Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Implementación de métodos computacionales

Evidencia #3

Implementación de un simulador de una máquina expendedora (segunda parte)

Docente: Román Martínez Martínez

Nombre: José Gerardo Cantú García **Matricula**: A00830760

Grupo: 850

Fecha de entrega: 13 de Junio del 2022

1. Una explicación breve del **diseño de las estructuras de datos** implementadas en el programa.

En el programa, implementa 3 diferentes estructuras de datos pero las 3 son con el mismo diseño lo único que cambia son sus datos. El diseño que escogí utilizar es el de base de datos ya que iba a facilitar la manipulación y la mutabilidad de los datos y cualquier otro tipo como una matriz o árbol binario iba a ser un poco más complejo.

Inventario de productos (Base de Datos):

((productoID precio cantidad)(productoID precio cantidad)(productoID precio cantidad)...)

((a 22 20) (b 16 20) (c 25 20) (d 14 20) (e 13 20) (f 15 20) (g 26 20) (h 19 20) (i 24 20) (j 12 20))

Inventario de Monedas (Base de Datos):

((moneda cantidad límite)(moneda cantidad límite)(moneda cantidad límite)...)

((1 500 1000) (2 250 500) (5 100 200) (10 50 100) (20 25 50) (50 10 20))

Lista de transacciones (Base de Datos):

((productoID lista-monedas)(productoID lista-monedas)(productoID lista-monedas))

((c (10 5 2 5)) (a (70 10)) (d (20 5)) (b (50)) (e (2 2 5 10 20)) (g (10 10 5)) (i (20 10 2)) (f (50 2 1)) (j (1 2 5 10 10)) (h (5 5 2)) (f (20 2 2 5)) (a (1 1 20)) (b (20 20)) (g (10 10 10)) (e (20 2)) (c (20 1 1 1 1 1)) (j (5 5 5 10)) (i (50)) (h (5 20 10)) (d (10 2 20)))

Lista de archivos (Base de Datos):

((info_arch1)(info_arch2)(.....)(info_archN))

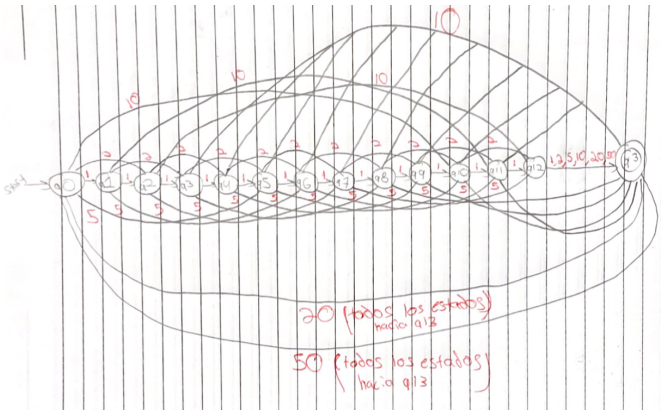
Info_archN:
(tranN.txt invN.txt)

tranN.txt = archivo transaccion

invN.txt = archivo inventarios

2. El **diseño del autómata** que sustenta el proceso de pago con monedas. No necesita ser gráfico, puede ser en el formato matemático.

Inicialmente, había creado el diseño de un autómata específicamente para el ejemplo de un producto con un precio de 13 pesos.



Como era específicamente para ese precio en específico, el autómata no era representativo para el resto de los productos de una máquina expendedora con diferentes productos cada uno con diferente precio así que tuve que generalizarlo para todos los productos. Logre identificar que el estado final de una transición en el autómata era la suma del estado anterior y el símbolo de la transición y de esta forma llegue al diseño del autómata generalizado.

