PRÁCTICA 0: INTRODUCCIÓN A LISP

IMPORTANTE: Utilizad la versión Allegro CL 6.2 ANSI with IDE.

EXPRESIONES

En LISP tanto el código como los datos se representan mediante las llamadas *expresiones S* (*S-exps*). Una *S-exp* sólo puede tener dos formas:

- *Atómo*.
- *Lista*.

Dentro de los *átomos*, se distinguen tres tipos:

- *Números*. Se expresan por su valor: 17, 0.001 ó 5e-4.
- *Símbolos*.Se representan mediante cadenas continuas de caracteres sin comillas y representan a variables o constantes: *i, j, var1*.
- *Cadenas de caracteres*. Se escriben como cadenas de caracteres entre comillas, pudiendo contener espacios: "hola", "buenas tardes".

Una *lista* se representa mediante un paréntesis de apertura y uno de cierre, entre los cuales puede aparecer a su vez cualquier tipo de S-exp: (), (17 i), (1 (2 3) 4), (5 ("seis" "siete") 8.0).

EVALUACIONES

Cuando se introduce una S-exp en el intérprete de LISP, este evalúa la expresión mediante las siguientes reglas:

- Valor de un átomo. Dependiendo de qué sea el átomo:
  - Si es un número, se evalúa al valor del propio número (p.e. 2 se evalúa a 2).
  - Si es una cadena de caracteres, se evalúa a la propia cadena (p.e. "hola" se evalúa a "hola").
  - Si es un símbolo, se evalúa a la S-exp que tenga asignada (p.e. *i* se evalúa a 3 si se le asignó antes dicho valor, y *mi_lista* se evalúa a (1 2 3) si previamente se le asignó la lista (1 2 3)).
- Valor de una lista. Una lista se interpreta como una llamada a función, en la que el nombre de la función es el primer elemento, y los argumentos pasados a la función son el resto de argumentos (notación prefija). Por ejemplo, (+ 2 3) se evalúa a 5, ya que + es el nombre de la función suma.

La evaluación de expresiones en LISP es ansiosa (*eager evaluation*). Antes de evaluarse una llamada a una función (es decir, una lista) se evalúan todos sus argumentos de izquierda a derecha. Los argumentos de entrada podrían ser a su vez expresiones que necesitan ser evaluadas. Por ejemplo, el evaluar la lista (+ (* 3 2) (- 9 i)) se hace en este orden:

1. Se evalúa el primer argumento: (* 3 2). Como esto a su vez es una lista, se aplica el mismo procedimiento: 3 se evalúa a 3 por ser un número, 2 a 2, se llama a la función * al no haber más argumentos, y se devuelve el resultado 6.

2. Se evalúa el segundo argumento: (- 9 i). Como esto a su vez es una lista, se aplica el mismo procedimiento: 9 se evalúa a 9 por ser un número, *i* se evalúa a la expresión que almacena (suponemos que es 3 de más arriba), y por tanto este segundo argumento también resulta ser 9 – 3 = 6.

3. Como ya no hay más argumentos para la función +, esta se invoca y devuelve 12.

Dado que una lista se interpreta siempre como una llamada a función, hace falta una manera de referirse a una lista de manera literal. Esto se puede hacer mediante la función *quote*, que normalmente se abrevia con un apóstrofo. Por ejemplo, (quote (1 2)) devuelve la lista (1 2) tal cual, y es equivalente a escribir '(1 2). El operador *quote* no sólo puede usarse con listas, sino con cualquier expresión: en cualquier caso devuelve la expresión tal cual.

Para asignar una expresión a un símbolo, la manera más directa es mediante *defparameter*: (defparameter s exp) le asignará el valor de la expresión exp al símbolo s. Una vez hecha la asignación, si se quiere desencadenar el proceso de evaluación del símbolo puede invocarse a la función *eval*: (eval s) evaluará lo que contenga el símbolo s.

