Conocimiento y Razonamiento Automatizado

PECL3

Curso 2021-2022

Isabel Blanco Martínez

Laura Ramos Martínez

Patricia Cuesta Ruiz

ÍNDICE

REPARTO DE TAREAS

No ha habido un reparto claro de las tareas ya que durante la realización de toda la práctica en la gran mayoría de ocasiones nos hemos estado reuniendo las 3 integrantes del grupo para realizarla.

GRADO DE CUMPLIMIENTO DE LOS REQUISITOS

Se han completado todos los requisitos llevando a cabo pruebas con cada uno de los apartados solicitados para la práctica.

Para todo el desarrollo de la práctica hemos declarado *mod* como variable global para poder asignar un módulo directamente a todas las operaciones sin pasarlas por parámetro.

Operaciones con modulares

- **Reducción a representante canónico**

Realizamos la reducción modular utilizando la función *restoent* proporcionada por los profesores en el archivo enteros.rkt.

```
(define representante_canonico (lambda (x)
                                 (lambda (p)
                                   ((restoent x) p))))
```

- **Aritmética: suma y producto**

Junto con estas funciones también hemos implementado la resta y el cociente. Para ello, hemos hecho uso de las funciones *sument*, *restaent*, *prodent* y *cocienteent* proporcionadas por los profesores, aplicándole la reducción al módulo (representante_canónico).

```
;; Recibe como parámetros dos numeros enteros y devuelve un procedimiento que calcula la suma de los mismos.

(define suma_enteros (lambda (m)
                       (lambda (n)
                         ((representante_canonico ((sument m) n)) mod))))

;; Recibe como parámetros dos numeros enteros y devuelve un procedimiento que calcula la resta de los mismos.

(define resta_enteros (lambda (m)
                        (lambda (n)
                          ((representante_canonico ((restaent m) n)) mod))))

;; Recibe como parámetros dos numeros enteros y devuelve un procedimiento que calcula el producto de los mismos.

(define prod_enteros (lambda (m)
                       (lambda (n)
                         ((representante_canonico ((prodent m) n)) mod))))

;; Recibe como parámetros dos numeros enteros y devuelve un procedimiento que calcula el cociente de los mismos.

(define cociente_enteros (lambda(m)
                           (lambda (n)
                             ((representante_canonico ((cocienteent m) n)) mod))))
```

- **Decisión sobre la inversibilidad y cálculo del inverso en caso de que exista.**

Para realizar este apartado hemos declarado cuatro funciones. *escero_enteros* nos ayuda a distinguir si un número entero es 0 o no. Con *inversa_enteros?* comprobamos si el número que recibe por parámetro mediante el uso de *mcd* y aplicando la *reducción de módulo*. Con *inverso_enteros* devuelve un procedimiento que se encarga de comprobar que el número pasado por parámetro tiene inverso llamando a la función anterior. Si tiene, llamamos a *calcular_inverso* y si no devuelve 0, ya que el inverso nunca puede ser 0. La función calcular_inverso, es un procedimiento recursivo que se encarga de encontrar el inverso de un número siguiendo la siguiente fórmula:

(A * A^-1) = 1

Es decir, un número por su inverso es igual a 1. Para ello utilizamos un contador que inicializamos a 0 que será nuestro número candidato actual, es decir, el que comprobaremos que sea el inverso del número dado. Se realizará esta operación hasta llegar al número del módulo establecido. Si supera este, devuelve cero ya que no tiene inverso. Si lo encuentra, devuelve el inverso.

```
;; Recibe como parametro un numero entero y devuelve un procedimiento que dice si dicho numero es cero o no. Si es cero no tiene inverso.

(define escero_enteros (lambda (x)
                         ((esigualent x) cero)))

;; Recibe como parámetro un número entero y devuelve un procedimiento que dice si tiene o no un inverso modular.

(define inversa_enteros? (lambda (x)
                           (((esigualent ((mcdent x) mod)) uno) true false)))

;; Recibe como parametro un número entero y devuelve un procedimiento que calcula su inverso.

(define inverso_enteros (lambda (n)
                          ((inversa_enteros? n)
                           ((calcular_inverso n) cero) ; inicializamos contador a 0
                           cero
                           )))

;; Función recursiva para calcular el inverso: mediante un contador va a comprobar con cada numero hasta llegar al modulo la siguiente formula:
;; (A * A^-1) = 1 (un numero por su inverso es igual a 1) --> siendo A^-1 nuestro candidato (c).

(define calcular_inverso (lambda (n) ; definimos A
                           (lambda (c) ; definimos el contador para el candidato
                             (((Y (lambda (f) ; mediante combinador de punto fijo creamos funcion recursiva f
                                    (lambda(x) ; equivalente a n
                                      (lambda(y) ; equivalente a c
                                        ((((esmayoroigualent y) mod) ; si el contador es mayor o igual al modulo
                                          (lambda (no_use)
                                            cero)                ; devuelve 0
                                          (lambda (no_use)
                                            ((((esigualent uno)((representante_canonico ((prodent x)y)) mod)) ; sino, si es igual el modulo de el producto del numero por el candidato a inverso a 1,
                                              ; quiere decir que ese valor es el inverso y lo retornamos
                                              (lambda (no_use)
                                                y)
                                              (lambda (no_use)
                                                ((f x)((sument y) uno)) ; aumentamos contador en 1
                                                )
                                              )cero))) cero) ; pasamos cero como parametro de no_use
                                        ))))
                                n)c))))
```