estado-inicial + símbolo = estado-final

simbolo = moneda

3. Una explicación breve de la manera en que se implementó el **algoritmo** que maneja el **autómata**.

Para la implementación del algoritmo que maneja el autómata, utilice la función de acepta? Que desarrolle en mi avance de la situación problema del entrenamiento día 38. La función de este avance estaba desarrollado para un autómata no generalizado así que lo que hice fue actualizar la función auxiliar de transición para que envés de que necesite la lista de transiciones del autómata, simplemente agregue el estado actual con el símbolo de la transición y con eso regresaba el estado siguiente o final. Mientras hace esta transición de una moneda a otra, le agregue una función que verifica si esa moneda es válida y sino regresa mensaje de error y no se completa la transacción ni se actualizan los inventarios. Esta función lo que hace es comparar esa moneda con el inventario de

monedas para ver si es igual a una moneda que existe dentro del inventario y si no encuentra esa moneda en el inventario, entonces significa que la máquina no lo acepta. El algoritmo sigue iterando y agregando de manera recursiva las diferentes monedas de la transacción hasta que la lista de monedas está vacía. Cuando está vacía, con un cond, compara con el precio del producto de la transacción para verificar si el pago se completo, si necesita regresar cambio o si le faltan monedas y lo despliega en la interfaz del programa.

4. Una explicación breve de la manera en que el programa gestiona la **mutabilidad** de los datos. Describir brevemente si el programa es **portable** a otro país en el que haya otro conjunto de monedas y otros precios. ¿Se necesitarían hacer cambios al código o sólo a los datos?

La manera en la que el programa gestiona la mutabilidad es recorriendo la lista de transacciones y por cada transacción, actualiza la base de datos de los inventarios y utiliza esa nueva versión de los inventarios para hacer la siguiente iteración de manera recursiva.

Lo explico de mejor manera en el video

También, el programa es efectivamente portable, esto significa que si se utiliza otro conjunto de monedas y otros precios para los diferentes productos, el programa seguirá funcionando siempre y cuando se utilice el mismo diseño de las estructuras de datos.

5. Una explicación breve de cómo se paralelizaron procesos para hacer más eficiente la ejecución del simulador. Acompañar la explicación con ejemplos de pruebas que demuestren la eficiencia lograda.

La forma en la que yo parallelize los procesos de la implementación de esta segunda parte de la solución a la evidencia fue que desarrolle un archivo que tenia dentro una base de datos en la que cada elemento de la base de datos o cada registro incluye los nombres de los dos archivos para cada máquina (transacciones e inventario). Con el uso del pmap, hice la paralelización de la lectura y ejecución de cada uno de los elementos de esta base de datos para que todos los registros, osea todas las máquinas, se evalúen y actualicen sus archivos de inventarios “al mismo tiempo” (productos y monedas). Para mis pruebas utilicé 15 máquinas. Como se observa en los ejemplos podemos ver el cambio sutil en velocidad y eficiencia de la ejecución sin embargo si llegaran a utilizarse miles de máquinas, la diferencia en eficiencia sería bastante más pronunciada y clara que en la simulación actual que se utilizó

6. Una explicación breve de las ventajas y/o desventajas que se apreciaron al desarrollar el simulador en un lenguaje funcional como Clojure.

Una de las ventajas más grandes de desarrollar el simulador en un lenguaje funcional es la recursividad y el uso de funciones de primer orden. Estas ventajas facilitan bastante el desarrollo del simulador porque de esta manera se podría ir iterando por las diferentes transacciones y actualizando los inventarios sin el uso de memoria lo cual significa que le da al programa la capacidad de correr de manera mucho más rápida a la hora de utilizar y ejecutar varias diferentes máquinas cada una con sus inventarios y transacciones únicas al mismo tiempo con el uso de la paralelización. Una gran ventaja del lenguaje funcional de clojure en específico es que con una simple llamada a “pmap” se puede lograr paralelizar sin ninguna otra necesidad de programación adicional. La única desventaja que yo vi que realmente no es desventaja en sí, es la misma cosa que mencione que es ventaja pero lo menciono porque si fue desventaja para mi personalmente ya que yo estaba acostumbrado a programar utilizando memoria y sin la recursividad y fue una curva de aprendizaje un poco más pronunciada. Al final del día, entiendo que ese es el reto en sí de poder cambiar nuestra forma de pensar de manera tradicional a manera recursiva y poder seguir utilizando esta forma de programar para futuros desarrollos por el resto de nuestras vidas.

