PRACTICA 3 - CRA:

Scheme - Listas

Blanca Calderón González

Franck Michael Fierro Chicaiza

Kevin Orlando Cancio Fernández

índice

Reparto de tareas

Tarea	Descripción	Miembro encargado
Requisitos obligatorios		
Codificar la operación de concatenación de listas	Operación que une dos listas sin cambiar su orden simplemente pone la segunda al final de la primera	Blanca Calderón Franck Michael Fierro
Codificar la operación de longitud de listas	Se ha codificado una operación que al pasarle una lista devuelva su longitud	Blanca Calderón Franck Michael Fierro
Codificar la operación de inversión	Operación que da la vuelta a una lista (invierte), recibe la original y devuelve su inversa	Blanca Calderón Franck Michael Fierro
Codificar la operación de pertenencia	Operación que indica si un elemento pertenece a la lista dada devolviendo true si lo hace.	Blanca Calderón Franck Michael Fierro
Codificar operación de suma de los elementos de una lista	Dada lista recibida suma todos los elementos de esta devolviendo la suma total.	Blanca Calderón Franck Michael Fierro
Codificar operación que calcule el máximo y mínimo	Dos funciones, una que devuelve el máximo y otra el mínimo dada una lista.	Blanca Calderón Franck Michael Fierro
Memoria	Desarrollo de una memoria explicativa del código desarrollado para la práctica	Blanca Calderón Franck Michael Fierro Kevin Orlando Cancio Fernández
Mejoras		
Operación que muestra por pantalla el contenido de una lista	Muestra los elementos de una lista por la consola.	Blanca Calderón Franck Michael Fierro
Operación que sume dos listas del misma tamaño	Dada dos listas recibidas del mismo tamaño sumar sus elementos dos a dos como si fuese suma de vectores devolviendo el resultado en otra lista	Blanca Calderón Franck Michael Fierro
Operación que devuelve el	Dada una lista y un número	Blanca Calderón

elemento de la lista que se encuentra en la posición indicada	(posición que queremos saber) se devuelve el número que se encuentra en la posición buscada	Franck Michael Fierro
Operación que devuelve los n primeros elementos de una lista	Dada una lista y un número que indica la posición hasta la cual queremos saber los elementos de la lista. Se devuelve lista formada por los elementos de la lista hasta la posición indicada	Blanca Calderón Franck Michael Fierro
Operación que devuelve los últimos elementos de una lista a partir de la posición indicada	Dada una lista y un número que indica la posición tras la cual queremos que nos devuelva la lista devuelve todos los elementos que están después de esta posición.	Blanca Calderón Franck Michael Fierro
Operación que elimina de una lista todas las apariciones del elemento dado	Dada una lista y el elemento que se desea eliminar se devuelve la lista quitando todas las apariciones de ese elemento	Blanca Calderón Franck Michael Fierro
Operación que elimina elemento de la lista que se encuentra en la posición indicada	Dada una lista y la posición cuyo elemento se desea borrar se devuelve la lista quitando el elemento que se encuentra en la posición dada	Blanca Calderón Franck Michael Fierro
Operación que ordena una lista de menor a mayor	Dada una lista la ordena de menor a mayor devolviendola ordenada	Blanca Calderón Franck Michael Fierro
Interfaz	Interfaz en la que se elijan dos listas de las presentadas como opciones con las que hacer las distintas operaciones definidas	Kevin Orlando Cancio Fernández

Requisitos obligatorios

Se han implementado al 100% los requisitos obligatorios de la práctica llevando a cabo la programación de todas las operaciones solicitadas.

Las listas de ejemplo usadas en las operaciones son:
(define lista1 ((const cinco) ((const tres) ((const dos) nil))))
(define lista2 ((const siete) ((const dos) ((const cinco) ((const uno) ((const dos) nil))))))
(define lista3 ((const dos) ((const dos) ((const cuatro) ((const cinco) ((const cero) nil))))))

Se han programado las siguientes funciones base usadas en varias operaciones de la práctica:

```
(define nil (lambda (z) z))
```

```
(define null primero)

(define const (lambda (x)
                (lambda (y)
                  ((par false) ((par x) y)))))

(define hd (lambda (z)
          (primero(segundo z))))

(define tl (lambda (z)
          (segundo (segundo z))))
```

Estas son lista vacía (null), el constructor de las listas (const) en el cual se construyen listas en pares, obtener la cabeza de la lista (hd) y obtener la cola (hd).

1º Operaciones usuales en listas

Se pueden probar todas las operaciones con el comando test.

- ## Codificar la operación de concatenación de listas

  Se ha programado la función **concatenar**, la cual recibe dos listas(l1 y l2) y las devuelve uniendo la segunda unida a la primera.

  Para simular la recursividad se ha hecho uso de la función auxiliar **concatenaraux** la cual también recibe las dos listas y es llamada por concatenar cuando la primera lista no es vacía (sigue habiendo elementos que concatenar).
  En **concatenaraux** si la primera lista (l1) está vacía (null) se termina la ejecución devolviendo la segunda lista (l2) ya que ya se han concatenado todos los elementos de la primera lista en la segunda.
  Si la lista 1 no es vacía se lleva a cabo la llamada a concatenar de nuevo en la que se usa el constructor **(const)** de las listas definido anteriormente en el programa para formar una lista con la cabeza de la primera lista y la llamada a concatenar pasandole la cola de la primera lista y la segunda lista uniendo de esta forma las dos listas con cada llamada.