- **Pruebas**

Hemos definido una función llamada *pruebaEnteros* con la que podemos ejecutar fácilmente unas pruebas con todas las operaciones anteriormente definidas.

```
(define (pruebaEnteros)
  (display "------------------------- PRUEBAS OPERACIONES DE ENTEROS -----------------------\n")
  (display "\n3 mod 7 = ")
  (display (testenteros ((representante_canonico tres) siete)))
  (display "\n20 mod 3 = ")
  (display (testenteros ((representante_canonico veinte) tres)))
  (display "\nSuma de enteros 2 + 3 = ")
  (display (testenteros ((suma_enteros dos) tres)))
  (display "\nResta de enteros 10 - 3 = ")
  (display (testenteros ((resta_enteros diez) tres)))
  (display "\nProducto de enteros 2 * 4 = ")
  (display (testenteros ((prod_enteros dos) cuatro)))
  (display "\nCociente de enteros 6 / 3 = ")
  (display (testenteros ((cociente_enteros seis) tres)))
  (display "\nInverso de 5 = ")
  (display (testenteros (inverso_enteros cinco)))
  (display "\n\n--------------------------------------------------------------------------------")
  )
```

Operaciones con matrices

- **Suma y producto**

Para realizar la suma de matrices se van sumando los enteros de cada una de las posiciones de la matriz (utilizando las funciones *primero* y *segundo* dadas por los profesores) utilizando la función *suma_enteros* creada para el apartado anterior.

```
#|
  * Recibe como parametro dos matrices
  * Devuelve un procedimiento que calcula la suma de ambos. El modulo lo aplica en suma_enteros pasado de forma global.
|#

(define suma_matrices (lambda (m)
                        (lambda (n)
                          ((((matriz
                              ((suma_enteros (primero (primero m))) (primero (primero n))))
                             ((suma_enteros (segundo (primero m))) (segundo (primero n))))
                            ((suma_enteros (primero (segundo m))) (primero (segundo n))))
                           ((suma_enteros (segundo (segundo m))) (segundo (segundo n)))))))
```

Para la multiplicación de matrices se aplica el método de multiplicación de matrices que consiste en sumar el primer elemento de A por el primero de B más el segundo elemento de A por el segundo de B.

```
#|
  * Recibe como parametro dos matrices
  * Devuelve un procedimiento que calcula el producto de ambas matrices.
|#

(define producto_matrices (lambda (m)
                            (lambda (n)
                              ((((matriz
                                  ((suma_enteros ((prodent (primero (primero m))) (primero (primero n)))) ((prodent (segundo (primero m))) (primero (segundo n)))))
                                 ((suma_enteros ((prodent (primero (primero m))) (segundo (primero n)))) ((prodent (segundo (primero m))) (segundo (segundo n)))))
                                ((suma_enteros ((prodent (primero (segundo m))) (primero (primero n)))) ((prodent (segundo (segundo m))) (primero (segundo n)))))
                               ((suma_enteros ((prodent (primero (segundo m))) (segundo (primero n)))) ((prodent (segundo (segundo m))) (segundo (segundo n))))))))
```