7. Una copia del **código** implementado, bien documentado con **comentarios significativos** en los elementos clave de la implementación.

```
(ns test2.core

 (:gen-class)

 (:require [clojure.java.io :as reader]))



; Actualizar archivo de inventarios (PRODUCTOS Y MONEDAS)

(defn update-arch [prod-nuevo mon-nuevo arch]

 (spit (clojure.java.io/resource arch) prod-nuevo) (spit (clojure.java.io/resource
arch) "\n" :append true) (spit (clojure.java.io/resource arch) mon-nuevo :append
true))



; ----- Leer archivo Principal -----
```

```clojure
;; (def archivo "main.txt")


(defn read-arch [arch]
 (read-string (slurp (clojure.java.io/resource arch))))


;; (def main (read-arch "main.txt"))


; ----- Read archivo de transacciones -----
(defn read-tran [arch]
 (read-string (slurp (clojure.java.io/resource arch))))


; ----- Read archivo de inventarios -----
(defn read-maquina [arch]
 (with-open [r (clojure.java.io/reader (clojure.java.io/resource arch))]
   (reduce conj [] (line-seq r))))


(defn get-maquina [maq]
 (map read-string maq))


; ----- Graba la info de los archivos de la maquina en especifica -----
(defn define [prod mon tran]
 (def productos prod)
 (def monedas mon)
 (def transacciones tran))
```

```
; ----- Funcion para sacar la actualizacion de inventarios despues de todas las transacciones -----

(defn ultimo [lista]

 (if (empty? (next lista))

   (first lista)

   (ultimo (next lista))))



;; ----- Interfaz de Usuario -----

; Ayuda a verificar si las monedas insertadas son validas

(defn checa-monedaI [monedas moneda]

 (cond

   (empty? monedas) false

   (= moneda (first (first monedas))) true

   :else (checa-monedaI (next monedas) moneda)))



(defn transicionI [estado0 moneda]

 (if (checa-monedaI monedas moneda)

   (+ estado0 moneda)

   (concat (list (quote "Moneda de") moneda (quote "no se acepta")))))



(defn verificarI [secuencia estado precio]

 (if (empty? secuencia)

   (cond

     (= estado precio) (quote "Pago exacto") ; Mensaje de pago exacto

     (> estado precio) (concat (list (quote "Cambio de: ") (- estado precio) (quote "pesos"))) ; Calcula y despliega el cambio
```

```
     :else (concat (list (quote "Te faltaron: ") (- precio estado) (quote "pesos"))))
; Calcula y despliega el dinero que faltaria para completar transaccion

   (if (list? (transicionI estado (first secuencia))) ; checa si la moneda es valida

     (transicionI estado (first secuencia)) ; Si no es valida manda el mensaje de
error

     (verificarI (next secuencia) (transicionI estado (first secuencia)) precio)))) ;
Si si es valida entonces pasa a la siguiente moneda



; 3. Si si existe el producto y tiene inventario, checa si el pago es suficiente y
calcula el cambio al igual que las monedas sobrantes en caso de que las haya

(defn aceptaI? [registro-producto secuencia]

 (verificarI secuencia 0 (first (next registro-producto))))



; 2. Si efectivamente existe una transaccion, verifica si el producto existe y tiene
inventario disponible

;; (defn productoI? [productos producto pago]

;;   (cond

;;     (empty? productos) (quote "Producto no existe") ; Despliega mensaje de que
producto no existe

;;     ((and (= (first (first productos)) producto) (> (first (next (next (first
productos)))) 1)) (aceptaI? (first productos) pago)) ; Pasa a verificar el pago

;;     ((and (= (first (first productos)) producto) (<= (first (next (next (first
productos)))) 1)) (quote "No hay producto suficiente")) ; Despliega mensaje de que el
producto no tiene suficiente inventario

;;     :else (productoI? (next productos) producto pago)))



(defn productoI? [productos producto pago]

 (if (empty? productos)

   (quote "Producto no existe") ; Despliega mensaje de que producto no existe
```

```
   (if (and (= (first (first productos)) producto) (> (first (next (next (first productos)))) 1))

     (aceptaI? (first productos) pago)

     (if (and (= (first (first productos)) producto) (<= (first (next (next (first productos)))) 1))

       (quote "No hay producto suficiente")

       (productoI? (next productos) producto pago))))) ; Pasa a verificar el pago



(defn maquina [transacciones]

 (if (empty? transacciones)

   '()