  El código completo es el siguiente:

```
(define concatenar (lambda (l1)
                (lambda (l2)
                  (((null l1) (lambda (no_use) l2) (lambda (no_use) ((concatenaraux  l1)
l2))) zero) )))

(define concatenaraux (lambda (l1)
                  (lambda (l2)
                    ((Y (lambda (f)
                          (lambda (x)
```

```
                ((
                  (null x)
                  (lambda (no_use)
                    l2
                    )
                  (lambda (no_use)
                    ((const (hd x)) ((concatenar (tl x)) l2))
                    )
                  )
                 zero)
                )
               ))
 l1)
 )))
```

Para llamar a esta función se debe ejecutar el siguiente **comando** → ((concatenar lista1) lista2)

y para mostrar la lista resultante → (mostrar ((concatenar lista1) lista2))

El resultado obtenido es : 5 3 2 7 2 5 1 2

- Codificar la operación de longitud de listas

La siguiente operación codificada es la usada para calcular la longitud de una lista dada (l). Para ello se ha programado la función **longitud** la cual recibe una lista y si esta no es vacía llama a **longitudaux** para simular la recursividad y recorrer la lista.

En **longitudaux** también se revisa si la lista es vacía (null l) y si lo es se termina devolviendo zero. Si no está vacía se recorre la lista llamando a longitud con la cola de la lista para seguir recorriendo la lista y sumando uno. De esta forma cuando se termina de recorrer la lista se devuelve el número de llamadas para recorrer la lista lo cual corresponde con la longitud de la lista.

El **código completo** es el siguiente:

```
(define longitud (lambda (l)
                (((null l) (lambda (no_use) zero) (lambda (no_use) (longitudaux l)))
zero)))

(define longitudaux (lambda (l)
                ((Y (lambda (f)
                    (lambda (x)
                      ((
                        (null x)
                        (lambda (no_use)
                          zero
                          )
                        (lambda (no_use)
```

```
            (sucesor (longitud (tl x)))
            )
          )
         zero)
        )
      ))
  l)
 ))
```

El **comando** para ejecutar esta operación es → (comprobar (longitud lista1))
El resultado de ejecutarlo con lista1 es: 3

- Codificar la operación de inversión

Para invertir una lista dada se ha programado el método **inversion** el cual recibe una lista y si no es vacía llama a su función auxiliar **inversionaux** en la cual si la lista es vacía devuelve nil (indica final de una lista) y si no hace haciendo uso de la función **concatenar** explicada anteriormente une la llamada a **inversion** a la que se le pasa la cola de la lista con la lista formada al unir la cabeza con la lista con el elemento nil. Al unir la lista formada con la recursividad con la cabeza de la lista a la derecha se lleva a cabo la inversión ya que el primer elemento se uniría al final y así sucesivamente.

El **código** es el siguiente:

```
(define inversion (lambda (l)
            (((null l) (lambda (no_use) nil) (lambda (no_use) (inversionaux l))) zero)))

(define inversionaux (lambda (l)
          ((Y (lambda (f)
              (lambda (x)
                ((
                  (null x)
                  (lambda (no_use)
                    nil
                    )
                  (lambda (no_use)
                    ((concatenar (inversion (tl x))) ((const (hd x)) nil))
                    )
                  )
                 zero)
                )
              ))
           l)
          ))
```

Para llamar a esta operación se usa el **comando** → (mostrar (inversion lista2))
El resultado del ejemplo con la lista 2 es : 2 1 5 2 7

- Codificar la operación de pertenencia

Esta operación comprueba si un elemento pasado pertenece a la lista dada devolviendo true si pertenece.
Para lograr esto se ha realizado **pertenece** el cual recibe la lista y el elemento y si la lista no está vacía llama a **perceneceaux** para recorrer la lista y comprobar si el elemento pertenece a ella.

En **perteneceaux** revisa si el elemento es igual a la cabeza de la lista y si lo es devuelve true. Si no lo es continúa recorriendo la lista llamando a pertenece con la cola de la lista.
Se sigue recorriendo la lista hasta que se encuentra el elemento caso en el que devuelve true o hasta que la lista está vacía (no se ha encontrado elemento) dónde devuelve false al no encontrar el elemento indicado.

El código completo es el siguiente:

```
(define pertenece (lambda (l)
                    (lambda (e)
                      (((null l) (lambda (no_use) false) (lambda (no_use) ((perteneceaux l)
e))) zero))))

(define perteneceaux (lambda (l)
                       (lambda (e)
                         ((Y (lambda (f)
                               (lambda (x)
                                 ((
                                   ((esigualent (hd l)) e)
                                   (lambda (no_use)
                                     true
                                     )
                                   (lambda (no_use)
                                     ((pertenece (tl x)) e)
                                     )
                                   )
                                  zero)
                                 )
                               ))
                          l)
                         )))
```

El **comando** para ejecutar la operación → ((pertenece lista2) cinco)
En este ejemplo se devolverá true.

