista es precedida por `quote`, la evaluación regresa la misma lista, en otro caso, la lista es evaluada como si fuese una llamada a una función.

```
1  > (list '(+ 2 3) (+ 2 3))
2  ((+ 2 3) 5)
```

Lista vacia

- En Lisp hay dos formas de representar la lista vacía, como un par de paréntesis o con el símbolo `NIL`.

```
1 > ()
2 NIL
3 > NIL
4 NIL
```

- Desafortunadamente `NIL` se usa también para denotar falso, por lo que es preferible usar `()` si queremos enfatizar lista vacía.

Constructor de listas (cons)

- La función `cons` construye listas. Si su segundo argumento es una lista, regresa una nueva lista con el primer argumento agregado en el frente.

```
> (cons 'a '(b c d))
(A B C D)
> (cons 'a (cons 'b nil))
(A B)
```

- El segundo ejemplo es equivalente a:

```
> (list 'a 'b)
(A B)
```

- Problema. ¿Qué sucede si el segundo argumento de `cons` no es una lista?

Acceso a listas (car y cdr)

- Las funciones primitivas para acceder a los elementos de una lista son `car` y `cdr`.
- El `car` de una lista es su primer elemento (el más a la izquierda) y el `cdr` es el resto de la lista (menos el primer elemento).

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

- Los nombres modernos para esta funciones son `first` y `rest`, pero...

Combinaciones de car y cdr

- Se pueden usar combinaciones de car y cdr para acceder a cualquier elemento de la lista.

```
1  > (car (cdr (cdr '(a b c d))))
2  C
3  > (caddr '(a b c d))
4  C
5  > (third '(a b c d))
6  C
7  > (cdadr '((alex 10) (luis 5) (juan 6)))
8  (5)
9  > (second (second '((alex 10) (luis 5) (juan 6))))
10 5
```

- Problema. ¿Cómo corregimos la línea 7?

Valores de verdad

- El símbolo `T` representa por default verdadero.
- La representación por default de falso es `NIL`!
- Ambos evaluan a si mismos.
- Ejemplo: El predicado `listp` regresa verdadero si su argumento es una lista:

```
1  > (listp '(a b c))
2  T
3  > (listp 34)
4  NIL
```

Predicados

- Una función cuyo valor de regreso se interpreta como un valor de verdad (verdadero o falso) se conoce como predicado.
- En lisp es común que el símbolo de un predicado termine en `p`.
- Como `nil` juega dos roles en Lisp, las funciones `null` (lista vacía) y `not` (negación) hacen exactamente lo mismo:

```
1 > (null nil)
2 T
3 > (not nil)
4 T
```

Condicional (if)

- El condicional más simple en Lisp es `if`.
- Normalmente toma tres argumentos: un predicado, una expresión then y una expresión else.
- El predicado es evaluado, si su valor es verdadero, la expresión then es evaluada; si su valor es falso, la expresión else es evaluada:

```
1  > (if (listp '(a b c d))
2        (+ 1 2)
3        (+ 3 4))
4  3
5  > (if (listp 34)
6        (+ 1 2)
7        (+ 3 4))
8  7
```

Universidad Veracruzana

If es una macro

- Como `quote`, `if` es un operador especial.
- No puede implementarse como una función, porque los argumentos de una función siempre se evalúan, y la idea al usar `if` es que sólo uno de sus argumentos sea evaluado.
- Para ello Lisp provee el concepto de macro [1].

Operadores Lógicos I

- Los operadores lógicos `and` y `or` se comportan como condicionales.
- Ambos toman cualquier número de argumentos, pero solo evalúan los necesarios para decidir que valor regresar.
- Si todos los argumentos son verdaderos (diferentes de `nil`), entonces `and` regresa el valor del último argumento evaluado.

```
1 > (and t (+ 1 2))
2 3
```

Operadores Lógicos II

- Pero si uno de los argumentos de `and` resulta falso, ninguno de los argumentos posteriores es evaluado y el operador regresa `nil`.
- De manera similar, `or` se detiene en cuanto encuentra un elemento verdadero.

```
1 > (or nil nil (+ 1 2) nil)
2 3
```

- Observen que los operadores lógicos son operadores especiales, definidos como macros.

Definición de funciones

- Es posible definir nuevas funciones con `defun` que toma normalmente tres argumentos:
  - Un nombre,
  - Una lista de parámetros y
  - Una o más expresiones que conforman el cuerpo de la función.
- Ejemplo: Así definiríamos `tercero`:

```
1  > (defun tercero (lst)
2      (caddr lst))
3  TERCERO
```

Defun

- El primer argumento de `defun` indica que el nombre de nuestra función definida será `tercero`.
- El segundo argumento `(lst)` indica que la función tiene un sólo argumento. Un símbolo usado de esta forma se conoce como variable. Cuando la variable representa el argumento de una función, se conoce como parámetro.
- El resto de la definición indica lo que se debe hacer para calcular el valor de la función –Para cualquier `lst`, se calculará el primer elemento, del resto, del resto del parámetro `(caddr lst)`:

```
1 > (tercero '(a b c d e))
2 C
```

Símbolos, listas y variables

- Ahora que hemos introducido el concepto de variable, es más sencillo entender lo que es un símbolo.
- Los símbolos son nombres de variables, que existen con derechos propios en el lenguaje Lisp.
- Por ello símbolos y listas deben protegerse con `quote` para ser denotados.
- Una lista debe protegerse porque de otra forma es procesada como si fuese código; un símbolo debe protegerse porque de otra forma es procesado como si fuese una variable.

