Conocimiento y Razonamiento Automatizado

Listas en $\lambda-$cálculo

Grado en Ingeniería Informática
Universidad de Alcalá

Pablo García García
Álvaro Jesús Martínez Parra
Alejandro Raboso Vindel

15 de mayo de 2023

Índice general

Capítulo 1

Introducción y objetivos

En esta práctica se pretende llevar el uso del λ–cálculo, a un ejemplo práctico real. Para ello, mediante el uso del lenguaje Racket, se implementarán una serie de operaciones sobre listas. Para ello, previamente partimos de una serie de λ–términos dados.

1.1. Codificaciones dadas

Vamos a explicar un poco en detalle qué estructuras se nos dan ya codificadas en Racket, de forma que pueda entenderse un poco mejor el código, y cómo funciona de lo que partimos.

1.1.1. Booleanos

Partiremos de la definición usual de los booleanos en λ–cálculo, así como los conectivos lógicos $\neg$, $\wedge$, y $\vee$:

$$\begin{array}{lcl} \textbf{true} & \equiv & \lambda xy.x \\ \textbf{false} & \equiv & \lambda xy.y \\ \textbf{not} & \equiv & \lambda x.x\ \textbf{false}\ \textbf{true} \\ \textbf{and} & \equiv & \lambda xy.xy\ \textbf{false} \\ \textbf{or} & \equiv & \lambda xy.x\ \textbf{true}\ y \end{array}$$

1.1.2. Pares ordenados

Tanto para el manejo de listas, y como para la codificación de números enteros, será muy útil el uso de pares ordenados:

$$\begin{array}{lcl} \textbf{pair} & \equiv & \lambda xyf.fxy \\ \textbf{fst} & \equiv & \lambda p.p\ \textbf{true} \\ \textbf{snd} & \equiv & \lambda p.p\ \textbf{false} \end{array}$$

1.1.3. Números naturales

Para poder representar el conjunto $\mathbb{Z}$ más adelante, primero debemos ser capaces de poder codificar el conjunto $\mathbb{N}$. Para ello emplearemos la codificación a la Church de los números

naturales:

$$\begin{array}{lcl} \underline{0} & \equiv & \lambda fx.x \\ \underline{1} & \equiv & \lambda fx.fx \\ \underline{2} & \equiv & \lambda fx.f(fx) \\ \vdots & \vdots & \vdots \\ \underline{n} & \equiv & \lambda fx.\underbrace{f(\cdots(f}_{n \text{ veces}} x)\cdots) \end{array}$$

También se dan ya implementados una serie de λ–términos como **escero**, **noescero**, **esigualnat**, **esmenornat**; entre otros, que codifican las operaciones usuales con el conjunto de lo números naturales. Es importante remarcar el λ–término **suc** $\equiv \lambda nfx.f(nfx)$, pues a partir de **suc** $\underline{0}$, podemos codificar cualquier elemento de $\mathbb{N}$.

1.1.4. Números enteros

Con ayuda de los números naturales y los pares ordenados, somos capaces de codificar los elementos de $\mathbb{Z}$. Lo que haremos será que el par (a, b), codifique al entero $a - b$, tomando $a, b \in \mathbb{N}$. Podemos observar que de esta forma, todo elemento de $\mathbb{Z}$ tiene infinitas formas de representarse, pues tomando un $k \in \mathbb{N}$, vemos que $(a + k, b + k)$ representa a $a + k - (b + k) = a - b$.

Para estos términos, ya se dan implementadas operaciones usuales como suma, resta, producto, división, comparaciones...

1.1.5. Combinadores de punto fijo

Si bien en Racket no es necesario un combinador de punto fijo para hacer uso de la recursividad, implementaremos uno para seguir la sintáxis del λ–cálculo. Este será el combinador de Church

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Es obvio que es un combinador de punto fijo, pues se cumple que $\mathbf{Y}F = F(\mathbf{Y}F)$:

$$\mathbf{Y}F \twoheadrightarrow F((\lambda x.F(xx))(\lambda x.F(xx))) \twoheadleftarrow F(\mathbf{Y}F)$$

1.2. Objetivos

Nuestro objetivo en esta práctica será implementar las listas en Racket, en concreto listas de enteros, para poder implementar a su vez las siguientes operaciones:

- Concatenación
- Longitud
- Inversión

- Pertenencia
- Suma de elementos de una lista
- Cálculo de elementos máximos y mínimos de una lista

También se implementarán algunas otras a modo de mejora. Estas se explicarán en el Capítulo 3.

