# Situación Problema 2 – Parte 2: Detrás de una máquina expendedora: el poder de un autómata y un lenguaje funcional. Evidencia #3: Implementación de un simulador de una máquina expendedora (segunda parte)

**Juan Daniel Rodríguez Oropeza A01411625**

**Monterrey, Nuevo León, México**

**Junio 15, 2022**

**Ing. Román Martínez Martínez**

**Instituto Tecnológico y de Estudios Superiores de Monterrey**

**Implementación de Métodos Computacionales**

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

   La estructura que tiene cada base de datos es la siguiente:

a) **Transacciones:** La estructura de datos que se utilizan aquí es la base de datos siguiendo esta estructura: *(id-maquina, código, producto, (lista-transiciones)).*

b) **Inventario y Repositorio:** Este archivo pertenecerá a cada máquina, el cual contendrá dos líneas, es decir, dos bases de datos, la primera correspondiente al inventario de productos con estos parámetros: *(codigo-producto, nombre-producto, precio, cantidad-disponible)*

   Y la segunda, la cual sería el repositorio de monedas, sería a esta forma: *(id-monedas, valor, cantidad-disponible, capacidad-maxima)*

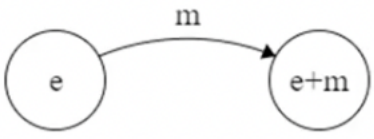
c) **Registro de las máquinas:** Este archivo contiene información de todas las máquinas existentes en una base de datos, tendrá la siguiente estructura: *(id-máquina, tipo-producto-principal, ubicación))*

   **Nota:** El parámetro de tipo-producto-principal se refiere a si la especialidad de la máquina son bebidas, frituras, electrónica, etc.

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

   Para el diseño del autómata hubo muchas implementaciones, primero me había basado en un modelo gráfico, pero debido a cuestiones de eficiencia y de portabilidad, se cambió a un modelo matemático el cual es representado en Clojure mediante funciones y operaciones.

   Básicamente lo que sucede es que la transición de los estados es mediante una suma de la secuencia de monedas hasta llegar al estado destino. El modelo en el que se basó dicha implementación es este:



Este es un modelo que generaliza al autómata de la máquina dispensadora, ya que prácticamente, como expliqué anteriormente, se puede resumir en ir sumando los valores de las monedas que se encuentran en la secuencia de transiciones.

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

   Para implementar el algoritmo que maneja al autómata se tuvo que utilizar primeramente la función de *acepta?* junto con otras funciones helper para poder primeramente validar que se encuentren las monedas que ya están registradas en el archivo de las bases de datos. Para esta parte se usó principalmente la función de *miembro?* que se programó en Scheme/Racket, pero ahora en Clojure se usó su equivalencia, que es la función de Java *.contains.*

   Para poder calcular las transiciones se tuvo que tomar en cuenta el precio del producto como el estado aceptor para que pudiese parar el autómata. También se toma en cuenta como estados aceptores todo aquel valor que sea mayor el precio, aunque esta vez se toma un diferente camino para proceder a calcular el cambio.

   Dentro de la función de transición se van sumando los estados mediante llamadas recursivas, tomando en cuenta siempre quien es el nuevo *estado_origen*. Si se llega al precio del producto, ahí es cuando paran los cálculos y procede a actualizar el inventario.

   Pero si se requiere dejar cambio, se pasa un proceso más complejo donde se tiene que tomar en cuenta el precio, el valor que pagó el cliente y la disponibilidad de las monedas, ahí se utiliza muchas llamadas recursivas y una constante actualización del repositorio de monedas para “regresar” en estados hasta que la diferencia entre el precio y el valor pagado sea de cero.

   Para ello se utilizó las funciones de *INDICE*, *ACCESA-N*, y *reemplaza* para poder verificar todas las posibilidades y que se complete el proceso de dar cambio.

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?**
   El programa gestiona la mutabilidad mediante la escritura y lectura de archivos. Cada vez que se termina de procesar una transacción se escribe en el archivo de texto donde se encuentran las bases de datos para reemplazar el contenido del inventario de productos y el repositorio de monedas. En caso de que no haya habido cambios por algún error en la transacción porque se haya introducido una moneda desconocida o por otra razón, simplemente se ignora y pasa a la siguiente transacción.
   También quiero mencionar que en la implementación de mi código está lo suficiente generalizado para poder modificar los valores de los registros del inventario de productos y el repositorio de monedas, así como quitar y agregar nuevas monedas y productos, sin importar el valor y la cantidad de registros agregados. Solamente hay que tomar en cuenta que tengan la misma estructura que se mencionó en el punto #1. Para poder llevar esto a cabo se usaron las funciones de *.contains, INDICE, ACCESA-N*, y *reemplaza* que se pueden encontrar en el código de Clojure ya documentado.
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.**
   Lo más difícil de la paralelización fue acomodar y ordenar los datos para que se pueda hacer el recorrido debido las listas.
   Primero que nada, el *pmap* tenía que recorrer 3 bases de datos y 1 lista plana a la vez, la colección de todos los inventarios de cada máquina, la colección de transferencias correspondientes a cada máquina, la colección de todos los repositorios de cada máquina, y el nombre del archivo de la base de datos (inventario y repositorio de cada máquina.

Para que se hiciera el recorrido de estas cuatro estructuras de datos al mismo tiempo, las cuatro debían tener la misma longitud. Para ello se generaron un archivo de base de datos (inventario y repositorio) y un archivo de transacciones por cada máquina.
Una vez hecho esto utilicé la función de *(map vector lista1, lista2, lista3, lista4)*, lo que provocaría lo siguiente *'((dato1-L1 dato1-L2 dato1-L3 fato1-L4) (dato2-L1 dato2-L2 dato2-L3 dato2-L4) (dato3-L1 dato3-L2 dato3-L3 dato3-L4)…)*
Para esta situación problema, quedaría así *'((invM1 transaccsM1 repM1 nombre-arch-BD-M1) (invM2 transaccsM2 repM2 nombre-arch-BD-M2) (invM3 transaccsM2 repM3 nombre-arch-BD-M3)…)*
De ahí especifique en el map que va dentro de pmap, una función que recibe *"x"*, y llamé ahí adentro a la función de *verif-exis-prod* (recibe múltiples parámetros), la cual inicializa el proceso para llevar a cabo las transacciones, los parámetros serían el *(first x)*, *(second x)*, y demás operaciones con la misma gran lista que incluye a las cuatro estructuras de datos previamente mencionados, permitiendo recorrerlas en el mismo *pmap*.
Hablando de ello, la partición de esta gran lista unida siempre será el 10% de la cantidad de ésta, por lo que sin importar el número de datos que se generen, siempre estará optimizado para ello. De esta manera los pedazos de información que agarra no son demasiado grandes y no se generán muchos hilos.
Sin embargo, con la aplicación del pmap no significa que se tomarán en cuenta todas las transacciones a la vez de una misma máquina, lo que se está paralelizando es el proceso de una sola transacción por máquina, una vez se completa esta paralelización, se llama recursivamente a la función (haciendo un *(map rest transacciones)* para que en la siguiente llamada se ejecute con las transacciones restantes de cada máquina) que ejecutaba el pmap para nuevamente leer el archivo de base de datos de cada máquina, el cual contiene el inventario de productos y repositorio de monedas y