Universidad Veracruzana

Funciones como generalización

- Podríamos decir que la definición de una función corresponde a la versión generalizada de una expresión Lisp.
- Ejemplo: La siguiente expresión verifica si la suma de 1 y 4 es mayor que 3:

```
1  > (> (+ 1 4) 3)
2  T
```

- Substituyendo los números particulares por variables, definimos una función que verifica si la suma de sus dos primeros argumentos es mayor que el tercero:

```
1  > (defun suma-mayor-que (x y z)
2      (> (+ x y) z))
3  SUMA-MAYOR-QUE
4  > (suma-mayor-que 1 4 3)
5  T
```

Programación incremental

- Lisp no distingue entre programa, procedimiento y función; todos cuentan como funciones y de hecho, casi todo el lenguaje está compuesto por funciones.
- Si se desea considerar una función en particular como *main*, es posible hacerlo, pero cualquier función puede ser llamada desde el top-level.
- Entre otras cosas, esto significa que es posible probar nuestros programas, pieza por pieza, conforme los vamos escribiendo, lo que se conoce como programación incremental (*bottom-up*).

Universidad Veracruzana

Recursividad

- Las funciones que hemos definido hasta ahora, llaman a otras funciones para hacer una parte de sus cálculos.
- Ejemplo: `suma-mayor-que` llama a las funciones $+$ y $>$.
- Una función puede llamar a cualquier otra función, incluida ella misma. Una función que se llama a si misma se conoce como recursiva.

Universidad Veracruzana

Miembro de

- En Lisp la función `member` verifica cuando algo es miembro de una lista.
- He aquí una versión recursiva simplificada de esta función:

```
1  > (defun miembro (obj lst)
2     (if (null lst)
3         nil
4         (if (eql (car lst) obj)
5             lst
6             (miembro obj (cdr lst)))))
7  MIEMBRO
```

- O usando `cond`

```
1  > (defun miembro (elt lst)
2      (cond ((null lst) nil)
3            ((eql elt (car lst)) lst)
4            (t (miembro elt (cdr lst)))))
5  MIEMBRO
```

Miembro de (Cont...)

- El predicado `eql` verifica si sus dos argumentos son idénticos, el resto lo hemos visto previamente.
- La llamada a `miembro` es como sigue:

```
1 > (miembro 'b '(a b c))
2 (B C)
3 > (miembro 'z '(a b c))
4 NIL
```

- Problema. ¿Podemos preguntar à la Prolog quién es miembro de la lista?

Miembro en castellano

- La descripción en castellano de lo que hace la función `miembro` es como sigue:
  1. Primero, verificar si la lista `lst` está vacía, en cuyo caso es evidente que `obj` no es un miembro de `lst`.
  2. De otra forma, si `obj` es el primer elemento de `lst` entonces es miembro de la lista.
  3. De otra forma, `obj` es miembro de `lst` únicamente si es miembro del resto de `lst`.

Estética

- Si bien los paréntesis delimitan las expresiones en Lisp, un programador en realidad usa los márgenes en el código para hacerlo más legible.
- Casi todo editor puede configurarse para verificar paréntesis bien balanceados. Ej. `:set sm` en el editor `vi`; o `M-x lisp-mode` en `Emacs`.
- Cualquier hacker en Lisp tendría problemas para leer algo como:

```
1  > (defun miembro (obj lst) (if (null lst) nil (if (eql
2      (car lst) obj) lst (miembro obj (cdr lst))))) MIEMBRO
```

Entradas y salidas

- Hasta el momento hemos procesado las E/S implícitamente, utilizando el top-level.
- Para programas interactivos esto no es suficiente. De hecho, casi todo Lisp actual incluye alguna librería para implementar interfaces gráficas, p. ej., CLIM, Common Graphics, LTK, etc.
- En este tutorial de introducción solo veremos algunas operaciones básicas de E/S.

Salida (format)

- Esta función toma dos o más argumentos: el primero indica donde debe imprimirse la salida, el segundo es una cadena que se usa como plantilla (*template*), y el resto son generalmente objetos cuya representación impresa será insertada en la plantilla.

```
1 > (format t "~A mas ~A igual a ~A. ~%" 2 3 (+ 2 3))
2 2 MAS 3 IGUAL A 5.
3 NIL
```

Evaluación de format

- Observen que dos líneas fueron desplegadas en el ejemplo anterior.
- La primera es producida por `format` y la segunda es el valor devuelto por la llamada a `format`, desplegada por el top-level como se hace con toda función.
- Normalmente no llamamos a `format` en el top-level, sino dentro de alguna función, por lo que el valor que regresa queda generalmente oculto.

Format en detalle

- El primer argumento de `format`, `t`, indica que la salida será desplegada en el dispositivo estándar de salida, generalmente el top-level.
- El segundo argumento es una cadena de control, sirve como molde de lo que será impreso.
- Dentro de esta cadena, cada `A` reserva espacio para insertar un objeto y cada % indica un salto de línea.
- Los espacios reservados de esta forma, son ocupados por el resto de los argumentos en el orden en que son evaluados.

Entrada (read)

- Sin argumentos, normalmente la lectura se hace a partir del top-level.
- Ejemplo: Una función que despliega un mensaje y lee la respuesta el usuario:

```
> (defun pregunta (s)
    (format t "~A" s)
    (read))
PREGUNTA
> (pregunta "Su edad: ")
Su edad: 34
34
```

Secuencias de expresiones

- La función `pregunta`, aunque corta, muestra algo que no habíamos visto antes: su cuerpo incluye más de una expresión Lisp.
- El cuerpo de una función puede incluir cualquier número de expresiones.
- Cuando la función es llamada, las expresiones en su cuerpo son evaluadas en orden y la función regresará el valor de la última expresión evaluada.
- Esto puede enfatizarse usando `progn`:

```
1  > (defun pregunta (string)
2      (progn
3        (format t "~A" string)
4        (read)))
5  PREGUNTA
```

Efectos colaterales y Lisp puro

- Hasta el momento, lo que hemos mostrado se conoce como *Lisp puro*, esto es, Lisp sin efectos colaterales.
- Un efecto colateral es un cambio en el sistema Lisp producto de la evaluación de una expresión.
- Cuando evaluamos `(+ 2 3)`, no hay efectos colaterales, el sistema simplemente regresa el valor 5. Pero al usar `format`, además de obtener el valor `nil`, el sistema imprime algo, esto es un tipo de efecto colateral.