Capítulo 2

Operaciones con listas

2.1. Listas

Para poder trabajar con listas, lo primero de todo será definirlas, pues no vamos a usar las que ya vienen por defecto en este lenguaje. Los λ–términos que las definen serán los siguientes:

$$\begin{array}{lcl} \textbf{nil} & \equiv & \lambda z.z \\ \textbf{cons} & \equiv & \lambda xy.\textbf{pair false (pair } xy) \\ \textbf{null} & \equiv & \textbf{fst} \\ \textbf{hd} & \equiv & \lambda z.\textbf{fst (snd } z) \\ \textbf{tl} & \equiv & \lambda z.\textbf{snd (snd } z) \end{array}$$

Con la codificación presentada para las listas, podemos crear listas de números enteros, naturales, booleanos, o la estructura que queramos. En esta práctica nos centraremos en crear λ–términos que funcionen correctamente con listas de números enteros. De esta manera, la lista `[1, 2, 3, 4]` puede ser representada como (**cons** $\underline{1}$(**cons** $\underline{2}$(**cons** $\underline{3}$(**cons** $\underline{4}$ **nil**)))), y siguiendo la sintaxis de Racket, `(define L ((cons uno)((cons dos)((cons tres)((cons cuatro)nil)))))`, siempre y cuando hayamos definido previamente `uno`, `dos`, `tres`, y `cuatro`.

2.2. $\lambda-$términos con listas

Veamos ahora las diferentes operaciones creadas para trabajar con listas en Racket.

2.2.1. Longitud

Esta operación calculará cuántos elementos tiene una lista. Para ello, veamos una forma recursiva de hacerlo:

$$\text{long}(L) = \begin{cases} 0 & \text{si} \quad L = \texttt{[]} \\ 1 + \text{long}(\text{tl}(L)) & \text{si} \quad L \neq \texttt{[]} \end{cases}$$

De esta manera, el λ–término adecuado para realizar esta operación será

$$\textbf{long} \equiv \mathbf{Y}(\lambda gl.(\textbf{null } l)\underline{0}(\textbf{suc}(g(\textbf{tl } l))))$$

Traducido a Racket, la idea es la misma, pero cambiando la sintaxis. Además, según lo explicado por los profesores, introducimos las funciones de variable `no_use`, para que Racket pueda trabajar con **Y**.

```
70 (define long  ;(printNum (long L3))
71   (lambda (l) ;lista
72       ((Y (lambda (f)
73          (lambda (x)
74           (((null x) ;si es una lista vacía devolvemos longitud 0
75             (lambda (no_use)
76               zero
77             )
78             (lambda (no_use)  ;si no es vacía anadimos 1 a la longitud de
                  la cola
79               (sucesor (f (tl x)))
80             )
81           )
82             zero)
83         )
84       ))
85         l) ; x <- l
86     )
87 )
```

Código 2.1: `long`

2.2.2. Pertenencia

Dada una lista L y un elemento x, esta operación devolverá **true** o **false** indicando si $x \in L$. Pensemos una forma recursiva de hacerlo:

$$\text{in}(L, x) = \begin{cases} \textbf{false} & \text{si} \quad L = [\,] \\ \textbf{true} & \text{si} \quad \text{hd}(L) = x \\ \text{in}(\text{tl}(L), x) & \text{si} \quad \text{hd}(L) \neq x \end{cases}$$

De esta manera, podemos deducir el λ–término que representa esto:

$$\textbf{in} \equiv \mathbf{Y}(\lambda glx.(\textbf{null } l)\ \textbf{false}\ ((\textbf{igual}(\textbf{hd } l)x)\ \textbf{true}\ (f(\textbf{tl } l))))$$

De aquí en adelante solo mostraremos el código una vez implementado el λ–término, pero en todos los casos se aplica el "*truco*" de la función con variable `no_use` para que Racket pueda hacer uso de **Y**.

```
90 (define in ; (display ((in L3)dos)) ó ((in L3)dos)
91   (lambda (l) ;lista
92     (lambda (y) ;elemento
93       ((Y (lambda (f)
94          (lambda (x)
95           (((null x) ; <- Nada pertence a nil
```

```
96              (lambda (no_use)
97                false
98              )
99              (lambda (no_use)  ; <- Cabeza == y?
100               (((esigualent y) (hd x))
101                 true
102                 (f (tl x))  ; si no, y pertenece a cola?
103               )
104             )
105           )
106             zero)  ; no_use <- x
107         )
108       ))
109         l) ; x <- l
110     )
111   )
112 )
```