2º Usando codificación de los enteros

- Codificar operación de suma de los elementos de una lista

Se ha codificado la suma de todos los elementos de una lista usando función **sumarlista** la cual recibe una lista y si no está vacía llama a **sumarlistaaux** y si lo es termina ejecución devolviendo cero.
En **sumarlistaaux** se comprueba si la lista no está vacía y si lo está se termina devolviendo cero y si no, se suma la cabeza de la lista a la llamada de sumarlista a la que se le pasa la cola de la lista para seguir recorriendola.

El **código** es:

```
(define sumarlista (lambda (l)
                ((((null l) (lambda (no_use) cero) (lambda (no_use) (sumarlistaaux
l))) cero)))

(define sumarlistaaux (lambda (l)
                    ((Y (lambda (f)
                        (lambda (x)
                          ((
                            (null x)
                            (lambda (no_use)
                              cero
                              )
                            (lambda (no_use)
                              ((sument (hd x)) (sumarlista (tl x)))
                              )
                            )
                           cero)
                          )
                        ))
                     l)
                    ))
```

Para ejecutar esta operación se usa **comando** → (testenteros (sumarlista lista2))

El resultado de este ejemplo es: 17

- Codificar operación que calcule el máximo y mínimo

Para realizar esta operación se han usado dos funciones distintas, una para calcular el máximo de la lista y otro para el mínimo, en ambas se pasa la lista y un elemento en el que se guardará el elemento resultado (el máximo de la lista o el mínimos según el caso) que comenzará siendo la cabeza de la lista.

Estas operaciones son:

-Cálculo del máximo de la lista → Se ha programado la función **maxilista** en el cual si la lista no está vacía se llama a **maxlistaaux** en la que se comprueba si la cabeza de la lista es mayor que el elemento pasado (máximo actual) y si lo es se llama a **maxlista** con la cola de la lista y el nuevo máximo (cabeza de la lista).
Si la cabeza no es mayor que el elemento mayor actual se hace la misma llamada pero sin cambiar el elemento pasado.
Cuando se termina de recorrer la lista se devuelve el elemento máximo actual (n).

-Cálculo del mínimo de la lista → Lo mismo que con el máximo pero la condición esta vez es si la cabeza es menor que el elemento mínimo actual y el nombre de las operaciones es **minlista y minlistaaux**.

El **código** es el siguiente:

```
(define maxlista (lambda (l)
                (lambda (n)
                 (((null l) (lambda (no_use) n) (lambda (no_use) ((maxlistaaux l)n)))
cero))))

(define maxlistaaux (lambda (l)
                    (lambda (n)
                     ((Y (lambda (f)
                         (lambda (x)
                           ((
                             ((esmayorent (hd l)) n)
                             (lambda (no_use)
                               ((maxlista (tl l)) (hd l))
                               )
                             (lambda (no_use)
                               ((maxlista (tl l)) n)
                               )
                             )
                            cero)
                           )
                          ))
                      l)
                     )))

(define minlista (lambda (l)
                (lambda (n)
                 (((null l) (lambda (no_use) n) (lambda (no_use) ((minlistaaux l)n)))
cero))))

(define minlistaaux (lambda (l)
                    (lambda (n)
                     ((Y (lambda (f)
                         (lambda (x)
                           ((
                             ((esmenorent (hd l)) n)
```

```
                (lambda (no_use)
                  ((minlista (tl l)) (hd l))
                  )
                (lambda (no_use)
                  ((minlista (tl l)) n)
                  )
                )
               cero)
              )
            ))
      l)
     )))
```

El **comando** para llamar a estas operaciones es:
(testenteros ((maxlista lista1) (hd lista1)))
(testenteros ((minlista lista1) (hd lista1)))

El resultado de este ejemplo sería → max lista1 = 5 y min lista1 = 2.

Mejoras

- ## Codificar operación que muestra los elementos de una lista

  Se ha programado una función que muestra todos los elementos de la lista con el objetivo de ver mejor el resultado de las operaciones anteriores.
  Para ello se han hecho dos funciones siendo éstas **mostrar** que si la lista no es vacía llama a **mostraraux** y si lo es muestra por consola espacio vacío terminando la ejecución.
  En **mostraraux** se muestra por pantalla con el display el elemento de la cabeza de la lista y si la lista no está vacía se llama a mostrar con la cola de la lista.

  El **código** es el siguiente:

```
(define mostrar (lambda (l)
            (((null l) (lambda (no_use) (display "")) (lambda (no_use) (mostraraux l)))
zero)))

(define mostraraux (lambda (l)
            ((Y (lambda (f)
                (display (testenteros(hd l))) (display " ")
                (lambda (x)
                  ((
                    (null x)
                    (lambda (no_use)
                      (display "")
                      )
                    (lambda (no_use)
```

```
                (mostrar (tl x))
                )
               )
              zero)
             )
            ))
        l)
       ))
```

Para ejecutar la operación se usa **comando** → (mostrar lista1)
cuyo resultado es: 5 3 2

- Codificar operación que sume dos listas del mismo tamaño

Operación que dadas dos listas del mismo tamaño suma sus elementos uno a uno como si fueran dos vectores.
Se usa **sumar2listas** en el cual se comprueba si la lista1 no está vacía (como son del mismo tamaño si una está vacía la otra también), si lo está devuelve nil y si no llama a **sumar2listasaux** en el que si la lista1 no está vacía se concatena la suma de la cabeza de la primera lista y de la segunda con la llamada a **sumar2listas** al que se le pasa el resto de ambas listas para seguir recorriendo las.