Universidad Veracruzana

Lisp puro y expresiones

- Cuando se escribe código sin efectos colaterales, no hay razón alguna para definir funciones cuyo cuerpo incluye más de una expresión.
- La última expresión evaluada en el cuerpo producirá el valor de la función, pero el valor de las expresiones evaluadas antes se perderá.

Variables locales (let)

- Uno de los operadores más comunes en Lisp es `let`, que permite la creación de nuevas variables locales.

```
1 > (let ((x 1)(y 2))
2       (+ x y))
3 3
```

Estructura de let

- Una expresión `let` tiene dos partes:
  - Una lista de expresiones definiendo las nuevas variables locales, cada una de ellas con la forma (*variable expresión*). Cada variable se inicializa con el valor de la expresión asociada.
  - Ejemplo. En el caso anterior se han creado dos variables, `x` e `y`, con los valores 1 y 2 respectivamente.
  - El cuerpo del let, donde las variables son evaluadas.
  - Ejemplo. `(+ x y)`

Preguntar con reservas

- Ejemplo. Veamos una función `preguntar` más selectiva:

```
1  > (defun preguntar-num ()
2      (format t "Por favor, escriba un numero: ")
3      (let ((val (read)))
4        (if (numberp val)
5            val
6            (preguntar-num))))
7  PREGUNTAR-NUM
```

Variables locales

- Esta función crea la variable local `val` para guardar el valor que regresa `read`.
- Como este valor puede ahora ser manipulado por Lisp, la función revisa que se ha escrito para decidir que hacer. Si el usuario ha escrito algo que no es un número, la función vuelve a llamarse a si misma:

```
1  > (preguntar-num)
2  Por favor, escriba un numero: a
3  Por favor, escriba un numero: (un numero)
4  Por favor, escriba un numero: 3
5  3
```

Universidad Veracruzana

Variables globales

- Se puede crear una variable global con un símbolo dado y un valor, usando `defparameter`:

```
1  > (defparameter *glob* 1970)
2  *GLOB*
```

- Esta variable es visible donde sea, salvo en contextos que definan una variable local con el mismo nombre.
- Para evitar errores accidentales con los nombre de las variables, se usa la convención de nombrar a las variables globales con símbolos que inicien y terminen en asterisco.

Constantes globales

- Se pueden definir también constantes globales usando `defconstant`:

```
1  > (defconstant limit (+ *glob* 1))
```

- No hay necesidad de dar a las constantes nombres distintivos porque si se intenta usar el nombre de una constante para una variable se produce un error.
- Para verificar si un símbolo es el nombre de una variable global o constante, se puede usar `boundp`:

```
1  > (boundp '*glob*)
2  T
```

Asignaciones (setf)

- En Lisp el operador de asignación más común es `setf`.
- Se puede usar para asignar valores a cualquier tipo de variable.

```
1  > (setf *glob* 2000)
2  2000
3  > (let ((n 10))
4      (setf n 2)
5      n)
6  2
```

Estructura de setf

- Cuando el primer argumento de `setf` es un símbolo que no es el nombre de una variable local, se asume que se trata de una variable global.

```
1 > (setf x (list 'a 'b 'c))
2 (A B C)
3 > (car x)
4 A
```

Versatilidad de setf

- Se puede hacer más que simplemente asignar valores a variables. El primer argumento de `setf` puede ser tanto una expresión, como un nombre de variable.
- En el primer caso, el valor representado por el segundo argumento es insertado en el lugar al que hace referencia la expresión.

```
1 > (setf (car x) 'n)
2 N
3 > x
4 (N B C)
```

Versatilidad de setf (cont...)

- Se puede dar cualquier número de argumentos pares a `setf`. Una expresión de la forma:

```
1 > (setf a 'b c 'd e 'f)
2 F
```

- es equivalente a:

```
1 > (set a 'b)
2 B
3 > (set b 'c)
4 C
5 > (set e 'f)
6 F
```

Programación funcional

- La programación funcional significa, entre otras cosas, escribir programas que trabajan regresando valores, en lugar de modificar cosas.
- Era el paradigma de programación dominante en Lisp, aunque ahora se favorece el paradigma orientado a objetos conocido como CLOS [2].
- También tiene aspectos imperativos como `loop`.

Remove

- La función `remove`, por ejemplo, toma un átomo y una lista y regresa una nueva lista que contiene todos los elementos de la lista original, menos el átomo indicado:

```
1 > (setf lst '(k a r a t e))
2 (K A R A T E)
3 > (remove 'a lst)
4 (K R T E)
```

Remove ¿remueve?

- ¿Por qué no decir simplemente que `remove` remueve un objeto dado de una lista? Porque esto no es lo que la función hace. La lista original no es modificada:

```
1 > lst
2 (K A R A T E)
```

Asignación destructiva

- Si se desea que la lista original sea afectada, se puede evaluar la siguiente expresión:

```
1 > (setf lst (remove 'a lst))
2 (K R T E)
3 > lst
4 (K R T E)
```

Mínimo uso de setf

- La programación funcional significa, esencialmente, evitar `setf` y otras expresiones con el mismo tipo de efecto colateral.
- Esto puede parecer contra intuitivo y hasta no deseable. Si bien programar totalmente sin efectos colaterales es inconveniente, a medida que practiquen Lisp, se sorprenderán de lo poco que en realidad se necesita este tipo de efecto.

Universidad Veracruzana

Verificación interactiva

- Una de las ventajas de la programación funcional es que permite la verificación interactiva.
- En código puramente funcional, se puede verificar cada función a medida que se va escribiendo.
- Si la función regresa los valores que esperamos, se puede confiar en que es correcta. La confianza agregada al proceder de este modo, hace una gran diferencia: un nuevo estilo de programación.