Código 2.2: `in`

2.2.3. Suma

Dada una lista L, queremos hallar la suma de todos sus elementos. Si lo pensamos de manera recursiva, podemos llegar a la siguiente conclusión:

$$\text{suma}(L) = \begin{cases} 0 & \text{si} \quad L = \texttt{[]} \\ \text{hd}(L) + \text{suma}(\text{tl}(L)) & \text{si} \quad L \neq \texttt{[]} \end{cases}$$

De esta manera, podemos deducir el λ–término que representa esto:

$$\textbf{suma} \equiv \mathbf{Y}(\lambda gl.(\textbf{null } l)\underline{0}(\textbf{add } (\textbf{hd } l)f(\textbf{tl } l)))$$

```
114 ;Suma de los elementos de una lista l
115 (define sumaL
116   (lambda (l) ;lista
117       ((Y (lambda (f)
118          (lambda (x)
119           (((null x) ; Suponemos que suma([]) = 0
120             (lambda (no_use)
121               zero
122             )
123             (lambda (no_use) ;sino
124               ((sument (hd x))(f (tl x))) ; cabeza + suma(cola)
125             )
126           )
127             zero)
128         )
129       ))
130         l) ; x <- l
131     )
132 )
```

Código 2.3: `sumaL`

2.2.4. Invertir

Dada una lista L, queremos hallar la lista en orden contrario. Podemos pensar esta operación de manera recursiva:

$$\text{inversa}(L) = \begin{cases} L & \text{si} \quad L = [] \\ \text{hd}(L) :: \text{inversa}(\text{tl}(L)) & \text{si} \quad L \neq [] \end{cases}$$

De esta manera, podemos escribir el λ–término asociado:

$$\textbf{reverse} \equiv \mathbf{Y}(\lambda gl.((\textbf{null}\ l)l(\textbf{concat}\ (\textbf{hd}\ l)(g(\textbf{tl}\ l)))))$$

```
141 (define reverse
142   (lambda (l) ;Lista
143     (((Y (lambda (f)
144            (lambda (lista)
145              (lambda (resultado) ;Donde se va a ir guardando la inversión
146                (((null lista) ;Si la lista es vacía se devuelve el
                      resultado
147                  (lambda (no_use)
148                    resultado
149                    )
150                  (lambda (no_use) ;Si no es vacía
151                    ((f (tl lista))((cons (hd lista))resultado)) ;Sino se
                          llama recursivamente con la cola y la variable
                          resultado que llevamos hasta ahora con la cabeza de
                          la lista delante
152
153                    )
154                  )
155                 cero)
156               )
157             )
158           ))
159       l) ; lista <- l
160     nil) ; resultado <- nil
161     )
162   )
```

Código 2.4: `reverse`

2.2.5. Concatenar

Dadas dos listas L_1 y L_2, queremos hallar la lista resultante de añadir L_2 a la derecha de L_1. Escribiendo esto de manera recursiva:

$$\text{append}(L_1, L_2) = \begin{cases} L_2 & \text{si} \quad L_1 = [] \\ \text{hd}(L_1) :: \text{append}(\text{tl}(L), L_2) & \text{si} \quad L_1 \neq [] \end{cases}$$

De esta manera, podemos escribir el λ–término correspondiente:

$$\textbf{append} \equiv \mathbf{Y}(\lambda glm.((\textbf{null}\ l)m(\textbf{concat}\ (\textbf{hd}\ l)(g(\textbf{tl}\ m)))))$$

```
(define append ;concatena la segunda lista al final de la primera
  (lambda (lista1)
    (lambda (lista2)
      (((Y (lambda (f)
             (lambda (l1)
               (lambda (l2)
                 (((null l1) ;Si l1 está vacía
                   (lambda (no_use)
                     l2 ;Se devuelve l2
                     )
                   (lambda (no_use) ;sino

                     ((cons (hd l1))((f (tl l1))l2)) ;se pone la cabeza de
                         l1 a lo que quede de llamar recurisvamente

                     )
                   )
                  cero)
                 ))
             ))
        lista1) ; l1 <- lista1
       lista2) ; l2 <- lista2
      )
    ))
```

Código 2.5: append

2.2.6. Elemento máximo y mínimo

Son dos λ–términos diferentes, pero la lógica a seguir es la misma. Razonemos de manera recursiva cómo obtener el máximo de una lista:

$$\text{maximo}(L, x) = \begin{cases} x & \text{si} \quad L = [] \\ \text{maximo}(\text{tl}(L), \text{máx}\{\text{hd}(L), x\}) & \text{si} \quad L \neq [] \end{cases}$$