procede nuevamente a paralelizar las siguientes transacciones de cada máquina.
En caso de que una máquina ya no tenga transacciones restantes, se despliega un mensaje respectivo, pero aún así se puede seguir haciéndole *rest* porque el *rest* de una lista vacía es igualmente una lista vacía *'()*, por lo que no da error.
De esta manera se estaría cuidando la condición de carrera, exclusión mutua, y del interbloqueo al paralelizar de manera recursiva cada transacción.
También hay que tomar en cuenta que el código debe ser eficiente, y debido a que el paradigma funcional puede no ser muy eficiente cuando se trata de complejidad de espacio. Así que para mitigar estos efectos decidí convertir las funciones a recursividad terminal para no sobrecargar al stack.

**Comparación de la ejecución de las transacciones de manera secuencial vs paralela:**

## Funciones:

## Secuencial

```
;; Función que ejecuta todas las funciones del codigo para que el proceso de las transacciones sea automático y secuencial.
(defn inicia-operaciones-secuencial [tod-transaccs]
  (let [todos-nuevo-inventario-productos (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máqui
                                                                                                  (filter (comp not #(.isDirectory %))
                                                                                                          (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-
        todos-nuevo-repositorio-monedas (map (fn [x] (with-open [rdr (io/reader x)]
                                                    (read-string (second (doall (line-seq rdr)))))) ;; Para que lea solamente la segunda línea del archivo.
                                             (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longi
                                                                                (filter (comp not #(.isDirectory %))
                                                                                        (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas))))))))]
    (if (not (every? empty? tod-transaccs)) ;; Si no se han procesado todas las transacciones correspondientes a cada máquina ...
      (when true
        (map (fn [x] (verif-exis-prod (first x) (second x) (caddr x) (cadddr x) (first x))) (map vector todos-nuevo-inventario-productos
                                                                                                 tod-transaccs
                                                                                                 todos-nuevo-repositorio-monedas
                                                                                                 (sort-by count (sort (map str (map #(.getPath %)
                                                                                                                                    (filter (comp not #(.isDirectory %))
                                                                                                                                            (tree-seq #(.isDirectory %) #(.listFiles
        (inicia-operaciones-secuencial (map rest tod-transaccs)))
      (Reporte-final todos-original-inventario-productos todos-nuevo-inventario-productos todos-nuevo-repositorio-monedas))))  ;; Llamada para ejecutar el reporte final.
```

```
transacciones sea automático y secuencial.

nt (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
                       (filter (comp not #(.isDirectory %))
                               (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))
:)]
l (line-seq rdr))))) ;; Para que lea solamente la segunda línea del archivo.
#(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
filter (comp not #(.isDirectory %))
       (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))]
transacciones correspondientes a cada máquina ...

first x))) (map vector todos-nuevo-inventario-productos
                tod-transaccs
                todos-nuevo-repositorio-monedas
                (sort-by count (sort (map str (map #(.getPath %)
                                                   (filter (comp not #(.isDirectory %))
                                                           (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-Transacciones-Maquinas)))))))))

roductos todos-nuevo-repositorio-monedas))))  ;; Llamada para ejecutar el reporte final.
```

## Paralelo

```
;; Función que ejecuta todas las funciones del codigo para que el proceso de las transacciones sea automático y paralela.
(defn inicia-operaciones-paralelo [tod-transaccs]
  (let [todos-nuevo-inventario-productos (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máqui
                                                                                                  (filter (comp not #(.isDirectory %))
                                                                                                          (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-
        todos-nuevo-repositorio-monedas (map (fn [x] (with-open [rdr (io/reader x)]
                                                    (read-string (second (doall (line-seq rdr)))))) ;; Para que lea solamente la segunda línea del archivo.
                                             (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longi
                                                                                (filter (comp not #(.isDirectory %))
                                                                                        (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas))))))))]
    (if (not (every? empty? tod-transaccs)) ;; Si no se han procesado todas las transacciones correspondientes a cada máquina ...
      (when true (doall (map (fn [y] (doall
                                      (pmap (fn [x] (verif-exis-prod (first x) (second x) (caddr x) (cadddr x) (first x))) y)))
                             (partition-all (quot (count todos-nuevo-inventario-productos) 10) (map vector todos-nuevo-inventario-productos
                                                                                                    tod-transaccs
                                                                                                    todos-nuevo-repositorio-monedas
                                                                                                    (sort-by count (sort (map str (map #(.getPath %)
                                                                                                                                       (filter (comp not #(.isDirectory %))
                                                                                                                                               (tree-seq #(.isDirectory %) #(.list
          (inicia-operaciones-paralelo (map rest tod-transaccs)))
      (Reporte-final todos-original-inventario-productos todos-nuevo-inventario-productos todos-nuevo-repositorio-monedas))))  ;; Llamada para ejecutar el reporte final.
```

```
nsacciones sea automático y paralela.

(sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
                    (filter (comp not #(.isDirectory %))
                            (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))

line-seq rdr)))))) ;; Para que lea solamente la segunda línea del archivo.
getPath %) ;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
ter (comp not #(.isDirectory %))
    (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))]
nsacciones correspondientes a cada máquina ...

ond x) (caddr x) (cadddr x) (first x))) y)))
oductos) 10) (map vector todos-nuevo-inventario-productos
                  tod-transaccs
                  todos-nuevo-repositorio-monedas
                  (sort-by count (sort (map str (map #(.getPath %)
                                                     (filter (comp not #(.isDirectory %))
                                                             (tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-Transacciones-Maquinas))))))))))

luctos todos-nuevo-repositorio-monedas)))  ;; Llamada para ejecutar el reporte final.
```

En ambas maneras se recorren las 4 colecciones de datos de manera simultánea como la había explicado anteriormente. Aquí la principal diferencia es la aplicación del *pmap* junto con la partición de las listas para hacer más eficiente dicho proceso en la forma paralela, permitiendo un mejor tiempo y evitando problemas como mencioné anteriormente porque siempre se está tratando de forma paralela con transacciones, inventarios y repositorios de diferentes máquinas.