Iteración

- Cuando deseamos programar algo repetitivo, algunas veces la iteración resulta más natural que la recursividad.

```
1  > (defun cuadrados (inicio fin)
2      (do ((i inicio (+ i 1)))
3          ((> i fin) 'final)
4        (format t "~A ~A ~%" i (* i i))))
5  CUADRADOS
6  > (cuadradros 2 5)
7  2 4
8  3 9
9  4 16
10 5 25
11 FINAL
```

La macro Do

- Es el operador básico de iteración en Lisp.
- Como `let`, `do` puede crear variables y su primer argumento es una lista de especificación de variables.
- Cada elemento de esta lista toma la forma (*variable valor-inicial actualización*).
- En cada iteración el valor de las variables definidas de esta forma, cambia como lo especifica la actualización.

La macro do (parada)

- El segundo argumento de `do` debe ser una lista que incluya una o más expresiones.
- La primera expresión se usa como prueba para determinar cuando debe parar la iteración. En el ejemplo, esta prueba es `(> i fin)`.
- El resto de la lista será evaluado en orden cuando la iteración termine. La última expresión evaluada será el valor de `do`, por lo que `cuadrados`, regresa siempre el valor `'final`.

La macro do (cuerpo)

- El resto de los argumentos de `do`, constituyen el cuerpo del ciclo y serán evaluados en orden en cada iteración, donde: las variables son actualizadas, se evalua la prueba de fin de iteración y si esta falla, se evalua el cuerpo de `do`.

La macro loop

- Más compacta y poderosa que do.

```
1  > (defun cuadrados (inicio final)
2      (loop for i from inicio to final finally (return 'final)
3        do (format t "~A ~A ~%" i (* i i))))
4  CUADRADOS
5  > (cuadrados 2 5)
6  2 4
7  3 9
8  4 16
9  5 25
10 FINAL
```

Cuadrados recursiva

- Para comparar, se presenta aquí una versión recursiva de `cuadrados`:

```
1 > (defun cuadrados (inicio fin)
2     (if (> inicio fin)
3         'final
4         (progn
5           (format t "~A ~A ~%"  inicio (* inicio inicio))
6           (cuadrados (+ inicio 1) fin))))
7 CUADRADOS
```

- También hay `time` en Lisp!

Iteración con dolist

- Lisp provee operadores de iteración más sencillos para casos especiales, por ejemplo, `dolist` para iterar sobre los elementos de una lista.
- Ejemplo. Una función que calcula la longitud de una lista:

```
1  > (defun longitud (lst)
2      (let ((len 0))
3        (dolist (obj lst)
4          (setf len (+ len 1)))
5        len))
6  LONGITUD
7  > (longitud '(a b c))
8  3
```

Estructura de dolist

- El primer argumento de `dolist` toma la forma (*variable expresión*), el resto de los argumentos son expresiones que constituyen el cuerpo de `dolist`.
- El cuerpo será evaluado con la `variable` instanciada con elementos sucesivos de la lista que regresa `expresión`.
- La función del ejemplo, dice – por cada `obj` en `lst`, incrementar en uno `len`.

Longitud recursiva

- La versión recursiva `naïf` de esta función longitud es:

```
1 > (defun longitud-rec (lst)
2     (if (null lst)
3         0
4         (+ (longitud-rec (cdr lst)) 1)))
5 LONGITUD
```

- Esta versión será menos eficiente que la iterativa porque no es recursiva a la cola (*tail-recursive*), es decir, al terminar la recursividad, la función debe seguir haciendo algo.

Recursión a la cola

- La definición recursiva a la cola de longitud es:

```
> (defun longitud-tr (lst)
    (labels ((longaux (lst acc)
              (if (null lst)
                  acc
                  (longaux (cdr lst)
                           (+ 1 acc)))))
      (longaux lst 0)))
LONGITUD-TR
```

- Lo nuevo aquí es `labels` que permite definir funciones locales como `longaux`.

Funciones de primera clase

- En Lisp las funciones son objetos regulares como los símbolos, las cadenas y las listas.
- Si le damos a `function` el nombre de una función, nos regresará el objeto asociado a ese nombre.
- Como `quote`, `function` es un operador especial, así que no necesitamos proteger su argumento.

```
1 > (function +)
2 #<Compiled-Function + 17BA4E>
```

Representación interna de funciones

- Este extraño valor corresponde a la forma en que una función sería desplegada en una implementación Lisp.
- Hasta ahora, hemos trabajado con objetos que lucen igual cuando los escribimos y cuando Lisp los evalua.
- Esto no sucede con las funciones, cuya representación interna corresponde más a un segmento de código máquina, que a la forma como la definimos.
- La mejor referencia a las tripas de Lisp es [4].

Abreviar function

- Al igual que usamos ' para abreviar quote, podemos usar #', para abreviar function.

```
1  > #'+
2  #<Compiled-Function + 17BA4E>
```

Funciones como argumentos

- Como sucede con otros objetos, en Lisp podemos pasar funciones como argumentos.
- Ejemplo. Una función que toma una función como argumento es `apply`.

```
1 > (apply #'+ '(1 2 3))
2 6
3 > (+ 1 2 3)
4 6
```

Apply

- Se le puede dar cualquier número de argumentos, si se respeta que el último de ellos sea una lista.

```
1  > (apply #'+ 1 2 '(3 4 5))
2  15
```

Funcall

- La función `funcall` hace lo mismo, pero no necesita que sus argumentos estén empaquetados en forma de lista.

```
> (funcall #'+ 1 2 3)
6
```

Funciones anónimas

- Para referirnos literalmente a un entero usamos una secuencia de dígitos:

```
1  > 120
2  120
```

- Para referirnos literalmente a una función usamos una expresión *lambda* cuyo primer elemento es el símbolo `lambda`, seguido de una lista de parámetros y el cuerpo de la función.

```
1  > (lambda (x y)
2      (+ x y))
3  #<FUNCTION (LAMBDA (X Y)) {22721EDB}>
```

Aplicación de una función anónima

- Una expresión lambda es una función.
- Ejemplo. Puede ser el primer elemento de una llamada de función:

```
1  > ((lambda (x) (+ x 100)) 1)
2  101
```

- o usarse con `funcall`:

```
1  > (funcall #'(lambda (x) (+ x 100)) 1)
2  101
```

- Entre otras cosas, esta notación nos permite usar funciones sin necesidad de nombrarlas.

