Tecnológico de Monterrey

Implementación de métodos computacionales (Gpo 820)

Actividad: Evidencia 2:Implementación de un simulador de almacén automatizado (primera parte)

Federico Castro Zenteno
A01660986

Contexto:

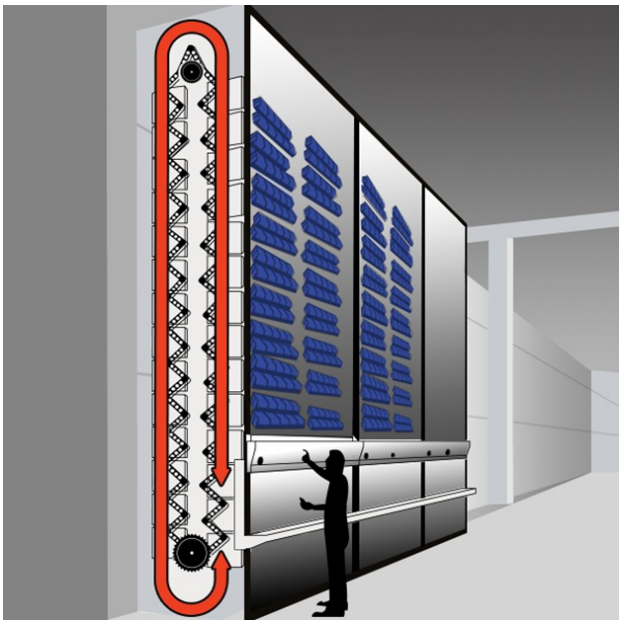
La gestión de almacenes, que conlleva también el control de inventarios, es una de las áreas más estudiadas por la ingeniería industrial, y la tecnología ha jugado un papel muy relevante en los últimos años para lograr que la logística de tiempos y movimientos de los productos sea cada vez más eficiente. Es por eso que ahora puedes recibir una compra en línea de un día para otro.

Para un almacén con un espacio relativamente pequeño, y con una alta variedad de productos pequeños (por ejemplo, los medicamentos en un hospital), el uso de carruseles verticales se ha vuelto una buena práctica en la que el humano hace una búsqueda mínima y directa.

¿Qué tecnología hay detrás de un carrusel automatizado de almacenamiento? Sin duda hay muchas cuestiones mecánicas, pero la inteligencia del carrusel está dada por el software que lo controla y que hace la interfase con el humano al que se le facilita su trabajo.

El software que acompaña a carrusel automatizado es factible desarrollarlo utilizando las herramientas que se están aprendiendo en el curso. Por un lado, los autómatas de estados finitos permiten representar procesos y lenguajes, y es justo lo que estas máquinas necesitan. Por otro lado, el paradigma de la programación funcional representa una ventaja para hacer un programa compacto y eficiente.

La primera parte por resolver en el segundo período del curso consistirá en implementar en Scheme/Racket un simulador de un almacén automatizado por medio de un carrusel vertical de productos pequeños (ej. materiales y medicamentos de un hospital), utilizando un autómata para el proceso de búsqueda, y que esté preparado para dar ciertos avisos al almacenista, como el estado y valor del inventario.



El tamaño del carrusel será configurable en el simulador que se programe, pero no podrá tener menos de 20 filas y menos de 5 recipientes por fila (posiciones horizontales para la ventanilla de acceso). El contenido del carrusel será modelado en Scheme/Racket con las

estructuras de datos necesarias para identificar claramente todos los datos asociados al inventario (cantidad de producto y precios). Esta información será guardada permanentemente en un archivo texto que contenga las listas con los datos. El programa leerá al inicio los datos de este archivo, y trabajará las transacciones de la simulación grabando cada vez en el archivo la información actualizada.