- Prueba a ejecutar el código del ejemplo 1 en el intérprete de LISP. En este código encontrarás una función adicional *list*, que forma una lista con los elementos que se le pasan. Explica los resultados obtenidos, con especial hincapié en las sentencias con un doble *eval*.

PREDICADOS

Un *predicado* es una función de LISP cuyo resultado es verdadero o falso. El valor falso se representa mediante *NIL*, que en el fondo se identifica con una lista vacía. El valor true se representa mediante *T*, y se identifica a su vez con una lista no vacía (también pueden usarse en minúsculas *nil* y *t*, que luego el intérprete pasa a mayúsculas).

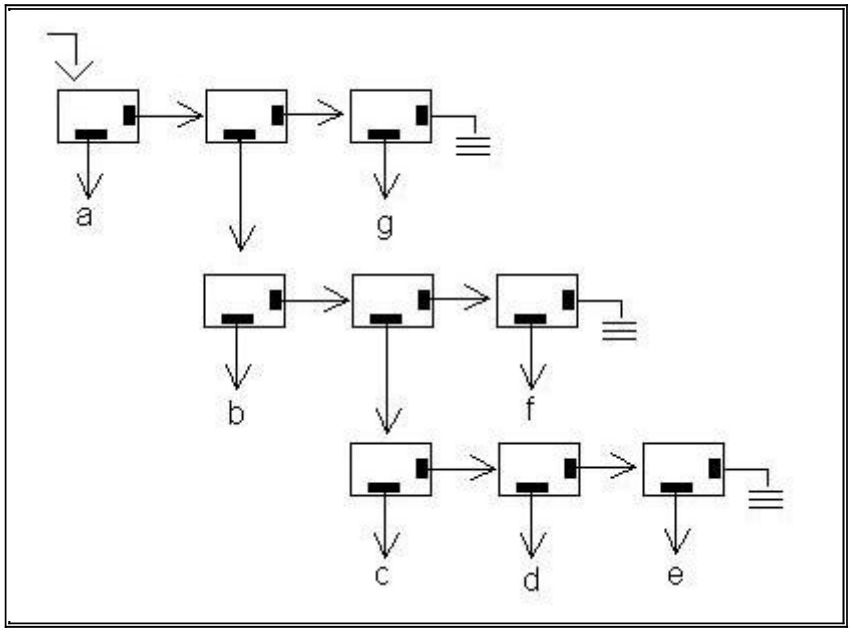
Algunos predicados utilizados frecuentemente son los siguientes:

- *null:* determina si se le pasa nil.
- *atom:* determina si se le pasa un átomo.
- *symbolp:* determina si se le pasa un símbolo.
- *numberp:* determina si se le pasa un número. Para saber si un número es 0, positivo, negativo, par o impar existen predicados más especializados como *zerop*, *plusp*, *minusp*, *evenp*, *oddp*.
- *listp:* devuelve si se le pasa una lista.

- Prueba a ejecutar el código del ejemplo 2 en el intérprete de LISP. Comenta brevemente los resultados obtenidos.

ESTRUCTURA Y FUNCIONES DE LISTAS

Las listas que usa LISP son listas enlazadas simples. Cada nodo de la lista es lo que se conoce como una *celda cons*. Una celda cons está formada por dos punteros. Uno de ellos, llamado históricamente *car* (Contents of Address Register) apunta al dato que tenemos en esa posición de la lista. El otro, llamado *cdr* (Contents of Decrement Register), apunta al siguiente elemento de la lista (NIL en caso de que la lista haya terminado). A continuación puedes ver la estructura que tiene en memoria la lista `(a (b (c d e) f) g)`.