En ambos casos, inicialmente, el valor de x será la cabeza de la lista. De esta manera, podemos escribir el λ–término correspondiente:

$$\textbf{maximo} \equiv \mathbf{Y}(\lambda glx.((\textbf{null } l)x(g(\textbf{tl } l)(\textbf{mayor}(\textbf{hd } l)x))))$$

```
(define maximum
  (lambda (lista)
    (((Y (lambda (f) ;Función recursiva
           (lambda (l)
             (lambda (maxActual)
               (((null l) ;Si la lista es vacía
                 (lambda (no_use)
                   maxActual ;El maximo es el maximo que lleve guardado
                   )
                 (lambda (no_use) ;Si no es vacía
```

```
200                     ((f (tl l)) ((mayor (hd l)) maxActual)) ;El maximo es
                           el mayor entre la cabeza y el maximo actual, y
                           llamar a la función con el resto de la lista
201                     )
202                   )
203                  cero)
204                ))
205             )
206           )
207       lista) ;l <- lista
208      (hd lista)) ;Inicializo el maxActual con la cabeza de la lista
209     ))
```

Código 2.6: `maximum`

2.2.7. Contar apariciones

Este λ–término contará las veces que aparece un número x en una lista. Pensándolo de manera recursiva, llegamos a lo siguiente:

$$\text{contar}(L,x) = \begin{cases} 0 & \text{si} \quad L = [] \\ \text{contar}(\text{tl}(L),x) & \text{si} \quad \text{hd}(L) \neq x \\ 1 + \text{contar}(\text{tl}(L),x) & \text{si} \quad \text{hd}(L) = x \end{cases}$$

Escribiéndolo en forma de λ–término:

$$\textbf{contar} \equiv \mathbf{Y}(\lambda glx.((\textbf{null}\ l)\underline{0}((\textbf{iguales}\ (\textbf{hd}\ L)x)(\textbf{suc}(g((\textbf{tl}\ L)x)))(g((\textbf{tl}\ L)x)))))$$

```
257 (define apariciones ;(printNum ((apariciones L4)dos)) -> Si se devolviera
       cero cuando la lista es vacía habría que usar printNumZ
258   (lambda (l)  ;lista
259     (lambda (y)  ;elemento
260       ((Y (lambda (f)
261          (lambda (x)
262           (((null x) ;si la lista esta vacia no habra ninguna aparicion
263             (lambda (no_use)
264               zero
265             )
266             (lambda (no_use) ;sino
267               (((esigualent y) (hd x)) ;si la cabeza de la lista es el
                     elemento que se esta buscando se suma 1 y se llama
                     recursivamente
268                 (sucesor (f (tl x)))
269                 (f (tl x)) ;en caso contrario se devuelve el resultado de
                     la funcion recursiva aplicada a la cola de la lista
270               )
271             )
272           )
273             zero)
274         )
275       ))
```

```
276           l) ; x <- l
277       )
278     )
279 )
```

Código 2.7: `apariciones`

2.2.8. Operaciones con vectores

Dadas dos listas L_1, y L_2, supondremos que son dos vectores $\vec{u}$ y $\vec{v}$, y se han implementado las operaciones $\vec{u}+\vec{v}$, $\vec{u}-\vec{v}$, multiplicación componente a componente, y producto escalar $\vec{u}\cdot\vec{v}$. Veamos un ejemplo de como se piensa la suma de forma recursiva, pues el resto es similar pero cambiando la operación básica:

$$\text{suma\_vec}(L_1, L_2) = \begin{cases} L_1[0] + L_2[0] :: \text{suma\_vec}(\text{tl}(L_1), \text{tl}(L_2)) & \text{si} \quad L_1 \neq [] \wedge L_2 \neq [] \\ L_1 & \text{si} \quad L_1 \neq [] \wedge L_2 = [] \\ L_2 & \text{si} \quad L_1 = [] \wedge L_2 \neq [] \\ [] & \text{si} \quad L_1 = [] \wedge L_2 = [] \end{cases}$$