La simulación consistirá en procesar las transacciones de operación del carrusel que estarán en otro archivo de texto con los datos de estas. Las opciones de transacción posibles son las siguientes:

Retirar producto. Esta transacción indicará el nombre del producto y la cantidad de este que se desea retirar. La ejecución de la transacción mostrará en pantalla la secuencia mínima de movimientos que necesitó hacer el carrusel para que el almacenista pueda hacer el retiro. Cuando la transacción no indique nombre de producto, se hará el retiro del producto que esté en la ventanilla en ese momento. Si la cantidad que se indica retirar es mayor a la existente, se retirará lo posible y se indicará con mensaje el error.
Agregar producto. Esta transacción indicará el nombre del producto y la cantidad de este que se desea agregar al carrusel. La ejecución de la transacción mostrará en pantalla la secuencia mínima de movimientos que necesitó hacer el carrusel para que el almacenista pueda hacer la inserción del producto. Cuando la transacción no indique nombre de producto, se hará el agregado del producto que esté en la ventanilla en ese momento. Se asume en esta transacción que la cantidad que se agrega cabe en el recipiente correspondiente.
Mover carrusel. Esta transacción indicará una secuencia de movimientos que se ejecutarán en el carrusel. Los movimientos posibles son arriba y abajo para mover una posición las filas del carrusel, o izquierda y derecha para mover una posición la ventanilla de acceso. Después de que se hayan ejecutado los movimientos, se mostrará en pantalla como resultado, el nombre del producto que quedó en ventanilla de acceso, con su precio e inventario. Si la secuencia de movimientos no es válida dentro del espacio de carrusel, se marcará el error correspondiente.
El diseño del lenguaje para las transacciones es libre, y no requerirá de un proceso formal de traducción, pues es un lenguaje interno del simulador, controlado por el propio programador.

Cuando el programa haya realizado todas las transacciones de operación del carrusel del archivo de entrada, dará los siguientes resultados:

Valor total del inventario en el carrusel (sumatoria de los productos existentes por sus precios).
Productos con poco inventario o inventario nulo, indicando el nombre, cantidad y posición en en carrusel. El criterio para identificar poco inventario puede ser diseñado libremente en el programa, pero deberá estar claramente documentado en el código.
La interfase de ejecución para este programa es completamente libre, y no se requiere nada gráfico ni especializado. Incluso, se puede manejar directamente la llamada a funciones desde el intérprete de Dr. Racket. Lo importante es que los datos se lean de los archivos correspondientes, se actualice el archivo de la máquina, y se desplieguen en pantalla los resultados esperados después de todas las transacciones. Obviamente, cada transacción

deberá actualizar los inventarios correspondientes. El diseño de esta interfase deberá facilitar la comprobación de lo que se ejecuta.

El diseño de este simulador DEBERÁ considerar el uso de un autómata de estados finitos para representar el comportamiento del carrusel. El lenguaje que representará este autómata está basado en las entradas de las 4 posibles direcciones de movimientos en el carrusel: arriba, abajo, izquierda y derecha, siendo cada estado del autómata el equivalente a un recipiente en el carrusel. La implementación de este modelo podrá hacerse de la manera que se considere más eficiente, y no se requerirá de un método de implementación en específico.

Definición de estructuras de datos:

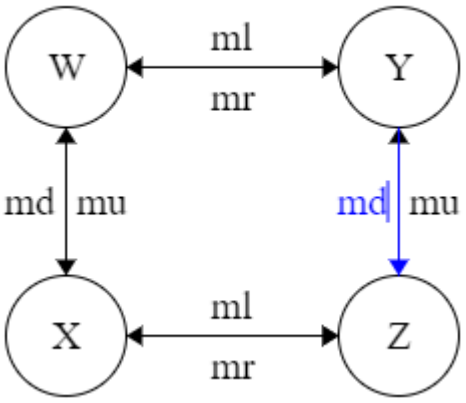
Esta estructura de datos es una “Matriz”, o sea, una lista de listas. Cada lista pequeña dentro de cada lista estaba diseñada con tres cosas en mente: Nombre, cantidad y coordenadas. Esta lista de listas tiene un elemento más, este elemento es el estado actual de la máquina. Este estado actual está ubicado en la última posición de la lista de listas. Esto con la intención de comodidad al momento de desarrollar y mover alrededor todos los datos que se encuentran. Por lo tanto, un ejemplo de esta lista de lista, implementada en Python, y caracterizada en 2D, se vería así:

```
[
[“NAME1”, 3,  0.0], [“NAME2”,  2, 0.1 ],[“NAME3”,  5, 0.2 ],
[“NAME4”, 3,  1.0], [“NAME5”,  2, 1.1 ],[“NAME6”,  5, 1.2 ],
[“NAME7”, 3,  2.0], [“NAME8”,  2, 2.1 ],[“NAME9”,  5, 2.2 ],
“NAME9”
]
```