   (cons (productoI? productos (first (first transacciones)) (first (next (first transacciones)))) (maquina (next transacciones)))))



; ----------ACTUALIZACION DE PRODUCTOS----------

(defn encuentra [producto-lista prod]

 (if (= producto-lista prod)

   1

   0)) ; checa si a ese producto en especifico es al que se le resta el inventario o no



(defn actualiza-productos [reg-producto prod]

 (cons (first reg-producto) (list (first (next reg-producto)) (- (first (next (next reg-producto))) (encuentra (first reg-producto) prod))))) ; Actualiza el producto especifico de la transaccion reduciendole 1 a su inventario



; 4. Actualizar inventario de monedas

(defn recorre-productos [lista-productos producto]
```

```
 (if (empty? lista-productos)

   '()

   (cons (actualiza-productos (first lista-productos) producto) (recorre-productos
(next lista-productos) producto)))) ; Crea el inventario de productos actualizado



; Ayuda a verificar si las monedas insertadas son validas

(defn checa-monedaP [monedas moneda]

 (cond

   (empty? monedas) false

   (= moneda (first (first monedas))) true

   :else (checa-monedaP (next monedas) moneda)))



(defn transicionP [estado0 moneda]

 (if (checa-monedaP monedas moneda)

   (+ estado0 moneda)

   (concat (list (quote "Moneda de") moneda (quote "no se acepta")))))



(defn verificarP [prod lista-productos secuencia estado precio]

 (if (empty? secuencia)

   (cond

     (= estado precio) (recorre-productos lista-productos prod) ; Como si se completa
el pago, va a actualiza el inventario de productos

     (> estado precio) (recorre-productos lista-productos prod) ; Como tambien se
completa el pago, va a actualiza el inventario de productos

     :else lista-productos) ; Si no se completa el pago entonces no actualiza
inventario de productos porque no se compro ni uno

   (if (list? (transicionP estado (first secuencia))) ; checa si la moneda es valida
```

```
    lista-productos ; Si no es valida no actualiza el inventario de productos

    (verificarP prod lista-productos (next secuencia) (transicionP estado (first
secuencia)) precio)))) ; Si si es valida entonces pasa a la siguiente moneda



; 3. Si si existe el producto y tiene inventario, checa si el pago es suficiente y
calcula el cambio al igual que las monedas sobrantes en caso de que las haya

(defn aceptaP? [lista-productos registro-producto secuencia]

 (verificarP (first registro-producto) lista-productos secuencia 0 (first (next
registro-producto))))



; 2. Si efectivamente existe una transaccion, verifica si el producto existe y tiene
inventario disponible

(defn productoP? [lista-productos productos producto pago]

 (if (empty? productos)

   lista-productos ; Si no existe el producto no actualiza inventario de productos
proque no hubo transaccion

   (if (and (= (first (first productos)) producto) (> (first (next (next (first
productos)))) 1))

     (aceptaP? lista-productos (first productos) pago)

     (if (and (= (first (first productos)) producto) (<= (first (next (next (first
productos)))) 1))

       lista-productos

       (productoP? lista-productos (next productos) producto pago))))) ; Pasa a
verificar el pago



; 1. Validar que exista una transaccion y si ya no hay transacciones entonces se acaba

(defn recorre-pagosP [lista-productos lista-pagos]

 (if (empty? lista-pagos)

   nil
```

```
   (cons (productoP? lista-productos productos (first (first lista-pagos)) (first
(next (first lista-pagos)))) (recorre-pagosP (productoP? lista-productos productos
(first (first lista-pagos)) (first (next (first lista-pagos)))) (next lista-pagos)))))

; Crea la lista de todas las iteraciones de actualizacion del inventario de productos



; ----------ACTUALIZACION DE MONEDAS----------

; Saca la lista de las monedas que forman parte del cambio con su cantidad para cada
una

(defn lista-cambio [cantidad lista-monedas]

 (cond

   (empty? lista-monedas) '()

   (= (quot cantidad (first (first lista-monedas))) 0) (lista-cambio cantidad (next
lista-monedas))

   (and (>= (quot cantidad (first (first lista-monedas))) 1) (>= (first (next (first
lista-monedas))) (quot cantidad (first (first lista-monedas))))) (cons (list (first
