LISP

César Ignacio García Osorio

Área de Lenguajes y Sistemas Informáticos

Universidad de Burgos

Febrero, 1999

El manejo de símbolos

- En todo programa de Inteligencia Artificial la manipulación de símbolos es una parte fundamental. Es por tanto conveniente el uso de un lenguaje que facilite dicha manipulación. LISP es un lenguaje muy adecuado para la manipulación simbólica que se requiere. En LISP:
  - Los elementos fundamentales son objetos semejantes a palabras y se denominan **átomos**.
  - Los grupos de átomos forman objetos parecidos a las oraciones y se denominan **listas**. Las listas pueden agruparse para formar a su vez listas de nivel superior.
  - En conjunto, los átomos y las listas se denominan **expresiones simbólicas**, o simplemente **expresiones**. El manejo de símbolos mediante LISP implica trabajar con expresiones simbólicas.
- Un programa que maneja símbolos se vale de expresiones simbólicas para trabajar con datos y procedimientos, igual que las personas emplean lápiz y papel y algún lenguaje para hacer lo mismo. Un programa que maneja símbolos, suele tener procedimientos para reconocer expresiones simbólicas particulares, separar elementos de ellas y formar nuevas expresiones.

Ejemplos de expresiones simbólicas

```
(arco (partes dintel poste1 poste2)
     (dintel debe-estar-sostenido-por poste1)
     (dintel debe-estar-sostenido-por poste2)
     (dintel es-un-tipo-de cuña)
     (poste1 es-un-tipo-de ladrillo)
     (poste2 es-un-tipo-de ladrillo)
     (poste2 no-debe-tocar poste1))
```

```
(mit (un-tipo-de universidad)
   (localidad (cambridge massachusetts))
   (telefono 253)
   (facultades (arquitectura
               administracion
               ingenieria
               humanidades
               ciencias))
   (fundador (william barton rogers)))
```

```
(identifica6
   ((? animal) tiene dientes puntiagudos)
   ((? animal) tiene garras)
   ((? animal) tiene ojos al-frente)
   ((? animal) es carnívoro))
```


Motivación

- La popularidad de LISP como lenguaje de IA se debe entre otros motivos a:
  - Es un lenguaje para la manipulación de símbolos.
  - Inicialmente era un lenguaje interpretado lo que suponía gran rapidez de desarrollo.
  - Es un lenguaje uniforme. Los programas y los datos tienen una misma estructura: **la lista**.
  - Existen excelentes compiladores que generan código muy eficiente
  - Existen entornos sofisticados de programación construidos sobre LISP.
- Otro lenguaje que se usa con frecuencia en IA es PROLOG (sobre en todo en Europa). Además, entrando en el mundo de la programación convencional, algunos sistemas de IA se están escribiendo en lenguajes de propósito general (C y C++). Lo anterior se debe en parte a que los programas de IA están dejando de ser sistemas aislados; en lugar de eso, forman parte de sistemas más grandes, en los que se incluyen programas convencionales y bases de datos de diversas clases.

Evolución

- LISP es, con sus más de 35 años, uno de los lenguajes de programación más antiguos que sigue en uso, sobreviviendo a los importantes cambios sufridos por la informática.
- LISP fue desarrollado inicialmente, en el MIT, por John McCarthy y sus alumnos basándose en el artículo "Recursive functions of symbolic expressions and their computation by machine" (1958) de John McCarthy.
- LISP es un lenguaje matemático formal y, con extensiones, un excelente lenguaje de programación.
  - Como lenguaje matemático basado en la teoría de las funciones recursivas y en el Cálculo Lambda
  - Como lenguaje de programación
    - orientado al procesado de estructuras simbólicas
    - no existen diferencias entre la representación de las estructuras de datos y las de los programas
- Inicialmente, era un lenguaje simple y "pequeño":
  - Funciones para manipular listas, definición de funciones, predicados de igualdad y funciones para "evaluar llamadas a funciones".
  - Recursión y condicional simple como único mecanismo de control.
- Posteriormente se crearon numerosos dialectos de LISP.
  ZetaLISP, MacLISP, InterLISP, LeLISP, Scheme, MultiLISP, QLISP
- Intentos de estandarización:
  - COMMON LISP (84): Descripción de una familia de lenguajes, haciendo hincapié en la portabilidad, estabilidad, eficiencia, potencia y expresividad.
  - CLOS: COMMON LISP Object System.
- En la práctica:
  - COMMON LISP se ha impuesto como estándar.

Características de COMMON LISP

- **Portabilidad**: COMMON LISP excluye aquellas características que no puedan ser implementadas en la gran mayoría de las máquinas. COMMON LISP se diseño para que fuera fácil construir programas lo menos dependientes posibles de la máquina.
- **Consistencia**: muchas implementaciones LISP son internamente inconsistentes en el sentido de que el interprete y el compilador pueden asignar distintas semánticas al mismo programa. Esta diferencia radica fundamentalmente en el hecho de que el interprete considera que todas las variables tienen alcance dinámico dinamically scope.
- **Expresividad**: coge las construcciones más útiles y comprensibles de dialectos de LISP anteriores (MacLISP, InterLISP y otros).
- **Eficiencia**: tiene muchas características diseñadas para facilitar la producción de código compilado de alta calidad.
- **Potencia**: suministradas por multitud de paquetes que corren sobre el núcleo de COMMON LISP.
- **Estabilidad**: Se pretende que COMMON LISP cambie lentamente y sólo cuando el grupo de expertos encargados del estándar así lo decidan por mutuo acuerdo después de examinar y experimentar con las nuevas características.

Programas de IA en LISP (1)

- En la actualidad, hay un creciente desarrollo de programas que presentan lo que la mayoría considera comportamiento inteligente. Muchos de ellos escrito en LISP. Algunos ejemplos:
  - Sistemas expertos:
    - Programas para configurar computadoras (R1, XCOM)
    - Diagnostico de infecciones de la sangre (MYCIN)
    - Entender circuitos electrónicos
    - Evaluar formaciones geológicas (PROSPECTOR)
    - Planear inversiones
    - Asignar puertas de embarque en un aeropuerto (ARIS)
    - Planificar fábricas
    - Demostrar teoremas matemáticos (OTTER)
    - Expertos en dinámica no lineal (KAM)

  Todos ellos escritos en LISP
  - Razonamiento con sentido común, gran parte del pensamiento humano parece implicar una pequeña cantidad de razonamiento y una gran cantidad de conocimiento. La representación de conocimiento implica elegir un vocabulario de símbolos y establecer algunos acuerdos en cuanto a cómo disponerlos. Las buenas representaciones hace que las cosas correctas resulten explícitas. LISP es un lenguaje en el cual se realiza la mayoría de la investigaciones acerca de la representación.

Programas de IA en LISP (2)

  - Aprendizaje: Programas que de grandes cantidades de datos obtienen las reglas que los relacionan (ID3 ha producido reglas de identificación en áreas que van desde asesorías sobre créditos hasta diagnóstico de enfermedades). Una vez más, LISP domina en este área.
  - Interfaces en lenguaje natural, utilizados para recuperar información de bases de datos, que de otra forma son difíciles de usar (START)
  - Educación y sistemas de apoyo inteligente. Los programas basados en LISP han empezado a hacer modelos de usuario analizando lo que éste hace. Estos programas usan estos modelos para ajustar o elaborar explicaciones.
  - Lenguajes y visión.
- Pero LISP también en otras áreas no relacionadas con la IA. El lenguaje de ampliación de aplicaciones adoptado por GNU es Scheme un dialecto de LISP . En las aplicaciones de CAD ya se venía usando AutoLISP otro dialecto de LISP. Además LISP es importante en educación en ciencias de la computación, ya que facilita la abstracción de procedimientos y de datos, dando énfasis con ello a dos ideas de suma importancia en la programación. Además LISP resulta adecuado para construir intérpretes y compiladores para una amplia gama de lenguajes (incluyendo el propio LISP).

Mitos de LISP

- No hay lenguaje de programación perfecto y LISP no es la excepción. Muchos de sus defectos originales se han corregido, aunque algunas personas continúan refiriéndolos erróneamente en la actualidad.
  - Mito: LISP es lento al operar con números. En efecto esto fue así en un principio, pero el problema se ha corregido mediante el diseño de eficaces compiladores.
  - Mito: LISP es lento. Inicialmente LISP era un lenguaje exclusivamente interpretado, con lo que las aplicaciones en LISP eran muy lentas. Esto también se ha solucionado con el desarrollo de excelentes compiladores de LISP.
  - Mito: Los programas en LISP son grandes. No es un mito, pero es que LISP permite crear programas los suficientemente grandes como para realizar una gran cantidad de tareas.
  - Mito: Los programas en LISP requieren ordenadores caros. Hace unos cuantos años, para empezar con LISP se necesitaba un millón de dólares (Las máquinas LISP son estaciones de trabajo de gran potencia, programadas enteramente en LISP. El sistema operativo, los programas para uso general, los editores, los compiladores y los intérpretes están todos escritos en LISP, lo que demuestra el poder y la versatilidad de este lenguaje.) Hoy en día estos sistemas tienen un costo inferior a los cien mi dólares, y existen incluso excelentes sistemas con LISP para ordenadores personales.
  - Mito: LISP es difícil de leer y depurar por todos los paréntesis que requiere. El problema de los paréntesis desaparece con el uso de editores adecuados.
  - Mito: LISP es difícil de aprender.