**Nota:** Mi computadora cuenta con 8 núcleos.

**Tiempos – Datos: 30 Máquinas, 40 Productos en cada Inventario, 200 Transacciones.**

**Secuencial**

```
"Elapsed time: 43.571083 msecs"
Ejecutado de forma Secuencial
```

**Paralela**

```
"Elapsed time: 259.926666 msecs"
Ejecutado de forma Paralela
```

Speed Up = Ts / Tp =  0.167628368687651

Eficiencia = Sp / p = 0.020953546085956

**Tiempos – Datos: 100 Máquinas, 50 Productos en cada Inventario, 300 Transacciones.**

**Secuencial**

```
"Elapsed time: 205.280458 msecs"
Ejecutado de forma Secuencial
```

**Paralela**

```
"Elapsed time: 1189.383125 msecs"
Ejecutado de forma Paralela
```

Speed Up = Ts / Tp =  0.172594056267614

Eficiencia = Sp / p = 0.021574257033452

**Tiempos – Datos: 50 Máquinas, 30 Productos en cada Inventario, 500 Transacciones.**

**Secuencial**

```
"Elapsed time: 127.645666 msecs"
Ejecutado de forma Secuencial
```

**Paralela**

```
"Elapsed time: 510.205417 msecs"
Ejecutado de forma Paralela
```

Speed Up = Ts / Tp = 0.250184850546187

Eficiencia = Sp / p = 0.031273106318273

**Tiempos – Datos: 50 Máquinas, 30 Productos en cada Inventario, 700 Transacciones.**

**Secuencial**

```
"Elapsed time: 117.872791 msecs"
Ejecutado de forma Secuencial
```

**Paralela**

```
"Elapsed time: 657.624167 msecs"
Ejecutado de forma Paralela
```

Speed Up = Ts / Tp = 0.179240357813675

Eficiencia = Sp / p = 0.022405044726709

**Tiempos – Datos: 30 Máquinas, 30 Productos en cada Inventario, 1000 Transacciones.**

**Secuencial**

```
"Elapsed time: 82.663166 msecs"
Ejecutado de forma Secuencial
```

**Paralela**

```
"Elapsed time: 781.854042 msecs"
Ejecutado de forma Paralela
```

Speed Up = Ts / Tp = 0.105727107055104

Eficiencia = Sp / p = 0.013215888381888

Como se puede observar, cuando se trata de pocos datos la manera secuencial hace mejor tiempo que de forma paralela, por lo que la eficiencia

es menor que 1. También se puede notar que influye la cantidad de archivos que se generen (con respecto a los inventarios y repositorios). Cuando se trabajan con una cantidad mucho mayor de datos es cuando se pueden notar los beneficios del paradigma paralelo.

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 de los lenguajes funcionales es que tienen transparencia referencial, es decir, que no hay efectos laterales en memoria que alteren el significado de un programa, lo que significa que no habrá problemas con el manejo de datos y archivos al momento de escribir y leer las bases de datos de inventario de productos y repositorio de monedas entre cada transacción.

   En la implementación de este código no se hacen declaraciones en memoria de ninguna variable (solamente para el manejo de archivos), lo cual evita problemas.

   Otro beneficio es la recursividad, la cual es muy conveniente usarla cuando se tiene un problema con un caso base, puesto que va dividiendo el problema en partes más pequeñas hasta llegar a dicho caso base, a partir de entonces comienza a regresar por donde vino, combinando las soluciones que ya calculó.

   Modelar un problema complejo utilizando la iteratividad puede resultar más complicado porque hay que encontrar la manera idónea de almacenar los resultados previamente calculados para después combinarlos, cosa que no sucede con la recursividad y los lenguajes funcionales por el tema de la transparencia referencial que fue explicado anteriormente.

   No obstante, y es aquí donde entran algunas desventajas del paradigma funcional, es que suelen consumir mucha memoria, teniendo una alta complejidad de espacio (a diferencia de la complejidad de tiempo). Esto es debido a que al momento de hacer llamadas recursivas se tiene que almacenar el estado de la función previa.

Este problema también se puede notar tanto en Clojure como en Scheme/Racket, siendo probable que también arroje errores de StackOverFlow, esto se puede mitigar con lo que es la recursividad terminal como mencioné en el punto anterior.