Al ser λ–términos un poco distintos, aunque con lógica similar, dejamos que se lean directamente del código. Obsérvese que no son como tal las operaciones matemáticas clásicas, pues se permite diferente dimensión entre los vectores. También comentar que el producto escalar lo conseguimos con operaciones previamente definidas, siendo estas, la suma de listas, y el producto de listas.

```
281 ;Suma de elementos de dos listas
282 (define sumaVectores ;(printLista ((sumaVectores L3)L4))
283   (lambda (lista1) ;lista1
284     (lambda (lista2) ;lista2
285       (((Y (lambda (f)
286          (lambda (x)
287            (lambda (y)
288            (((null x)  ;si la lista 1 es vacía habrá que devolver la lista
                  2
289              (lambda (no_use)
290                y
291              )
292              (lambda (no_use)
293                ((null y)  ;si la lista 2 es vacía habrá que devolver la
                      lista 1
294                  x
295                  ((cons ((sument (hd x))(hd y)))((f (tl x))(tl y)))  ;en
                        caso de que haya elementos en las dos listas, habrá que
                         devolver una nueva lista con la suma de las cabezas de
                         ambas y la suma de las colas
296                )
297              )
298            )
299              zero)
300         ))
```

```
301       ))
302         lista1) ;x <- lista1
303        lista2) ;y <- lista2
304     )
305   )
306  )
307
308 ;Resta de elementos de dos listas
309 (define restaVectores ;(printLista ((restaVectores L3)L4))
310   (lambda (lista1) ;lista1
311     (lambda (lista2) ;lista2
312       (((Y (lambda (f)
313          (lambda (x)
314            (lambda (y)
315            (((null x)  ;si la lista 1 es vacía habrá que devolver la lista
                 2
316              (lambda (no_use)
317                y
318              )
319              (lambda (no_use)
320                ((null y)  ;si la lista 2 es vacía habrá que devolver la
                      lista 1
321                  x
322                  ((cons ((restaent (hd x))(hd y)))((f (tl x))(tl y)))  ;en
                        caso de que haya elementos en las dos listas, habrá que
                         devolver una nueva lista con la resta de las cabezas
                        de ambas y la resta de las colas
323              )
324            )
325           )
326             zero)
327         ))
328       ))
329         lista1) ;x <- lista1
330        lista2) ;y <- lista2
331     )
332   )
333  )
334
335 ;Producto de elementos de dos listas
336 (define multVectores ;(printLista ((multVectores L3)L4))
337   (lambda (lista1) ;lista1
338     (lambda (lista2) ;lista2
339       (((Y (lambda (f)
340          (lambda (x)
341            (lambda (y)
342            (((null x)  ;si la lista 1 es vacía habrá que devolver la lista
                 2
343              (lambda (no_use)
344                y
345              )
346              (lambda (no_use)
347                ((null y)  ;si la lista 2 es vacía habrá que devolver la
                      lista 1
```

```
348                 x
349                 ((cons ((prodent (hd x))(hd y)))((f (tl x))(tl y)))  ;en
                        caso de que haya elementos en las dos listas, habrá que
                         devolver una nueva lista con el producto de las
                        cabezas de ambas y el producto de las colas
350               )
351             )
352           )
353             zero)
354         ))
355       ))
356         lista1) ;x <- lista1
357        lista2) ;y <- lista2
358     )
359   )
360  )
```

Código 2.8: Operaciones con vectores

```
502 (define productoEscalar
503   (lambda (l1)
504        (lambda (l2)
505                     (sumaL ((multVectores l1)l2))
506                     )
507    ))
```

Código 2.9: `productoEscalar`

2.2.9. Sustituciones

Dada una lista L y dos elementos x e y, reemplaza todas las apariciones de x por y en L. Podemos definirlo de manera recursiva así:

$$\text{sust}(L,x,y) = \begin{cases} [] & \text{si} \quad L = [] \\ \text{hd}(L) :: \text{sust}(\text{tl}(L),x,y) & \text{si} \quad \text{hd}(L) \neq x \\ y :: \text{sust}(\text{tl}(L),x,y) & \text{si} \quad \text{hd}(L) = x \end{cases}$$

Escrito en forma de λ–término:

$$\textbf{sust} \equiv \mathbf{Y}(\lambda glxy.((\textbf{null } l)\textbf{nil}((\textbf{iguales}(\textbf{hd}(l)x))(\textbf{cons } y(g(\textbf{sust}(\textbf{tl } l)xy))))(\textbf{cons hd}(l)(g(\textbf{sust}(\textbf{tl } l)xy)))))$$

```
363 (define sustituir
364   (lambda (l)  ;lista
365     (lambda (x)  ;elemento
366       (lambda (y)
367       ((Y (lambda (f)
368          (lambda (l1)
369           (((null l1) ;si la lista esta vacia no habra ninguna aparicion
370             (lambda (no_use)
371               l1
372             )
```

```
373             (lambda (no_use) ;sino
374               (((esigualent x) (hd l1)) ;si la cabeza de la lista es el
                      elemento que se esta buscando se suma 1 y se llama
                      recursivamente
375                 ((cons y)(f (tl l1)))
376                 ((cons (hd l1))(f (tl l1))) ;en caso contrario se devuelve
                        el resultado de la funcion recursiva aplicada a la
                        cola de la lista
377               )
378             )
379           )
380             zero)  ; no_use <- x
381         )
382       ))
383         l) ; l1 <- l
384     )
385   )
386  )
387 )
```