Un aperitivo

- Veamos las respuestas del interprete para las siguiente secuencia de entradas:

```
>(+ 3.14 2.71)
5.85
>(setf abiertos '(c d e f))
(c d e f)
>abiertos
(c d e f)
>(setf cerrados '(b a))
(a b)
>(setf abiertos (remove 'c abiertos))
(d e f)
>abiertos
(d e f)
>(setf cerrados (cons 'c cerrados))
(c b a)
>cerrados
(c b a)


>(defun mueve-nodo(nodo)
     (setf abiertos (remove nodo abiertos))
     (setf cerrados (cons nodo cerrados)))
mueve-nodo
>(mueve-nodo d)
(d c b a)
>abiertos
(e f)
>cerrados
(d c b a)
```


Juego de caracteres (1)

- **Caracteres no estándar**. Dependiendo de la implementación puede haber caracteres como:

  `#\HomeUp` `#\Escape` `#\Break ...`
- **Caracteres semi-estándar**:

`#\backsapce`	(BS, 010)	`#\tab`	(HT,011)
`#\linefeed`	(LF, 012)	`#\page`	(FF, 014)
`#\return`	(CR, 015)	`#\rubout`	(DEL, 177)
- **Caracteres estándar**:
  - **`#\space #\newline`**
  - **Alfabéticos**: Letras y dígitos:

    a, b, … z, A, B, … Z, 0, 1, … 9
  - **Pseudo-alfabéticos**:

    `+ - * / @ $ % ^ & _ \ < > ~ ! . { } [ ]|`

    Pseudo-alfabéticos cuyo uso se debe evitar:

    `? ! { } [ ] ^ _ ~`

Juego de caracteres (2)

- **Caracteres estándar**:
  - **Especiales**:

    `( ) ' ; " \ | # ` , :`

    (Un paréntesis izquierdo indica el comienzo de una lista de objetos. La lista puede contener un número indefinido de objetos e incluso ninguno. Las listas pueden estar anidadas. Por ejemplo:

    `(car (quote (1 2 3))) (cons (car x)(cdr y))`

) Un paréntesis derecho termina una lista de objetos.

    ' Un apóstrofo, también llamado "quote", seguido de un objeto, es la abreviatura de la función (`quote` objeto), así pues `'hola` es la abreviatura de `(quote hola)` y `'(1 2 3)` de `(quote (1 2 3))`.

    ; El punto y coma es el carácter que indica el principio de un comentario, a partir de su aparición, el interprete ignora todos los caracteres hasta el final de línea.

    " Las doble comillas sirve para delimitar las cadenas de caracteres.

    **** El backslash es un carácter de escape. Sintácticamente, provoca que el siguiente carácter se considere alfabético. Por ejemplo, `A\(B` es un símbolo de tres caracteres: `A`, `(`, `B`, y `"\""` representa una cadena de caracteres cuyo único carácter es la doble comilla.

Juego de caracteres (3)

- **Caracteres estándar**:
  - **Especiales**:

| La barra vertical se usa en pares para rodear el nombre (o parte del nombre) de un símbolo que este formado o por varios caracteres especiales. Es totalmente equivalente a poner un backslash delante de todos los caracteres entre las barras. Por ejemplo, `|A(B)|`, `A|(|B|)|` y `A\(B\)` representan todos un símbolo cuyo nombre esta compuesto por cuatro caracteres: A, (, B,).

El signo de numeral señala el inicio de una estructura sintáctica compleja:

`#o105` (el número octal 105),

`#b1011` (el número binario 1011),

`#x10A` (el número hexadecimal 10A),

`#\L` (el carácter L),

`#(a b c)` (un vector de tres elementos),

`#'fn` (abreviatura de `(function fn)`.

` El backquote inhibe la evaluación del objeto que le sigue indicando que es un patrón que puede contener comas. Los elementos precedidos por una coma si que se evalúan. Es útil para crear listas dentro de una macro. Por ejemplo:

```
`(if (\verb|>=| ,valor ,umbral) ,accion)
```

, La coma se usa dentro de la sintaxis del backquote.

: Los dos puntos se usan para indicar el paquete al que pertenece un símbolo. Por ejemplo: `network:reset` es el símbolo reset del paquete network. Un doble punto al comienzo de un símbolo indica una palabra clave, un símbolo que siempre se evalúa a si mismo.

Algunos conceptos (1)

- Cuando un paréntesis izquierdo y uno derecho rodean algo, decimos que es una **lista** y hablamos de sus **elementos**. Por ejemplo, la lista **`(+ 3.14 2.71)`** tiene tres elementos, **`+`**, **`3.14`** y **`2.71`**.
- Un **procedimiento** es una especificación detallada de cómo hacer algo expresada en un lenguaje de programación.
- Un procedimiento proporcionado por LISP, como `+`, se llama una **primitiva**.
- Un procedimiento proporcionado por le programador en términos de las primitivas de LISP se denomina un **procedimiento definido por el usuario**.
- Un **programa** es un conjunto de procedimientos que trabajan juntos.

Algunos conceptos (2)

- Los elementos indivisibles que tienen un significado propio, como **`27`**, **`3.14`** y **`+`**, y elementos como `HOLA`, `B27` y `SIMBOLO-CON-GUIONES` se denominan **átomos**.
- Los átomos como `27` y `3.14` se denominan **átomo numéricos**, o simplemente **números**.
- Los átomos como `HOLA`, `B27`, `SIMBOLO-CON-GUIONES`, `FIRST` y `+` se denominan **átomos simbólicos** o sencillamente **símbolos**.
- Una **lista** consta de un paréntesis izquierdo, seguida por cero o más átomos o listas, y un paréntesis derecho.
- Los átomos y las listas se denomina **expresiones simbólicas** o simplemente **expresiones**.
- Cuando se está interesado en el valor de una expresión, se acostumbra referirse a ella como una **forma**. Si esta forma es una lista, el primer elemento es el nombre del procedimiento que se empleará para producir el valor. El proceso de calcular el valor de una forma se denomina **evaluación**.
- Se dice que el procedimiento especificado en una forma es **aplicado** o **llamado** y el resultado es el **valor de la forma** o el **valor devuelto** por el procedimiento.

PRIMITIVAS BÁSICAS

Manipulación de listas

- Acceso básico a los elementos de una lista:
  - **first** lista => elto
  - **rest** lista => resto
  - Y abreviaturas: **second**, **third**, ..., **tenth**
  - **car**, **cdr** y abreviaturas: **cadr**, **cddr**, ..., **cddddr**
  - **nth** n lista => elemento

Ejercicios

- Otras funciones de acceso:
  - **nthcdr** n lista => sub-lista
  - **butlast** lista **&optional** n => lista-truncada
  - **last** lista => lista-truncada

Ejercicios

- Constructores de listas:
  - **cons** objeto lista => nueva-lista
  - **push** objeto lista => nueva-lista
  - **pop** lista => objeto (lista modificada)
  - **append &rest** listas => lista
  - **list &rest** objetos => lista

Ejercicios

- Otras funciones de lista:
  - **length** lista => número-elementos
  - **reverse** lista => lista-al-reves
  - **subst** nuevo viejo lista => nueva-lista

Ejercicios

- Listas de asociación: lista compuesta por pares de elementos
  - **assoc** clave a-list **&key :test** pred => elemento

Ejercicios

Funciones numéricas

- Comparación de números
  - **max** número **&rest** mas-números => número-mayor
  - **min** número **&rest** mas-números => número-menor
- Funciones aritméticas
  - **+ &rest** números => suma
  - **-** número **&rest** mas-números => diferencia
  - *** &rest** números => producto
  - **/** número **&rest** mas-números => cociente

Ejercicios

- Funciones irracionales y transcendentes
  - **exp** potencia => número (devuelve e elevado a potencia)
  - **expt** base potencia => número (devuelve base elevado a potencia)
  - **log** número **&optional** base => número (devuelve el logaritmo de número en base, e por defecto)
  - **sqrt** número => número (devuelve la raíz cuadrada de número)
  - **abs** número => número (devuelve el valor absoluto de número)
  - **sin** número => número (devuelve el seno en radianes)
  - **cos** número => número (devuelve el coseno en radianes)
  - **tan** número => número (devuelve la tangente en radianes)
  - **atan** y **&optional** x => número (devuelve el arcotangente de y/x, por defecto y, en radianes)

PREDICADOS

Predicados (1)

- Predicados de identificación de objetos
  - **null** objeto => T si objeto es **nil** (**endp** lista => T si lista es **nil**)
  - **symbolp** objeto => T si objeto es de tipo symbol
  - **atom** objeto => T si objeto no es de tipo cons
  - **listp** objeto => T si objeto es de tipo list
  - **numberp** objeto => T si objeto es cualquier tipo de número
- Predicados numéricos

Ejercicios

  - **zerop** número => T si número vale cero
  - **plusp** objeto => T si número es estrictamente mayor que cero
  - **minusp** objeto => T si número es estrictamente menor que cero
  - **=** número **&rest** mas-números => T si todos los argumentos tienen el mismo valor
  - **/=** número **&rest** mas-números => T si todos los argumentos tienen distinto valor
  - **<** número **&rest** mas-números => T si la secuencia de argumentos es monótona creciente
  - **>** número **&rest** mas-números => T si la secuencia de argumentos es monótona decreciente
  - **<=** número **&rest** mas-números => T si la secuencia de argumentos es creciente
  - **>=** número **&rest** mas-números => T si la secuencia de argumentos es decreciente

Predicados (2)

