Programación Lógica y Funcional
Tarea 4. Ejercicios en LISP

Angel García Báez
ZS24019400@estudiantes.uv.mx

Maestría en Inteligencia Artificial

IIIA Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección 2a, No 112
Nuevo Xalapa, Xalapa, Ver., México 91097

27 de febrero de 2025

1. Implemente en Lisp las siguientes operaciones sobre conjuntos representados como listas. [15 puntos]

- Subconjunto:

```
> (subset '(1 3) '(1 2 3 4))
true.
> (subset '() (1 2)).
true.
```

- Intersección:

```
> (inter '(1 2 3) '(2 3 4)).
(2 3)
```

- Unión:

```
> (inter '(1 2 3) '(2 3 4)).
(2 3)
```

- Diferencia:

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

Para lograr la correcta implementación de las operaciones de conjuntos, considerando el hecho de ser un principiante en LISP, fue necesario recurrir a las diapositivas de Guerra-Hernández (2024), a el libro de Clocksin (2003) donde las explicaba en prolog y a el libro de Seibel (2006) que explica el porque usar LISP y como empezar en el.

Despues de un proceso de mucha prueba y error, las funciones que de se lograron crear fueron las siguientes

- Subconjunto:

```
;; Subconjunto
(defun subset (a b &key (test #'eql))
"Devuelve T si A es subconjunto de B, NIL en caso contrario.
```

```
El parámetro opcional TEST permite personalizar la comparación."
(every (lambda (element)
(find element b :test test))
a))
```

- Intersección:

```
;; 1.2 Intersección
(defun inter (conjunto1 conjunto2)
"Devuelve la intersección de dos conjuntos representados como listas."
(remove-duplicates
(remove-if-not (lambda (x) (member x conjunto2))
conjunto1)))
```

- Unión:

```
;; 1.4 Diferencia
;; Capitulo 3 del libro de commond lisp
(defun dif (conjunto1 conjunto2)
"Devuelve la diferencia de dos conjuntos representados como listas."
(remove-if (lambda (x) (member x conjunto2)) conjunto1))

```

- Diferencia:

```
;; 1.4 Diferencia
;; Capitulo 3 del libro de commond lisp
(defun dif (conjunto1 conjunto2)
"Devuelve la diferencia de dos conjuntos representados como listas."
(remove-if (lambda (x) (member x conjunto2)) conjunto1))
```

2. Escriba un programa en que elimine todas las ocurrencias de un elemento en una lista. Por ejemplo:

```
> (eliminar 3 '(1 3 2 4 5 3 6 7))
(1 2 4 5 6 7)
```

Explique brevemente cómo es que LISP evalúa esta expresión. [10 puntos]

Dicho programa se implemento recordando las ideas de prolog y quedando de la siguiente forma:

```
;; 2. Escriba un programa en que elimine todas las ocurrencias
;; de un elemento en una lista.
>(defun eliminar (elemento lista)
    "Elimina todas las ocurrencias de un elemento en la lista"
    (cond
      ((null lista) nil) ; Caso base si la lista es vacia
      ((eql (car lista) elemento) ; Si el primer elemento coincide, eliminar
       (eliminar elemento (cdr lista))) ; Continuar con el resto de la lista.
      (t ; Caso contrario, conservar el elemento y seguir recorriendo
       (cons (car lista) (eliminar elemento (cdr lista))))))
>(eliminar 3 '(1 3 2 4 5 3 6 7))
(1 2 4 5 6 7)
```

La función fue implementada bajo una lógica recursiva y condicional, por lo que en un primer momento, se plantea el caso base que es tener una lista vacía. Si es el caso, la función unicamente devuelve dicha lista vacía, si no es el caso, se verifica la ocurrencia del siguiente caso donde se verifica si el primero elemento de la lista coincide con el elemento que se esta buscando eliminar las ocurrencias lo que hace es eliminarlo, si no es el caso, continua para el resto de la lista y aplica el caso recursivo donde se vuelve a llamar,

Lisp ejecuta las llamadas recursivas de forma que evalúa cada rama de las condicionales en orden. Para cada nivel toma la decisión de si debe eliminar el primer elemento (cabeza) o conservarlo. Cuando llega al caso base que es tener la lista vacía se detiene la recursión y reconstruye la lista con los elementos que no fueron eliminados.