- **Determinante**

Para hallar el determinante de una matriz hemos implementado el método *determinante* que, al ser siempre una función 2x2, resta los elementos de la primera diagonal multiplicados entre sí, menos los elementos de la segunda diagonal multiplicados entre sí. A todo ello le aplicamos *representante_canonico* al que le pasamos *mod*.

```
#|
  * Recibe como parametro una matriz.
  * Devuelve un procedimiento que calcula el determinante de una matriz.
  * Como las matrices son 2 x 2, el determinante es la resta de los elementos de la primera diagonal multiplicados entre sí,
    menos los elementos de la segunda diagonal multiplicados entre sí.
  * El módulo se aplica con la función representante_canónico pasándole el mod global.
|#

(define determinante (lambda (m)
                       ((representante_canonico ((restaent ((prodent (primero (primero m))) (segundo (segundo m))))
                                                  ((prodent (segundo (primero m))) (primero (segundo m))))) mod)))
```

- **Decisión sobre la inversibilidad y cálculo de la inversa y del rango.**
    - *Cálculo de la inversa*

Para la realización del cálculo de la inversa de la matriz hemos realizado varias funciones para implementar el siguiente método para su cálculo utilizando la fórmula:

$$A^{-1} = \frac{(Adj(A))^T}{|A|}$$

En primer lugar, hemos realizado una función llamada *prod_escalar* donde dada una matriz y un número devuelve el resultado de la multiplicación del número por los elementos de la matriz.

```
#|
 * Dada una matriz y un número entero, devuelve un procedimiento con el resultado de la multiplicación
 * de dicho número por todos los elementos de la matriz.
|#

(define prod_escalar (lambda (m)
                       (lambda (n)
                         ((((matriz
                             ((prod_enteros (primero (primero m))) n))
                            ((prod_enteros (segundo (primero m))) n))
                           ((prod_enteros (primero (segundo m))) n))
                          ((prod_enteros (segundo (segundo m))) n)))))
```

La función *inversa_matriz?* comprueba si la matriz que se pasa tiene o no inversa dependiendo si su determinante es 0.

```
#|
 * Dada la matriz comprueba si tiene inversa.
 * Devuelve un procedimiento que dice si tiene o no inversa.
|#

(define inversa_matriz? (lambda (m)
                          ((esceroent (determinante m)) false true)))
```

La función *inversa_matriz* comprueba si la matriz que se pasa tiene o no inversa dependiendo si su determinante es 0, por lo que si es 0 no tiene inversa y devuelve la *matriz_nula*, y si no, llama a la función *inversa_matriz_aux* para calcular su inversa.

```
#|
 * Decisión sobre inversibilidad de una matriz dependiendo de si el determinante es 0 (si es 0 no tiene).
 * Dada la matriz calcula la inversa.
 * Si es 0 el determinante de la matriz devuelve matriz nula (no tiene inverso). Sino llama a inversa_matriz_aux para calcular su inversa.
|#

(define inversa_matriz (lambda (m)
                         ((esceroent (determinante m)) matriz_nula (inversa_matriz_aux m))))
```

La función *inversa_matriz_aux* recibe una matriz y multiplica el inverso del determinante de por la matriz traspuesta de la adjunta.

```
#|
 * Recibe como parametro una matriz
 * Multiplica el determinante por la matriz adjunta con el producto escalar.
 * Al ser una matriz de 2x2, se cumple que la matriz  a b   al hacer la adjunta y despues la transpuesta de la adjunta
 *                                                    c d
 * queda siempre como  d -b .
 *                    -c  a
|#

(define inversa_matriz_aux (lambda (m)
                             ((prod_escalar (matriz_traspuesta (matriz_adjunta m))) (inverso_enteros (determinante m)))))
```

La función *matriz_adjunta* calcula la matriz adjunta, que consiste en poner los números de la diagonal secundaria en negativo. Esto se hace aplicando el representante canónico (para aplicar el módulo) a cada una de las posiciones de la nueva matriz, realizando la resta de 0 menos el valor original para obtener el número negativo.

```
#|
 * Recibe como parametro una matriz.
 * Dada una matriz devuelve la matriz adjunta, que es la diagonal secundaria con los números en negativo.
|#

(define matriz_adjunta (lambda (m)
                          ((((matriz
                              ((representante_canonico (segundo (segundo m)))mod))
                             ((representante_canonico ((restaent cero) (primero (segundo m))))mod))
                            ((representante_canonico ((restaent cero) (segundo (primero m))))mod))
                           ((representante_canonico (primero (primero m)))mod))
                          ))
```

La función *matriz_traspuesta* recibe una matriz y calcula la traspuesta, que consiste en intercambiar los valores creando una matriz nueva.

```
#|
 * Recibe como parametro una matriz.
 * Dada una matriz devuelve la matriz traspuesta, intercambiando los valores a fuerza bruta mediante la creación de
 * una matriz nueva.
|#

(define matriz_traspuesta (lambda (m)
                             ((((matriz
                                 (primero (primero m)))
                                (primero (segundo m)))
                               (segundo (primero m)))
                              (segundo (segundo m)))))
```