Es decir, que una lista se representa en LISP como un par de punteros: uno al primer elemento de la lista y otro al resto de la lista. Para juntar dos elementos en una *celda cons* existe la propia función *cons*, al igual que también se proporciona un predicado *consp* para saber si su argumento es o no una *celda cons*. Para construir una lista de más de dos elementos ya sabemos que podemos utilizar la función *list*. Por ejemplo, (list 'a 'b 'c) es equivalente a (cons 'a (cons 'b (cons 'c nil))). Para formar una lista mediante concatenación de otras listas puede usarse *append*.

Como equivalente a *car* tenemos *first*, y como equivalente a *cdr* está *rest*. Mediante combinaciones de *car* y *cdr* se puede acceder a cualquier elemento de la lista, aunque para los casos más habituales existen atajos como *second*, *third*, etc. La longitud de una lista la da la función *length*.

- Prueba a ejecutar el código del ejemplo 3 en el intérprete de LISP.
- Una vez comprendidos los resultados, escribe el pseudocódigo de una implementación propia de *length* que utilice recursión sobre la lista pasada. Llama a esta función *my-length*.

OPERADORES LÓGICOS Y CONDICIONALES

Los operadores lógicos básicos de LISP son:

- *not*: niega la expresión pasada.
- *or*: evalúa las expresiones pasadas hasta encontrar una que no sea nil, devolviendo su valor. Si no encuentra ninguna, devuelve nil.
- *and*: evalúa las expresiones pasadas hasta encontrar una que sea nil, devolviendo dicho nil. Si no encuentra ninguna, devuelve el valor de la última expresión.

En cuanto a los condicionales más habituales:

- *if*: si su primer argumento se evalúa a t, devuelve el valor de su segundo argumento. En caso contrario, devuelve el valor de su tercer argumento.
- *when*: si su primer argumento se evalúa a t, evalúa el resto de sus argumentos, devolviendo el valor del último. En caso contrario, devuelve nil.
- *unless*: si su primer argumento se evalúa a nil, evalúa el resto de sus argumentos, devolviendo el valor del último. En caso contrario, devuelve nil.
- *cond*: se le pasa una lista de condiciones de la forma (condición exp1…expN). Cuando se encuentra la primera no nil, evalúa sus expresiones exp1…expN y devuelve el valor de la última.

En caso de que no se encuentre ninguna condición no nil, devuelve nil. Suele ponerse una última condición a t para asegurarse de que siempre hace algo (al estilo del *default* en un *case* de C).

- Prueba a ejecutar el código del ejemplo 4. Asegúrate de entender por qué cada caso devuelve lo que devuelve.
- Prueba a modificar el valor de nota para que el cond devuelva los otros casos. Prueba también a modificar el cond para que devuelva nil.

DEFINICIÓN DE FUNCIONES

Para definir una función se utiliza la construcción *defun*, que recibe tres parámetros:

1. El nombre que se le va a dar a la función.
2. Una lista con los argumentos que recibe.
3. El cuerpo de la función (es decir, las expresiones que conforman el código).

Por ejemplo, el siguiente código:

```
(defun my-third (x)
   (first (rest (rest x))))
```

Define una función *my-third* que recibe un único argumento x, y su única instrucción es la contenida en la segunda línea. Puede comprobarse fácilmente cómo dicha función es equivalente a la función nativa *third*.

- Prueba a ejecutar el código del ejemplo 5. Asegúrate de entender cómo funcionan las funciones ahí definidas.
- A continuación, implementa en LISP la función *my-length* cuyo pseudocódigo escribiste anteriormente, y comprueba su correcto funcionamiento mediante diferentes pruebas.

APLICACIÓN REPETIDA DE FUNCIONES

Cuando queremos aplicar una función de manera repetida sobre una o varias listas, LISP nos ofrece dos opciones básicas:

- *mapcar*: aplica la función que se le diga sobre el first de las listas (el car), luego sobre el second (el car del cdr), y así sucesivamente, es decir, elemento a elemento. Devuelve la lista resultante.
- *maplist*: aplica la función que se le diga sobre las listas tal cual, luego sobre el cdr de cada una de ellas, luego sobre el cdr del cdr, y así sucesivamente. Devuelve la lista resultante.

