

## Actividad 4

Leonardo Flores Torres

1 de diciembre de 2022

Primero, antes de comenzar con las soluciones a la actividad presente, quisiera mencionar que el uso de `sbcl` en realidad fue hasta los últimos incisos de esta tarea. Personalmente, encontré `racket` bastante atractivo como para continuar estudiándolo y usándolo en proyectos personales en donde usaría algo como `python`, o incluso en un carácter más serio. Algunas referencias usadas para aprender y consultar dudas de `racket` fueron [1] y [2].

- 10/10 1. Escriba un programa que elimine todas las ocurrencias de un elemento en una lista. Por ejemplo:

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

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

### Solución:

Leyendo la documentación de `racket`, y con mi experiencia en `julia`, noté que podía usar una función familiar para seleccionar elementos de acuerdo a una condición, en `racket` esta condición es `filter`. `filter` toma una condición y la mapea a todos los elementos de una lista quedándose solo con aquellos que pasan la prueba. Entonces, `filter` viene bien para resolver este inciso. Se puede aprovechar el uso de aplicar una función- $\lambda$  como condición del `filter` para checar si un valor se encuentra dentro de una lista. Lo descrito anteriormente se muestra a continuación:

```
1 ;; my-remove removes all instances of a value from a list.
2 (define (my-remove val lst)
3   (filter (lambda (x) (not (eq? x val))) lst))
```




Encontré esta solución bastante atractiva, sencilla y fácil de entender. Cabe mencionar que no fue la primera implementación que hice de esta función. Mi primer intento consistió en aprovechar las facilidades intrínsecas de este lenguaje dentro de la familia de lenguajes `lisp`.

Además, casi que esto seguro que filtre es más eficiente que la recursividad en Racket.

```

1  ;; my-remove-rec recursively removes all instances of a value from a list.
2  (define (my-remove-rec val lst)
3    (if (empty? lst)
4        empty
5        (if (eq? val (first lst))
6            (my-remove-rec val (rest lst))
7            (cons (first lst) (my-remove-rec val (rest lst))))))

```




La recursividad fue bastante intuitiva para resolver las operaciones entre conjuntos de este inciso, se hizo uso de esto en la función `my-remove-rec`. El caso base es cuando se ha recorrido toda la lista pasando por sus elementos, pero si no está vacía entonces se debe verificar que el primer elemento de la lista es igual o no al valor a remover. Finalmente se hace la llamada recursiva con el resto de la lista, y es aquí que se quita de la lista al llamar recursivamente a la función con el resto de la lista. Si no es igual al valor a remover entonces se agrega a una lista auxiliar construida a partir de `cons`, y nuevamente se hace la llamada recursiva. Las dos funciones `my-remove` y `my-remove-rec` fueron escritas en `racket`.

Un ejemplo de ejecución de estas funciones se muestra a continuación:

```

1  > (my-remove 3 '(1 3 2 4 5 3 6 7))
2  '(1 2 4 5 6 7)
3  > (my-remove-rec 3 '(1 3 2 4 5 3 6 7))
4  '(1 2 4 5 6 7)

```



La forma en que `lisp` evalúa esta función es haciendo un símil a como el lenguaje `AL` evalúa funciones mediante el `EVAL` visto durante el curso. De hecho, la secuencia de pasos a seguir es muy similar a la que se vió para la función factorial `fac` incluida en las notas de clase [3].

- 
- 10/10** 2. Implemente una función en Lisp que dada una lista de átomos, regresa las posibles permutaciones de sus miembros. [10 puntos]

```

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

```

### Solución:

La solución a este inciso fue obtenida de una respuesta a una pregunta<sup>1</sup> en StackOverflow. La idea es tomar un elemento de la lista, permutar el resto de la lista, e insertar nuevamente al inicio de la permutación el elemento que se tomó. Además, está escrita en `racket`.