- Predicados de igualdad
  - **equal** objeto1 objeto2 => T si objeto1 y objeto2 son isomórficos (de igual tipo y estructura). En la mayoría de los casos, dos objetos son equal si tienen la misma representación impresa.
  - **eq** objeto1 objeto2 => T si objeto1 y objeto2 son el mismo objeto, se limita a comparar apuntadores
  - **eql** objeto1 objeto2 => T si objeto1 y objeto2 son **eq** o si son números del mismo tipo y valor
- La relación entre estos predicados de igualdad es la siguiente
  - EQUAL primero verifica si sus argumentos satisfacen EQL. Si no lo hace trata de verificar si son listas cuyos elementos satisfacen EQUAL
  - EQL primero verifica si sus argumentos satisfacen EQ. Si no lo hacen trata de ver si son números del mismo tipo y con igual valor.
  - EQ verifica que sus argumentos estén representados en las mismas localidades de memoria, es decir que sean símbolos idénticos.
  - = verifica que sus argumentos representen el mismo número, aún cuando no sean del mismo tipo numérico

Ejercicios

- Otros predicados de igualdad son:
  - **neq** objeto1 objeto2 => T si objeto1 y objeto2 no son el mismo objeto.
  - **neql** objeto1 objeto2 => T si objeto1 y objeto2 no son ni **eq** ni números del mismo tipo y valor
  - **member** item lista **&key :test** => Este predicado es cierto si lista contiene un elemento que satisface **:test** con item, en este caso devuelve el final de lista comenzando por item. Por defecto **:test** es **eql**

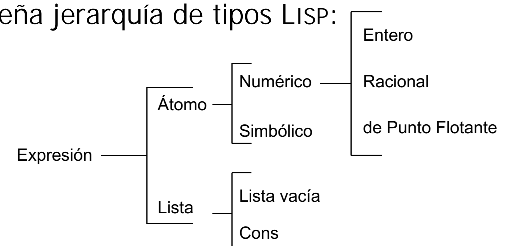
Funciones lógicas

- Operadores lógicos
  - **not** objeto => T si objeto es **nil**; **nil** en caso contrario
  - **and** {forma}* => **nil**/valor-última-foma. **and** evalúa secuencialmente sus argumentos hasta que alguna forma es **nil**, en cuyo caso detiene la evaluación y devuelve **nil**.
  - **or** {forma}* => valor-primera-foma-no-nil/valor-última-foma. **or** evalúa secuencialmente sus argumentos hasta que una forma no es **nil**, en cuyo caso detiene la evaluación y devuelve el valor de esta forma.

LOS TIPOS LISP

Los tipos de LISP

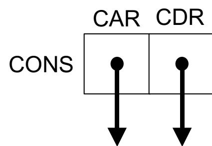
- Las variables/símbolos no tienen tipo: se las pueda asignar como valor cualquier objeto.
- Los que tienen tipo son los objetos no las variables. Cada objeto pertenece como mínimo a un tipo (posiblemente a varios).
- Aunque es posible declarar que una variable tome como valores solo determinados tipos de objetos.
- Los tipos se definen como conjuntos de objetos (posiblemente infinitos).
- Algunos de los tipos definidos en COMMON LISP:
  - **NUMBER**, CHARACTER, **SYMBOL**, **LIST**, **ARRAY**, HASH-TABLE, READTABLE, PACKAGE, **PATHNAME**, **STREAM**, RANDOM-STATE, STRUCTURE, **FUNCTION**
- Los restantes tipos se pueden definit como subtipos o supertipos de los anteriores.
  - Vector: array unidimensional
  - secuence: unión de los tipos vector y list
  - null: subtipo de list y symbol
- Una pequeña jerarquía de tipos LISP:



- Par punteado, celdas cons, listas
- Funciones: symbol-name, symbol-value

El tipo list

- El tipo list se define como la unión de los tipos cons y null.
- Un **cons** (par punteado) es una secuencia de posiciones de memoria, que tiene dos componentes: **car** y **cdr**. Cada componente contiene una dirección de memoria (un apuntador). Gráficamente, lo podemos representar por una caja con dos compartimentos y dos apuntadores.



- Una lista se define recursivamente como:
  - la lista vacía, o
  - cons cuyo cdr es una lista.
- Dicho de otro modo, una lista es una cadena de conses, unidos por su cdr y terminada por un nil. Los car de los conses son los componentes de la lista.

Variables (1)

- Existen dos tipos de variables (visión simplificada):
  - léxicas (estáticas) variables locales
    - definidas por el contexto en el cual están escritas dentro del programa (alcance léxico y extensión indefinida)
  - especiales (dinámicas) variables globales
    - definidas por el contexto en el cual se han creado durante la ejecución del programa (alc. indef. ext. din.)
- Los símbolos **`t`** y **`nil`** no pueden utilizarse como variables o tener valores asignados a ellos.
  - **`t`** se interpreta como CIERTO
  - **`nil`** se interpreta como FALSO (también representa la lista vacía)
- Funciones de acceso para variables:
  - (**`setq`** var1 form1 var2 form2 ...)
    - forms son evaluadas y el resultado es almacenado en las variables vars
    - vars no son evaluadas
    - devuelve el último valor asignado
  - (**`set`** symbol value)
    - modifica el valor de una variable especial
    - no puede modificar el valor de una variable local (asignación léxica)
    - devuelve value

Variables (2)

- Función de actualziación universal: setf
  - (**`setf`** lugar valor-nuevo)
    - Evalúa lugar para acceder a un objeto
    - Almacena el resultado de evaluar valor-nuevo en el lugar apuntado por la función de acceso lugar.
    - setf puede tener el mismo efecto que setq cuando lugar es el nombre de una variable
    - setf va más allá de setq, puede modificar cualquier posición de memoria

Alcance y extensión.

- El **alcance** de un nombre indica la región de texto en un programa lisp donde el nombre se puede referenciar
- La **extensión** de un nombre indica el intervalo de tiempo durante el cual la referencia al nombre accede a una determinada entidad.
- Hay varios tipos de alcance y extensión:
  - **Alcance léxico**: el nombre solo se puede referenciar en la región de programa que lo establece.
  - **Alcance indefinido**: una vez establecido, el nombre se puede referenciar en cualquier lugar del programa
  - **Extensión dinámica**: la relación entre el nombre y la entidad solo se mantiene durante la evaluación de la construcción que lo establece
  - **Extensión indefinida**: la relación entre el nombre y la entidad se mantiene mientras es posible referenciarlo.

Variables (3)

Tipos de variables

- Variables **globales**: cualquier variable que no sea establecida en la lambda-lista de una función o en la región de ligadura de variables de una forma especial (ej: **`let`**) es una variable global con alcance indefinido y extensión dinámica.
  - Las variables globales referencian al último objeto al que han sido ligadas y se pueden referenciar en cualquier parte del programa.
  - Aunque no es necesario, es recomendable declarar las variables globales:
    - (**`defvar`** name [valor-inicial])
    - **`defvar`** hace de nombre una variable global y si nombre no esta ligada, le asigna el valor-inicial, generalmente se usa a nivel de interprete, y por convenio el nombre de la variable global se precede y finaliza por asterisco.
- Variables **localmente ligadas**: una variable esta localmente ligada respecto a una función o forma especial si es dicha forma la que establece la variable. Tienen, alcance léxico y extensión indefinida.
  - Solo podemos referenciar dichas variables dentro del cuerpo de la forma que las establece; las referencias a estas variables acceden a la última entidad que se le halla asignado en dicha forma.

Variables (4)

- Variables **libres**: Una variable es libre respecto a una función o forma especial si la variable no ha sido establecida en dicha forma.
  - El valor de una variable libre es el valor de la ligadura más interna en la región texto donde se la referencia. Si no existe esta ligadura, se usa su valor global: el más recientemente asignado a la variable.
- Variables **especiales**: La combinación de **`declare`** y **`special`** hace que una variable tenga alcance indefinido y extensión dinámica: se puede referenciar en cualquier parte del programa, pero solo mientras se evalúa la forma que la establece.
  - (**`declare`** (**`special`** var))
  - la forma **`declare`** se denomina declaración; solo puede aparecer al comienzo del cuerpo de una función o de algunas formas especiales (ej. **`let`**)
  - las variables globales son variables especiales, pero con la diferencia de que como no las establece ninguna forma especial, las ligaduras permanece durante el período de vida del interprete.

Gestión de memoria

- LISP esta diseñado para liberar al programador de los detalles de gestión de memoria: utiliza una lista de espacio libre, de la que toma memoria para construir cualquier tipo de estructura.
- Inconvenientes: Las celdas de memoria no se pueden devolver a la lista de espacio libre cuando se modifica el valor de un símbolo, pues las celdas de memoria pueden estar compartidas por más de un símbolo (u otras estructuras).
- Para evitar que se agote la memoria del sistema LISP utiliza un mecanismo de liberación de memoria denominado Garbage Collection (recolector de basura), GC). Los GC más sencillo operan en dos fases y son invocados cuando está apunto de agotar la memoria (90 o 95%):
  - 1ª fase: **marcado**. Se trata de marcar todas las posiciones de memoria que usa el sistema. Ello incluye todas las posiciones a las que apuntan los símbolos, definiciones de funciones, cálculos actuales ...
  - 2ª fase: **barrido**. Se recorren secuencialmente todas las posiciones de memoria. Las celdas no marcadas se pasan a la lista de espacio libre y se eliminan las marcas de las posiciones marcadas.

Ciclo read-eval-print

- **Forma**: Expresión-s cuya evaluación no produce error
- **Evaluación de formas**
  - si es un número, devolver el valor del número
  - si es un símbolo, devolver
    - su valor, si el símbolo ha sido ligado previamente
    - error, en caso contrario
  - si es una lista:
    1. considerar que el primer elemento es una función
    2. evaluar los restantes elementos
    3. aplicar la función tomando como argumentos los valores de ②
    4. devolver el valor obtenido en ③
- Las denominadas **formas especiales** alteran el flujo normal de evaluación. La mayoría son estructuras de control.
- Si queremos que algo no se evalúe se usa el **`quote`**

Ciclo read-eval-print

Ciclo read-eval-print.