El código es:

```
(define sumar2listas (lambda (l1)
                       (lambda (l2)
                         (((null l1) (lambda (no_use) nil) (lambda (no_use) ((sumar2listasaux
l1) l2))) zero))))

(define sumar2listasaux (lambda (l1)
                          (lambda (l2)
                            ((Y (lambda (f)
                                  (lambda (x)
                                    ((
                                      (null x)
                                      (lambda (no_use)
                                        nil
                                        )
                                      (lambda (no_use)
                                        ((concatenar ((const ((sument (hd x)) (hd l2))) nil))
((sumar2listas (tl x)) (tl l2)))
                                        )
                                      )
                                     cero)
                                    )
                                  ))
                             l1)
                            )))
```

El **comando** para ejecutarlo es → (mostrar ((sumar2listas lista2) lista3))
El resultado de este ejemplo es: 9 4 9 6 2

- ## Codificar operación que devuelve el elemento de la posición indicada

Dada una lista y la posición cuyo elemento se quiere saber se devuelve el elemento que se encuentra en esa posición. Para ello, se ha programado la operación **buscarelem** que recibe la lista y la posición a borrar y se revisa si esa posición es menor que 0 o si la lista es nula lo que indica que esa posición está fuera de la lista caso en el cual se devuelve -1. Si estas condiciones no se cumplen se llama a **buscarelemaux** el cual revisa si la posición pasada es mayo que 0 en cuyo caso se hace la llamada a buscarelem con el resto de la lista y disminuyendo posición en uno ya que se avanza en la lista, si la posición no es mayor que 0 significa que se ha encontrado la posición y se devuelve el elemento actual de la cabeza el cual corresponde a esa posición terminando la ejecución.
Si se devuelve -1 es que no se ha podido encontrar la posición indicada.

El **código** es el siguiente:

```
(define buscarelem (lambda (l)
                     (lambda (p)
                       (((((esmenorent p) cero) or (null l)) (lambda (no_use) -uno) (lambda
(no_use) ((buscarelemaux l)p))) zero))))

(define buscarelemaux (lambda (l)
                    (lambda (p)
                     ((Y (lambda (f)
                           (lambda (x)
                             ((
                               ((esmayorent p) cero)

                               (lambda (no_use)
                                 ((buscarelem (tl x)) ((restaent p) uno))

                                 )
                               (lambda (no_use)
                                 (hd x)

                                 )
                               )
                              zero)
                             )
                           ))
                      l)
                     )))
```

Para ejecutarlo se hace uso del **comando:** (testenteros ((buscarelem lista2) dos))

Cuyo resultado es: 2

- ## Codificar operación que devuelve elementos hasta llegar a la posición indicada

Operación que devuelve los primeros elementos de la lista hasta llegar a la posición indicada (n).
Se hace uso de la operación **obtenerinicio** la cual recibe la lista y la posición hasta donde se quiere devolver comprobando que la lista no sea nula o que la posición no sea cero ya que si lo es se devuelve nil y se termina. Si no se dan estas condiciones se llama a **obtenerinicioaux** en el cual se revisa si la posición pasada es mayor que 0 en cuyo caso se llama a obtenerinicio con el resto de la lista y posición menos uno para seguir recorriendola concatenando a su vez la cabeza de la lista para ir guardando los elementos a devolver. Se termina cuando la posición no es mayor que 0 (se ha llegado a la posición indicada) en cuyo caso se devuelve nil devolviendo al final los primeros elementos concatenados en una lista.

El **código** es:

```
(define obtenerinicio (lambda (l)
                        (lambda (p)
                          (((((esmenorent p) cero) or (null l)) (lambda (no_use) nil) (lambda
(no_use) ((obtenerinicioaux l) p))) zero))))

(define obtenerinicioaux (lambda (l)
                           (lambda (p)
                             ((Y (lambda (f)
                                   (lambda (x)
                                     ((
                                       ((esmayorent p) cero)

                                       (lambda (no_use)
                                         ((concatenar ((const (hd l)) nil)) ((obtenerinicio (tl l))
((restaent p) uno)))
                                       )
                                       (lambda (no_use)
                                         nil
                                       )
                                      )
                                     zero)
                                    )
                                   ))
                              l)
                             )))
```

El **comando** para su ejecución es: (mostrar ((obtenerinicio lista2) tres))
y su resultado : 7 2 5

- ## Codificar operación que devuelve elementos que se encuentran después de la posición indicada

  Se ha codificado operación **obtenerfinal** que es igual que obtenerinicio pero llamando a **obtenerfinalaux** en la cual en vez de concatenar la cabeza de la lista cuando la posición es mayor que 0 solo hace la llamada a obtenerfinal con el resto de la lista y la posición restada de forma que no guarde los primeros elementos. Cuando la posición ya no es mayor que cero significa que se ha encontrado la posición por lo que se devolvería lo que queda de la lista lo que corresponde a los elementos que se encuentran después de la posición indicada.