**El contenido del documente continua en las siguientes páginas.**

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

```
(ns a01411625-sp2-2.core
  (:gen-class))

;; Situación Problema #2 Parte 2
;; Juan Daniel Rodríguez Oropeza A01411625

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

;; Nombres archivos y carpetas
(def nombre-archivo-Registro-Maquinas "/Users/danny/Documents/ITESM/4to Semestre/IMECO/Clojure/a01411625_sp2-
2/Maquinas.txt")
(def nombre-carpeta-BD-Maquinas "/Users/danny/Documents/ITESM/4to Semestre/IMECO/Clojure/a01411625_sp2-2/BD-
Maquinas")
(def nombre-archivo-TTMaquinas "/Users/danny/Documents/ITESM/4to Semestre/IMECO/Clojure/a01411625_sp2-
2/Transacciones-Maquinas/TTMaquinas.txt")
(def nombre-carpeta-Transacciones-Maquinas "/Users/danny/Documents/ITESM/4to Semestre/IMECO/Clojure/a01411625_sp2-
2/Transacciones-Maquinas")
;; Funciones helpers

;; Función de miembro->= que devuelve valor si se cumple con la condición, de lo contario regresa falso.
(defn miembro->= [atomo lista]
  (cond (empty? lista) false
        (>= atomo (first lista)) (first lista)
        :else (miembro->= atomo (rest lista))))

;; Función que cuenta cuántas coincidencias hay de un elemento en una lista.
```

```clojure
(defn CUANTOS [atomo lista]
  (cond (empty? lista) 0
        (= atomo (first lista)) (+ (CUANTOS atomo (rest lista)) 1)
        :else (CUANTOS atomo (rest lista))))

;; Función que regresa el elemento de una determinada posición de una lista.
(defn ACCESA-N [pos lista]
  (cond (empty? lista) '()
        (zero? pos) '()
        (= pos 1) (first lista)
        :else (ACCESA-N (- pos 1) (rest lista))))

;; Función que regresa la posición de un elmento en una lista.
(defn INDICE [atomo lista]
  (if (.contains lista atomo)
    (+ 1 (count (take-while (partial not= atomo) lista)))
    '()))


;; Función que reemplaza un elmento en una determinada posicón de una lista.
(defn reemplaza [dato pos lst fin-lst]
  (if (= pos 1)
    (concat (reverse (cons dato fin-lst)) (rest lst))
    (reemplaza dato (- pos 1) (rest lst) (cons (first lst) fin-lst))))

;; Función equvialente al caddr de Scheme/Racket.
(defn caddr [lista]
  (second (rest lista)))

;; Función equivalente al cadddr de Scheme/Racket.
(defn cadddr [lista]
  (second (rest (rest lista))))
```

```clojure
;; Función para convertir de string a int.
(defn parse-int [s]
  (Integer. (re-find  #"\d+" s)))

;; Función helper de validador-espacio-monedas que se dedica principalmente a contar las coincidencias de cada moneda en la secuencia de transiciones.
(defn validador-espacio-monedas-helper [repositorio-monedas transaccion r]
  (if (empty? (rest repositorio-monedas)) ;; Si ya es el último elemento...
    (if (nil? (first r))
      0
      (first r)) ;; Solamente despliega un valor.
    (validador-espacio-monedas-helper (rest repositorio-monedas) transaccion
                                      (concat r (list (CUANTOS (first (nfirst repositorio-monedas)) transaccion))))))
;; Llama recursivamente a la función para agrupar en una lista el número de coincidencias de cada moneda.

;; Función que valida si la transacción es posible de realizar, tomando en cuenta la capacidad máxima de las monedas que hay dentro de la máquina.
(defn validador-espacio-monedas [repositorio-monedas transaccion]
  (if (empty? repositorio-monedas) ;; Si está vacío el repositorio de monedas significa que se han hecho todas las comparaciones y NO se ha encontrado ninguna moneda desconocida.
    true ;; Regresa verdadero.
    (if (> (+ (second (nfirst repositorio-monedas)) (validador-espacio-monedas-helper repositorio-monedas transaccion '())) (second (rest (nfirst repositorio-monedas)))) ;; Hace la comparación para saber si la cantidad de dicha moneda que se encuentra en la secuencia de transicones va a superar el límite del repositorio.
      (str "Error: Se ha superado la capacidad maxima de la moneda de " (first (nfirst repositorio-monedas)) " pesos.") ;; Si se supera el límite del respositorio, se despliega este mensaje.
      (validador-espacio-monedas (rest repositorio-monedas) transaccion)))) ;; Llama recursivamente a la función.


;; Función como predicado que decide si aceptar o no una secuencia de transiciones.
(defn acepta? [repositorio-monedas transaccion]
```

```
  (if (true? (validador-espacio-monedas repositorio-monedas transaccion))
    (cond (empty? transaccion) false
          (.contains transaccion (first (nfirst repositorio-monedas))) true ;; Regresa verdadero si el valor de la moneda se encuentra en la secuencia de transiciones.
          :else (acepta? (rest repositorio-monedas) transaccion)) ;; Llama a la función recursivamente haciendo rest al repositorio de monedas.
    (validador-espacio-monedas repositorio-monedas transaccion))) ;; Llama a la función de validador-espacio-monedas para regresar el mensaje de error correspondiente en pantalla.


;; Función que actualiza el inventario de los productos.
(defn actualiza-inventario-productos [codigo inventario-productos nuevo-archBD]
  (spit nuevo-archBD (prn-str (reemplaza (reemplaza (- (ACCESA-N (INDICE (some #{codigo} (map first inventario-productos)) (map first inventario-productos)) (map cadddr inventario-productos)) 1)
                                                     4 ;; Se sustitye en la posición 4.
                                                     (ACCESA-N (INDICE (some #{codigo} (map first inventario-productos)) (map first inventario-productos)) inventario-productos) ;; Indica que desea sustituir en la sublista donde se encuentra el valor del producto.
                                                     nil)
                                          (INDICE (some #{codigo} (map first inventario-productos)) (map first inventario-productos)) ;; En esta función se checa que coincida con la misma posición del valor del producto cuya cantidad se está actualizando.
                                          inventario-productos
                                          nil))))


;; Funciones de Escritura de Archivos

;; Función que actualiza el repositorio de monedas.
(defn actualiza-repositorio-monedas [repositorio-monedas precio valor-pagado nuevo-archBD]
```

```
  (if (and (number? (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))) ;; Verifica
cuales son las monedas que sean igual o mayor a la diferencia del valor pagado por el usuario en comparación con el
precio del producto.
          (> (ACCESA-N (INDICE (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))
(reverse (map second repositorio-monedas))) (reverse (map caddr repositorio-monedas))) 0))
    (if (= (- valor-pagado (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))) precio)
;; Si con esta acción ya se acompletó el cambio...
      (spit nuevo-archBD (pr-str (reverse (reemplaza (reemplaza
                                                      (- (ACCESA-N (INDICE (miembro->= (- valor-pagado precio)
                                                                                       (reverse (map second
repositorio-monedas))) ;; lista con valores de monedas de manera descenedente
                                                                           (reverse (map second repositorio-
monedas))) ; lista con valores de monedas de manera descendente
                                                                 (reverse (map caddr repositorio-monedas))) ;;
lista con valores de la capacidad de las monedas en el mismo orden que la lista anterior
                                                         1) ;; Se resta 1
                                                      3 ;; Se reemplaza en la posición 3
                                                      (ACCESA-N (INDICE (miembro->= (- valor-pagado precio) ;; Todo
esto indica que desea sustituir en la sublista donde se encuentra el valor de la moneda.