- El término lector LISP designa a la función read lee una secuencia de caracteres, hasta que el parser decide que dicha secuencia constituye la representación de una expresión-s
  - Lee caracteres del stream de entrada
  - construye una expresión-s
  - devuelve dicha expresión
- La función eval es la responsable de evaluar formas. De hecho, la ejecución de código LISP consiste en la aplicación de la función eval sobre expresiones-s
- La función print imprime una línea en blanco y la representación de un objeto LISP en el stream de salida
- El interprete LISP ejecuta siempre el siguiente ciclo:
  1. read una forma
  2. eval la forma
  3. print el resultado de ②
  4. mostrar el prompt e ir a ①

Ejercicios

- El programa responsable de ejecutar este ciclo se denomina Lisp listener

DEFINICIÓN DE FUNCIONES

Definición de funciones (1)

- (**defun** nombre lambda-lista {declaración}* {forma}*)
defun asocia al símbolo nombre la función cuyo cuerpo es {forma}* y cuya lista de parámetros viene dada por lambda-lista
- La sintaxis de una lambda-lista es:

({**var**}*
  [**&optional** {var|(var [initform [svar]])}*]
  [**&rest** var]
  [**&key** {var|({var|(keyword var)}[initform [svar]])}*]
  [**&aux** {var|(var [initform])}*])

- Para llamar a una función hay que aplicar el nombre sobre sus argumentos (evaluación). La evaluación pasa por las siguientes etapas
  1. evaluar los argumentos
  2. asignar a los parámetros de la lambda-lista los valores de los argumentos
  3. evaluar, secuencialmente, las formas del cuerpo
  4. devolver la última forma evaluada

Definición de funciones (2)

- Una lambda-lista tiene cinco partes:
  - parámetros requeridos: todos los que están antes de la primera clave.
  - parámetros opcionales: entre **&optional** y la siguiente clave
  - parámetro rest: un único parámetro **&rest**
  - parámetros clave: entre **&key** y la siguiente clave
  - variables auxiliares: a partir de **&aux**
- Proceso de los argumentos de aplicación: Ejercicios
  - Si hay n parámetros requeridos, hay que suministrar al menos n argumentos. Los parámetros quedan ligados al valor de los n primeros argumentos.
  - Si se ha incluido **&optional** y hay argumentos sin procesar, los parámetros opcionales quedan ligados a los siguientes argumentos, si no hay argumentos por procesar, los parámetros opcionales se ligan a **nil** o al valor de initform.
  - Si se ha incluido **&rest**, el parámetro rest queda ligado a la lista formada con todos los argumentos que quedan por procesar.
  - Si se ha incluido **&key** las variables se ligan al valor que siga a la clave en la lista de argumentos (que no sean requeridos u opcionales), si no aparece se ligan a **nil** o al valor de initform
  - Si se ha incluido **&aux** los siguientes elementos de la lambda-lista no son en realidad parámetros, sino variables auxiliares que quedan localmente ligadas en la función

Definición de funciones (3) - Funciones anónimas

- Las llamadas expresiones lambda constituyen la forma básica de representar funciones
  (**lambda** lambda-lista {declaración}* {forma}*)
- Una función anónima es aquella expresión **lambda** que permite definir "in situ" un conjunto de acciones que se quieren realizar a partir de una serie de argumentos.
- Tiene que haber el mismo número de parámetros que de argumentos en la llamada a la función
- Los argumentos "se cogen" uno a uno, de izquierda a derecha de los valores devueltos por la evaluación de las expresiones situadas a la derecha de la propia expresión **lambda**.
- Devuelve la evaluación de la última expresión dentro del cuerpo de la función (conjunto de expresión que la formas)
  ((**lambda** lambda-lista {declaración}* {forma}*)
valores sobre los que actúa)

Ejercicios

ESTRUCTURAS DE CONTROL

Estructuras de control (1) Condicionales I

if test forma-then [forma-else] => resultado-última-forma-evaluada

- si test no es **nil**, evalúa y devuelve el valor de forma-then; si test es **nil** evalúa y devuelve forma-else (**nil** por defecto).

when test {forma}* => **nil**/resultado-última-forma

- si test no es **nil**, evalúa cada forma secuencialmente y devuelve el valor de la última forma evaluada, en caso contrario devuelve **nil**.

unless test {forma}* => **nil**/resultado-última-forma

- si test es **nil**, evalúa cada forma secuencialmente y devuelve el valor de la última forma evaluada, en caso contrario devuelve **nil**.

cond {(test {forma}*)}* => **nil**/resultado-última-forma

- Es el condicional básico de Common Lisp.
- Cada (test forma-1 forma-2 ... Forma-n) se denomina cláusula. Los test de las cláusulas se evalúan secuencialmente y se selecciona la primera cláusula cuyo test no sea **nil**. A continuación se evalúan secuencialmente las formas de la cláusula seleccionada y se devuelve el valor de la última cláusula.
- Si todos los test son **nil**, **cond** devuelve **nil**.

Estructuras de control (2) Condicionales II

case forma-clave {({({clave}*)|clave}{forma}*)}*

- Esta forma especial evalúa como máximo una de sus cláusulas, en base a las claves.
- Cada cláusula tienen la forma (especificación-clave forma-1 ... forma-n). **especificación-clave** puede ser una lista de claves (cualquier objeto) **t** u **otherwise** o una única clave. **t** y **otherwise** solo pueden aparecer en la última cláusula.
- En primer lugar se evalúa forma-clave para producir el **selector-clave**.
- A continuación se compara selector-clave con cada una de las especificación-clave, sin evaluar, usando **member** para las listas y **eql** si es una única clave. El orden en que se compara las claves no está definido, salvo que si una cláusula usa **t** u **otherwise** se compara la última. Se selecciona una cláusula que satisfaga el test. Si no hay ninguna y se incluye **t** u **otherwise**, se selecciona esta.
- Por último, se evalúan secuencialmente las formas de la cláusula seleccionada y se devuelve el valor de la última forma (**nil** si no se selecciona ninguna cláusula).

Ejercicios

Estructuras de control (3) Recursión

- Un **procedimiento** es **recursivo** si se define en términos de si mismo. Directa o indirectamente.
- Muchos problemas se resuelven de modo natural en forma recursiva.
- Para examinar el comportamiento de un procedimiento recursivo, es útil simular la cadena de llamadas gráficamente, construyendo un árbol:
  - nodos: número de llamada y nombre.
  - Entrada nodo: argumentos de llamada.
  - Salida nodo: valor devuelto
- **Recursión CDR**: la recursion CDR es un procedimiento utilizado para realizar la misma operación sobre todos los elementos de una lista. El procedimiento genérico es el siguiente:
  - si la lista esta vacía, salir
  - realizar la operación sobre el primer elemento; recurrir sobre el resto
- **Recursión CAR/CDR**: La recursión CAR/CDR se utiliza si hay que recorrer todos los niveles de una lista. El procedimiento genérico es el siguiente:
  - si el argumento es la lista vacía, salir.
  - Si el argumento es un átomo, realizar la operación y salir
  - recurrir sobre le first y el rest de la lista.

Ejercicios

Estructuras de control (4) Ligadura de variables

let ({Var|(Var valor)}*){declaracion}*{forma}* => resultado-última-forma

- liga Var a valor (por defecto **nil**); las ligaduras se realizan en paralelo y las variables tiene alcance léxico y extensión indefinida. A continuación evalúa secuencialmente cada forma.

let* ({Var|(Var valor)}*){declaracion}*{forma}* => resultado-última-forma

- es idéntica a let, excepto que las variables se ligan secuencialmente.

progv lista-símbolos lista-valores {forma}* => resultado-última-forma

- lista-símbolos es una lista de variables especiales. La n-esima variable de lista-símbolos es ligado con el n-esimo valor de lista-valores. A continuación se evalúan secuencialmente las formas del cuerpo.

Ejercicios

Estructuras de control (5) Secuenciamiento

progn {forma}* => resultado-última-forma

- Evalúa secuencialmente cada forma y devuelve el valor de la última

prog1 {forma}* => resultado-primera-forma

- Evalúa secuencialmente cada forma y devuelve el valor de la última

- Estas formas son poco utilizadas, pues la evaluación secuencial de formas esta implícito en casi todas las formas LISP. Pueden ser útiles con **if** porque solo admite una forma.

Estructuras de control (6) Bloques

block nombre {forma}* => resultado-última-forma

- establece un bloque denominado nombre y a continuación evalúa secuencialmente las formas hasta que encuentra un return-from o alcanza la última forma. Nombre tiene alcance léxico y extensión dinámica.

return-from nombre resultado => resultado

- provoca la salida del bloque más reciente denominado nombre devolviendo resultado. Debe estar en el alcance de dicho bloque (nombre).

return resultado => resultado

- provoca la salida del bloque denominado **nil** más reciente, devolviendo resultado . Se usa en algunas formas especiales que establecen bloques **nil**. (do, prog, …)

Ejercicios

Estructuras de control (7) Alteración flujo ctrl

tagbody {tag|forma} * => nil

- El cuerpo de **tagbody** está formado por {tag|forma}. tag es un entero o un símbolo. Cada elemento del cuerpo se evalúa secuencialmente, ignorándose los tags, cuyo único efecto es quedar establecidos. Si se alcanza la última forma, **tagbody** devuelve nil.
- Los tags tienen alcance léxico y extensión dinámica.
- **tagbody** se suele utilizar dentro de un bloque.

go tag

- produce un salto al tag de un **tagbody**

prog ({var|(var [ini])}*){declaración}*{tag|forma}* => nil

- Similar a **let**, pero el cuerpo es un **tagbody** y está dentro de un bloque nil (permite utilizar **go** y **return**)

Estructuras de control (8) Iteración I

dotimes (var forma-contador [forma-result]) {declaración}* {tag|forma}*=> resultado forma-result

- **dotimes** proporciona la forma más sencilla de iteración contador sobre un conjunto de enteros. El cuerpo de **dotimes** ({tag|forma}*) es un bloque nil y un tadbody.
- Inicialmente se evalúa forma-contador, que deber ser un entero **n**. Para cada entero en el rango **0** a (**n-1**), la variable var se liga a ese entero y se evalúa el cuerpo. Finalmente var se desliga y se devuelve el valor de forma-resultado (por defecto nil).

dolist (var forma-lista [forma-result]) {declaración}* {tag|forma}*=> resultado forma-result

- Similar al **dotimes**, pero itera sobre los elementos de forma-lista, que al evaluarse debe producir una lista.

loop {forma}*=> valor de retorno

- **loop** evalúa repetidamente las formas del cuerpo, hasta que alguna forma fuerce la salida. El cuerpo de llop esta dentro de un bloque nil (**return** permite abandonar **loop**).