La implementación en scheme se ve algo así con ejemplos:

```
((((“Leche” 2 1.9) (“Café” 193 4.0) (“Azúcar” 101 5.2)
(“Arroz” 250 2.3) (“Chocolate” 273 5.6)) ((“Aceite” 35 6.6)
(“Sal” 66 5.6) (“Harina” 126 2.7) (“Cerveza” 140 7.6) (“Vino”
188 1.9))) “Arroz”)
```

Diseño de autómata:



Este es el diseño del autómata de la forma más sencilla posible, los movimientos son:

ml = Move-left (Moverse a la izquierda)
mr = Move-right (Moverse a la derecha)
mu = Move-up (Mover arriba)
md = Move-down (Mover abajo)
[EL AUTÓMATA ESTÁ USANDO ACRÓNIMOS, EN LA IMPLEMENTACIÓN SE USARÍAN LOS NOMBRES. LA HERRAMIENTO NO PERMITÍA GENERAR TEXTOS LARGOS]

Este diseño es completamente escalable, pues el agregar los estados no cambia para nada el funcionamiento del movimiento.

Handling de autómata:
La función dedicada para el funcionamiento del autómata solo se encarga de leer linea por linea cada una de las funciones que se encuentran dentro del texto, sin embargo no hice que pudiera funcionar de manera correcta. Un texto que podría ser utilizado para este problema podría ser:
(move-right)
(move-left)
(move-up)
(add 10 “Chocolate)

Mutabilidad:
El código, al estar apoyado completamente en la recursividad, es completamente mutable. Se puede tener de incrementar el tamaño de este carrusel de manera continua sin ningún problema. Solo es cuestión de agregar una cantidad en serie de productos.

Ventajas/Desventajas:
Este tipo de lenguajes están dedicados a que sean elegantes, minimalistas y con un enfoque en el funcionalismo, un lenguaje dinámico que permite escribir funciones de

diferentes formas. Sin embargo, es increíblemente empinada la curva de aprendizaje de este tipo de lenguajes. Es complicadísimo. Sin duda es el lenguaje más difícil que he tenido que aprender, aunque se hubieran generado unos 2-3 archivos para hacer esto en python, escribir este tipo de programas en python o en javascript. Es increíblemente dificil terminar de envolver mi cabeza alrededor de muchos conceptos de scheme, solo practicandolos y haciéndolos me ayudaron, pero a diferencia de python, es que aquí realmente es muy desalentador el progreso. También, la recursividad es uno de los mayores problemas que tengo con este tipo de lenguaje de programación.