                                                                        (reverse (map second repositorio-
monedas)))
                                                                  (reverse (map second repositorio-monedas)))
                                                            (reverse repositorio-monedas))
                                                     nil)
                                                    (INDICE (miembro->= (- valor-pagado precio) ;; En esta función
se checa que coincida con la misma posición del valor de la moneda cuya cantidad se está actualizando.
                                                             (reverse (map second repositorio-monedas)))
                                                    (reverse (map second repositorio-monedas)))
                                                   (reverse repositorio-monedas)
                                                   nil))) :append true)
      (actualiza-repositorio-monedas (reverse (reemplaza (reemplaza
                                                          (- (ACCESA-N (INDICE (miembro->= (- valor-pagado precio)
```

```
                                                                                (reverse (map second
repositorio-monedas))) ;; lista con valores de monedas de manera descenedente
                                                                                    (reverse (map second repositorio-
monedas))) ;; lista con valores de monedas de manera descendente
                                                                            (reverse (map caddr repositorio-monedas))) ;;
lista con valores de la capacidad de las monedas en el mismo orden que la lista anterior
                                                                  1) ;; Se resta 1
                                                               3 ;; Se reemplaza en la posición 3
                                                               (ACCESA-N (INDICE (miembro->= (- valor-pagado precio) ;;
Todo esto indica que desea sustituir en la sublista donde se encuentra el valor de la moneda.
                                                                                        (reverse (map second
repositorio-monedas)))
                                                                                    (reverse (map second repositorio-
monedas)))
                                                                          (reverse repositorio-monedas))
                                                               nil)
                                                              (INDICE (miembro->= (- valor-pagado precio) ;; En esta
función se checa que coincida con la misma posición del valor de la moneda cuya cantidad se está actualizando.
                                                                                    (reverse (map second repositorio-
monedas)))
                                                                                (reverse (map second repositorio-monedas)))
                                                              (reverse repositorio-monedas)
                                                              nil))
                                        precio ;; El precio es igual.
                                        (- valor-pagado (miembro->= (- valor-pagado precio) (reverse (map second
repositorio-monedas)))) ;; Se actualiza la diferencia de cambio.
                                        nuevo-archBD)) ;; Se selecciona el archivo del que se está leeyendo.
    (actualiza-repositorio-monedas (reverse (rest (reverse repositorio-monedas))) precio valor-pagado nuevo-archBD)))
;; Si no hay coincidencia sigue buscando recursivamente la moneda que esté disponible.

;; (actualiza-repositorio-monedas '((1 1 49 50) (2 2 13 45) (3 5 36 40) (4 10 10 25) (5 20 15 20) (6 50 1 5)) 19 20
"/Users/danny/Documents/ITESM/4to Semestre/IMECO/Clojure/a01411625_sp2-2/CopiaBasesDatos.txt")
```

```
;; Función que actualiza el repsoitorio de monedas tomando en cuenta las monedas que recibe por parte de la
transacción.
(defn actualiza-repositorio-monedas-helper-toma-cuenta-recibe [repositorio-monedas transaccion precio valor-pagado
nuevo-archBD]
  (if (empty? transaccion) ;; Si ya terminó de checar todas las monedas de la transacción.
    (if (= precio valor-pagado) ;; Si el pago fue exacto...
      (spit nuevo-archBD (pr-str repositorio-monedas) :append true) ;; Escribe en el archivo el repositorio de
monedas sin dar cambio.
      (actualiza-repositorio-monedas repositorio-monedas precio valor-pagado nuevo-archBD)) ;; Actualiza el
repositorio de monedas tomando en cuenta el cambio.
    (actualiza-repositorio-monedas-helper-toma-cuenta-recibe (reemplaza (reemplaza
                                                                         (inc (ACCESA-N (INDICE ;; Se suma 1 porque
se están añadiendo monedas.
                                                                                        (some #{(first transaccion)}
                                                                                              (map second
repositorio-monedas)) ;; lista con valores de las monedas
                                                                                        (map second repositorio-
monedas)) ;; lista con valores de las monedas
                                                                                       (map caddr repositorio-
monedas))) ;; lista con la cantidad disponible de monedas
                                                                         3 ;; Se reemplaza en la posición 3.
                                                                         (ACCESA-N (INDICE (some #{(first
transaccion)} ;; Todo esto indica que desea sustituir en la sublista donde se encuentra el valor de la moneda.
                                                                                           (map second
repositorio-monedas)) ;; lista con valores de las monedas
                                                                                     (map second repositorio-
monedas)) ;; lista con valores de las monedas
                                                                                repositorio-monedas) ;; base de
datos con el repositorio de monedas
                                                                         nil)
```

```
                                                                    (INDICE ; En esta función se checa que
coincida con la misma posición del valor de la moneda cuya cantidad se está actualizando.
                                                                     (some #{(first transaccion)} (map second
repositorio-monedas))
                                                                     (map second repositorio-monedas))
                                                                    repositorio-monedas ;; base de datos con el
repositorio de monedas
                                                                    nil)
                                                            (rest transaccion)
                                                            precio ;; El precio es igual.
                                                            valor-pagado ;; El valor pagado es igual.
                                                            nuevo-archBD))) ;; Se selecciona el archivo del que se
está leeyendo.

;; Función que regresa el cambio. En esta función se utiliza mucho el comando de reverse para que primero tome en
cuenta las monedas de más valor primero.
(defn cambio [repositorio-monedas precio valor-pagado r]
  (if (and (number? (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))) ;; Verifica
cuales son las monedas que sean igual o mayor a la diferencia del valor pagado por el usuario en comparación con el