(first lista-monedas)) (quot cantidad (first (first lista-monedas)))) (lista-cambio
(rem cantidad (first (first lista-monedas))) (next lista-monedas)))

   (and (= (quot cantidad (first (first lista-monedas))) cantidad) (>= (first (next
(first lista-monedas))) (quot cantidad (first (first lista-monedas))))) (cons (list
(first (first lista-monedas)) (quot cantidad (first (first lista-monedas))))
(lista-cambio cantidad (next lista-monedas)))

   :else (lista-cambio cantidad (next lista-monedas))))



; Sacar la cantidad de la moneda que sea parte del cambio

(defn cant-cambio-moneda [lista moneda]

 (cond

   (empty? lista) 0

   (= moneda (first (first lista))) (first (next (first lista)))

   :else (cant-cambio-moneda (next lista) moneda)))
```

```
(defn cuenta [moneda pago]

 (cond

   (empty? pago) 0

   (= moneda (first pago)) (+ 1 (cuenta moneda (next pago)))

   :else (cuenta moneda (next pago))))



(defn actualiza-monedas [moneda pago cant-cambio]

 (cons (first moneda) (list (- (+ (cuenta (first moneda) pago) (first (next moneda)))
(cant-cambio-moneda (lista-cambio cant-cambio (reverse monedas)) (first moneda)))
(first (next (next moneda))))))

; Actualiza la moneda especifica de la transaccion haciendo la suma de la cantidad de
veces que se encuentra en la transaccion y reduciendole la cantidad que forma parte
del cambio



; 4. Actualizar inventario de monedas

(defn recorre-monedas [lista-monedas pago cant-cambio]

 (if (empty? lista-monedas)

   nil

   (cons (actualiza-monedas (first lista-monedas) pago cant-cambio) (recorre-monedas
(next lista-monedas) pago cant-cambio)))) ; Crea el inventario de monedas actualizada



; Ayuda a verificar si las monedas insertadas son validas

(defn checa-moneda [monedas moneda]

 (cond

   (empty? monedas) false

   (= moneda (first (first monedas))) true

   :else (checa-moneda (next monedas) moneda)))
```

```
(defn transicion [estado0 moneda]

 (if (checa-moneda monedas moneda)

   (+ estado0 moneda)

   (list (quote "Moneda de") moneda (quote "no se acepta"))))



(defn verificar [lista-monedas toda-secuencia secuencia estado precio]

 (if (empty? secuencia)

   (cond

     (= estado precio) (recorre-monedas lista-monedas toda-secuencia 0) ; Como si se
completa el pago, va a actualiza el inventario de monedas

     (> estado precio) (recorre-monedas lista-monedas toda-secuencia (- estado
precio)) ; Como tambien se completa el pago, va a actualiza el inventario de monedas

     :else lista-monedas) ; Si no se completa el pago entonces no actualiza inventario
de momedas porque no se compro ni uno

   (if (list? (transicion estado (first secuencia))) ; checa si la moneda es valida

     lista-monedas ; Si no es valida manda el mensaje de error

     (verificar lista-monedas toda-secuencia (next secuencia) (transicion estado
(first secuencia)) precio)))) ; Si si es valida entonces pasa a la siguiente moneda



; 3. Si si existe el producto y tiene inventario, checa si el pago es suficiente y
calcula el cambio al igual que las monedas sobrantes en caso de que las haya

(defn acepta? [lista-monedas registro-producto secuencia]

 (verificar lista-monedas secuencia secuencia 0 (first (next registro-producto))))



; 2. Si efectivamente existe una transaccion, verifica si el producto existe y tiene
inventario disponible

(defn producto? [lista-monedas productos producto pago]

 (if (empty? productos)
```

```
   lista-monedas ; Si el producto no existe, no actualiza el invenrtario de monedas porque no hubo transaccion

   (if (and (= (first (first productos)) producto) (> (first (next (next (first productos)))) 1))

     (acepta? lista-monedas (first productos) pago)

     (if (and (= (first (first productos)) producto) (<= (first (next (next (first productos)))) 1))

       lista-monedas

       (producto? lista-monedas (next productos) producto pago))))) ; Pasa a verifiar el pago



; 1. Validar que exista una transaccion y si ya no hay transacciones entonces se acaba

(defn recorre-pagos [lista-monedas lista-pagos]

 (if (empty? lista-pagos)

   nil