- *Cálculo del rango*

Para calcular el rango, primero debemos comprobar que la matriz no sea nula ya que su rango sería 0. Para ello, desarrollamos la función *matriz_nula?* que comprueba uno a uno los elementos de una matriz 2x2 y devuelve true en caso de ser nula, o false en caso de tener algún valor diferente de 0.

```
#|
 * Recibe como parametro una matriz.
 * Recibe como parametro una matriz y mediante la comprobación de cada uno de sus elementos devuelve
 * true o false según la matriz sea nula o no.
|#

(define matriz_nula? (lambda (m)
                      ((esceroent ((representante_canonico (primero (primero m))) mod))
                       ((esceroent ((representante_canonico (segundo (primero m))) mod))
                        ((esceroent ((representante_canonico (primero (segundo m))) mod))
                         ((esceroent ((representante_canonico (segundo (segundo m))) mod)) true false) false) false)
                       false)))
```

Para calcular el rango definimos *rango_aux* con la que comprobamos si el determinante de la matriz es 0, ya que en caso de ser 0 el rango es uno, y en caso contrario es dos.

```
#|
 * Recibe como parametro una matriz.
 * Procedimiento auxiliar que nos devuelve si el rango es uno o dos.
 * Usada para calcular el rango.
|#

(define rango_aux (lambda (m)
                    ((esceroent (determinante m)) uno dos)))
```

Por último, creamos la función *rango*, que recibe una matriz y lo primero que hace es comprobar si es nula. En caso de serlo, devuelve 0, ya que el rango de una matriz nula es 0. En caso de no serlo, llamamos a *rango_aux* que devolverá 1 o 2 dependiendo del determinante de la matriz.

```
#|
 * Recibe como parametro una matriz.
 * Dada una matriz calcula el rango de esta.
 * Si es matriz nula devuelve cero directamente.
|#

(define rango (lambda (m)
                ((matriz_nula? m) cero (rango_aux m))))
```

- **Cálculo de potencias (naturales) de matrices.**

Para llevar a cabo el cálculo de las potencias hemos creado dos procedimientos diferentes para seguir el procedimiento de exponenciación binaria. El primero es *cuadrado_matrices* que nos hará falta para calcular las potencias. Dicho método recibe una matriz por parámetro y mediante *producto_matrices* implementado en el apartado (d) calcula el cuadrado de la matriz.

```
#|
 * Recibe como parametro una matriz.
 * Devuelve un procedimiento que calcula el cuadrado de una matriz dada.
|#

(define cuadrado_matrices (lambda (m)
                            ((producto_matrices m) m)))
```

El método *potencias_matrices* es un procedimiento recursivo que calcula el resultado de elevar el exponente indicado a la matriz obtenida por parámetro. Para ello, seguimos el procedimiento de exponenciación binaria:

$$x^n = \begin{cases} x & \text{si } n = 1 \\ \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ es par} \\ x \times x^{n-1} & \text{si } n \text{ es impar} \end{cases}$$

Es decir, en el procedimiento lo primero que hacemos es comprobar que el exponente sea 0. Si lo es, devuelve la matriz identidad. En caso de no serlo, comprobamos si el exponente es par: calculamos el cuadrado de la matriz y llamamos recursivamente dividiendo entre 2 el exponente. Si es impar: se multiplica la matriz por el valor del exponente menos 1.

En todo momento se hace en función del módulo sin tener que pasarlo por parámetro, ya que en la función de *producto_matrices* lo reducimos a representante canónico.

```
#|
  * Recibe como parametros una matriz y un número.
  * Calcula la matriz resultante de elevar una matriz a un numero natural.
  * Devuelve un procedimiento que calcula cual seria la matriz resultado al elevar la matriz parametro al numero parametro siguiendo
    el algoritmo recursivo de la exponenciacion binaria.
|#

(define potencia_matrices (lambda (m)
                            (lambda (p)
                              ((Y (lambda (f) ; declaramos llamada recursiva como f, pasando el exponente
                                    (lambda (e)
                                      (((esceroent e) ; si el exponente es cero
                                        (lambda (no_use)
                                          identidad ; devuelve la matriz identidad
                                          )
                                        (lambda (no_use)
                                          (((esceroent ((restoent e) dos)) ; si no es cero se comprueba si el exponente es par
                                            (lambda (no_use)
                                              (cuadrado_matrices (f ((cocienteent e) dos)))) ; si es par se calcula el cuadrado de la matriz y se
                                                                                             ; llama a recursion dividiendo entre 2 el exponente
                                            (lambda (no_use)
                                              ((producto_matrices m)(f ((restaent e) uno))))) cero)  ; si es impar se multiplica la matriz
                                                                                                     ; por el valor del exponente menos uno
                                          )
                                        )
                                       cero)
                                      )
                                    ))
                               p)
                              )
                            )
  )
```