precio del producto.
          (> (ACCESA-N (INDICE (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))
                               (reverse (map second repositorio-monedas)))
                       (reverse (map caddr repositorio-monedas))) 0))
    (if (= (- valor-pagado (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))) precio)
;; Si con esta acción ya se acompletó el cambio...
      (concat r (list (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas)))))
      (cambio (reverse (reemplaza ;; Llama recursivamente a la función de cambio tomando en cuenta que se debe
actualizar el repositorio de monedas.
                       (reemplaza ;; Primero se reemplaza el valor de la cantidad de monedas disponibles de dicho
valor correspondiente, antes de reemplazar en el repositorio con dicha sublista generada.
                        (- (ACCESA-N (INDICE (miembro->= (- valor-pagado precio)
```

```
                                                    (reverse (map second repositorio-monedas))) ;; lista con
valores de monedas de manera descenedente
                                              (reverse (map second repositorio-monedas))) ; lista con valores de
monedas de manera descendente
                                    (reverse (map caddr repositorio-monedas))) ;; lista con valores de la capacidad
de las monedas en el mismo orden que la lista anterior
                           1) ;; Se resta 1
                         3 ;; Se reemplaza en la posición 3
                         (ACCESA-N (INDICE (miembro->= (- valor-pagado precio) ;; Todo esto indica que desea
sustituir en la sublista donde se encuentra el valor de la moneda.
                                                   (reverse (map second repositorio-monedas)))
                                          (reverse (map second repositorio-monedas)))
                                    (reverse repositorio-monedas))
                         nil)
                        (INDICE (miembro->= (- valor-pagado precio) ;; En esta función se checa que coincida con la
misma posición del valor de la moneda cuya cantidad se está actualizando.
                                         (reverse (map second repositorio-monedas)))
                                 (reverse (map second repositorio-monedas)))
                        (reverse repositorio-monedas)
                        nil))
              precio ;; El precio es igual.
              (- valor-pagado precio (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas))))
              (concat r (list (miembro->= (- valor-pagado precio) (reverse (map second repositorio-monedas))))))) ;;
Se actualiza la diferencia de cambio.
    (cambio (reverse (rest (reverse repositorio-monedas))) precio valor-pagado r))) ;; Si no hay coincidencia sigue
buscando recursivamente la moneda que esté disponible.

;; (apply + (cambio '((1 1 49 50) (2 2 13 45) (3 5 36 40) (4 10 10 25) (5 20 15 20) (6 50 1 5)) 17 20 '()))

;; (cambio '((1 1 49 50) (2 2 13 45) (3 5 36 40) (4 10 10 25) (5 20 15 20) (6 50 1 5)) 16 20 '())

;; Función de transción que va haciendo la suma de las monedas introducidas por el usuario.
```

```
(defn transicion [listaT precio repositorio-monedas origen codigo nuevo-archBD og-inventario-productos og-listaT id-maq]
  (if (empty? listaT)
    (println (str "Transaccion de la maquina " (pr-str id-maq) " con codigo de producto " (pr-str codigo) " --- Error: No se ingresó la cantidad suficiente de dinero.")) ;; Si ya recorrió toda la lista de transiciones se despliega este mensaje.
    (if (true? (acepta? repositorio-monedas listaT)) ;; Primero verifica si se acepta la secuencia de monedas.
      (cond (= (+ origen (first listaT)) precio) (when true
                                                   (actualiza-inventario-productos codigo og-inventario-productos nuevo-archBD) ;; En caso de que el pago sea exacto, solamente se actualiza el inventario.
                                                   (actualiza-repositorio-monedas-helper-toma-cuenta-recibe repositorio-monedas og-listaT precio (+ origen (first listaT)) nuevo-archBD) ;; Escribe el repositiorio de monedas.
                                                   (println (str "Transaccion de la maquina " (pr-str id-maq) " con codigo de producto " (pr-str codigo) " --- Transaccion realizada con exito. El pago fue exacto."))) ;; Despliega el mensaje de que la transacción fue exitosa.
            (> (+ origen (first listaT)) precio) (when true
                                                   (actualiza-inventario-productos codigo og-inventario-productos nuevo-archBD) ;; En caso de que se requiera dar cambio, primero se actualiza el inventario de productos.
                                                   (actualiza-repositorio-monedas-helper-toma-cuenta-recibe repositorio-monedas og-listaT precio (+ origen (first listaT)) nuevo-archBD) ;; Actualiza el repositorio de monedas.
                                                   (println (str "Transaccion de la maquina " (pr-str id-maq) " con codigo de producto " (pr-str codigo) " --- Transaccion realizada con exito. Su cambio fue de " (+ 1 2) " pesos. Las monedas que se dieron de cambio fueron: " (pr-str '(2 1))))) ;;Despliega el mensaje de que la transacción fue exitosa junto con su cambio.
            :else (transicion (rest listaT) precio repositorio-monedas (+ origen (first listaT)) codigo nuevo-archBD og-inventario-productos og-listaT id-maq)) ;; Llama recursivamente a la función para seguir haciendo las transiciones.
      (println (str "Transaccion de la maquina " (pr-str id-maq) " con codigo de producto " (pr-str codigo) " --- " (acepta? repositorio-monedas listaT))))));; Hace la llamada para desplegar el mensaje correspondiente.

;; Función que verifica si existe un producto basado en el codigo que fue insertado.
```

```
(defn verif-exis-prod [inventario-productos codigo-transaccion repositorio-monedas nuevo-archBD og-inventario-productos]
  (cond (empty? codigo-transaccion) (println (str "Ya no hay más transacciones de la maquina " (pr-str (subs nuevo-archBD (+ (count nombre-carpeta-Transacciones-Maquinas) 9) (- (count nuevo-archBD) 4)))))
        (= (second (first codigo-transaccion)) "Nulo") (println (str "Transaccion de la maquina " (pr-str (ffirst codigo-transaccion)) " con codigo de producto " (pr-str (second (first codigo-transaccion))) " --- Error: Transaccion Vacia."))
        (empty? inventario-productos) (println (str "Transaccion de la maquina " (pr-str (ffirst codigo-transaccion)) " con codigo de producto " (pr-str (second (first codigo-transaccion))) " --- Error: Usted ha ingresado un codigo que no existe."))
        (= (second (first codigo-transaccion)) (ffirst inventario-productos)) (transicion ;; Si hay coincidencia llama a la función de transición.
                                                                                          (second (nfirst codigo-transaccion))
                                                                                          (second (nfirst inventario-productos))
                                                                                          repositorio-monedas
                                                                                          0 ;; Origen de la transición.
                                                                                          (second (first codigo-transaccion))
                                                                                          nuevo-archBD
                                                                                          og-inventario-productos
                                                                                          (second (nfirst codigo-transaccion))