3. Implemente una función en Lisp que dada una lista de átomos, regresa las posibles permutaciones de sus miembros. [10 puntos]

```
> (perms '(1 2 3))
((1 2 3) (1 3 2) (2 3 1) (2 1 3) (3 1 2) (3 2 1))
```

Para lograr la implementación de dicha función, nuevamente, fue extremadamente necesario acudir a las notas de prolog Guerra-Hernández (2024), al libro de Clocksin (2003) y a Seibel (2006) para poder hacer la traducción de esas ideas de prolog a LISP, resultando en la siguiente función:

```
>(defun perms (lista)
   "Devuelve una lista con todas las permutaciones posibles de LISTA."
   (if (null lista)
       '(()) ; Caso base si la lista es vacia
       (mapcan
        (lambda (x)
         (mapcar (lambda (perm)
                   (cons x perm))
                 (perms (remove x lista :count 1))))
       lista)))
>(perms '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

Para la implementación de dicha función que devuelve las permutaciones, fue necesario emplear algunas funciones de más alto nivel como lo son mapcan, mapcar y remove.

La lógica que sigue la función es nuevamente un caso recursivo, donde el caso base es tener una lista vacía, si no es el caso, la función mapcan recorre cada uno de los elementos de la lista y aplica una función lambda a cada elemento de tal forma que se va consumiendo la lista pero al llegar hasta el caso base, comienza a devolver hacia atrás la reconstrucción de las listas con las diferentes permutaciones de los elementos.

4. ¿Qué diferencias importantes puede señalar entre la implementación de los ejercicios anteriores y la llevada a cabo con Prolog. Sean concisos en la respuesta. [10 puntos]

Una de las principales diferencias viene dada desde el cambio de paradigma, si bien, LISP es un lenguaje que permite trabajar con varios paradigmas de programación, desde el enfoque que se viene manejando en el curso prolog es el paradigma lógico y LISP el funcional. Por lo que cambia el hecho de que en uno se están manejando como clausulas y hechos lógicos, mientras que en LISP se estan manejando como si fueran funciones de funciones.

Otro detalle importante es el estilo, puesto que en los mismos ejercicios pero implementados en prolog, no era necesario ser 100 % explicito en lo que debía de ocurrir, puesto que bastaba con decirle al programa lo que se sabia del problema para que el motor de inferencias junto con el backtracking hicieran el resto vía unificación, aquí es necesario ser más explicito para llevar el flujo de lo que se quiere que el programa haga debido a que no se cuenta con dichas bondades de prolog.

5. Lea el capítulo 22 (*LOOP for Black Belts*) del libro de Peter Seibel, Practical Common Lisp. Utilice la macro loop para resolver alguno de los ejercicios propuestos en esta tarea. [15 puntos]

Se propone resolver el ejercicio 2 que pide eliminar todas las ocurrencias de un elemento en una lista haciendo uso del macro LOOP para que recorra toda la lista en lugar de hacerlo de forma recursiva.

El código de la función propuesta con ayuda de Seibel (2006) es:

```
>(defun eliminarlp (elemento lista)
   "Elimina todas las ocurrencias de ELEMENTO en LISTA usando LOOP."
   (loop for x in lista
     unless (eql x elemento) ; Si X no es igual, incluyelo
     collect x))
>(eliminarlp 3 '(1 3 2 4 5 3 6 7))
(1 2 4 5 6 7)
```

Aquí se define el macros loop para realizar un recorrido sobre cada uno de los elementos de la lista de forma que, estando dentro de esta estructura de control más familiar como lo que viene siendo un for, se evalúa elemento por elemento si el elemento actual coincide con el elemento que se desea excluir. Si el elemento coincide, se elimina y si no coincide, se guarda en una nueva lista que va a ser la lista de salida.

6. Defina una macro repeat que tenga el siguiente comportamiento. Evidentemente, la expresión que se repite puede ser cualquier expresión válida en Lisp. [15 puntos]:

```
> (repeat 3 (print 'hi))
HI
HI
HI
NIL
```

A continuación se muestra el codigo de la macro REPEAT propuesta con mucha ayuda del libro de Seibel (2006):

```
>(defmacro repeat (n &rest body)
   "Repite la expresión BODY N veces."
   (let ((i (gensym)))  ; Genera un nombre único para el índice
     `(dotimes (,i ,n)   ; Repite el cuerpo N veces
        ,@body)))         ; Expansión del cuerpo (se ejecuta en cada iteración)

>(repeat 5 (print "Hola, mundo!"))
"Hola, mundo!"
"Hola, mundo!"
"Hola, mundo!"
"Hola, mundo!"
"Hola, mundo!"
NIL
```

El macro repeat toma como parámetros de entrada un numero n que indica cuantas veces se va a repetir la cosa así como una expresión para que se aplique dicha repetición, en este caso esa expresión es un print que a su vez necesita un mensaje para devolver y ser repetido.

Dentro de la logica de la macro, se utiliza el gensym para crear símbolos que garanticen que no ocurrirá una parasitación de variables o símbolos, la expansión de la cosa con dotimes permite ejecutar el cuerpo especifico de la función y replicarlo una cantidad n de veces.

El resultado es una macro que permite hacer lo solicitado.

7. La siguiente función me permite definir una entrada en un registro de mis libros:

```
(defun crea-libro (titulo autor ed precio)
   (list :titulo titulo :autor autor :ed ed :precio precio))
```

Puedo usar una variable global como *db* para llevar un registro de entradas como sigue:

```
(defvar *db* nil)
(defun agregar-reg (libro) (push libro *db*))
```

De forma que:

```
> (agregar-reg (crea-libro "Pericia Artificial" "Alejadro Guerra" "UV" 90.50))
((:TITULO "Pericia Artificial" :AUTOR "Alejadro Guerra" :ED "UV" :PRECIO 90.5))
```

Agregue más entradas al registro y escriba una función con ayuda de format (Ver capítulo 18 del libro de Seibel) que despliegue las entradas como sigue [15 puntos]:

```
> (listado-db)
TITULO: Pericia Artificial
AUTOR: Alejadro Guerra
ED: UV
PRECIO: 90.50
```

Para la resolución de dicho ejercicio con las funciones propuestas, se tomo de referencia tambien el ejemplo de los CD's que viene explicado en el capitulo 3 del libro de Seibel (2006).

Para ello se hizo el siguiente código con las funciones mencionadas en la misma instrucción como sigue:

```
;; 7. La siguiente función me permite definir una entrada en un registro de mis
;; libros:

; Función para crear un registro de libro
> (defun crea-libro (titulo autor ed precio)
    (list :titulo titulo :autor autor :ed ed :precio precio))

```

```
; Definir la variable global para almacenar las entradas
> (defvar *db* nil)

; Definir la función que agrega libros a la base de datos db
> (defun agregar-reg (libro) (push libro *db*))
```

Una vez que se tiene implementada la lógica de como va a funcionar este almacenador de registros, se procede a llenar la base de datos con el registro de 5 libros y se crea la función que permite proyectar la colección de libros con un formato más legible:

```
;Agregue entradas al registro
>(agregar-reg (crea-libro "El Aleph" "Jorge Luis Borges" "Debolsillo" 199.00))
>(agregar-reg (crea-libro "Ficciones"  "Jorge Luis Borges" "Debolsillo" 259.00))
>(agregar-reg (crea-libro "El libro de los sueños" "Jorge Luis Borges" "Debolsil
>(agregar-reg (crea-libro "Análisis de algoritmos" "Homero Vladimir Ríos Figuero
>(agregar-reg (crea-libro "Pericia Artificial" "Alejandro Guerra Hernández" "UV"

; Definir la función para que se proyecte bonito
>(defun dump-db ()
   (dolist (libro *db*)
     (format t "~{~a:~10t~a~%~}~%" libro)))

> (dump-db)

TITULO:   Pericia Artificial
AUTOR:    Alejandro Guerra Hernández
ED:       UV
PRECIO:   90.5

TITULO:   Análisis de algoritmos
AUTOR:    Homero Vladimir Ríos Figueroa
ED:       UV
PRECIO:   20.0

TITULO:   El libro de los sueños
AUTOR:    Jorge Luis Borges
ED:       Debolsillo
PRECIO:   229.0

```

```
30 TITULO:   Ficciones
31 AUTOR:    Jorge Luis Borges
32 ED:       Debolsillo
33 PRECIO:   259.0
34
35 TITULO:   El Aleph
36 AUTOR:    Jorge Luis Borges
37 ED:       Debolsillo
38 PRECIO:   199.0
39
```

8. Defina una función para recuperar una entrada en el registro buscando por autor [10 puntos].

Basado fuertemente en la función que permite hacer las consultas en el ejemplo de los CD's del capitulo 3 de Seibel (2006), se adapto la función para que pudiera correr con la base de datos de los registros de los libros y les diera su correspondiente formato estilizado como se muestra a continuación:

```
; 8.- Defina una función para recuperar una entrada en el registro buscando por

> (defun buscar-por-autor (autor)
    "Busca y devuelve una lista de libros escritos por el autor dado."
        (remove-if-not
      (lambda (libro)
            (equal (getf libro :autor) autor))
                    *db*))

;; Ejemplos de uso

> (let ((resultado (buscar-por-autor "Alejandro Guerra Hernández")))
   (dolist (libro resultado)
    (format t "~{~a:~10t~a~%~}~%" libro)))
TITULO:   Pericia Artificial
AUTOR:    Alejandro Guerra Hernández
ED:       UV
PRECIO:   90.5

> (let ((resultado (buscar-por-autor "Jorge Luis Borges")))
   (dolist (libro resultado)
     (format t "~{~a:~10t~a~%~}~%" libro)))

TITULO:   El libro de los sueños
AUTOR:    Jorge Luis Borges
ED:       Debolsillo
PRECIO:   229.0

TITULO:   Ficciones
AUTOR:    Jorge Luis Borges
ED:       Debolsillo
PRECIO:   259.0
```

```
33
34 TITULO:   El Aleph
35 AUTOR:    Jorge Luis Borges
36 ED:       Debolsillo
37 PRECIO:   199.0
```

Referencias

Clocksin, William F. & Mellish, C. S. (2003). *Programming in PROLOG.* Springer Science & Business Media.

Guerra-Hernández, D. A. (2024). Programación Lógica y Funcional - Programación Lógica.

Seibel, P. (2006). *Practical Common Lisp.* Apress, Berkeley, CA, USA.