<sup>1</sup><https://stackoverflow.com/questions/20319593/creating-permutation-of-a-list-in-scheme>

```

1  ;; The general idea of the algorithm to compute the permutations of a list is
   ↪ explained
2  ;; below.
3  ;; For each of the n elements in the list, first find all the permutations of the
4  ;; remaining n-1 elements. Then, prepend the current element to each permutation.
   ↪ Finally,
5  ;; return a list of all resulting permutations.
6  (define (permutations lst)
7    (if (empty? lst) '())
8        (apply append
9              (map (λ (element)
10                   (map (λ (permutation)
11                        (cons element permutation))
12                          (permutations (remove element lst))))
13                    lst))))

```



Buen uso del map

Esta implementación es sencilla comparada con otras encontradas en la red para computar las permutaciones de una lista. Elegí tomar esta por la simplicidad sobre como implementa la lógica de los pasos a seguir anteriormente explicados de manera muy general. Ejemplos de esta función en acción se muestran a continuación:

```

1  > (permutations '(1 2 3))
2  '((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
3  > (permutations '(1 2 3 4))
4  '((1 2 3 4)
5    (1 2 4 3)
6    (1 3 2 4)
7    (1 3 4 2)
8    (1 4 2 3)
9    (1 4 3 2)
10   (2 1 3 4)
11   (2 1 4 3)
12   (2 3 1 4)
13   (2 3 4 1)
14   (2 4 1 3)
15   (2 4 3 1)
16   (3 1 2 4)
17   (3 1 4 2)
18   (3 2 1 4)
19   (3 2 4 1)
20   (3 4 1 2)
21   (3 4 2 1)
22   (4 1 2 3)
23   (4 1 3 2)
24   (4 2 1 3)
25   (4 2 3 1)
26   (4 3 1 2)
27   (4 3 2 1))

```



Se encontraron otras respuestas en StackOverflow usando `common lisp` pero las im-

plementaciones<sup>2</sup> que se proponían no las encontré entendibles del todo. Incluso en la famosa página de algoritmos, Rosetta Code, vienen muchas soluciones<sup>3</sup> para encontrar las permutaciones en diferentes lenguajes de programación dentro de los cuales también se incluye a `racket` y `common lisp`. También se encontrará un post dentro de un blog<sup>4</sup> donde se explica el algoritmo y varias implementaciones del mismo usando `common lisp`.



**15/15** 3. Implemente en Lisp las siguientes operaciones sobre conjuntos representados como listas. [15 puntos]

- Subconjunto:

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

### Solución:

Para resolver este inciso primero se creó una función para determinar si un valor dado aparece en una lista, con que aparezca al menos una vez es suficiente para que el resultado sea verdadero `#t`, de otra manera el resultado es falso `#f`.

```
1 ;; my-member-rec recursively iterates through a list and returns #t if a given
   ↳ value
2 ;; exists at least once in a list, else returns #f.
3 (define (my-member-rec val lst)
4   (if (empty? lst)
5       false
6       (if (eq? val (first lst))
7           true
8           (my-member-rec val (rest lst)))))
```



Con la función miembro, también definida de manera recursiva como para el caso de `my-remove-rec`, se implementa la función recursiva `my-subset-rec` que itera sobre todos los valores de la primera lista y realiza el chequeo si cada uno de ellos se encuentra en una segunda lista. Esto para estar en acuerdo con la definición de subconjunto,  $A \subseteq B$ , donde  $A$  es `lsta` y  $B$  es `lstb`. Si se llega al final de la lista es porque todos los elementos de `lsta` están en `lstb`. En caso contrario, si alguno de los elementos de `lsta` es miembro de `lstb`, entonces  $A \not\subseteq B$ .

```
1 ;; my-subset-rec is the recursive implementation of my-subset.
2 (define (my-subset-rec lsta lstb)
3   (if (empty? lsta)
```

<sup>2</sup><https://stackoverflow.com/questions/2087693/how-can-i-get-all-possible-permutations-of-a-list-with-common-lisp>


<sup>3</sup><https://rosettacode.org/wiki/Permutations>

<sup>4</sup><http://www.lispology.com/show?1FZG>

```

4      true      ; If the end of lsta is reached it means all elements of lsta
      ↪ are in lstb
5      (if (my-member-rec (first lsta) lstb)
6          (my-subset-rec (rest lsta) lstb)
7          false))) ; If one element in lsta is not in lstb then lsta is not
      ↪ a subset

```




A pesar de que la función `my-subset-eq` haga lo que se pretenda que haga y que sea fácilmente de entender lo que está haciendo al verla, es posible hacer otra implementación que a mi parecer tiene un carácter más funcional.