                                                                                          (ffirst codigo-transaccion))
        :else (verif-exis-prod (rest inventario-productos) codigo-transaccion repositorio-monedas nuevo-archBD og-inventario-productos))) ;; Sigue buscando recursivamente

;; Generar máquinas
(defn genera-maquinas [n id og-n lst]
  (let [lista-tipo-productos '(Frituras Bebidas Ropa Electronica) ;; Lista de los tipos de producto
        lista-ubicaciones '(Cumbres Zona-Tec Alta-Vista Valle-Oriente Contry Satelite Centro-MTY Obrera Centro-GDLP Linda-Vista)] ;; Lista de ubicaciones
    (if (zero? n) ;; Si ya se generaron todas las máquinas expendedoras...
      (when true
        '() ;; Lista vacía para que no se agreguen más datos.
```

```
        (spit nombre-archivo-Registro-Maquinas (pr-str lst)) ;; Escribe en el archivo la base de datos con todas las
máquinas generadas.
        (println (str "Se generaron " og-n " maquinas."))) ;; Despliega mensaje al usuario con la cantidad de
imágenes generadas.
      (genera-maquinas (dec n) ;; Desciende el conteo para seguir generando máquinas.
                       (inc id) ;; Conteo de las máquinas generadas
                       og-n ;; Cantidad de maquinas
                       (concat lst (list (list
                                          (str "M" id) ;; Id de la máquina
                                          (first (shuffle lista-tipo-productos)) ;; Valor al azar de la lista de tipo
de productos.
                                          (first (shuffle lista-ubicaciones))))))))) ;; Valor al azar de la lista de
ubicaciones.

;; Función que corrige la generación del repositorio de monedas.
(defn corregir-repositorio-monedas [lista-repositorio-monedas arch-BD]
  (if (empty? (filter (fn [x] (> (caddr x) (cadddr x))) lista-repositorio-monedas)) ;; Si no hay registros donde la
cantidad de monedas disponible es mayor que la cantidad máxima...
    (spit arch-BD (pr-str lista-repositorio-monedas) :append true) ;; Escribe en el archivo de la base de datos donde
se encuentra también el inventario.
    (corregir-repositorio-monedas (reemplaza (reemplaza ;; Reemplaza en el repositorio los registros con datos
erróneos.
                                             (cadddr (first (filter (fn [x] (> (caddr x) (cadddr x))) lista-
repositorio-monedas))) ;; Agarra el valor de la cantidad máxima de los registros donde la cantidad de monedas
disponbile es mayor que la cantidad máxima.
                                             3 ;; Lo reemplaza en la posición de la cantidad de monedas disponible.
                                             (first (filter (fn [x] (> (caddr x) (cadddr x))) lista-repositorio-
monedas)) ;; Lo reemplaza en el registro (sublista) donde está el error de todos los registros que son erróneos.
                                             nil)
                                            (INDICE (some #{(first (filter (fn [x] (> (caddr x) (cadddr x))) lista-
repositorio-monedas))};; La sublista se reemplaza en la posición donde se encuentra el registro con el error junto
con todos los demás registros que se habían generado..
```

```
                                                    lista-repositorio-monedas)
                                            lista-repositorio-monedas)
                                    lista-repositorio-monedas ;; Se reemplaza en el repsoitorio de monedas.
                                    nil)
                            arch-BD))) ;; archivo de la Basse de datos.

;; Función que genera el repositorio de monedas.
(defn genera-repositorio-monedas [n id lst arch-BD lista-monedas]
  (if (zero? n) ;; Si ya se generó la cantidad deseafa...
    (corregir-repositorio-monedas lst arch-BD) ;; Llama a la función de corregir si es que hubo errores durante la
generación de datos.
    (genera-repositorio-monedas (dec n) ;; Decrece para tener en cuenta cuanats llamadas recrusivas faltan.
                                (inc id) ;; Incrementa el número del id.
                                (concat lst (list ;; Concatena en un registro los siguientes parámetros
                                            (list
                                             id ;; ID
                                             (first lista-monedas) ;; El primer valor de la lista de monedas.
                                             (rand-int 51) ;; Número aleatorio de cantidad disponible de monedas.
                                             (rand-int 51)))) ;; Número aleatorio de cantidad máxima de monedas.
                                arch-BD ;; Archivo de la base de datos de la máquina.
                                (rest lista-monedas)))) ;; Avanza en la lista de monedas.

;; Función que corrige la generación del inventario de productos.
(defn corregir-inventario-productos [lista-inventario-productos num-maq]
  (if (empty? (filter (fn [x] (> (second x) 1)) (frequencies lista-inventario-productos))) ;; Si hay códigos de
productos que se repiten...
    (let [arch-BD (str nombre-carpeta-BD-Maquinas "/Maquina" num-maq ".txt")]
      (spit arch-BD (prn-str lista-inventario-productos)) ;; Escribe el inventario en el archivo.
      (genera-repositorio-monedas (count '(1 2 5 10 20 50)) 1 '() arch-BD '(1 2 5 10 20 50))) ;; Llama a la función
para generar el repositorio de monedas.
    (let [lista-letras '(A B C D E)]
```

```
      (corregir-inventario-productos (reemplaza (reemplaza ;; Reemplaza en el inventario los registros con códigos
repetidos.
                                                 (str (first (shuffle lista-letras)) (rand-int 11)) ;; Genera
nuevamente un código de producto aleatorio.
                                                 1 ;; Lo reemplaza en la primera posición, es decir, la posición del
código de producto.
                                                 (ACCESA-N (INDICE (some #{(ffirst (filter (fn [x] (> (second x) 1))
(frequencies (map first lista-inventario-productos))))} ;; Lo reemplaza en el registro (sublista) donde se encuentra
el código repetido.
                                                                                         (map first lista-inventario-productos))
                                                                                 (map first lista-inventario-productos))
                                                                         lista-inventario-productos)
                                                 nil)
                                                (INDICE (some #{(ffirst (filter (fn [x] (> (second x) 1))
(frequencies (map first lista-inventario-productos))))} ;; Lo reemplaza en la posición donde se encuentra el registro
junto con todos los demás registros que se habían generado en la función anterior..
                                                                                   (map first lista-inventario-productos))
                                                                           (map first lista-inventario-productos))
                                                 lista-inventario-productos
                                                 nil)
                                       num-maq)))) ;; Número de la máquina con la que se está tratando.

;; Función que genera el inventario de productos de una máquina expendedora.
(defn genera-inventario-productos [n maqs lst num-maq]
  (if (zero? n) ;; Si ya se generaron todas las máquinas...
    (corregir-inventario-productos lst num-maq) ;; Llama a la función de corregir por si hay errores.
    (let [lista-letras '(A B C D E) ;; Lista de las letras que puede haber en una máquina expendedora.
          lista-productos-frituras '(Sabritas Ruffles-Originales Ruffles-Queso Cheetos Cheetos-Torciditos Cheetos-
Poffs) ;; Lista de frituras