Universidad Veracruzana

Mapcar

- `Mapcar` aplica una función a los elementos de una lista.

```
1  > (mapcar #'(lambda (x) (* x x))
2            '(1 2 3 4))
3  (1 4 9 16)
```

Tipificación manifiesta

- Lisp utiliza un inusual enfoque flexible sobre tipos.
- En muchos lenguajes, las variables tienen un tipo asociado y no es posible usar una variable sin especificar su tipo.
- En Lisp, los valores tienen un tipo, no las variables.
- Imaginen que cada objeto en Lisp tiene asociada una etiqueta que especifica su tipo.
- Este enfoque se conoce como tipificación manifiesta. No es necesario declarar el tipo de una variable porque cualquier variable puede recibir cualquier objeto de cualquier tipo.
- De cualquier forma, es posible definir el tipo de una variable para optimizar el código antes de la compilación.

Universidad Veracruzana

Jerarquía de tipos

- Lisp incluye una jerarquía predefinida de subtipos y supertipos.
- Un objeto siempre tiene más de un tipo.
- Ejemplo. El número 27 es de tipo `fixnum`, `integer`, `rational`, `real`, `number`, `atom` y `t`, en orden de generalidad incremental.
- El tipo `t` es el supertipo de todo tipo en Lisp.

Typep

- La función typep toma como argumentos un objeto y un especificador de tipo y regresa t si el objeto es de ese tipo.

```
1 > (typep 27 'integer)
2 T
```

Ideas grandes

- Aunque este documento presenta un bosquejo rápido de Lisp, es posible apreciar ya el retrato de un lenguaje de programación inusual.
- Un lenguaje con una sola sintáxis para expresar programas y datos.
- Esta sintaxis se basa en listas, que son a su vez objetos en Lisp.
- Las funciones, que son objetos del lenguaje también, se expresan como listas.
- Y Lisp mismo es un programa Lisp, programado casi por completo con funciones Lisp que en nada difieren a las que podemos definir.

A practicar

- No debe preocuparles que la relación entre todas estas ideas no sea del todo clara.
- Lisp introduce tal cantidad de conceptos nuevos que toma tiempo acostumbrarse a ellos y como usarlos.
- Solo una cosa debe quedar clara: Lisp fue creado para hacer Inteligencia Artificial [3]

Para concluir la introducción

- Si C es el lenguaje para escribir UNIX (Richard Gabriel), entonces podríamos describir a Lisp como el lenguaje para escribir Lisp, pero eso es una historia totalmente diferente.
- Un lenguaje que puede ser escrito en si mismo, es algo distinto de un lenguaje para escribir una clase partícular de aplicaciones.
- Ofrece una nueva forma de programar: así como es posible escribir un programa en el lenguaje, ¡el lenguaje puede mejorarse para acomodarse al programa!
- Un compendio de aplicaciones "contempóraneas" de Lisp se puede ver en [5].

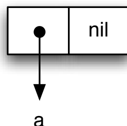
LISP = LISt Processor

- Las listas fueron originalmente la principal estructura de datos en Lisp.
- El nombre del lenguaje es un acrónimo de "LISt Processing".
- Esta sesión muestra qué es lo que uno puede hacer con las listas y las usa para introducir algunos conceptos generales de Lisp.

Cons

- Lo que cons hace es combinar dos objetos, en uno formado de dos partes llamado *cons*.
- Un *cons* es un par de apuntadores: el primero es el *car* y el segundo el *cdr*.

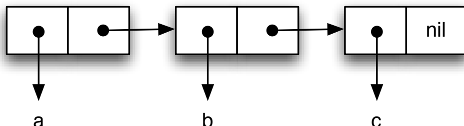
```
1  > (cons 'a nil)
2  (A)
3  > (car '(a))
4  A
5  > (cdr '(a))
6  NIL
```



Listas y Conses

- Cuando construimos una lista con múltiples componentes, el resultado es una cadena de *conses*.

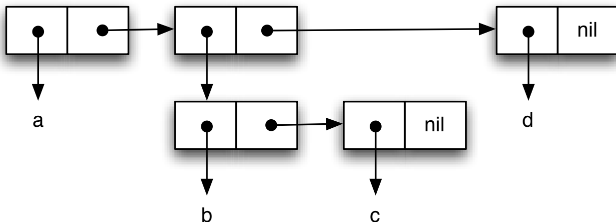
```
1  > (list 'a 'b 'c)
2  (A B C)
3  > (cdr (list 'a 'b 'c))
4  (B C)
```



Listas y Conses anidados

- Esto también aplica a las listas anidadas:

```
1  > (list 'a (list 'b 'c) 'd)
2  (A (B C) D)
```



¿Es lista o átomo?

- La función consp regresa *true* si su argumento es un *cons*, así que podemos definir:

```
1  > (defun mi-listp (x)
2      (or (null x) (consp x)))
3  MI-LISTP
4  > (defun mi-atomp (x)
5      (not (consp x)))
6  MI-ATOMP
7  > (mi-listp '(1 2 3))
8  T
9  > (mi-atomp '(1 2 3))
10 NIL
11 > (mi-listp nil)
12 T
13 > (mi-atomp nil)
14 T
```

Cons e Igualdad

- Cada vez que invocamos a *cons*, Lisp reserva memoria para dos apuntadores.
- Si llamamos a *cons* con el mismo argumento dos veces, Lisp regresa dos valores que aparentemente son el mismo, pero en realidad se trata de diferentes objetos:

```
> (eql (cons 1 nil) (cons 1 nil))
NIL
```

Iguales si tienen los mismos elementos

- La función `mi-eql` regresa *true* cuando dos listas tienen los mismos elementos:

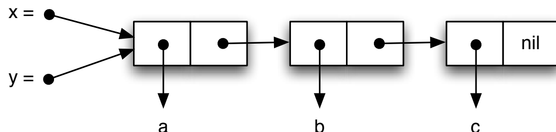
```
1 > (defun mi-eql (lst1 lst2)
2     (or (eql lst1 lst2)
3         (and (consp lst1)
4              (consp lst2)
5              (mi-eql (car lst1) (car lst2))
6              (mi-eql (cdr lst1) (cdr lst2)))))
7 MI-EQL
8 > (mi-eql (cons 1 nil) (cons 1 nil))
9 T
```

- Esta función es equivalente a `equal`.