  El **código** es:

```
(define obtenerfinal (lambda (l)
                       (lambda (p)
                        (((((esmenorent p) cero) or (null l)) (lambda (no_use) nil) (lambda
(no_use) ((obtenerfinalaux l) p))) zero))))

(define obtenerfinalaux (lambda (l)
                          (lambda (p)
                           ((Y (lambda (f)
                                 (lambda (x)
                                   ((
                                     ((esmayorent p) cero)

                                     (lambda (no_use)
                                       ((obtenerfinal (tl x)) ((restaent p) uno))

                                       )
                                     (lambda (no_use)
                                       x
                                       )
                                     )
                                    zero)
                                   )
                                 ))
                            l)
                           )))
```

  El **comando** para ejecutarlo: (mostrar ((obtenerfinal lista2) dos))
  Su resultado: 5 1 2

- ## Codificar operación que elimina todas las apariciones del elemento pasado de la lista

  Se usa operación **eliminarelem** en el cual se comprueba si la lista pasada está vacía y si lo está devuelve nil, si no llama a **eliminarelemaux** en el igual se revisa si el elemento pasado es igual a la cabeza de la lista en cuyo caso se hace llamada a eliminarelem con el resto de la lista para no guardar ese elemento y así eliminarlo de la lista resultante. Si el elemento no es igual se concatena para guardarlo y se llama

a eliminarelem con el restos de elementos, de esta forma se eliminan todas las apariciones del elemento al solo guardar los elementos distintos a éste.

El **código** es:

```
(define eliminarelem (lambda (l)
                       (lambda (p)
                        (((null l) (lambda (no_use) nil) (lambda (no_use) ((eliminarelemaux
l) p))) zero))))

(define eliminarelemaux (lambda (l)
                          (lambda (p)
                           ((Y (lambda (f)
                                 (lambda (x)
                                  ((
                                    ((esigualent p) (hd x))

                                    (lambda (no_use)
                                     ((eliminarelem (tl x)) p)

                                     )
                                    (lambda (no_use)
                                     ((concatenar ((const(hd x)) nil)) ((eliminarelem (tl x)) p))
                                    )
                                   zero)
                                  )
                                 ))
                            l)
                           )))
```

El **comando** para ejecutar operación es: (mostrar ((eliminarelem lista3) dos))
Cuyo resultado es: 4 5 0

- ## Codificar operación que elimina elemento en la posición indicada

Se ha programado **eliminarpos** el cual recibe la lista y la posición a borrar, se comprueba si la posición pasada es menor que cero o si la lista no es nula para ver si nos hemos salido de los límites de la lista y si es el caso devuelve nil y si no llama a **eliminarposaux** dónde se revisa si la posición es mayor que cero lo que significa que todavía no se ha encontrado posición aborrar por lo que se llama a eliminarpos con el resto de la lista y disminuyendo la posición concatenando además la cabeza de la lista para guardarnos el elemento. Si la posición no es mayor que cero es que se ha encontrado la posición por lo que se elimina el elemento simplemente devolviendo el resto de la lista sin él.
Destacar que la cuenta de las posiciones comienza en cero.

El **código** es:

```
(define eliminarpos (lambda (l)
                      (lambda (p)
```

```
                    (((((esmenorent p) cero) or (null l)) (lambda (no_use) nil) (lambda
(no_use) ((eliminarposaux l) p))) zero))))

(define eliminarposaux (lambda (l)
                    (lambda (p)
                     ((Y (lambda (f)
                          (lambda (x)
                           ((
                             ((esmayorent p) cero)

                             (lambda (no_use)
                               ((concatenar ((const(hd x)) nil)) ((eliminarpos (tl x))
((restaent p) uno)))
                             )
                             (lambda (no_use)
                               (tl x)
                               )
                             )
                            zero)
                           )
                          ))
                      l)
                     )))
```

El **comando** para ejecutarlo es: (mostrar ((elimarpos lista2) dos))
El resultado: 7 2 1 2

- ## Codificar operación que ordena una lista de menor a mayor

Dada una lista que recibe la ordena de menor a mayor usando **ordenacion** el cual recibe la lista original y la lista ordenada comprobando si la lista original está vacía se termina ordenación y se devuelve la lista ordenada (l2), si no está vacía se llama **ordenacionaux** el cual revisa si la lista ordenada está vacía y si lo está llama a ordenacion con el resto de la lista introduciendo la cabeza de la lista en la ordenada.Si no está vacía se llama a ordenacion con el resto de la lista y a **colocar** pasandole la lista ordenada y la cabeza de la lista.