- **Pruebas**

Hemos definido una función llamada *pruebaMatrices* con la que podemos ejecutar fácilmente unas pruebas con todas las operaciones anteriormente definidas.

```
#| ----------------------------------------------------------------------------------------------------------------------------------
  *    PRUEBAS: pruebas de operaciones con matrices
 ---------------------------------------------------------------------------------------------------------------------------------- |#

(define (pruebaMatrices)
  (display "----------------------- PRUEBAS DE MATRICES DE ENTEROS -----------------------\n")
  (display "\nMatriz identidad: ")
  (display (testmatrices identidad))
  (display "\nMatriz prueba 1: ")
  (display (testmatrices matriz_prueba1))
  (display "\nMatriz prueba 2: ")
  (display (testmatrices matriz_prueba2))
  (display "\nMatriz de prueba 1 + Matriz de prueba 2: ")
  (display (testmatrices ((suma_matrices matriz_prueba1) matriz_prueba2)))
  (display "\nMatriz de prueba 1 x Matriz de prueba 2: ")
  (display (testmatrices ((producto_matrices matriz_prueba1) matriz_prueba2)))
  (display "\nDeterminante matriz de prueba 2: ")
  (display (testenteros (determinante matriz_prueba2)))
  (display "\nRango matriz prueba 1: ")
  (display (testenteros (rango matriz_prueba1)))
  (display "\nRango matriz prueba 2: ")
  (display (testenteros (rango matriz_prueba2)))
  (display "\nRango matriz nula: ")
  (display (testenteros (rango matriz_nula)))
  (display "\nInversa matriz prueba 1: ")
  (display (testmatrices(inversa_matriz matriz_prueba1)))
  (display "\nInversa matriz prueba 2: ")
  (display (testmatrices(inversa_matriz matriz_prueba2)))
  (display "\nMatriz prueba 1 ^ 2: ")
  (display (testmatrices((potencia_matrices matriz_prueba1) dos)))
  (display "\nMatriz prueba 2 ^ 3: ")
  (display (testmatrices((potencia_matrices matriz_prueba2) tres)))
  (display "\n\n--------------------------------------------------------------------------------")
  )
```

ERRORES O ASPECTOS NO IMPLEMENTADOS

Un aspecto que podríamos considerar como no implementado es que, en las operaciones con enteros, en el cálculo de la inversa no hemos aplicado el algoritmo extendido de Euclides ya que considerábamos mucho más sencillo aplicar la fórmula (A * A^-1) = 1 mediante un procedimiento recursivo para hallar la solución.

Sin embargo, aunque esto no lo consideramos un error, también mencionar el hecho de que no pasamos por parámetro de cada operación el módulo, sino que lo creamos de forma global para que se aplique a cada función el mismo módulo.

BIBLIOGRAFÍA

https://docs.racket-lang.org/guide/lambda.html

https://www.campusmvp.es/recursos/post/Que-es-la-Currificacion-en-programacion-funcional.aspx

https://es.wikipedia.org/wiki/Inverso_multiplicativo

https://www.superprof.es/blog/aritmetica-aprender-a-dividir/

https://es.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses

https://es.wikipedia.org/wiki/Exponenciación_binaria

https://www.matricesydeterminantes.com/matrices/potencias-de-matrices-2x2-y-3x3-ejemplos-y-ejercicios-resueltos/

https://www.superprof.es/apuntes/escolar/matematicas/algebralineal/determinantes/matriz-inversa.html

https://es.planetcalc.com/3311/ (comprobacion inverso modular)

https://www.matesfacil.com/calculadoras/matrices/calculadora-online-matriz-inversa-adjunta-2x2-3x3-matrices.html (comprobacion adjunta)

https://es.onlinemschool.com/math/assistance/matrix/transpose/ (comprobacion traspuesta)

https://es.onlinemschool.com/math/assistance/matrix/multiply/ (comprobación multiplicación matrices)