Variables, Conses y Apuntadores

- Así como los *conses* tienen apuntadores a sus elementos, las variables tienen apuntadores a sus valores.

```
1 > (setf x '(a b c))
2 (A B C)
3 > (setf y x)
4 (A B C)
5 > (eql x y)
6 T
```



Copiando una lista

- La función copia-lista recibe una lista y regresa una copia de ella (en conses diferentes).

```
> (defun copia-lista (lst)
    (if (atom lst)
        lst
        (cons (car lst) (copia-lista (cdr lst)))))
COPIA-LISTA
> (setf x '(a b c)
        y (copia-lista x))
(A B C)
> (eql x y)
NIL
```

Un algoritmo de compresión de datos

- RLE (*run-length encoding*) es un algoritmo de compresión de datos muy sencilla.
- Funciona como los meseros: si los comensales pidieron una tortilla española, otra, otra más y una ensalada verde; el mesero pedirá tres tortillas españolas y una ensalada verde.

El código

- El código de esta estrategia es como sigue:

```
1  (defun rle (lst)
2    (if (consp lst)
3        (compress (car lst) 1 (cdr lst))
4        lst))
5
6  (defun compress (elem n lst)
7    (if (null lst)
8        (list (n-elems elem n))
9        (let ((sig (car lst)))
10         (if (eql sig elem)
11             (compress elem (+ n 1) (cdr lst))
12             (cons (n-elems elem n)
13                   (compress sig 1 (cdr lst)))))))
14
15 (defun n-elems (elem n)
16   (if (> n 1)
17       (list n elem)
18       elem))
```

Llamada a RLE

- Una llamada a *rle*/1 sería como sigue:

```
1  > (rle '(1 1 1 0 1 0 0 0 0 1))
2  ((3 1) 0 1 (4 0) 1)
```

- Programen una función inversa a *rle*: dada una lista que es un código *rle*, regrese la cadena original.
- Observen que este método de comprensión no tiene perdida de información. Prueben su solución con la ayuda de un generador de listas de *n* elementos aleatorios.

Biblioteca básica de funciones sobre listas

```
1  (proclaim '(inline last1 single append1 conc1 mklist))
2  (proclaim '(optimize speed))
3
4  (defun last1 (lst)
5    (car (last lst)))
6
7  (defun single (lst)
8    (and (consp lst) (not (cdr lst))))
9
10 (defun append1 (lst obj)
11   (append lst (list obj)))
12
13 (defun conc1 (lst obj)
14   (nconc lst (list obj)))
15
16 (defun mklist (obj)
17   (if (listp obj) obj (list obj)))
```

last1

- La función `last1` regresa el último elemento de una lista.
- La función predefinida `last` regresa el último *cons* de una lista, no su último elemento.
- `last1` no lleva a cabo ningún chequeo de error:

```
1 > (last1 "prueba")
2 value "prueba" is not of the expected type LIST...
```

- Cuando las utilidades son tan pequeñas, forman una capa de abstracción tan delgada, que son transparentes.

single

- La función `single` prueba si algo es una lista de un elemento. Los programas Lisp necesitan hacer esta prueba bastantes veces. Al principio, uno está tentado a utilizar la traducción natural del español a Lisp:

```
1  (= (length lst) 1)
```

- pero escrita de esta forma, la función sería muy ineficiente.

append1 y conc1

- Las funciones `append1` y `conc1` agregan un elemento al final de una lista, `conc1` de manera destructiva.
- Estas funciones son pequeñas, pero se usan tantas veces que vale la pena incluirlas en la librería.
- De hecho, `append1` ha sido predefinida en muchos dialectos Lisp (pero no en Lispworks).

mklist

- La función `mklst` nos asegura que su argumento sea una lista.
- Muchas funciones Lisp están escritas para regresar una lista o un elemento. Supongamos que `lookup` es una de estas funciones. Si queremos colectar el resultado de aplicar esta función a todos los miembros de una lista, podemos escribir:

```
1  (mapcar #'(lambda (d) (mklist (lookup d)))
2          data)
```

Longer y filter

```
1  (defun longer (x y)
2    (labels ((compare (x y)
3                (and (consp x)
4                     (or (null y)
5                         (compare (cdr x) (cdr y))))))
6      (if (and (listp x) (listp y))
7          (compare x y)
8          (> (length x) (length y)))))
9
10 (defun filter (fn lst)
11   (let ((acc nil))
12     (dolist (x lst)
13       (let ((val (funcall fn x)))
14         (when val (push val acc))))
15     (nreverse acc)))
```

Group

```
1  (defun group (src n)
2    (if (zerop n) (error "Los grupos son de longitud 0")
3        (labels ((rec (src acc)
4                     (let ((rest (nthcdr n src)))
5                       (if (consp rest)
6                           (rec rest (cons (subseq src 0 n) acc))
7                           (nreverse (cons src acc))))))
8          (if src (rec src nil) nil))))
```

Corridas

```
1   > (filter #'null '(nil t nil t 5 6))
2   (T T)
3   > (filter #'(lambda (x)
4                   (when (> x 0) x))
5             '(-1 2 -3 4 -5 6))
6   (2 4 6)
7   > (filter #'(lambda (x)
8                   (if (numberp x) (1+ x)))
9             '(a 1 2 b 3 c d 4))
10  (2 3 4 5)
11  > (group '(a b c d e f g) 2)
12  ((A B) (C D) (E F) (G))
```