La función pasada a estas operaciones puede ser una función nativa de LISP, una propia definida con *defun*, o una *función lambda*, que actúa como una función anónima definida sólo en dicha orden. Las funciones lambda se definen con la misma sintaxis que las funciones estándar, sólo que sustituyendo la palabra *defun* por *lambda*. Por ejemplo, la siguiente función le suma 1 a su argumento:

```
(lambda (x) (+ x 1))
```

Para referirse a una función concreta se puede usar el operador #', que viene a ser un quote de funciones. Dado que una función lambda es anónima, salvo que la función se le haya asignado a un símbolo (por ejemplo mediante defparameter), sólo podremos referirnos a ella mediante #'.

La función *reduce* también aplica una función sobre una lista, pero la función tiene que ser obligatoriamente binaria, de forma que se puedan ir acumulando resultados de izquierda a derecha. Por ejemplo, (reduce #'* '(1 2 3 4)) es equivalente a (* (* (* 1 2) 3) 4)

Para llamar a una función concreta sólo una vez, en vez de para una lista como hacen *mapcar* y *maplist*, pueden emplearse tanto *apply* como *funcall*. La única diferencia es que *apply* espera recibir una lista con los argumentos, mientras que en *funcall* se dan separados.

- Estudia el código del ejemplo 6.
- Una vez comprendido, implementa una función *sum-range* que reciba un número n y devuelva el resultado de sumar todos los números desde 1 hasta n. Por ejemplo, (sum-range 10) debe dar 55 y (sum-range 100) es 5050. Si el número pasado no es positivo haz que sum-range devuelva nil. Puedes definir funciones auxiliares si te facilita la tarea.

COMPARACIONES Y BÚSQUEDAS

En LISP existen diferentes predicados para comparar elementos:

- *eq*: devuelve t si y sólo si los dos argumentos que se le pasan son el mismo objeto.
- *eql*: devuelve t si sus dos argumentos son iguales bajo *eq* o si son números del mismo tipo y con el mismo valor.
- *equal*: devuelve t si sus dos argumentos son similares estructuralmente (isomorfos). Esto suele querer decir que *equal* devolverá t si la representación por escrito de ambos objetos es la misma.
- =: para números devuelve t si el valor numérico es el mismo, independientemente del tipo. Existen también los operadores relacionados /=, >, >=, <, <=. Pueden recibir más de dos argumentos, en cuyo caso se aplica el operador elemento a elemento, y sólo devuelve t si es cierto para todos.

Estos predicados de igualdad se pueden usar en funciones que se basan en búsquedas en listas. Por defecto usan *eql*, pero este comportamiento se puede cambiar mediante la keyword *:test* (con los dos puntos delante). Las funciones más comunes son:

- *member*: determina si un determinado elemento está en una lista o no. Si no está devuelve nil, y si sí que está devuelve la sublista cuyo primer elemento es el deseado.
- *position*: determina si un determinado elemento está en una lista o no, devolviendo su posición (empezando en 0) o nil si no se encuentra.
- *find*: determina si un determinado elemento está en una lista o no, devolviendo su primera ocurrencia o nil si no se encuentra.
- *count*: devuelve cuántas veces se encuentra el elemento en la lista, o 0 si no se encuentra.
- *remove*: devuelve la lista resultante de eliminar el elemento especificado.

A su vez, todas estas funciones tienen variantes if (*member-if*, *position-if*, etc.) que comprueban el predicado que se le pase en lugar de la igualdad. Otras dos funciones que reciben un predicado son *every* y *some*. Como indican sus nombres, la primera devuelve t si el predicado devuelve t para todos los elementos de la lista, mientras que la segunda devuelve t si el predicado devuelve t para algún elemento de la lista.