En colocar se recibe la lista dónde se ordena y el elemento mandado desde ordenacionaux y su la lista está vacía se crea una nueva lista con el elemento y si no llama a **colocaraux** en el cual se comprueba si el elemento pasado es mayor que la cabeza de la lista y si lo es se concatena la cabeza con la llamada a colocar con el resto de la lista de forma que la cabeza que es menor quede delante del elemento ordenando así de menor a mayor. Si el elemento es menor se concatena al principio de la lista ordenada.

El **código completo** es el siguiente:

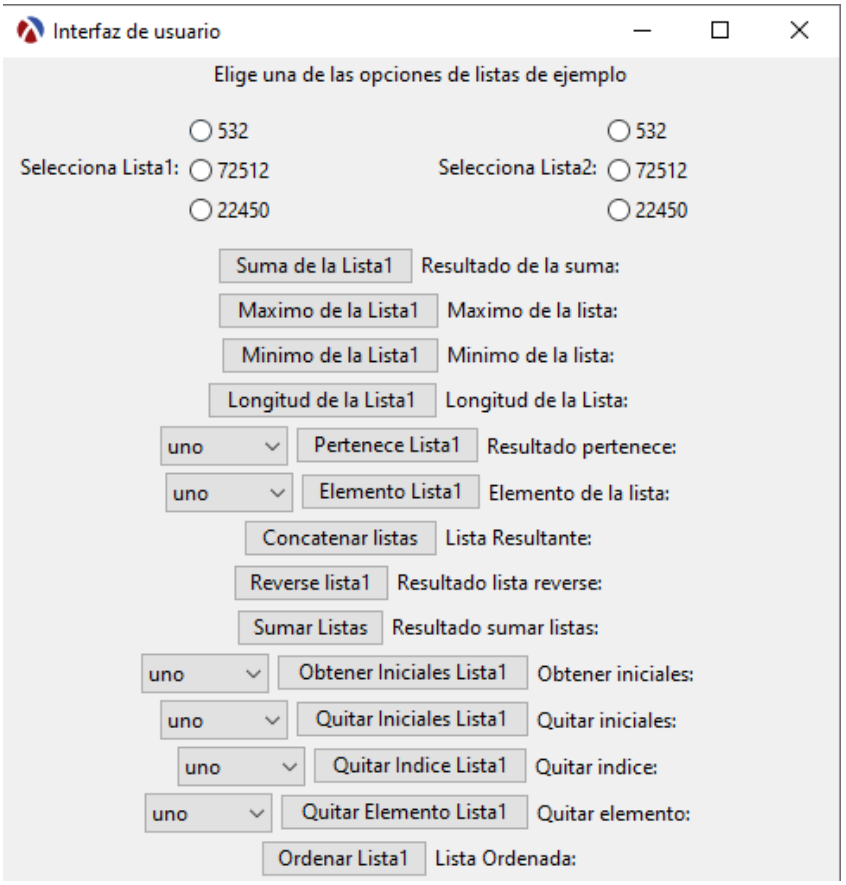
```
(define ordenacion (lambda (l1)
                    (lambda (l2)
                     (((null l1) (lambda (no_use) l2) (lambda (no_use) ((ordenacionaux
l1) l2))) zero))))
```

```
(define ordenacionaux (lambda (l1)
                (lambda (l2)
                  ((Y (lambda (f)
                        (lambda (x)
                          ((
                            (null l2)
                            (lambda (no_use)
                              ((ordenacion (tl x)) ((const(hd l1)) l2))
                              )
                            (lambda (no_use)
                              ((ordenacion (tl x)) ((colocar l2) (hd x)))
                              )
                            )
                           zero)
                          )
                        ))
                   l1)
                  )))


;-----------------------------------------------------------------------------------------------------------
(define colocar (lambda (l)
                  (lambda (p)
                    (((null l) (lambda (no_use) ((const p) nil)) (lambda (no_use)
((colocaraux l) p))) zero))))


(define colocaraux (lambda (l)
                     (lambda (p)
                       ((Y (lambda (f)
                             (lambda (x)
                               ((
                                 ((esmayorent p) (hd x))
                                 (lambda (no_use)
                                   ((concatenar ((const(hd x)) nil)) ((colocar (tl x)) p) )
                                   )
                                 (lambda (no_use)
                                   ((concatenar ((const p) nil)) x)
                                   )
                                 )
                                zero)
                               )
                             ))
                        l)
                       )))
```

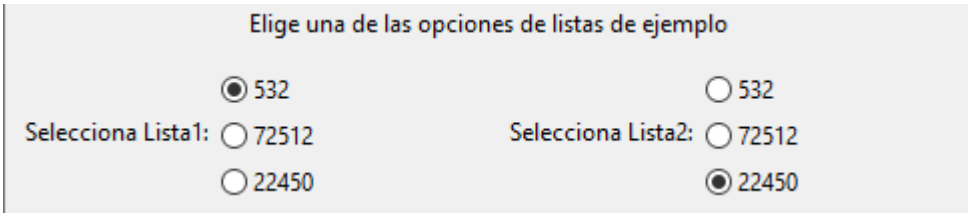
El **comando** para ejecutarlo es: (mostrar ((ordenacion lista2) nil))
Su resultado: 1 2 2 5 7