Código 2.10: `sustituir`

2.2.10. Ordenar

Dada una lista L, ordenaremos sus elemento de mayor a menor, o de menor a mayor. Esto son dos operaciones diferentes, aunque mostraremos solo una como ejemplo. En este caso estamos viendo la ordenación de mayor a menor. En el caso contrario, buscamos el mínimo en vez del máximo. Podemos hacer esto de forma recursiva:

$$\text{sort}(L) = \begin{cases} [] & \text{si} \quad L = [] \\ \text{maximo}(L) :: \text{sort}(\text{eliminar\_prim}(L, \text{maximo}(L))) & \text{si} \quad L \neq [] \end{cases}$$

Veremos más adelante que se ha implementado otro λ–término `eliminar_prim`, que lo que hace es eliminar la primera aparición de un elemento en una lista. Además podemos escribir el λ–término de la ordenación.

$$\textbf{sort} \equiv \mathbf{Y}(\lambda gl.((\textbf{null}\ l)\textbf{nil}(\textbf{cons}(\textbf{max}(l)g(\textbf{eliminar\_p}\ l(\textbf{max}\ l))))))$$

```
415 (define ordenarMayorMenor
416   (lambda (l)  ;lista
417       ((Y (lambda (f)
418          (lambda (l1)
419           (((null l1) ;si la lista esta vacia no habra ninguna aparicion
420             (lambda (no_use)
421               l1
422             )
423             (lambda (no_use) ;sino
424               ((cons (maximum l1))(f ((eliminarPrimeraAparicion l1)(
                      maximum l1))))
425             )
426           )
```

```
427             zero)  ; no_use <- x
428         )
429       ))
430         l) ; x <- l
431     )
432 )
```

Código 2.11: `ordenar`

2.2.11. Tomar y dejar

Dada una lista L, y n elementos, tomar devuelve los n primeros elementos de L, y dejar, quita los n primeros elementos de L y devuelve el resto. Al ser muy similares, explicaremos solo tomar, y el razonamiento de dejar, será similar. Escribiéndolo de manera recursiva:

$$\text{tomar}(L,n) = \begin{cases} [] & \text{si} \quad L = [] \\ \text{hd}(L) :: \text{tomar}(\text{tl}(L), n-1) & \text{si} \quad L \neq [] \end{cases}$$

Escrito en forma de λ–término:

$$\textbf{tomar} \equiv \textbf{Y}(\lambda gln.((\textbf{null}\ l)\textbf{nil}(\textbf{cons}(\textbf{hd}\ l)g((\textbf{tl}\ l)(\textbf{pre}\ n)))))$$

```
454 (define tomaNElementos
455   (lambda (numero) ;numero de elementos a tomar
456     (lambda (lista) ;lista
457       (((Y (lambda (f)
458              (lambda (n)
459                (lambda (l)
460                  (((esceroent n) ;Si la lista esta vacia
461                    (lambda (no_use)
462                      nil
463                      )
464                    (lambda (no_use) ;sino
465
466                      ((cons (hd l))((f ((restaent n)uno))(tl l))) ;
                             Concatenamos la cabeza con n números de la cola
467
468                      )
469                    )
470                   cero)
471                  ))
472              ))
473         numero) ;n <- numero
474        lista) ;l <- lista
475       )
476     ))
477
478 (define dejaNElementos
479   (lambda (numero) ;numero de elementos a dejar
480     (lambda (lista) ;lista
481       (((Y (lambda (f)
482              (lambda (n)
```

```
483                (lambda (l)
484                  (((esceroent n) ;Si la lista esta vacia
485                    (lambda (no_use)
486                      l
487                      )
488                    (lambda (no_use) ;sino
489
490                      ((f ((restaent n)uno))(tl l)) ;Cogemos solo números a
                             partir del n primero
491
492                      )
493                    )
494                   cero)
495                  ))
496              ))
497         numero) ;n <- numero
498        lista) ;l <- lista
499       )
500     ))
```

Código 2.12: Tomar y dejar

Capítulo 3

Mejoras

En esta parte veremos una serie de mejoras que se han realizado a la práctica, siendo algunas de ellas más operaciones extra, y un menú explicativo.