Ejercicios

Estructuras de control (9) Iteración II

do ({(var [ini [act]])}*) (test {forma-test}*) {declaración}* {tag|forma}* => resultado-ultima-forma-test/nil

- **do** proporciona la forma más general de realizar iteraciones. Toda la estructura **do** está dentro de un bloque nil y el cuerpo es un **tagbody**
- **do** tiene tres partes:
  - indice: ((var$_1$ ini$_1$ act$_1$) ... (var$_n$ ini$_n$ act$_n$))
  - fin: (test forma-test$_1$ ... forma-test$_n$)
  - cuerpo: {tag|forma}*
- En primer lugar, las variables del índice quedan localmente ligadas y se les asigna el valor de ini (por defecto nil). La ligadura y asignación se realiza en paralelo. A continuación comienza la iteración. Cada ciclo realiza los siguientes pasos:
  1) Se evalúa el test de fin. Si el resultado no nil finaliza la iteración. Se evalúan secuencialmente las formas forma-test y se devuelve el valor de la última (nil si no hay ninguna)
  2) Si el valor de test es nil se evalúan secuencialmente las formas del cuerpo.
  3) Cuando se alcanza el fin del cuerpo, se asigna a cada var el valor de act. Si no se proporciona act, var no es actualizada.

do* ({(var [ini [act]])}*) (test ... .. .

- como **do**, pero las variables de índice se ligan secuencialmente

Ejercicios

Edición, documentación y corrección de errores (1)

load pathname

- Carga el archivo pathname en el interprete. En freelisp es más cómodo usar el comando Load... Del menú File

trace {nombre-función}* => último nombre-función/nil

- Provoca el trace de la evaluación de las llamadas a las funciones indicadas (su nombre no se evalúa). Sin argumentos devuelve la lista de funciones sometidas a traceo.

untrace {nombre-función}* =>lista traceo/nil

- Deshace el efecto de trace. Sin argumentos deshace el traceo de todas las funciones.

documentation símbolo tipo-doc => cadena-doc/nil

- Muestra la cadena de documentación asociada a símbolo si no existe dicha cadena devuelve nil.
- tipo-doc debe ser: **function**, **variable** o **type**.

time forma => resultado de evaluar forma

- Retorna el resultado de evaluar forma, pero como efecto lateral imprime diversas informaciones sobre el tiempo de ejecución de forma.

dribble &optional pathname => nil

- Imprime las salidas y entradas en pathname

Ejercicios

Edición, documentación y corrección de errores (2)

- El evaluador de expresiones (listener) ignora cualquier texto que aparezca entre un punto y un carácter de "retorno"
- Se siguen las siguientes convenciones para documentar el código:
  - Las cabeceras, secciones y subsecciones se documentan escribiendo delante del comentario correspondiente la secuencia ;;;;
  - La secuencia ;;; precede partes significativas del programa (secciones, funciones, etc)
  - La secuencia ;; se utiliza para documentar el código que se encuentra debajo del comentario
  - Con ; se documentan las expresiones incluidas en esa misma línea

EVALUACIÓN Y APLICACIÓN DE FUNCIONES

Aplicación (1)

eval form => valor

- Evalúa y devuelve el valor de forma y devuelve el resultado de su evaluación.
- Notar que el argumento de **eval** se evalúa dos veces: una por ser el argumento de una función y otra por tratarse de la función **eval**.

function nombre-funcion => objeto-funcion

- Retorna la función asociada al símbolo nombre-funcion. Como tiene un uso frecuente se puede abreviar como **#'**.

apply funcion arg **&rest** args=> valor

- Aplica la funcion a una lista de argumentos y devuelve el resultado de dicha aplicación. Si funcion es un símbolo, su definición de función no debe ser una macro.
- El último argumento de **apply** debe ser una lista.

funcall funcion **&rest** args=> valor

- Produce la llamada a funcion con argumentos args retornando el resultado de esta llamada.
- Su primer argumento puede ser una función o un símbolo, en este último caso su definición funcional no puede ser una macro.

(**apply** #'fn (**list** arg1 ... argn))≡ (**funcall** #'fn arg1 ... argn)

Ejercicios

Aplicación (2) a listas

- Las que siguen son funciones que aceden iterativamente a los elementos o "CDRs sucesivos" que componente una lista, realizan acciones en función de dicho elemento o "CDR" y devuelven una lista con los resultados de dichas acciones o devuelven la lista original (en este caso lo importante es el resultado de cada acción y no el valor devuelto).

mapcar funcion list **&rest** mas-listas => lista-resultado

- Esta función aplica la funcion a los sucesivos enesimos elementos de las listas (al FIRST de cada lista luego a los SECOND y así sucesivamente) hasta que se alcance el final de la lista más corta
- La funcion debe tener tantos argumentos como listas
- Devuelve una lista con los resultados dados por las sucesivas aplicaciones de funcion

mapcan funcion list **&rest** mas-listas => lista-resultado

- Esta función actúa como **mapcar** pero utiliza la función **nconc** para combinar los resultados de las sucesivas aplicaciones de funcion

(**mapcan** funcion list1 ... listn) ≡
(**apply** #'**nconc** (**mapcar** funcion list1 ... listn))

Aplicación (3) a listas

mapc funcion list **&rest** mas-listas => lista-resultado

- Actúa como **mapcar** salvo que no acumula los resultados de las sucesivas aplicaciones de funcion
- Devuelve su segundo argumento

(**mapc** funcion list1 ... listn) ≡
(**prog1** list1 (**mapcar** funcion list1 ... listn))

maplist funcion list **&rest** mas-listas => lista-resultado

- Esta función aplica la funcion a las n sucesivos sublistas de las listas (primero a cada lista luego a los CDR, a los CDDR y así sucesivamente) hasta que se una de las listas sean NIL
- La funcion debe tomar tantos argumentos de tipo lista como listas le sigan
- Devuelve una lista con los resultados dados por las sucesivas aplicaciones de funcion

mapcon funcion list **&rest** mas-listas => lista-resultado

- Esta función actúa como **maplist** pero utiliza la función **nconc** para combinar los resultados de las sucesivas aplicaciones de funcion

(**mapcon** funcion list1 ... listn) ≡
(**apply** #'**nconc** (**maplist** funcion list1 ... listn))

Ejercicios

Aplicación (4) a listas

mapl funcion list **&rest** mas-listas => lista-resultado

- Actúa como **maplist** salvo que no acumula los resultados de las sucesivas aplicaciones de funcion
- Devuelve su segundo argumento

(**mapl** funcion list1 ... listn) ≡
(**prog1** list1 (**maplist** funcion list1 ... listn))

- Resumen de funciones de transformación de lista

	Opera sobre **CAR**s	Opera sobre **CDR**s
Efectos laterales	MAPC	MAPL
La lista devuelta se construye con **LIST**	MAPCAR	MAPLIST
La lista devuelta se construye con **NCONC**	MAPCAN	MAPCON

Ejercicios

Valores múltiples

- A veces es conveniente que las funciones devuelvan más de un valor, sin que para ello se deba recurrir a su agrupación en una lista

values &rest args => lista-resultado

- Retorna n valores cuando se le dan n argumentos
- La sentencia values al final de una función es la forma estándar de lograr que una función devuelva más de un valor

multiple-value-setq (var1 var2 ...) forma => resultado-forma

- Se asignan sucesivamente los valores devueltos por sentencia a las variables incluidas en la lista de variables
- Si hubiera más variables que valores, a estas se les asigna **nil**
- Si hay mas valores que variables, los valores en exceso no se usan

multiple-value-bind (var1 var2 ...) forma-value {declaración}* {tag|forma}* => valor-ultima-forma

- Funciona de forma análoga al let. Los valores devueltos por forma-value se asignan localmente a las variables incluidas en la lista de variables y luego se ejecutan sucesivamente el resto de las sentencias

Ejercicios

ESTRUCTURAS DE DATOS

Listas de propiedades

- Uno de los componentes básicos de cualquier símbolo en la memoria es el puntero a su lista de propiedades (plist)
- Una lista de propiedades es una celda de memoria que contiene una lista con un número par de elementos (puede ser 0)

get símbolo propiedad => valor-de-la-propiedad/**nil**

- Devuelve el valor asociado a la propiedad en la lista de propiedades del símbolo
- La búsqueda se realiza tomando como base de la comparación el predicado **eq**
- Si el par no existiera se devuelve **nil**

symbol-plist símbolo => lista-de-propiedades/**nil**

- Devuelve la lista de propiedades completa de símbolo

setf (**get** símbolo propiedad) valor) => valor

- **get** accede a la posición donde se encuentra el valor actual de la propiedad del símbolo
- **setf** es la forma genérica de cambiar el contenido de posiciones que apuntan a objetos de LISP