- Estudia el código del ejemplo 7 y razona las semejanzas y las diferencias entre los resultados obtenidos con *eq*, *eql* y *equal*.
- Una vez entendidas, implementa una versión propia de *member* llamada *my-member* que reciba 3 argumentos: el elemento, la lista y el comparador a usar. Prueba con distintos ejemplos tanto con *eql* como con *equal*, y asegúrate de que devuelve lo mismo que el *member* nativo con el *:test* adecuado.
- Cuando hayas comprobado su corrección, codifica una función *my-count* que haga uso *de my-member*. De nuevo, comprueba que obtienes los mismos resultados que el *count* nativo.

COMPARACIÓN DE IMPLEMENTACIONES

Como en cualquier lenguaje de programación, existen diferentes maneras de implementar una misma tarea. Sin embargo, ya habrás ido intuyendo que la estructura de LISP basada en listas lo hace mucho más propenso a implementaciones recursivas que otros lenguajes como C o Java, donde para hacer operaciones sobre listas o arrays se suelen utilizar bucles.

LISP también ofrece opciones a caballo entre la recursividad pura y los bucles, como por ejemplo utilizar *mapcar*. Aunque también existen bucles en LISP (*do*, *dotimes*, *dolist*…), estos suelen reservarse para circunstancias muy concretas. De hecho, salvo que se diga lo contrario, en las prácticas de esta asignatura no se deberá usar ningún bucle.

Obviamente, en caso de que ya exista una función nativa en LISP que haga una determinada tarea, es preferible usarla a implementarla nosotros mismos, usemos el paradigma que usemos. En esta práctica inicial probamos a implementar versiones propias, pero simplemente para familiarizarnos con el lenguaje; es casi seguro que la función nativa será mejor en simplicidad y velocidad de ejecución.

- Estudia el código del ejemplo 8, donde se dan tres versiones de una función que obtiene los elementos pares de una lista. Comenta las ventajas e inconvenientes que le ves a cada una de esas implementaciones.
- Comprueba la eficiencia de estas implementaciones. Para ello, construye una lista grande con ayuda de la función *make-list* (por ejemplo de tamaño 1000) y mide tiempos de ejecución haciendo uso de la función *time*. Analiza los resultados obtenidos.
- A la vista de estos resultados, ¿crees que podrías implementar más eficientemente *my-length* y *my-member*? En caso de que sí, prueba a reimplementarlas y compara ejecuciones con time a ver si estás en lo cierto.

FUNCIONES DESTRUCTIVAS

Existen algunos pares de funciones nativas en LISP que realizan el mismo cometido, pero que en una versión no alteran los argumentos originales (no destructivas) y en otros sí que los modifican (destructivas). Por ejemplo, ya hemos visto que *remove* no modifica la lista pasada, sino que devuelve una lista copia en la que se ha eliminado el elemento deseado. La versión destructiva de esta tarea es *delete*, que elimina el elemento directamente de la lista original.

En otros casos, existe sólo una función para realizar una tarea y es destructiva. El ejemplo por excelencia es la función de ordenación *sort*. Si se quiere que la lista original se mantenga inalterada debe pasarse una copia con *copy-list*.

En aras de la programación funcional, debe evitarse en lo posible el uso de funciones destructivas. En caso de necesitar utilizarlas se les pasarán copias, de manera que ninguna función modifique nunca sus argumentos y no desencadene así efectos colaterales.

- Estudia el código del ejemplo 9 y verifica cómo las funciones destructivas modifican los argumentos que se les pasan, mientras que las no destructivas no lo hacen.
- Codifica una función *sorted-occurrences* que cuente el número de veces que aparece cada elemento distinto en una lista. Debe devolver pares de la forma (elemento, número de ocurrencias), donde el orden es de mayor a menor según el número de ocurrencias. Por ejemplo, (sorted-occurrences '(3 1 4 2 3 1 3 2 3 1)) debe devolver ((3 4) (1 3) (2 2) (4 1)). Asegúrate de que la función no es destructiva.