3.1. $\lambda-$términos extra

Además de las operaciones solicitadas por los profesores, se han definido una nueva colección, para poder trabajar de forma más cómoda en Racket. Estas son las siguientes:

- `church2N`: dado un número natural codificado a la Church, devuelve el número natural (como número de Racket).
- `church2Z`: dado un entero, representado como $m-n$ en el par (m, n), lo devuelve como número de Racket.
- `printLista`: dada una lista en la codificación empleada para la práctica, la imprime. Por ejemplo, dada la lista `(define L ((cons uno)((cons cero)((cons dos)nil))))`, imprimirá `[1, 0, 2, nil]`.
- `printNumZ`: dado un número entero, lo muestra por pantalla
- `printNum`: dado un número natural, lo muestra por pantalla
- `concatNumList`: dada una lista L, y un número n, devuelve la lista $n :: L$.
- `mayor` y `menor`: dados dos números n y m, devuelven el mayor y el menor de los dos números.
- `apariciones`: ya explicada en la Sección 2.2.7.
- `sumaVectores`: ya explicada en la Sección 2.2.8.
- `restaVectores`: ya explicada en la Sección 2.2.8.
- `multVectores`: ya explicada en la Sección 2.2.8.

- `productoEscalar`: ya explicada en la Sección 2.2.8.
- `sustituir`: ya explicada en la Sección 2.2.9.
- `eliminarPrimeraAparicion`: dada una lista L, y un elemento x, elimina la primera aparición por la izquierda de x en L.
- `ordenarMayorMenor`: ya explicada en la Sección 2.2.10.
- `ordenarMenorMayor`: ya explicada en la Sección 2.2.10.
- `tomaNElementos`: ya explicada en la Sección 2.2.11
- `dejaNElementos`: ya explicada en la Sección 2.2.11

3.2. Menú explicativo

Se ha puesto a disposición del usuario, un menú que muestra todas y cada una de las operaciones que puede realizar, así como una breve explicación de cómo puede ejecutar cada operación con la lista que él quiera. Además, se puede volver a mostrar el menú, con la llamada a `(menu cero)`.

```
FUNCIONALIDADES
1. Imprimir una lista -> Introduce (printLista L), siendo L la lista que deseas imprimir.
2. Sacar longitud de una lista -> Introduce (printNum(long L)), siendo L la lista.
3. Test de pertenenencia -> Introduce ((in L)N), siendo L la lista y N el número (escrito en letras) que quieres buscar dentro de la lista.
4. Sumar los elementos de una lista -> Introduce (printNumZ(sumaL L)) siendo L la lista.
5. Anadir un elemento a la izquierda de una lista -> Introduce (printLista((concatNumList L)N)),
siendo L la lista y N el número (escrito en letras) que quieres anadir a la izquierda.
6. Invertir una lista -> Introduce (printLista(reverse L)), siendo L la lista.
7. Concatenar dos listas -> Introduce (printLista((append L1)L2)),
siendo L1 una lista y L2 la lista que se quiere concatenar a L1
8. Sacar el número máximo de una lista -> Introduce (printNumZ(maximum L)), siendo L la lista.
9. Sacar el número mínimo de una lista -> Introduce (printNumZ(minimum L)), siendo L la lista.
10. Sacar el número de apariciones de un elemento en una lista -> Introduce (printNum((apariciones L)N)),
siendo L la lista y N el elemento que quieres contar (escrito en letras).
11. Sacar la suma de dos listas -> Introduce (printLista((sumaVectores L1)L2)),
siendo L1 y L2 las dos listas.
12. Sacar la resta de dos listas -> Introduce (printLista((restaVectores L1)L2)),
siendo L1 y L2 las dos listas.
13. Sacar el producto de dos listas -> Introduce (printLista((productoVectores L1)L2)),
siendo L1 y L2 las dos listas.
14. Sustituir un elemento X en una lista por otro -> Introduce (printLista(((sustituir L)N)M)),
siendo L la lista, N el elemento que se quiere sustituir (escrito en letras) y M el nuevo elemento (escrito en letras).
15. Eliminar la primera aparicion de un elemento X en una lista -> Introduce (printLista((eliminarPrimeraAparicion L)N)),
siendo L la lista y N el elemento (escrito en letras).
16. Ordenar la lista de menor a mayor -> Introduce (printLista(ordenarMenorMayor L)), siendo L la lista.
17. Ordenar la lista de mayor a menor -> Introduce (printLista(ordenarMayorMenor L)), siendo L la lista.
18. Coger los N primeros elementos de una lista. Introduce (printLista((tomaNElementos N)L)),
siendo N el número de elementos (escrito en letras) que se quieren coger y L la lista.
19. Quitar los N primeros elementos de una lista. Introduce (printLista((dejaNElementos N)L)),
siendo N el número de elementos (escrito en letras) que se quieren dejar y L la lista.
20. Calcular el producto escalar de dos listas. Introduce (printNumZ((productoEscalar L1)L2)),
siendo L1 y L2 dos listas.
NOTA: Los números enteros definidos son del -veinte al veinte por lo que si se quieren usar otros distintos,
habrá que definirlos primero de la siguiente forma:
Para el veinte: (define vingt (sucesor dix-neuf)) y (define veinte ((par vingt) zero))
Para el -veinte: (define -veinte ((par zero) vingt))
Para desplegar este menú de nuevo, escribir (menu cero)
```