Ejercicios

Estructuras

- Este tipo de datos es semejante a los registros de otros lenguajes de programación
- Las estructuras permiten utilizar una técnica de representación simple y eficiente
- El Common LISP crea automáticamente funciones de acceso y actualización para cada una de las instancias de las estructuras creadas por el usuario
- Las estructuras proporcionan una extensión de la representación utilizada en las listas de propiedades
- La programación basada en estructuras constituye una técnica más depurada que la programación basada en listas

defstruct nombre-estructura nombre-campo-1 nombre-campo-2 nombre-campo-n => nombre-estructura

- Esta macro define un tipo de dato estructura. Todas las funciones de acceso y de creación de instancias de la estructura se crean al mismo tiempo
- Para actualizar los valores de los campos de una estructura se utiliza la función genérica de actualización **setf**

Ejercicios

Matrices y vectores (1)

- Son estructuras de datos con un tamaño prefijado
- La gran ventaja es que facilitan el acceso directo al elemento requerido a partir de su posición
- El tiempo de acceso a un elemento concreto es independiente de su posición (al contrario que en las listas)
- En general se puede afirma que el acceso y la actualización (Dada la forma en como se almacenan internamente) es más rápido con las matrices que con las listas
- En general se puede afirmar que la inserción de un nuevo elemento es más rápida con las listas (dada la flexibilidad de su estructura interna) que con las matrices

Matrices y vectores (2)

make-array dimensiones
  [**:element-type** tipo]
  {[**:initial-element** valor-comun] |
  [**:element-contents** lista-elementos] }
  [**:adjustable** valor]
  => array

- dimensiones es una lista de entero no negativos que representa las dimensiones de la matriz. Para el caso de las matrices de una dimensión (vectores) basta con proporcionar un entero
- [**:element-type** tipo] indica la posibilidad de especificar el tipo de los elementos. Si tipo es **t** entonces puede ser cualquier tipo (es el valor por omisión)
- [**:initial-element** valor-comun] Crea una matriz con todos sus elementos iguales a valor-comun
- [**:element-contents** lista-elementos] lista-elementos es una lista cuya estructura debe corresponderse con el valor dado en dimensiones
- [**:adjustable** valor] si valor es distinto de nil señala que puede ajustarse dinámicamente el tamaño de la matriz (mediante **adjust-array**)
- Si no se le proporciona ningún argumento, los elementos de la matriz se rellenan con **nil**
- A cada una de las "dimensiones" de la matriz se le llama eje y tanto los ejes como los elementos incluidos en cada uno de éstos comienza a numerarse en el 0

Matrices y vectores (3)

array-dimension array número-eje =>dimension

- Devuelve el número asociado a la dimensión del numero-eje especificado

array-rank array =>dimensiones

- Devuelve el número de dimensiones de la matriz

aref array indice1 indice2 ... =>elemento

- Es la función general de acceso a los elementos de una matriz. Retorna el valor del elemento que ocupa la posición indicada por los indices

vector elto1 elto2 ... =>vector

- Crea un vector. Matriz de una sola dimensión

Ejercicios

Otras estructuras de datos

- **Par punteado**
  - Celda cons cuyo cdr no es nil
- **Listas de asociación**
  - Lista compuesta de pares de elementos
  - El car de los elementos de una a-list se llama clave y el cdr se conoce como dato (clave dato)

assoc clave a-list [**:test** pred] =>**nil**/par-de-asociación

MACROS

Macros (1)

- Ventajas de usar macros
  - Hace que el código sea más claro y sencillo de escribir y entender. Permiten definir funciones que convierten cierto código LISP en otro, generalmente más farragoso. Esta conversión se realiza antes de que el código sea evaluado
  - Al igual que en el resto de los lenguajes de programación, las macros se diferencian de las funciones en que a la hora de generar el código compilado las macros "se expanden en línea" (es decir, cada llamada a una macro obliga al compilador a repetir el código de la macro in situ)
  - Ayuda a escribir código modular
  - Son útiles para definir ampliaciones en las sentencias incluidas en el lenguaje
  - Elimina tener que escribir repetidas veces partes complicadas del código a lo largo del programa
- Desventajas:
  - Son difíciles de depurar

Ejercicios

Macros (2)

- En las llamadas a funciones se evalúan todos los argumentos y posteriormente la función se aplica a los resultados de dichas aplicaciones.
- Las macros provocan una doble evaluación
  - En la primera "pasada" se produce una forma o sentencia de LISP.
  - En la segunda se evalúa de nuevo el resultado producido en la primera pasada y se produce el valor correspondiente.

defmacro nombre lambda-lista {sentencias}* =>nombre

- No se evaluan los argumentos
- Puede tener una gran variedad de argumentos; la lambda-lista es semejante a la de las funciones
- No puede haber una macro con el mismo nombre que una función

macroexpand forma =>expansión-macro boolean

- Realiza todas las expansiones de macros necesarias hasta conseguir que no haya una macro sin expandir en **forma**

Ejercicios

Macros (3)

- El carácter de comilla invertida "` " (quote) simplifica la escritura del código que es doblemente evaluado dentro del cuerpo de las macros.
- Se utiliza para realizar una evaluación selectiva del código que aparece después de dicho carácter
  - En la primera "pasada", el código que se encuentra a la derecha del carácter de comilla invertida ("` ") deja de ser evaluado, por tanto su contenido se deja intacto (para ser nuevamente evaluado en el segundo paso si aparece dentro de una macro)
  - Si a la derecha del carácter de comilla invertida ("` ") apareciera el carácter de coma (",") delante de alguna expresión, dicha expresión sí que es evaluada en la primera pasada, y el objeto devuelto en dicha evaluación se introduce en el lugar donde aparecía dicha expresión.
  - Si a la derecha del carácter de comilla invertida ("` ") apareciera la combinación de caracteres de coma (",") seguido de arroba ("@"), es decir: (",@") entonces la expresión que siga a dicha combinación se evalúa y su valor se introduce en lugar de dicha expresión de tal forma que si su valor es una lista de elementos, dichos elementos se introducen como nuevos elementos en la estructura que contuviera a la expresión y no se introduce como tal la lista de elementos

Ejercicios

ENTRADA/ SALIDA

Entrada/Salida (1)

- Los flujos de datos ("streams") permiten leer y escribir datos en un fichero
- Un flujo es un objeto que se utiliza como fuente o sumidero de datos
- El flujo puede ser de catacteres, enteros, binario
- La comunicación del bucle read-eval-print con el usuario se realiza a través de flujos de datos (o canales)
  - Existen variables que contienen los streams estándares que permiten dicha comunicación
    - ***standard-input*** es la fuente de donde se leen los datos del usuario
    - ***standard-output*** es el stream donde se escriben los datos que recibe el usuario
- El usuario también puede crear sus propios flujos de datos

Entrada/Salida (2)

with-open-file (stream nombrefichero {opciones}*) {sentencias}*

- Esta macro se encarga de abrir un flujo de datos con un fichero para permitir realizar acciones con sus contenido a través de las sentencias evaluadas como si hubiera un **progn** implícito
- Tiene la gran ventaja que al terminar de ejecutarse cierra automáticamente el flujo con el fichero correspondiente
- Las opciones posibles se especifican a través de argumentos de tipo clave
  - **:direction** Este argumento especifica si el cana o flujo es de entrada (valor **:input**, es el valor por omisión), de salida (valor **:output**) o bidireccional (valor **:io**)
  - **:element-type** este argumento especifica el tipo de datos intercambiados en la comunicación. Puede ser character (en cuyo caso se utilizarán las funciones read-char y write-char), bit, etc
  - **:if-exists** este argumento especifica la acción que deberá realizarse si la dirección de apertura es **:output** o **:io**. Algunos posibles vlaores son **:new-version**, **:overwrite**, **:append**, **:supersede**
  - **:if-does-not-exist** puede tener dos valroes **:error** o **:create** que indican si se señala un error (por omisión si la **:direction** es **:input**) o si se crea un fichero nuevo con el nombre especificado

Entrada/Salida (3)

read [flujo-entrada [eof-error-p [eof-value]]])
- flujo-entrada debe ser un stream del cual se obtienen los datos (si no se especifica nada se supone que provienen de ***standard-input***)
- eof-eror-p controla lo que sucede cuando se alcanza el carácter de fin de fichero, el valor por omisión es **t** y en dicho caso se señala un error. Si su valor es **nil** la función **read** devuelve al encontrar dicho carácter el valor indicado en el argumento eof-value
- la función **read-line** con los mismos argumentos permite leer el contenido de un fichero línea a línea

print objeto stream
- escribe primero un carácter de nueva línea sobre el stream, luego el objeto de tal forma que pueda ser leído más adelante (por tanto incluye los caracteres de escape) debe ser un stream del cual se obtienen los datos (si no se especifica nada se pone que provienen de ***standard-input***) y finalmente escribe un espacio en blando

princ objeto stream
- escribe el objeto sobre el stream sin incluir los caracteres de escape

Ejercicios

BIBLIOGRAFÍA

Patrick Henry Winston,
"LISP", 3ª edición,
Addison-Wesley Iberoamericana

Guy L. Steele,
"Common Lisp the Language", 2ª edición,
Digital Press

Jesús González Boticario,
"Introducción al lenguaje Common Lisp",
UNED

?¿?¿
Documentación del GOLDEN COMMON LISP
Gold Hill Computers, Inc.