Pues sí!

```

1  ;; my-subset returns #f if a set A is subset of another set B, "A subset B",
   ↪ else
2  ;; returns #f.
3  ;; Note. This implementation seemed to be easily achievable with a map!
4  (define (my-subset lsta lstb)
5    (andmap (λ (x) (my-member-rec x lstb)) lsta))

```



La función `my-subset` hace uso de un mapeo donde aplica la función `my-member-rec` a todos los elementos de la lista `lsta`, si se usara un `map` se obtendría una lista de booleanos `#t` ó `#f`, tantos como cumplan con la condición de la función- $\lambda$  aplicada dentro del mapeo. El punto de usar `andmap` en vez de `map` es que si todos los elementos a los que se les aplica el mapeo cumplen con la condición, entonces el retorno de `andmap` será un solo `#t`, o en su defecto `#f` si al menos un elemento de `lsta` no se encuentra en `lstb`.


Las soluciones para las operaciones entre conjuntos fueron escritas usando `racket` a menos que se especifique lo contrario.

La función `my-subset` y su equivalente recursivo se muestran en acción:

```

1  > (my-subset '(1 3) '(1 2 3 4))
2  #t
3  > (my-subset-rec '(1 3) '(1 2 3 4))
4  #t
5  > (my-subset '() '(1 2))
6  #t
7  > (my-subset-rec '() '(1 2))
8  #t
9  > (my-subset '(1 7) '(1 2 3 4))
10 #f
11 > (my-subset-rec '(1 7) '(1 2 3 4))
12 #f

```



#### ■ Intersección:

También puedes definir tu filter  
Para hacer estas operaciones  
más funcionalmente

```

... regresa los elementos para los que fn elt es diferente de false
... > (filter zero? '(1 2 0 3 0 4 0 5))
... '(0 0 0)
... > (filter (compose not zero?) '(1 2 0 3 0 4 0 5))
... '(1 2 3 4 5)

```

```

5  (define (filter fn lst)
    (let [(acc '())]
      (for-each (λ (arg)
                  (when (fn arg) (set! acc (cons arg acc))))
                lst)
      (reverse acc)))

```

```

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

```


**Solución:**

La definición de la Intersección entre conjuntos resulta en tomar aquellos valores que ambos conjuntos comparten,  $A \cap B = \{x | x \in A \wedge x \in B\}$ . Para este caso basta con usar el mismo patrón que se ha visto hasta ahora en las funciones ya definidas, y construir una lista usando `cons` con el primer elemento de la lista por la que se está corriendo a la que se le agregaran más elementos si la cabeza de la lista es miembro de la segunda lista, y finalmente llamar recursivamente `my-intersection-rec` para ambos casos del `if` justo como se muestra a continuación:

```

1 ;; my-intersection-rec recursively computes the intersection of two sets A and
  ↪ B,
2 ;; "A intersection B".
3 (define (my-intersection-rec lsta lstb)
4   (if (empty? lsta)
5       empty
6       (if (my-member-rec (first lsta) lstb)
7           (cons (first lsta) (my-intersection-rec (rest lsta) lstb))
8           (my-intersection-rec (rest lsta) lstb))))

```




Algunos ejemplos de como funciona la función `my-intersection-rec` se muestran inmediatamente en el siguiente snippet:

```

1 > (my-intersection-rec '(1 2 3) '(2 3 4))
2 '(2 3)
3 > (my-intersection-rec '(1 5 7) '(2 3 4))
4 '()
5 > (my-intersection-rec '() '(2 3 4))
6 '()

```



Hace lo que uno esperaría al computar la intersección entre dos conjuntos manualmente.