Código:

```
#lang racket
(define (read-file file)
  ; Reads the file that contains the carousel and the current state.
  (define input (open-input-file file))
  (define content (read input))
  (close-input-port input)
  content)

(define (get-carousel)
  ; Generates the carousel list every time a transaction is executed.
  (car (read-file "carrusel.txt")))

(define (get-current-state)
  (cadr (read-file "carrusel.txt")))

(define rows (length (get-carousel)))
(define columns (length (car (get-carousel))))

;Get coordinates
(define (get-coordinates lst f x y)
  (if (null? lst) f
      (if (<= y columns)
          (get-coordinates (cdr lst) (append f (list (list (caar lst) (cons x y))))
x (+ y 1))
          (get-coordinates lst f (+ x 1) 1))))
;Crear diccionario
(define (get-dictionary)
  (get-coordinates (apply append (get-carousel)) '() 1 1))

(define (search-product product)
  ; Searches for a product by its name and returns its coordinates.
  (cadar (filter (lambda (p) (equal? product (car p))) (get-dictionary))))

(define (search-product-by-user product)
  ; Searches for a product specified by the user. If it returns false, it means it
does not exist.
  (define x (filter (lambda (p) (equal? product (car p))) (get-dictionary)))
  (if (null? x) #f
      (cadar x)))
```

```
(define (get-details x y lst)
  ; Receives the coordinates of the product and returns its name, quantity, and
price.
  (define (get-element y lst)
    (if (not (= 1 y))
        (get-element (- y 1) (cdr lst))
        (car lst)))
  (if (not (= 1 x))
      (get-details (- x 1) y (cdr lst))
      (get-element y (car lst))))

(define (replace-element product x y lst f)
  ; Changes the quantity of the product in the carousel and then overwrites the
file.
  (define (replace-row product y lst f)
    (cond
      ((null? lst) (list f))
      ((= y 1) (replace-row product (- y 1) (cdr lst) (append f (list product))))
      (else (replace-row product (- y 1) (cdr lst) (append f (list (car lst)))))))
  (cond
    ((null? lst) f)
    ((= 1 x) (replace-element product (- x 1) y (cdr lst) (append f (replace-row
product y (car lst) '()))))
    (else (replace-element product (- x 1) y (cdr lst) (append f (list (car
lst)))))))

(define total-value (apply + (map (lambda (x) (* (cadr x) (caddr x))) (apply append
(get-carousel))))) ; Calculates the total value of the inventory.
;Low stock
(define (get-low-stock-products-aux lst f)
  (if (null? lst) f
      (get-low-stock-products-aux (cdr lst) (append f (list (list (caar lst) (cadar
lst) (search-product (caar lst))))))))
(define low-stock-products (get-low-stock-products-aux (filter (lambda (x) (> 20
(cadr x))) (apply append (get-carousel))) '())) ; Returns the products with less
than 20 units.

(define (write-file lst)
  ; Overwrites the carousel file with the received list.
  (define output (open-output-file "carrusel.txt" #:exists 'replace))
  (write lst output)
  (close-output-port output))
;Arriba
(define (move-up)
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (if (= 0 (- (car coords) 1))
      (begin
        (displayln (get-details rows (cdr coords) lst)))
      (begin
        (displayln (get-details (- (car coords) 1) (cdr coords) lst))
```

```
        (write-file (list (get-carousel) (car (get-details (- (car coords) 1) (cdr
coords) lst)))))))
;Abaj
(define (move-down)
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (if (= rows (+ (car coords) 1))
      (begin
        (displayln (get-details 1 (cdr coords) lst)))
      (begin
        (displayln (get-details (+ (car coords) 1) (cdr coords) lst))
        (write-file (list lst (car (get-details (+ (car coords) 1) (cdr coords)
lst)))))))
;Izq
(define (move-left)
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (if (= 0 (- (cdr coords) 1))
      (displayln "Invalid movement.")
      (begin
        (displayln (get-details (car coords) (- (cdr coords) 1) lst))
        (write-file (list lst (car (get-details (car coords) (- (cdr coords) 1)
lst)))))))
;Der
(define (move-right)
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (if (< columns (+ (cdr coords) 1))
      (displayln "Invalid movement.")
      (begin
        (displayln (get-details (car coords) (+ (cdr coords) 1) lst))
        (write-file (list lst (car (get-details (car coords) (+ (cdr coords) 1)
lst)))))))

(define (move-up-aux x1 x2 lst)
  (if (= x1 x2) lst
      (if (= x1 1)
          (move-up-aux rows x2 (append lst (list 'move-up)))
          (move-up-aux (- x1 1) x2 (append lst (list 'move-up))))))

(define (move-down-aux x1 x2 lst)
  (if (= x1 x2) lst
      (if (= x1 rows)
          (move-down-aux 1 x2 (append lst (list 'move-down)))
          (move-down-aux (+ x1 1) x2 (append lst (list 'move-down))))))

(define (get-distance-y y1 y2 lst)
  (if (= y1 y2) lst
      (if (> y1 y2)
          (get-distance-y (- y1 1) y2 (append lst (list 'move-left)))
          (get-distance-y (+ y1 1) y2 (append lst (list 'move-right))))))
```

```
(define (get-distance x1 y1 x2 y2 lst)
  ; Calculates the distance between the current product and the product to be
reached, determines the fastest route.
  (cond
    ((< x1 x2) (if (> (- x2 x1) (/ rows 2))