   (cons (producto? lista-monedas productos (first (first lista-pagos)) (first (next (first lista-pagos)))) (recorre-pagos (producto? lista-monedas productos (first (first lista-pagos)) (first (next (first lista-pagos)))) (next lista-pagos)))))

; Crea la lista de todas las iteraciones de actualizacion del inventario de monedas




;
-------------------------------------------------------------------------------

; Lista de productos actualizadas

(defn prod-ids [updated-p]

 (conj updated-p))



; Lista de monedas actualizadas
```

```
(defn mon-ids [updated-m]

 (conj updated-m))



; Lista de ganancias de cada maquina

(defn ganancia [updated-mon mon]

 (conj (- (apply + (map (fn [x] (* (first x) (second x))) updated-mon)) (apply + (map
(fn [x] (* (first x) (second x))) mon)))))




; ----- definir listas -----

(defn read-main [arch]

 (let [transacciones (read-tran (str (first arch))) productos (first (get-maquina
(read-maquina (str (second arch))))) monedas (second (get-maquina (read-maquina (str
(second arch))))) maq (str (second arch))]

   (define productos monedas transacciones)

   (let [updated-prod (ultimo (recorre-pagosP productos transacciones))
updated-monedas (ultimo (recorre-pagos monedas transacciones)) lista-tran (maquina
transacciones)]

     (update-arch updated-prod updated-monedas maq) ; Actualiza el archivo de
inventario

     (list (ganancia updated-monedas monedas) (prod-ids updated-prod) (mon-ids
updated-monedas)) ; Genera una lista con todo lo que necesito para mis 5 Resultados

     ;lista-tran ; Activar para hacer pruebas

     )))



; ----- EJECUTAR PARALELIZACION -----

(def ejecutar (time (doall (pmap read-main (read-arch "main.txt")))))
```

```
;; ; ========= RESULTADOS ==========

(def lista-prod-final (map second ejecutar))

(def lista-prod-ids (map list (iterate inc 1) lista-prod-final))



(def lista-mon-final (map (fn [x] (second (next x))) ejecutar))

(def lista-mon-ids (map list (iterate inc 1) lista-mon-final))



;; ; Ganancia total del negocio obtenida después de todas las transacciones de venta
procesadas.

(println (apply + (map first ejecutar)))



;; ; Lista del top 10% de máquinas con más ganancia mostrando el identificador de
máquina y la ganancia correspondiente.









;; ; Lista de identificadores de máquinas que necesitan resurtido de algún producto
por estar en el punto de reorden de su inventario. (<= 2)

(def restock '())



(defn no-prod [prod]

 (if (<= (second (next prod)) 2)

   1

   0))



(defn maq-restock? [lista]

 (cond
```

```
   (empty? lista) 0

   (= 1 (no-prod (first lista))) 1

   :else (maq-restock? (next lista))))



(defn id-maq-restock [lista res]

 (cond

   (empty? lista) res

   (= (maq-restock? (second (first lista))) 1) (concat res (cons (first (first lista))
(id-maq-restock (next lista) res)))

   :else (id-maq-restock (next lista) res)))




(println (id-maq-restock lista-prod-ids restock))



;; ; Lista de identificadores de máquinas que necesitan resurtido de alguna
denominación de moneda por tener menos del límite permitido.

(def restock-m '())



(defn no-mon [mon]

 (if (<= (second mon) 2)

   1

   0))



(defn maq-restock-m? [lista]

 (cond

   (empty? lista) 0

   (= 1 (no-mon (first lista))) 1
```

```
   :else (maq-restock-m? (next lista))))


(defn id-maq-restock-m [lista res]
 (cond
   (empty? lista) res
   (= (maq-restock-m? (second (first lista))) 1) (concat res (cons (first (first lista)) (id-maq-restock-m (next lista) res)))
   :else (id-maq-restock-m (next lista) res)))



(println (id-maq-restock-m lista-mon-ids restock-m))


; Lista de identificadores de máquinas que necesitan retiro de monedas de alguna denominación por estar llenas o en el límite para el retiro.
(def retirar-m '())


(defn full-mon [mon]
 (if (> (second mon) (- (second (next mon)) 2))
   1
   0))


(defn maq-retirar-m? [lista]
 (cond
   (empty? lista) 0
   (= 1 (full-mon (first lista))) 1
   :else (maq-retirar-m? (next lista))))
```

```
(defn id-maq-retirar-m [lista res]

 (cond

   (empty? lista) res

   (= (maq-retirar-m? (second (first lista))) 1) (concat res (cons (first (first
lista)) (id-maq-retirar-m (next lista) res)))

   :else (id-maq-retirar-m (next lista) res)))




(println (id-maq-retirar-m lista-mon-ids retirar-m))
```