---

■ Unión:

```

1 > (union '(1 2 3 4) '(2 3 4 5))
2 (1 2 3 4 5)

```

**Solución:**


La unión de conjuntos se denota como  $A \cup B = \{x | x \in A \vee x \in B\}$ , la función `my-union-rec` es muy similar a la función `my-intersection-rec`, con un par de diferencias. En vez de crear una lista usando `cons` se toma una de las listas, en

este caso `lstb` y se le agregan valores al inicio si se cumple con que alguno de los elementos dentro de `lsta` no se encuentra en `lstb`. Cuando se llega al final de la lista `lsta` significa que ya se han agregado todos los elementos a `lstb` que cumplen con la condición, y se puede regresar el resultado.

```

1  ;; my-union-rec returns the set which is the union of two other sets A and B,
2  ;; "A union B".
3  (define (my-union-rec lsta lstb)
4    (if (empty? lsta)      ; When lsta is empty it means we have finished
5        (sort lstb <)      ; Just to order the returned list, sorting can be
        ↪ omitted
6        (if (my-member-rec (first lsta) lstb)
7            (my-union-rec (rest lsta) lstb)
8            (my-union-rec (rest lsta) (cons (first lsta) lstb)))))

```




Ejemplos del uso de esta función se muestran a continuación:

```

1  > (my-union-rec '(1 2 3 4) '(2 3 4 5))
2  '(1 2 3 4 5)
3  > (my-union-rec '(1 2 3 4) '())
4  '(1 2 3 4)
5  > (my-union-rec '() '())
6  '()

```



#### ■ Diferencia:

```

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

```


#### Solución:

La diferencia  $A - B$  en `my-difference-rec` se muestra a continuación:

```

1  ;; my-difference-rec recursively computes the difference between two sets,
2  ;; "A difference B".
3  (define (my-difference-rec lsta lstb)
4    (if (empty? lstb)
5        lsta
6        (if (my-member-rec (first lstb) lsta)
7            (my-difference-rec (my-remove-rec (first lstb) lsta) (rest lstb))
8            (my-difference-rec lsta (rest lstb)))))

```



Aquí lo que se busca es quitar de la primera lista `lsta` los elementos que comparta con la segunda lista `lstb`. Se itera recursivamente por los elementos de `lstb` para verificar si alguno de ellos es miembro de `lsta`, y si es el caso entonces se quita

de `lsta`, y se llama recursivamente a `my-difference-rec` con el resto de `lstb`. Si dicho elemento no es miembro entonces se llama nuevamente la función con el resto de la segunda lista `lstb`, hasta quedar vacía.

Finalmente, ejemplos de la diferencia se incluyen a continuación:

```
1 > (my-difference-rec '(1 2 3 4) '(2 3 4 5))
2 '(1)
3 > (my-difference-rec '(1 2 3) '(1 4 5))
4 '(2 3)
5 > (my-difference-rec '(2 3 4 5) '(1 2 3 4))
6 '(5)
7 > (my-difference-rec '(1 4 5) '(1 2 3))
8 '(4 5)
```



Nótese que el orden importa para esta operación, no es lo mismo  $A - B$  que  $B - A$ .

Correcto

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

#### Solución:

La diferencia que se podría mencionar primero, pero que resulta trivial, es la diferencia en la sintaxis de ambos lenguajes de programación. Trivial en el sentido de que resulta evidente, al ver ambos lenguajes, que tienen sintaxis distintas. Tiende a ser cada vez más interesante que al observar cada vez más las implementaciones de las operaciones sobre conjuntos hechas en `prolog` resultan tener una estructura muy similar a lo que implementé ahora en esta actividad usando `racket` y `sbcl`, en ambos casos se hace un fuerte uso de la recursión.



En `prolog` se usó la recursión y se definieron reglas para lidiar con los casos base y especiales que podrían surgir, aunque el algoritmo usado ahí es el algoritmo de resolución y constantemente se está aplicando resolución y backtracking. Por otra parte, en `lisp` se usa el cálculo lambda. Un buen ejemplo de como funciona el cálculo lambda es mediante lo visto en clase [3] respecto al lenguaje `AL`, y resulta ser la forma en que los lenguajes de programación dentro de la familia `lisp` realizan las evaluaciones sobre una sencilla estructura de datos como lo son las listas.