                   (get-distance 0 y1 0 y2 (move-up-aux x1 x2 lst))
                   (get-distance 0 y1 0 y2 (move-down-aux x1 x2 lst))))
    ((> x1 x2) (if (> (- x1 x2) (/ rows 2))
                   (get-distance 0 y1 0 y2 (move-down-aux x1 x2 lst))
                   (get-distance 0 y1 0 y2 (move-up-aux x1 x2 lst))))
    (else (begin
            (display "Shortest route: ")
            (displayln (get-distance-y y1 y2 lst))))))

(define (remove-n quantity)
  ; Removes the quantity of the product that is currently displayed.
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (define product (get-details (car coords) (cdr coords) lst))
  (if (< (- (cadr product) quantity) 0)
      (begin
        (displayln "Attempted to remove more than the existing quantity.")
        (write-file (list (replace-element (list (car product) 0 (caddr product))
(car coords) (cdr coords) lst '()) (car product))))
      (write-file (list (replace-element (list (car product) (- (cadr product)
quantity) (caddr product)) (car coords) (cdr coords) lst '()) (car product)))))

(define (remove-p quantity prod)
  ; Removes the quantity of the specified product.
  (define origin (search-product (get-current-state)))
  (define destination (search-product-by-user (car prod)))
  (define lst (get-carousel))
  (if (not destination)
      (displayln "Product not found.")
      (begin
        (if (equal? origin destination)
            (void)
            (get-distance (car origin) (cdr origin) (car destination) (cdr
destination) '()))
        (if (< (- (cadr (get-details (car destination) (cdr destination) lst))
quantity) 0)
            (begin
              (displayln "Attempted to remove more than the existing quantity.")
              (write-file (list (replace-element (list (car (get-details (car
destination) (cdr destination) lst)) 0 (caddr (get-details (car destination) (cdr
destination) lst))) (car destination) (cdr destination) lst '()) (car (get-details
(car destination) (cdr destination) lst)))))
            (write-file (list (replace-element (list (car (get-details (car
destination) (cdr destination) lst)) (- (cadr (get-details (car destination) (cdr
destination) lst)) quantity) (caddr (get-details (car destination) (cdr
```

```
destination) lst))) (car destination) (cdr destination) lst '()) (car (get-details
(car destination) (cdr destination) lst))))))))

(define (remove quantity . prod)
  ; Specifies an optional argument for the remove function.
  (if (null? prod)
      (remove-n quantity)
      (remove-p quantity prod)))

(define (add-n quantity)
  ; Adds the quantity to the product that is currently displayed.
  (define coords (search-product (get-current-state)))
  (define lst (get-carousel))
  (define product (get-details (car coords) (cdr coords) lst))
  (write-file (list (replace-element (list (car product) (+ (cadr product)
quantity) (caddr product)) (car coords) (cdr coords) lst '()) (car product))))

(define (add-p quantity prod)
  ; Adds the quantity to the specified product.
  (define origin (search-product (get-current-state)))
  (define destination (search-product-by-user (car prod)))
  (define lst (get-carousel))
  (if (not destination)
      (displayln "Product not found.")
      (begin
        (if (equal? origin destination)
            (void)
            (get-distance (car origin) (cdr origin) (car destination) (cdr
destination) '()))
        (write-file (list (replace-element (list (car (get-details (car
destination) (cdr destination) lst)) (+ (cadr (get-details (car destination) (cdr
destination) lst)) quantity) (caddr (get-details (car destination) (cdr
destination) lst))) (car destination) (cdr destination) lst '()) (car (get-details
(car destination) (cdr destination) lst)))))))

(define (add quantity . prod)
  ; Specifies an optional argument for the add function.
  (if (null? prod)
      (add-n quantity)
      (add-p quantity prod)))

(define (execute-transactions file)
  ; Reads and executes transactions line by line.
  (load file))

;(execute-transactions "transacciones.txt")
(display (get-carousel))
(newline)
(search-product "Azúcar")
(display "Total inventory value: ") (displayln total-value)
(newline)
```

```
(display "Products with low or no inventory: ") (displayln low-stock-products)
```

Reflexion:

Me llevo que NO me gustan este tipo de lenguajes de programación, simplemente no me gusta como es que se tienen que implementar muchas cosas. Este tipo de lenguaje no es para mí, este parcial ha sido un reto para mí. Pues prácticamente mis conocimientos y práctica la pude estar haciendo durante toda esta semana que estaba desarrollando este proyecto, realmente es un reto comprender a profundidad este tipo de funcionamientos. Sin embargo, muchas cosas de la recursión me han servido muchísimo, creo que me siento muchísimo más seguro al respecto de la recursividad. Es increíble el conocimiento que he aprendido, pues cada elemento de scheme/racket apoyan este tipo de pensamiento, sin embargo sigo creyendo que es un lenguaje de programación anticuado y obsoleto. Estos proyectos serían muchísimo más fáciles dentro de cualquier otro lenguaje de programación.