Versiones más cortas

- filter con la ayuda de loop:

```
1  (defun filter2 (fn lst)
2    (loop for elt in lst if (funcall fn elt) collect elt))
```

Funciones doblemente recursivas

```
1  (defun flatten (lst)
2    (labels ((rec (lst acc)
3                  (cond ((null lst) acc)
4                        ((atom lst) (cons lst acc))
5                        (t (rec (car lst) (rec (cdr lst) acc))))))
6      (rec lst nil)))
7
8  (defun prune (test tree)
9    (labels ((rec (tree acc)
10                 (cond ((null tree) (nreverse acc))
11                       ((consp (car tree))
12                          (rec (cdr tree)
13                               (cons (rec (car tree) nil) acc)))
14                       (t (rec (cdr tree)
15                               (if (funcall test (car tree))
16                                   acc
17                                   (cons (car tree) acc)))))))
18     (rec tree nil)))
```

Corridas

- La primera de ellas, `flatten` regresa la lista de átomos que son elementos de su lista argumento:

```
1  > (flatten '(a (b c) ((d e) f)))
2  (A B C D E F)
```

- La segunda función, `prune` remueve de una lista todo átomo que satisface el predicado `test`, de forma que:

```
1  > (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
2  (1 (3 (5)) 7 (9))
```

Nuestros propios mapeos

```
1  (defun map0-n (fn n)
2    (mapa-b fn 0 n))
3
4  (defun map1-n (fn n)
5    (mapa-b fn 1 n))
6
7  (defun mapa-b (fn a b &optional (step 1))
8    (do ((i a (+ i step))
9         (result nil))
10       ((> i b) (nreverse result))
11     (push (funcall fn i) result)))
12
13 (defun map-> (fn start test-fn succ-fn)
14   (do ((i start (funcall succ-fn i))
15        (result nil))
16       ((funcall test-fn i) (nreverse result))
17     (push (funcall fn i) result)))
```

Universidad Veracruzana

Corridas

```
1  > (map0-n #'1+ 5)
2  (1 2 3 4 5 6)
3  > (map1-n #'1+ 5)
4  (2 3 4 5 6)
5  > (mapa-b #'1+ 1 4 0.5)
6  (2 2.5 3.0 3.5 4.0 4.5 5.0)
7  >  (map-> #'(lambda(x) x) 0 #'(lambda(x) (> x 10)) #'1+)
8  (0 1 2 3 4 5 6 7 8 9 10)
```

Más mapeos

```
1  (defun mapcars (fn &rest lsts)
2    (let ((result nil))
3      (dolist (lst lsts)
4        (dolist (obj lst)
5          (push (funcall fn obj) result)))
6      (nreverse result)))
7
8  (defun rmapcar (fn &rest args)
9    (if (some #'atom args)
10       (apply fn args)
11       (apply #'mapcar
12              #'(lambda (&rest args)
13                  (apply #'rmapcar fn args))
14              args)))
```

Corridas

```
1  > (mapcars #'1+ '(1 2 3) '(4 5 6))
2  (2 3 4 5 6 7)
3  > (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
4  123456789
5  (1 2 (3 4 (5) 6) 7 (8 9))
6  > (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
7  (11 (22 (33) 44))
```

mapa-b en términos de map->

```
1  (defun mi-mapa-b (fn a b &optional (step 1))
2    (map-> fn
3           a
4           #'(lambda(x) (> x b))
5           #'(lambda(x) (+ x step))))
```

Programas que escriben programas

- Una macro es esencialmente el de una función que genera código Lisp –Un programa que genera programas.
- Una función produce un resultado, pero una macro produce una expresión que al ser evaluada produce un resultado.

Ejemplo: nil!

- La macro `nil!` asigna a su argumento el valor `nil`.
- Queremos que `(nil! x)` tenga el mismo efecto que `(setq x nil)`

```
1  > (defmacro nil! (var)
2        (list 'setq var nil))
3  NIL!
4  > (nil! x)
5  NIL
6  > x
7  NIL
```

- ¿Cómo evalúa Lisp la llamada en la línea 4?

Evaluación de una macro

- Lisp identifica a `nil!` como una macro y la evalúa en dos pasos:

  Macro expansión. Lisp construye la expresión especificada por la definición de la macro: `(setq x nil)`; y entonces

  Evaluación. Se evalúa la expresión obtenida en el paso anterior

Deteniendo la evaluación sólo en algunas partes

- El backquote es una versión especial de nuestro conocido `quote`, que puede ser usado para definir "moldes" de expresiones Lisp.

```
1 > (list 'a 'b 'c)
2 (A B C)
3 > `(a b c)
4 (A B C)
5 > (setf a 1 b 2 c 3)
6 3
7 > `(a (,b c))
8 (A (2 C))
```

Ventajas de backquote

- Las expresiones son más fáciles de leer, debido a su similitud con su macro-expansión.
- Vean las definiciones de `nil!`

```
1 (defmacro nil! (var)
2   (list 'setq var nil))
3 (defmacro nil! (var)
4   `(setq ,var nil))
```

Un ejemplo más complejo: nif

- Queremos que `nif` regrese la etiqueta correspondiente si su argumento es un número (p)ositivo, (c)ero o (n)egativo:

```
1  > (mapcar #'(lambda(x)
2                  (nif x 'p 'c 'n))
3              '(0 2.5 -8))
4  (C P N)
```

- Aunque observen que el usuario define las etiquetas en la llamada (línea 2).

nif con apóstrofo

```
1  (defmacro nif (expr pos zero neg)
2    `(case (truncate (signum ,expr))
3       (1 ,pos)
4       (0 ,zero)
5       (-1 ,neg)))
```

nif sin apóstrofo

```
1 (defmacro nif (expr pos zero neg)
2    (list 'case
3       (list 'truncate (list 'signum expr))
4       (list 1 pos)
5       (list 0 zero)
6       (list -1 neg)))
```

Coma-arroba

- Funciona como la coma normal, pero inserta su valor removiendo los paréntesis más externos:

```
1 > (setq b '(1 2 3))
2 (1 2 3)
3 > `(a ,b c)
4 (A (1 2 3) C)
5 > `(a ,@b c)
6 (A 1 2 3 C)
```

Universidad Veracruzana

mi-when

- Definición:

```
1 (defmacro mi-when (test &body body)
2   '(if ,test
3      (progn ,@body)))
```

- El parámetro `&body` toma un número arbitrario de argumentos y el operador coma-arroba los inserta en un sólo `progn`.
- En la práctica es igual a `&rest`, pero resalta el uso pretendido del argumento.