EJERCICIOS

Ejercicios - Acceso

- Evaluar las siguientes formas:
  - (first '(1 2 3))
  - (rest '(1 2 3))
  - (setf amigos '(juan ana alba))
  - (car amigos)
  - (first nil)
  - (fitst ())
  - (rest '((a (b c))(x y)))
  - (first (rest '((a b)(x y))))
  - (second '(1 2 3))
  - (nth 2 '(1 2 3 4))
  - (nth 0 '(1 2 3 4))
  - (cadadr '(1 ((2 3 4)(3 2))))
- Que primitivas son necesarias para obtener 3 a partir de cada una de las siguientes listas:
  - (1 2 3 4)
  - (1 (2 (3 (4))))
  - ((((1) 2) 3) 4)
  - ((1 2)((3)4))
  - (((1)(2)(3)(4)))
  - ((((1)))((2))(3) 4)
  - (((1)((2 3) 4)))

Ejercicios - Acceso

- Evaluar las siguientes formas:
  - (setf lista '(a b c))
  - (nthcdr 0 lista)
  - (nthcdr 1 lista)
  - (nthcdr 2 lista)
  - (nthcdr 9 lista)
  - (butlast lista)
  - (butlast lista 0)
  - (butlast lista 2)
  - (butlast lista 9)
  - (last lista)
  - (last ())
  - (last '(1 2 3 4 5) 3)
  - (last nil 4)

Ejercicios - Constructores

- Evaluar las siguientes formas:
  - (cons 'a '(1 2))
  - (setf lst1 '(3 2 1) lst2 '(c b a))
  - (cons nil nil)
  - (cons lst1 lst2)
  - (cons (fitst lst1)(rest lst1))
  - lst1
  - lst2
  - (setf lst1 (cons 4 lst1))
  - (setf lst2 (cons 'd lst2))
  - lst1
  - lst2
  - (push 'e lst2)
  - (push 'f lst2)
  - lst2
  - (pop lst1)
  - (pop lst1)
  - lst1
  - (append '(1) '(2) '(3))
  - (append nil nil)
  - (append lst1 lst2)
  - (list 1 2 3)
  - (list '(a b) '(c d))
  - (list lst1 lst2 '(x y z))
  - lst1
  - (cons '(a) '())
  - (list '(a) '())
  - (append '(a) ())
  - (cons '() '(a))
  - (list '() '(a))
  - (append '() '(a))

Ejercicios - Otras

- Evaluar las siguientes formas:
  - (length '(a b c))
  - (length '((a b)((c) d e)))
  - (length (list lst1 lst2))
  - (length (append lst1 lst2))
  - (length "Hola mundo")
  - (reverse '(1 2 3 4 5))
  - (reverse '((1 2 3)(4 5 6))
  - (reverse "anita lava la tina")
  - (subst 'a 'b '(a b ((a) (b))))
- Usando únicamente la lista lst=(a b c d) y el átomo e obtener las listas:
  - (a b c e)
  - ((a b d)(d b a))
  - (c e a b)
  - (a b c d 4)
  - (4 (d c b a))
  - (a b e d)
  - (4 c b a)

Ejercicios - Lista asociación

- Evaluar las siguientes formas:
  - (assoc 'a '((b c d)(e g)(a b c)(1 2)))
  - (assoc 'a '((a 1)(a 2)(a 3))
  - (assoc 'a (reverse '((a 1)(a 2)(a 3)))
  - (setf alba '((estatura 1.70)(peso 65)(cintura 60)))
  - (assoc 'peso alba)
  - (assoc 'cintura alba)

Ejercicios - Numéricas

- Evaluar las siguientes formas:
  - (max 4)
  - (min 4 3 2 5 6)
  - (+)
  - (+ 1 3 4 -5)
  - (- 10 4 3 2 1)
  - (/ (* 4 3 2 1) 4 3 2 1)
  - (exp 1)
  - (exp 0)
  - (log 1000 10)
  - (log (exp 69))
  - (sin 1)
  - (sin 0)
  - (/ 4 3)
  - (/ 4 3.0)
- Si en los símbolos c1 y c2 tenemos almacenada la longitud de los catetos de un triángulo rectángulo, como se calcularía en LISP la longitud de la hipotenusa
- Si en los símbolos a, b y c tenemos almacenados los valores de los coeficientes de la ecuación: $ax^2+bx+c=0$ como se calcularía en LISP el valor de x que hace cierta la igualdad

Ejercicios - Predicados

- Evaluar las siguientes formas:
  - (null nil)
  - (null 'nil)
  - (null ())
  - (null '())
  - (null '(nil))
  - (symbolp 4)
  - (setq S 'hola)
  - (symbolp S)
  - (symbolp 'S)
  - (setq S 3)
  - (symbolp S)
  - (atom nil)
  - (atom ())
  - (atom t)
  - (atom (quote hola))
  - (atom S)
  - (listp nil)
  - (listp '(a . b))
  - (numberp S)
  - (numberp (+ 3 4))
  - (numberp 2/7)
  - (numberp 2.4)
  - (zerop 0)
  - (plusp 0)
  - (minusp (- 3))
  - (minusp -4)
  - (= S 3 (+ 1 2))
  - (/= S 1 2 3)
  - (< 1 2 (+ 1 1) 4)
  - (>= (exp 10) 3 S (- 4 1))
  - (null '(lista no vacia))
  - (endp '(lista no vacia))
  - (endp ())
  - (null 'simbolo)
  - (endp 'simbolo)

Ejercicios - Igualdad

- Evaluar las siguientes formas:
  - (equal (+ 2 2) 4)
  - (equal (+ 2 3) 3)
  - (equal '(hola mundo) (setf lista '(hola mundo)))
  - (equal '(hola mundo) lista)
  - (equal 'hola mundo (setf invlista (reverse '(hola mundo)))
  - (equal lista (reverse invlista))
  - (setq subl '(1 2))
  - (setq l1 (list 1 2 subl))
  - (setq l2 '(1 2 (1 2)))
  - (eq l1 l2)
  - (eq subl (third l1))
  - (equal l1 l2)
  - (eq 1.0 1)
  - (eq 1.0 1.00)
  - (eql 1.0 1.00)
  - (= 1.0 2/2)
  - (member subl l1)
  - (member subl l2)
  - (member subl l2 :test (function equal))
  - (member subl l2 :test #'equal)
  - (not (member subl l1))
  - (setq predicado #'equal)
  - (member subl l2 :test predicado)

Ejercicios - Read-eval-print

- Evaluar las siguientes formas:
  - (read) Y teclear: hola que tal?
  - (read) Y teclear: (hola que tal)
  - (read) Y teclear: "hola que tal"
  - (setq saludo (read)) Y teclear: hola
  - saludo
  - (setq cad (read)) Y teclear: (cdr '(b c d))
  - cad
  - (eval 'cad)
  - (eval cad)
  - (eval (second '(1 (cdr '(a b c)) 2 3)))
  - (print (cdr '(1 2 3)))
  - (print (eval (read)))

Ejercicios - Definición de funciones

- Evaluar las siguientes formas:
  - (defun foo (x &optional (y nil) (z 8 b) &rest l)(list x y z b l))
  - (foo 4) (foo 4 3)
  - (foo 4 3 2) (foo 4 3 2 1 0)
  - (defun bar (a b &key c d)(list a b c d)))
  - (bar 1 2) (bar 1 2 :c 6)
  - (bar 1 2 :d 8) (bar 1 2 :c 6 :d 8)
  - (bar 1 2 :d 8 :c 6)
  - (bar :a 1 :d 8 :c 6)
  - (bar :a :b :c :d)
  - (defun pin (a &optional (b 3) &rest x &key c (d a))(list a b c d x))
  - (pin 1 2) (pin :c 7)
  - (pin 1 6 :c 7) (pin 1 6 :d 8)
  - (pin 1 6 :d 8 :c 9 :d 10)

Ejercicios - Definición de funciones

- (defun foo (&key radix (type 'integer)) ...)
- (defun foo (&key ((:radix radix) ((:type type) 'integer)) ...)
- (defun f (&key ((c d) nil) p)

  (if (eq d 'run) "running")

  "Not found"))
- (f :p 'edit)
- (f :p 'edit 'c 'run)

Ejercicios - Funciones sin nombre

- Evaluar las siguientes formas:
  - ((lambda (a b)(+ a (* b 3))) 4 5)
  - ((lambda (a &optional (b 2))(+ a (* b 3))) 4 5)
  - ((lambda (a &optional (b 2))(+ a (* b 3))) 4)
  - ((lambda (a b &key c d)(list a b c d))) 1 2 :d 8 :c 6)

Ejercicios - Condicionales

- Evaluar las siguientes formas:
  - (defun test-case (in)

```
(case in
  ((1 3 6 10) "Triangular")
  ((1 3 7) "Primo")
  ((2 4 6 8) "Par")
  (Hola "Saludo")
  ((pera fresa) "Fruta")
  (otherwise "No tengo ni idea")))
```

  - (test-case 3)
  - (test-case 'fresa)
  - (test-case '(1 3 7))

Ejercicios - Recursión

- Evaluar las siguientes formas:
  - (defun sataniza-lista (lista)
    (if (null lista)
    lista
    (cons 6 (sataniza-lista (rest lista)))))
  - (sataniza-lista '(1 2 3 4))
  - (sataniza-lista '(1 (2 3)((4 (5)) 6) 7))
  - (defun sataniza-arbol (a)
    (cond ((null a) a)
    ((not (listp a)) 6)
    (t (cons (sataniza-arbol (first a))
    (sataniza-arbol (rest a))))))
  - (sataniza-arbol '(1 2 3 4))
  - (sataniza-arbol '(1 (2 3)((4 (5)) 6) 7))