A pesar de estas diferencias internas, si se comparan las implementaciones de los ejercicios de mis respuestas solamente, se vería que las implementaciones recursivas de los incisos de esta actividad son muy similares a las implementaciones hechas en `prolog`. Aquí no se hizo uso de funciones auxiliares como si se hace en `prolog` para definir reglas de casos específicos como ya se había mencionado, para después dejar a `prolog` hacerlo suyo, búsqueda y resolución.



- 15/15 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]

Es como si en Prolog, tu definieras conceptos y estos te ayudarían a computar valores que satisfacen las Definiciones, vía resolución (con unificación y backtracking). Aquí defines algoritmos, al menos que seas más funcional y entonces aplicas y/o compones funciones para obtener los resultados deseados.



**Solución:**

El capítulo contiene tips bastante interesantes respecto a como utilizar la macro `loop`, tanto es así que asemeja mucho a lo que se realizan en lenguajes no funcionales, como por ejemplo en `julia` donde se aconseja realizar los procesos de una manera iterativa al estilo de `C`; los `for-loop`'s son rápidos en `julia`.

Justo esa es la idea de loop.

Decidí resolver el problema del inciso 6 de esta actividad usando la macro `loop`, dicha solución se mostrará en el siguiente inciso. Mientras tanto, también se reimplementaron algunas de las operaciones entre conjuntos del inciso 3 con esta macro.

En contraste con el inciso 3 donde se usó `racket`, de aquí en adelante se usó `sbcl`. Las razones de esto es que la macro que se necesita usar no existe en `racket` como tal, para esto es necesario utilizar una librería adicional para la cual habría sido necesario revisar su documentación más a fondo y así asegurarme de que no habría cambios en el comportamiento de la macro. Por otra parte, las macros en `common lisp` (`sbcl`) no son equivalentes a las macros en `racket`. Además, el libro recomendado [4] para este inciso contiene muy buenas explicaciones para `common lisp` las cuales no son todas directamente aplicables a `racket`.

ok

Y créeme que se hecha de menos cuando usas mucho loop.

La primera función que se implementó con esta macro fue una función auxiliar:

```
1 ;; my-member operation to check whether a value is member of a list at least
   ↪ once.
2 (defun my-member (val lst)
3   (loop for item in lst thereis (equal val item)))
```



Encontré interesante que exista la palabra clave `thereis` la cual funciona como una cláusula de terminación pronta para salir del loop, en el momento en el que la condición evalúa un valor no nulo (non-nil) el ciclo termina regresando ese valor. En este caso el valor de retorno es un booleano, `t` si un valor se encuentra dentro de la lista y `nil` si no es el caso. Por lo tanto, se itera sobre los elementos de la lista hasta que da con el valor buscado o se llega al final de la lista, y se sale del loop antes si lo encuentra.

Ahora, con esta función `my-member` decidí implementar nuevamente la operación  $A \subseteq B$ . La función que implementa esto se muestra a continuación, `my-subset`.

```
1 ;; my-subset operation equivalent to that done in racket, and making use of loop
2 (defun my-subset (lsta lstb)
3   (loop for item in lsta
4     always (my-member item lstb)))
```



Nótese que se usa nuevamente la macro `loop` pero ahora en conjunto con otra cláusula de terminación, `always`, que termina tempranamente el loop si es que existe alguna evaluación que no cumpla con la condición establecida en cuyo caso corresponde a que todos los elementos de la lista `lsta` siempre sean miembros en la lista `lstb`.


De manera similar traté de hacer una segundo intento para implementar esta misma operación. Lo que se hace en `my-subset2` es generar una lista dentro del loop de los

elementos de la lista `lsta` dentro de la lista `lstb`. Si la lista generada es igual a la lista original `lsta`, entonces se regresa `t`, en caso contrario regresa `nil`.

```

1  ;; my-subset2 just collects the items from a list lsta that appear in another
   ↪ list
2  ;; lstb, if those items are the same as those in lsta then lsta is subset of
   ↪ lstb.
3  ;; Although, the function my-subset is cleaner, more lispy even?
4  (defun my-subset2 (lsta lstb)
5    (if
6      (equal
7        lsta
8        (loop for item in lsta when (my-member item lstb) collect item))
9      t))