Definiendo macros (truco)

- Se comienza con la llamada a la macro que queremos definir.
- Escribirla en un papel y abajo escriban la expresión que quieren producir con la macro:

```
1  (mem-eq obj lst)
2
3
4
5  (member obj lst :test #'eq)
```

Parámetros de la macro

- La llamada nos sirve para definir los parámetros de la macro.
- En este caso, como necesitamos dos argumentos, el inicio de la macro será:

```
1  (defmacro mem-eq (obj lst)
2
3
4
5  (member obj lst :test #'eq)
```

Cuerpo de la macro

- Para cada argumento en la llamada a la macro, tracen un línea hacía donde son insertadas en la expansión.
- Comiencen el cuerpo de la macro con un apóstrofo invertido.
- Ahora lean la expansión expresión por expresión:
    1. Si no hay una línea conectado la expresión en la llamada, entonces escribir la expresión tal cual en el cuerpo de la macro.
    2. Si hay una conexión, escriban la expresión precedida por una coma.

```
1  (defmacro mem-eq (obj lst)
2    `(member ,obj ,lst :test #'eq))
```

Macros de aridad indeterminada

- Hay que recurrir a coma-arroba:

```
1  (defmacro while (test &body body)
2    `(do ()
3       ((not ,test))
4     ,@body))
```

- y obtenemos una nueva estructura de control

```
1        > (let ((i 0))
2             (while (< i 10)
3                           (princ i)
4                           (setf i (1+ i))))
5        0123456789
6        NIL
```

Macro expansiones

- Para macros más complejas, es necesario poder observar si la expansión ha sido correcta.
- Para ello lisp provee dos funciones predefinidas:
  - `macroexpand` muestra como la macro se expandiría antes de ser evaluada. Si la macro hace uso de otras macros, esta revisión de la expansión es de poca utilidad.
  - La función `macroexpand-1` muestra la expansión de un sólo paso.

Ejemplos de macro expansión

```
> (pprint (macroexpand
            '(while (puedas) (rie))))
(BLOCK NIL
  (LET ()
    (DECLARE (IGNORABLE))
    (DECLARE)
    (TAGBODY
     #:G747  (WHEN (NOT (PUEDAS)) (RETURN-FROM NIL NIL))
             (RIE)
             (GO #:G747))))

> (pprint (macroexpand-1
            '(while (puedas) (rie))))
(DO () ((NOT (PUEDAS)))  (RIE))
```

Una meta macro

- Si vamos a hacer esto muchas veces, nos convine definir una macro:

```
1 (defmacro mac (expr)
2   `(pprint (macroexpand-1 ',expr)))
```

de forma que podemos evaluar ahora:

```
1 > (mac (while (puedas) rie))
2 (DO () ((NOT (PUEDAS)))  RIE)
3 ; Not value
```

Evaluando la expansión

- La expansión obtenida puede evaluarse en el `TOP-LEVEL` para experimentar con la macro:

```
1 > (setq aux (macroexpand-1 '(mem-eq 'a '(a b c))))
2 (MEMBER 'A '(A B C) :TEST #'EQ)
3 > (eval aux)
4 (A B C)
```

for, in y random-choice

```
1   (defmacro for (var start stop &body body)
2     (let ((gstop (gensym)))
3       `(do ((,var ,start (1+ ,var))
4             (,gstop ,stop))
5            ((> ,var ,gstop))
6          ,@body)))
7
8   (defmacro in (obj &rest choices)
9     (let ((insym (gensym)))
10      `(let ((,insym ,obj))
11         (or ,@(mapcar #'(lambda(c) `(eql ,insym ,c))
12                       choices)))))
13
14  (defmacro random-choice (&rest exprs)
15    `(case (random ,(length exprs))
16       ,@(let ((key -1))
17           (mapcar #'(lambda(expr)
18                       `(,(incf key) ,expr))
19                   exprs))))
```

Corridas

```
1  > (for x 1 8 (princ x))
2  12345678
3  NIL
4  > (in 3 1 2 3)
5  T
6  > (random-choice 1 2 3)
7  1
8  > (random-choice 1 2 3)
9  3
10 > (random-choice 1 2 3)
11 3
12 > (random-choice 1 2 3)
13 1
14 > (random-choice 1 2 3)
15 1
16 > (random-choice 1 2 3)
17 2
```

Universidad Veracruzana

macro expansiones

```
1  > (mac (for x 1 9 (princ x)))
2  (DO ((X 1 (1+ X))
3       (#:G1009 9))
4      ((> X #:G1009))  (PRINC X))
5  ; No value
6  > (mac (in 3 1 2 3))
7  (LET ((#:G1010 3)) (OR (EQL #:G1010 1)
8                         (EQL #:G1010 2)
9                         (EQL #:G1010 3)))
10 ; No value
11 > (mac (random-choice 1 2 3))
12 (CASE (RANDOM 3) (0 1) (1 2) (2 3))
13 ; No value
```

Referencias I

[1] P Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall International, 1993.

[2] G Kiczales, J des Rivières y DG Bobrow. *The Art of Metaobject Protocol*. Cambridge, MA, USA: The MIT Press, 1991.

[3] P Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Burlington, MA, USA: Morgan Kauffman Publishers, 1992.

[4] C Queinnec. *Lisp in Small Pieces*. Cambridge, UK: Cambridge University Press, 1996.

[5] P Seibel. *Practical Common Lisp*. New York, NY, USA: Apress, 2005.