Ejercicios - Ligadura

- Evaluar las siguientes formas:
  - (setq v1 69)
  - (let ((v1 1)(v2 (+ v1 1))) (+ v1 v2))
  - (let* ((v1 1)(v2 (+ v1 1))) (+ v1 v2))
  - (let ((c 0))
    (defun inc() (setq c (+ c 1)))
    (defun dec() (if (zerop c) c (setq c (- c 1))))
    (defun show() c))
  - c
  - (inc)
  - (show)
  - (dec)
  - (defun foo(x)(list a x b))
  - (foo 2)
  - (progv '(a b) '(1 3) (foo 2))
  - (let ((a 1)(b 2))
    (declare (special a))(declare (special b))
    (foo 2))

Ejercicios - Bloques

- Evaluar las siguiente formas, primero tal como esta, después sustituyendo **interno** por medio y externo:

```
(block externo
  (print "dentro de externo")
  (block medio
    (print "dentro de medio")
    (block interno
      (print "dentro de interno")
      (return-from interno 7)
      (print "salgo de interno"))
    (print "salgo de medio"))
  (print "salgo de externo"))
```

Ejercicios - Iteración I

- Evaluar las siguiente formas
  - (dotimes (c (* 2 4) 'fin)
    (format t "~%~D * 2 = ~D" c (* c 2)))
  - (dolist (elto '(a b c) 'fin) (print elto))
  - (let ((el '(a b c)))
    (loop (print (car el))
    (setq el (cdr el))
    (if (null el) (return 'fin))))
  - (defun mi-expt (base exponente)
    (let ((res 1))
    (dotimes (c exponente res)
    (setq res (* pot base)))))
  - (defun mi-rev (lista &aux res)
    (dolist (elto lista res)
    (setq res (con elto res))))

Ejercicios - Iteración II

- Evaluar las siguiente formas
  - (do ((c 0 (+ c 1)) (lim (* 2 4)))
    ((= c lim) 'fin)
    (format t "~%~D * 2 = ~D" c (* c 2))))
  - (do* ((l '(a b c)(cdr l))(elto (car l)(car l)))
    ((null l) 'fin)
    (print elto))
  - (defun mi-expt (base exponente)
    (do ((res 1 (* base res))
    (exp exponente (- exp 1)))
    ((zerop exp) res)
    (format t "~%res: ~D, exp: ~D" res exp)))

Ejercicios - Depuración

- Evaluar las siguiente formas
  - (defun f0 ()(dolist (x '(1 2 3) 'fin) (print x)))
  - (defun f1 (n)(dotimes (c n 'fin) (print c)))
  - (defun f2 (a b)
    (dotimes (c1 a 'fin0) f0)
    (dotimes (c2 b 'fin1) (f1 a)))
  - (defun fr (a b)
    "Funcion recursiva"
    (if (zerop a) b
    (fr (- a 1)(+ b 1))))
  - (trace f0 f1 f2 fr)
  - (fr 3 5)
  - (f2 3 2)
  - (untrace fr)
  - (fr 3 5)
  - (time (fr 3 5))

Ejercicios - Evaluación

- Evaluar las siguiente formas
  - (eval '(+ 2 3))
  - (setf A 'B B 'C)
  - (eval A)
  - (defun foo () (print "hola"))
  - (setf foo 'adios)
  - foo
  - (function foo)
  - #'foo
  - (function (lambda (arg) (* arg 2))
  - (apply #'+ 1 2 '(3 4))
  - (apply #'cdr '((a b c)))
  - (apply 'cdr '((a b c)))
  - (setq fun-lista '(append list nconc))
  - (funcall (first fun-lista) '(a b c) '(1 2 3) '(y z))

Ejercicios - Trasnf. listas 1

- Evaluar las siguiente formas
  - (mapcar #'list '(A B C))
  - (mapcar #'list '(A B C) '(1 2))
  - (mapcar #'+ '(1 2 3) '(4 5 6))
  - (mapcan #'(lambda (l e)
    (when (member e l) (list l)))
    '((1 2 3) (a b c) (8 9 0) (k o p) (y u 9))
    '(2 c 1 p v))
  - (mapcan #'(lambda (x)
    (and (number x) (list x)))
    '(a 1 b c 3 4 d 5))
  - (apply #'nconc (**mapcar** #'(lambda (x)
    (and (number x) (list x)))
    '(a 1 b c 3 4 d 5)))
  - (let ((foo '(1 2 3 4 5)) (bar 0))
    (mapc #'(lambda (x) (setf bar (+ bar x))) foo)
    bar)

Ejercicios - Trasnf. listas 2

- Evaluar las siguiente formas
  - (maplist #'(lambda (x) x) '(a b c))
  - (maplist #'(lambda (x y) (list x y)) '(a b c) '(1 2 3))
  - (maplist #' append '(a b c) '(1 2 3))
  - (let ((foo '(1 2 7 4 6 5)))
    (maplist #'(lambda (xl yl) (< (car xl) (car yl)))
    foo (cdr foo)))
  - (mapcon #'(lambda (x y) (list x y)) '(a b c) '(1 2 3))
  - (mapcon #'(lambda (x) (if (null (rest x))
    (list (first x)) (list (first x) 'and)))
    '(a b c d e))
  - (mapl #'(lambda (x y) (print (list x y))) '(a b c) '(1 2 3))
  - (mapl #'(lambda (x)(unless (null (rest x))
    (setf (second x)(+ (first x)(second x)))))
    '(1 2 3 4 5))

Ejercicios - Valores múltipl.

- Evaluar las siguiente formas
  - (values 1 2 3)
  - (values (values 1 2) (values 3 4))
  - (defun doble-exp (x y) (values (expt x y)(expt y x)))
  - (doble-exp 3 2)
  - (let (a b c)
    (multiple-value-setq (a b c)
    (values 1 2 3 4))
  - (values a b c))
  - (multiple-value-setq (n1 n2) (doble-exp 2 3))
  - (multiple-value-setq nil (values 1 2))
  - (multiple-value-bind (conciente resto)
    (floor 17 3)(* cociente resto))

Ejercicios - Lista propiedad

- Evaluar las siguiente formas
  - (setq a 5)
  - (get 'a 'estatura)
  - (symbol-plist 'a)
  - (progn (setf (get 'foo 'frob) 7)
    (symbol-plist 'foo)) => (frob 7)
  - (setf (get 'alba 'estatura) 1.70)
  - (setf (get 'alba 'peso) 58)
  - (setf (get 'alba 'cintura) 60)
  - (symbol-plist 'alba)
  - (get 'alba 'estatura)

Ejercicios - Estructuras

- Evaluar las siguiente formas
  - (defstruct persona
    nombre
    estatura
    peso
    cintura)
  - (setq pers1
    (make-persona :nombre 'alba))
  - (persona-nombre pers1)
  - (setf (persona-estatura pers1) 1.70)
  - pers1

Ejercicios - Arrays

- Evaluar las siguiente formas
  - (make-array '(2 2) :initial-element 'a)
  - ```
    (let
        ((a (make-array '(2 3)
              :initial-contents '((a b c)(1 2 3))))
        (b (make-array 2
              :initial-element 'a)))
        (values a b))
    ```
  - (setq a (make-array '(2 3)))
  - (values (array-dimension a 0)(array-dimension a 1))
  - (array-rank a)
  - (array-rank (make-array '(1 2 3 4 5)))
  - (setf (aref a 0 1) 'hola)
  - (setf (aref a 0 0) 'inicial)
  - (setf (aref a 1 2) 'final)
  - (setf (aref a 1 1) '(1 1))
  - a
  - (vector 'uno 2 'tres 4 5)

Ejercicios - Macros 1

- Evaluar las siguiente formas
  - ```
    (defmacro mi-unless (test &rest cuerpo)
         (list 'cond (cons (list 'not test) cuerpo)
                        (list t nil)))
    ```
  - (setf a 3)
  - (mi-unless (listp a) (list a))
  - (macroexpand '(mi-unless (listp a)(list a)))
  - ```
    (defmacro mi-if (cond then &optional else)
         (list 'cond (list cond then)
                        (list t else)))
    ```
  - (mi-if (> 2 1) (print "2>1"))
  - (mi-if (> 5 7) (print "error")(print "5<7"))

Ejercicios - Macros 2

- Evaluar las siguiente formas
  - (setq c 'fin)
  - `(a b c)
  - `(a b ,c)
  - (setq b '(una mini lista))
  - `(a ,b c)
  - `(a ,@b c)
  - `(a ,@(cdr b) ,c)
  - ```
    (defmacro mi-unless (test &rest cuerpo)
         `(cond ((not ,test) ,@cuerpo)
                (t nil)))
    ```
  - (mi-unless (listp a) (list a))
  - (macroexpand '(mi-unless (listp a)(list a)))
  - ```
    (defmacro mi-if (cond then & optional else)
         `(cond (,cond ,then)
                (t ,else)))
    ```

Ejercicios - E/S

- Evaluar las siguiente formas
  - ```
    (with-open-file (mi-stm "test"
              :direction :output :if-exists :supersede)
         (prin1 "primero" mi-stm))
    ```
  - ```
    (with-open-file (mi-stm "test" :direction :input)
         (read mi-stm))
    ```
  - (with-open-file (mi-stm "nuevo-f"
  - ```
    :direction :output :if-exists :overwrite
                          :if-does-not-exist :create)
         (princ "Mi nombre es nuevo-f" mi-stm)
         (terpri mi-stm)
         (princ "esto es otra línea"))
    ```
  - (setq stm (open "nuevo-f" :direction :input))
  - (read-line stm nil 'fin)
  - (read-line stm nil 'fin)
  - (read-line stm nil 'fin)
  - (close stm)
  - (print "hola")
  - (princ "hola")
  - (prin1 "hola")