Figura 3.1: Menú

También se han definido una serie de listas como caso de prueba, listas para ser usadas. Finalmente, al ejecutar el código, se aplica de forma automática todas las operaciones creadas a los casos de prueba, con motivo de verificar que están todas correctamente realizadas.

```
La lista A es: [-1, 2, -3, -4, 5, 6, nil]

La lista B es: [9, 2, 5, 1, 3, 2, nil]

La longitud de la lista A es: 6
La longitud de la lista B es: 6
El número 7 está en la lista A:
#<procedure:false>

El número 2 está en la lista B:
#<procedure:true>

La suma de los elementos de la lista A es: 5
La suma de los elementos de la lista B es: 22
Anadir el 1 al principio de la lista A: [1, -1, 2, -3, -4, 5, 6, nil]

Anadir el 3 al principio de la lista B: [3, 9, 2, 5, 1, 3, 2, nil]

Lista A invertida: [6, 5, -4, -3, 2, -1, nil]

Lista B invertida: [2, 3, 1, 5, 2, 9, nil]

Concatenar a la lista A la lista B: [-1, 2, -3, -4, 5, 6, 9, 2, 5, 1, 3, 2, nil]

Concatenar a la lista B la lista A: [9, 2, 5, 1, 3, 2, -1, 2, -3, -4, 5, 6, nil]

El máximo de la lista A es: 6
El máximo de la lista B: 9
El mínimo de la lista A es: -4
El mínimo de la lista B: 1
Número de 6 que hay en la lista A: 1
Número de 2 que hay en la lista B: 2
La suma de las dos listas es: [8, 4, 2, -3, 8, 8, nil]

La resta de las dos listas es: [-10, 0, -8, -5, 2, 4, nil]

La multiplicacion de las dos listas es: [-9, 4, -15, -4, 15, 12, nil]

Si sustituimos en la lista A los 6 por 8 nos queda: [-1, 2, -3, -4, 5, 8, nil]

Si sustituimos en la lista B los 2 por 11 nos queda: [9, 11, 5, 1, 3, 11, nil]

Si eliminamos el primer 5 de la lista A nos queda: [-1, 2, -3, -4, 6, nil]

Si eliminamos el primer 2 de la lista B nos queda: [9, 5, 1, 3, 2, nil]

La lista A ordenada de menor a mayor es: [-4, -3, -1, 2, 5, 6, nil]

La lista B ordenada de menor a mayor es: [1, 2, 2, 3, 5, 9, nil]

La lista A ordenada de mayor a menor es: [6, 5, 2, -1, -3, -4, nil]

La lista B ordenada de mayor a menor es: [9, 5, 3, 2, 2, 1, nil]

Si quitamos los tres primeros elementos de la lista A nos queda: [-4, 5, 6, nil]

Si quitamos los dos primeros elementos de la lista B nos queda: [5, 1, 3, 2, nil]

Si tomamos los tres primeros elementos de la lista A nos queda: [-1, 2, -3, nil]

Si tomamos los dos primeros elementos de la lista B nos queda: [9, 2, nil]

Si tomamos las listas A y B, su producto escalar es: 3
```

Figura 3.2: Casos de prueba

Apéndice A

Otros detalles

Se han cubierto todos los objetivos solicitados, funcionando correctamente todas las operaciones, así como un gran banco de operaciones a modo de mejora, sin ningún error de implementación.

Respecto al reparto de tareas, la práctica se ha realizado por todos los integrantes del grupo a la vez en una llamada de Discord, aportando cada uno soluciones y alternativas a los problemas que se iban encontrando.

Para finalizar, las referencias consultadas han sido, los apuntes de la asignatura, el manual de Racket, y StackOverFlow.