8. Explicación de las **pruebas** que se realizaron al programa y cuáles fueron los resultados.

Producto no existe: Esta prueba lo que hace es verificar si el producto de la transacción existe dentro de la estructura de datos del inventario de productos.

Producto no tiene inventario disponible: Esta prueba lo que hace es verificar si hay más de 1 en cantidad del producto de la transacción y si no manda mensaje de error. Lo que hace es comparar el id del producto de la transacción con cada producto del inventario de productos y si se encuentra el producto verifica si su valor de cantidad es mayor a 1 o no.

Moneda no existe: Esta prueba lo que hace es verificar si alguna moneda dentro de la transacción no es válida dentro de la máquina. Esta prueba lo que hace es que durante la función de transición que forma parte del acepta? Verifica que cada moneda exista dentro del inventario de monedas y si no hace la transición al siguiente estado y envés manda el mensaje de error.

Pago Exacto: Esta prueba lo que hace es que despliega mensaje de pago exacto y no regresa cambio. Lo que hace es que al terminar las transiciones consigue el total del pago en una transacción, lo compara con el precio del producto de la transacción y si son iguales manda el mensaje.

Pago con Cambio: Esta prueba lo que hace es que despliega la cantidad de cambio que regresó al usuario por esa transacción en específica. Lo que hace es que al terminar las

transiciones consigue el total del pago en una transacción, lo compara con el precio del producto de la transacción y si es mayor manda el mensaje calculando el dinero que se regresa.

Pago incompleto: Esta prueba lo que hace es que despliega la cantidad de dinero que falto para poder alcanzar a comprar el producto seleccionado en esa transacción. Lo que hace es que al terminar las transiciones consigue el total del pago en una transacción, lo compara con el precio del producto de la transacción y si es menor manda el mensaje calculando el dinero que falto.

9. Una breve reseña de cómo fue la **experiencia de aprendizaje** al desarrollar la solución.

La experiencia de aprendizaje al desarrollar la solución a esta segunda parte de la evidencia 2 fue bastante más agradable y rápida que la primera parte. Todos los conocimientos que logre acumular durante el desarrollo de la primera parte me ayudaron bastante a terminar de agregar lo que faltaba de paralelización con bastante más facilidad y conocimiento de lo que tenía que realizar. Aunque el tema de la paralelización fue unos de los temas que más se complicó de todos los temas vistos en clase durante el semestre, la implementación fue bastante sencilla y todo fue gracias al lenguaje de Clojure y la simpleza con la que se puede implementar la paralelización a un map simplemente con agregar la letra p al principio. Para esta segunda parte de la evidencia, otro reto no necesariamente desventaja que me enfrente durante mi aprendizaje era el cambio de sintaxis de Scheme que ya estaba logrando dominar, a Clojure aunque sin embargo me ayudo bastante mi conocimiento anterior de Scheme para poder aprender Clojure con mayor velocidad.

Video de aproximadamente 5 minutos en el que se muestre la ejecución del programa con algunas pruebas significativas y explicando algunos de los puntos que se han documentado.

Liga: https://youtu.be/Yg8NCLzD5xw