- Interfaz

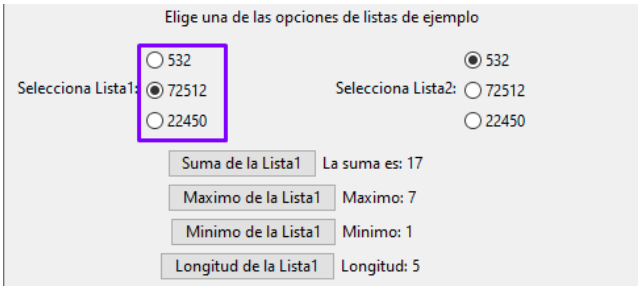
Se ha implementado una interfaz gráfica para hacer uso de todas las funciones programadas en la práctica. Para su realización se ha usado la librería “racket gui” y se ha establecido un frame “padre” de 500x500:



Su funcionamiento es mostrar el resultado de las diferentes operaciones que realiza el programa, combinando las listas de ejemplo que tiene introducidas. Para las operaciones con dos listas hace combinación de la lista seleccionada con el botón en “Lista1” y la lista seleccionada con el botón en “Lista2”. Para las operaciones con una sola lista, usa como referencia la lista seleccionada en “Lista1”:

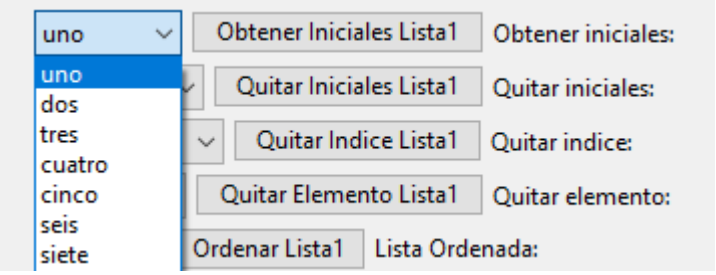


Para crear estos botones de selección se ha usado un “radio checkbox” de la librería “gui”.

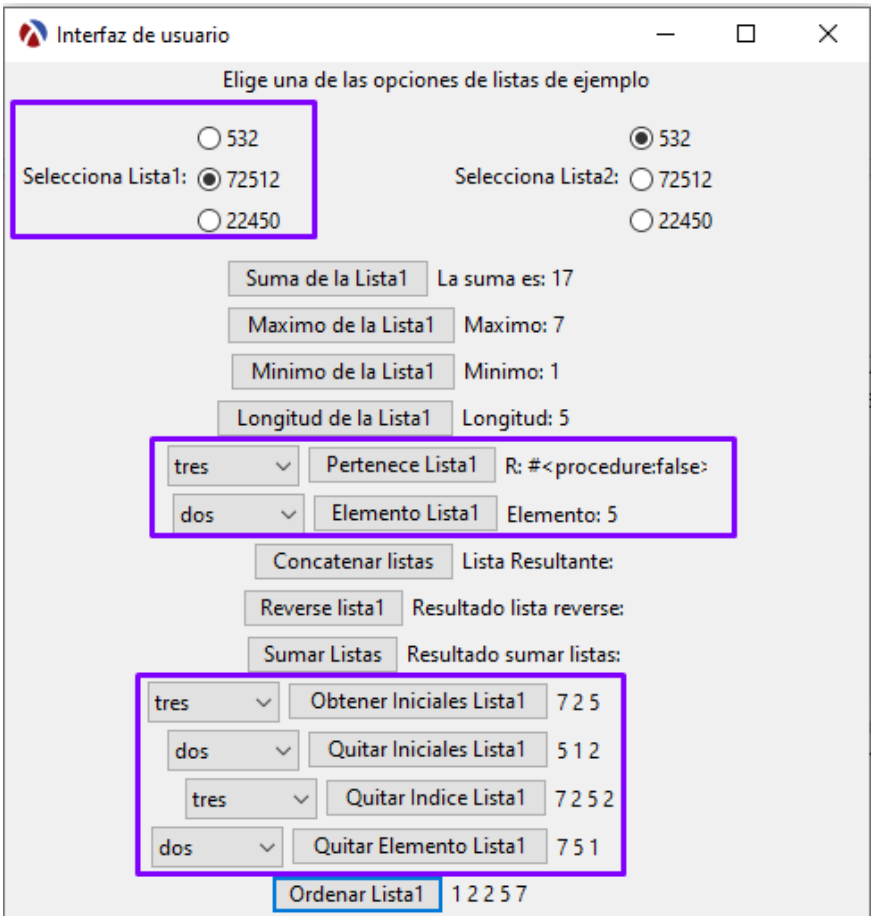


Ejemplo funcionamiento operaciones con 1 lista

Para las operaciones que hagan uso de un índice o número específico elegido por el usuario, se hace uso de un botón tipo “choice”, que permite elegir entre los diferentes números:

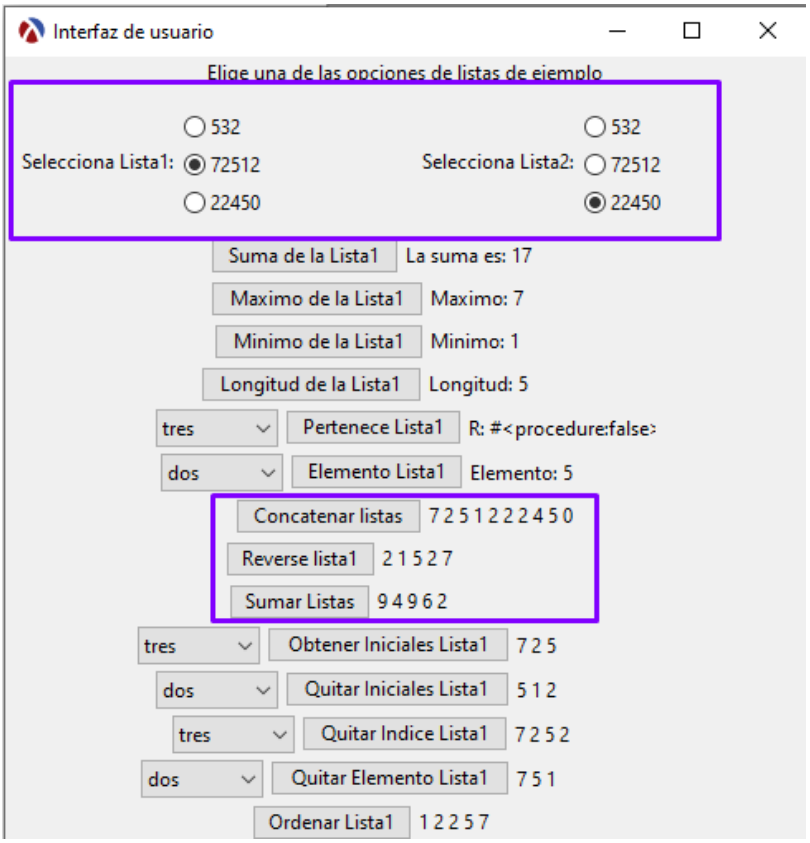


Para que funcione ese número elegido con los diferentes predicados, es necesario hacerle un cast de string a symbol para que pueda funcionar y ser evaluado.



Ejemplo funcionamiento operaciones con selector de número

Para las operaciones con dos listas, como se ha comentado, se hace combinación de la lista elegida en “Lista1” y “Lista2”, mediante la interacción de estas dos, se pueden realizar las distintas funciones programadas:



Ejemplo funcionamiento operaciones con dos listas

Para realizar la ordenación de los elementos visuales, se ha dotado de un identificador a cada objeto botón/choice/radio checkbox/message para poder agruparlos en una misma línea horizontal y mostrar el resultado de cada operación. Para agruparlos en la misma línea horizontal el grupo de botón/message se ha hecho uso de una estructura denominada “spacer” que son separadores horizontales que ayudan a la agrupación de elementos. Todos los elementos siguen una estructura similar:

```
;-------------------- CONCATENAR LISTAS -------------------------
(define spacer7 (new horizontal-panel% [parent frame] [min-width 10][alignment '(center cente
(new button% [parent spacer7] ;Boton para seleccionar la longitud de una lista
     [label "Concatenar listas"]
     [callback (lambda (button event)
                 (cond
                   [(and (= (send rb get-selection) 0) (= (send rb2 get-selection) 0)) (beg
                   [(and (= (send rb get-selection) 0) (= (send rb2 get-selection) 1)) (beg
                   [(and (= (send rb get-selection) 0) (= (send rb2 get-selection) 2)) (beg

                   [(and (= (send rb get-selection) 1) (= (send rb2 get-selection) 0)) (beg
                   [(and (= (send rb get-selection) 1) (= (send rb2 get-selection) 1)) (beg
                   [(and (= (send rb get-selection) 1) (= (send rb2 get-selection) 2)) (beg

                   [(and (= (send rb get-selection) 2) (= (send rb2 get-selection) 0)) (beg
                   [(and (= (send rb get-selection) 2) (= (send rb2 get-selection) 1)) (beg
                   [(and (= (send rb get-selection) 2) (= (send rb2 get-selection) 2)) (beg
                   )
 )])

(define msg8 (new message% [parent spacer7]; Msg
                           [label "Lista Resultante: "]))

;-----------------------------------------------------------
```

Para poder mostrar el resultado de operaciones que involucran dos listas o que devuelven una lista como resultado, ha sido necesario crear una nueva estructura denominada “mostrar2(...)” que es capaz de transformar el resultado obtenido a string, para poder ser mostrado en la interfaz, ya que string es el tipo de dato que usa la “gui” para mostrar mensajes.
También como se ha comentado anteriormente se han implementado operaciones de casteo de string a símbolo para que el programa pueda hacer uso de ellos como parámetros.

Finalmente para que el programa funcione correctamente, hay que enviar la interfaz generada al programa:

```
;-------------------- FINAL -------------------------
(send frame show #t) ; Manda la interfaz
;----------------------------------------------------
```

Errores

No hay ningún error en la ejecución de la práctica.

Aspectos no implementados

Se han implementado todos los aspectos solicitados en la práctica.

Bibliografía

https://docs.racket-lang.org/gui/