```




Estas funciones escritas en `sbcl` se muestran en acción a continuación:

```

1  * (my-subset '(1 3) '(1 2 3 4))
2  T
3  * (my-subset2 '(1 3) '(1 2 3 4))
4  T
5  * (my-subset '() '(1 2))
6  T
7  * (my-subset2 '() '(1 2))
8  T
9  * (my-subset '(1 7) '(1 2 3 4))
10 NIL
11 * (my-subset2 '(1 7) '(1 2 3 4))
12 NIL

```



- 
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]

```

1  > (repeat 3 (print 'hi))
2  HI
3  HI
4  HI
5  NIL

```

### Solución:

Jugando con la macro `loop` se escribieron dos funciones. Cabe resaltar que la primera no es idea propia, surgió como resultado de las discusiones entrabladas en grupo, y se muestra a continuación:

```

1  ;; One way to define the repeat macro using a loop and the for keyword
2  (defmacro myrepeat (reps expr)

```

```
3  `(loop for i from 1 to ,reps
4      do (eval expr)))
```



Pero esta parte debería estar protegida como se indica. Backquote, coma.

La segunda, cambiando la cláusula `for` por `repeat` dentro de la macro `loop`:

```
5  ;; The repeat macro using a loop and the repeat keyword
6  (defmacro myrepeat2 (reps expr)
7    (loop repeat reps
8          do (eval expr)))
```



Revisa como se macro expanden las dos versiones.

Ambas implementaciones se hicieron con la macro `loop` para satisfacer los requerimientos del inciso 5 en conjunto de este. Ejemplos de las macros `myrepeat` y `myrepeat2` se muestran a continuación:

```
1  * (myrepeat 3 (print 'hi))
2
3  HI
4  HI
5  HI
6  NIL
7  * (myrepeat2 3 (print 'hi))
8
9  HI
10 HI
11 HI
12 NIL
```



15/15 7. La siguiente función me permite definir una entrada en un registro de mis libros:

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



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

```
1  (defvar *db* nil)
2
3  (defun agregar-reg (libro)
4    (push libro *db*))
```

De forma que:

```
1  > (agregar-reg (crea-libro "Pericia Artificial" "Alejandro Guerra" "UV" 90.50))
2  ((:TITULO "Pericia Artificial" :AUTOR "Alejandro 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]:

```

1 > (listado-db)
2 TITULO:    Pericia Artificial
3 AUTOR:    Alejandro Guerra
4 ED:       UV
5 PRECIO:    90.50

```


### Solución:

Un ejemplo muy similar a este inciso y al octavo se puede encontrar en el libro de referencia [4], adaptándolo se llega a la siguiente función:

```

1 ;; Muestra la base de datos dandole formato para facilitar la lectura
2 (defun muestra-bd ()
3   (dolist (libro *base-de-datos*)
4     (format t "~{a:~10t~a~%~%}" libro)))

```




Incluso se añade una alternativa sin usar `dolist`:

```


1 ;; Definicion alternativa para formatear la base de datos
2 (defun muestra-bd2 ()
3   (format t "~{~{a:~10t~a~%~%}~%}" *base-de-datos*))

```



Este format tampoco lo  
tienes en Racket

Para esta alternativa, `format` consume listas dentro de la lista de listas que compone a la base de datos, el primer `a` consume las propiedades de cada sublista y el segundo `a` consume el valor asociado a dicha propiedad. El `t` que se encuentra en medio no consume ningún valor, solamente indica que incluya cierta cantidad de espacios entre la propiedad y su valor al mostrarlas en el `repl`. Esta notación `~{ ... ~}` indica el consumir un elemento de una lista, entonces `~{~{ ... ~}~}` es consumir elementos para cada sublista dentro de una lista hasta consumir todas las sublistas.



Se lanza `sbc1` en terminal, y se carga el programa en el `repl`:

```

1 * (load "bookrecord.lsp")
2 T


```

Después de cargar el archivo se agregaron los libros como se muestra a continuación:

```

3 * (agregar-reg (crea-libro "The catcher in the rye" "Jerome David Salinger"
4   ↪ "Alianza Editorial" 175.0))
5 (:TITULO "The catcher in the rye" :AUTOR "Jerome David Salinger" :EDIT
6   "Alianza Editorial" :PRECIO 175.0)
7 (:TITULO "Kitchen" :AUTOR "Banana Yoshimoto" :EDIT "TusQets" :PRECIO 150.5)
8 (:TITULO "Rebellion en la granja" :AUTOR "George Orwell" :EDIT "Debolsillo"
9   :PRECIO 200.0)
10 (:TITULO "El cuento veracruzano" :AUTOR "Luis Leal" :EDIT "UV" :PRECIO 50.0)
11 (:TITULO "Pericia Artificial" :AUTOR "Alejandro Guerra" :EDIT "UV" :PRECIO
    90.5))

```



Posteriormente, se muestra la salida de la base de datos de los libros incluidos hasta ahora con el formato requerido para este inciso:

```

12 * (muestra-bd)
13 TITULO:  The catcher in the rye
14 AUTOR:   Jerome David Salinger
15 EDIT:    Alianza Editorial
16 PRECIO:  175.0
17
18 TITULO:  Kitchen
19 AUTOR:   Banana Yoshimoto
20 EDIT:    TusQets
21 PRECIO:  150.5
22
23 TITULO:  Rebellion en la granja
24 AUTOR:   George Orwell
25 EDIT:    Debolsillo
26 PRECIO:  200.0
27
28 TITULO:  El cuento veracruzano
29 AUTOR:   Luis Leal
30 EDIT:    UV
31 PRECIO:  50.0
32
33 TITULO:  Pericia Artificial
34 AUTOR:   Alejandro Guerra
35 EDIT:    UV
36 PRECIO:  90.5
37
38 NIL
    
```

Checa lo que defstruct puede hacer por ti. Define estructuras como las de cada libro, pero te da gratis las funciones de acceso y el constructor:

(defstruct libro ...)

Te crea automáticamente:

(make-libro ...)

Y

(libro-autor ...)

Etc,



10/10

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

### Solución:

Este inciso es una continuación del inciso anterior. Igualmente, me basé en el ejemplo del capítulo 3 del libro de referencia escrito por Seibel [4]. A continuación se muestra una llamada a la función `selecciona-por-autor` la cual filtra la base de datos y regresa las entradas que corresponden a dicho autor:

```

39 * (selecciona-por-autor "Banana Yoshimoto")
40 ((:TITULO "Kitchen" :AUTOR "Banana Yoshimoto" :EDIT "TusQets" :PRECIO 150.5))
    
```



La implementación de `selecciona-por-autor` escrita para mostrar las entradas dentro del registro de un autor dado es la siguiente:

```

1 ;; Busca en la base de datos los libros que corresponden al autor de interes
2 (defun selecciona-por-autor (autor)
3   (remove-if-not
    
```



```
4  #'(lambda (libro) (equal (getf libro :autor) autor))
5  *base-de-datos*)
```

Lo que hace ese `remove` de la base de datos aquellos elementos que no cumplen con la condición establecida por la función- $\lambda$ . Dicha función `lambda` busca para cada una de las entradas de la base de datos su autor correspondiente, y si el valor de la propiedad `:autor` de dicha entrada coincide con el valor dado por el usuario la remueve. De hecho, si se hace la llamada de `muestra-bd` para checar las entradas, sigue teniendo las mismas, en realidad el `remove-if-not` no altera la base de datos.



---

## Referencias

- [1] Matthew Butterick. Beautiful racket: An introduction to language-oriented programming using racket. <https://beautifulracket.com/>, 2016. Visitado: 2022-11-26.
- [2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. How to design programs. <https://htdp.org/from-first-to-second-edition.html>, 2018. Visitado: 2022-11-25.
- [3] Alejandro Guerra-Hernandez. Programación para la inteligencia artificial. <https://www.uv.mx/personal/aguerra/pia/>, 2022. Visitado: 2022-10-07.
- [4] Peter Seibel. *Practical common lisp*. Apress, 2006.