          lista-productos-bebidas '(Lata-Coca-Cola Botella-Coca-Cola-500ml Botella-Ciel-600ml Botella-JUMEX-Manzana
Botella-JUMEX-Mango) ;; Lista de bebidas
```

```
          lista-productos-ropa '(Camisa-Polo Jeans-Completos Jeans-Hoyos Shorts-Mezclilla Bermudas Blusa Sombrero
Gorra) ;; Lista de ropa
          lista-productos-electronica '(Procesador Cables-Varios Placa-Base Tarjeta-Madre Tornillos iPhone-13 iPhone-
SE Samsung-Galaxy-S22 Realme-GT-5G)] ;; Lista de electrónica
      ;; Para esta sección se generan el inventario de productos en base a su tipo de producto, para cada inventario
se genera de manera al azar el código del producto, el producto de manera al azar dentro de la categoría, un precio
al azar, y una cantidad disponible al azar.
      (cond (= (second (first maqs)) (quote Frituras)) (genera-inventario-productos (dec n) maqs (concat lst (list
(list (str (first (shuffle lista-letras)) (rand-int 6)) (first (shuffle lista-productos-frituras)) (rand-int 52)
(rand-int 101)))) num-maq)
            (= (second (first maqs)) (quote Bebidas)) (genera-inventario-productos (dec n) maqs (concat lst (list
(list (str (first (shuffle lista-letras)) (rand-int 6)) (first (shuffle lista-productos-bebidas)) (rand-int 52)
(rand-int 101)))) num-maq)
            (= (second (first maqs)) (quote Ropa)) (genera-inventario-productos (dec n) maqs (concat lst (list (list
(str (first (shuffle lista-letras)) (rand-int 6)) (first (shuffle lista-productos-ropa)) (rand-int 52) (rand-int
101)))) num-maq)
            (= (second (first maqs)) (quote Electronica)) (genera-inventario-productos (dec n) maqs (concat lst (list
(list (str (first (shuffle lista-letras)) (rand-int 6)) (first (shuffle lista-productos-electronica)) (rand-int 52)
(rand-int 101)))) num-maq)))))

;; Función que elimina todos los archivos que se encuentran en la carpeta donde están las bases de datos (inventario
y repositorio) de cada máquina expendedora.
(defn eliminar-archivos-BD [carpeta dir?]
  (if (empty? (map #(.getPath %) ;; Si ya está vacía la carpeta...
                   (filter (comp not dir?)
                           (tree-seq dir? #(.listFiles %) carpeta))))
    (println "Se han eliminado las bases de datos existentes para iniciar con la generacion de bases de datos.") ;;
Despliega este mensaje.
    (when true ;; Borra recursivamente los archivos que se encuentran en la carpeta.
      (io/delete-file (str (first (map #(.getPath %)
                                       (filter (comp not dir?)
                                               (tree-seq dir? #(.listFiles %) carpeta))))))
```

```
      (eliminar-archivos-BD carpeta dir?))))) ;; Llama recursivamente a la función.

;; Función que inicia con los procesos para generar el inventario de prodcutos y repositorio de monedas de cada
máquina.
(defn inicia-generacion-BD [n-BDs n-Inv arch-Maq num-maq]
  (if (zero? n-BDs) ;; Si ya se crearon todas las bases de datos...
    (println "Se generado todas las bases de datos.") ;; Se despliega este mensaje.
    (when true
      (cond (= num-maq 1) (eliminar-archivos-BD (io/file nombre-carpeta-BD-Maquinas) #(.isDirectory %))) ;; La
primera vez que se ejecuta está función procede a borrar todas las bases de datos existentes que había anteriormente.
      (genera-inventario-productos n-Inv arch-Maq '() num-maq) ;; Genera inventario de la máquina actual.
      (inicia-generacion-BD (dec n-BDs) n-Inv (rest arch-Maq) (inc num-maq))))) ;; Llama recursivamente a la
siguiente función para generar la base de datos para la siguiente máquina.

;; Función que elimina todos los archivos que contienen las transacciones.
(defn eliminar-archivos-transacciones [carpeta dir?]
  (if (empty? (map #(.getPath %) ;; Si ya está vacía la carpeta...
                   (filter (comp not dir?)
                           (tree-seq dir? #(.listFiles %) carpeta))))
    (println "Se han eliminado las transacciones existentes para iniciar con la generacion de transacciones.") ;;
Despliega este mensaje.
    (when true ;; Borra recursivamente los archivos que se encuentran en la carpeta.
      (io/delete-file (str (first (map #(.getPath %)
                                       (filter (comp not dir?)
                                               (tree-seq dir? #(.listFiles %) carpeta))))))
      (eliminar-archivos-BD carpeta dir?)))) ;; Llama recursivamente a la función.

;; Función que ordena las transacciones por su id de máquina.
(defn ordena-transacciones [arch-TTMaq]
  (let [transacciones-ordenadas (sort (map first arch-TTMaq))]
    (if (empty? transacciones-ordenadas) ;; Si ya no hay más transacciones por ordenar.
      (when true
```

```
        (io/delete-file nombre-archivo-TTMaquinas) ;; Borra el archivo en donde se encontraban todas las
transacciones en el mismo archivo.
        (println "Se han generado y ordenado todas las transacciones de cada maquina en su perspectivo archivo.")) ;;
Se despliega este mensaje.
      (when true ;; Crea y escribe un archivo de transacciones dedicado a la máquina con la que está tratando en ese
momento, tomando en cuenta su id en el archivo con todas las transacciones de todas las máquinas.
        (spit (str nombre-carpeta-Transacciones-Maquinas "/TMaquina" (first transacciones-ordenadas) ".txt") (pr-str
(filter (fn [x] (= (first transacciones-ordenadas) (first x))) arch-TTMaq)))
        (ordena-transacciones (remove (fn [x] (= (first transacciones-ordenadas) (first x))) arch-TTMaq)))))) ;;
Llama recursivamente a la función removiendo las trnasacciones de la máquina con la que ya trató.

;; Función que agrega transacciones nulas a máquinas que no cuentan con ninguna transacción después de la generación
de datos, esto es para que cada máquina tenga su propio archivo de transacciones.
(defn agrega-transacciones-faltantes [arch-Maq arch-TTMaq]
  (cond (empty? arch-Maq) (when true ;; Si ya está vació la lista que geneeraba la función que creaba transacciones.
                            (spit nombre-archivo-TTMaquinas (pr-str arch-TTMaq)) ;; Sobreescribe la base de datos con
todas las transacciones en el archivo que se había generado anteriormente.
                            (ordena-transacciones (read-string (slurp nombre-archivo-TTMaquinas)))) ;; Llama a la
función que ordena las transacciones.
        (.contains (map first arch-TTMaq) (first (map first arch-Maq))) (agrega-transacciones-faltantes (rest arch-
Maq) arch-TTMaq) ;; Si la máquina cuenta con transacciones, se llama recursivamente a la función haciendo rest a la
base de datos contiene a todas las transacciones.
        :else (agrega-transacciones-faltantes arch-Maq (concat (list (list (str (first (map first arch-Maq))) "Nulo"
'())) arch-TTMaq)))) ;; Si la máquina no cuenta con alguna transacción, se le agrega una con la notación de nulo en
la posición donde debe ir el código del producto.

;; Función que genera transacciones.
(defn genera-transacciones [n lst arch-Maq vez]
  (cond (= vez 1) (eliminar-archivos-transacciones (io/file nombre-carpeta-Transacciones-Maquinas) #(.isDirectory
%))) ;; La primera vez que se ejecuta esta función borra todas las transacciones que ya se habían creado
anteriormente.
  (if (zero? n) ;; Si ya se crearon todas las transacciones.
```

```
    (when true
      (spit nombre-archivo-TTMaquinas (pr-str lst)) ;; Escribe todas las transacciones en un mismo archivo.
      (agrega-transacciones-faltantes arch-Maq (read-string (slurp nombre-archivo-TTMaquinas)))) ;; Llama a la
función que agrega transacciones a máquinas que no tengan.
    (let [lista-letras '(A B C D E) ;; Lista de posibles letras que pueden tener un código de producto.
          lista-monedas '(1 2 5 10 20 50)] ;; Lista de monedas estipuladas por las máquinas.
      (genera-transacciones (dec n) (concat lst (list (list (str "M" (first (shuffle (range 1 (inc (count arch-
Maq)))))) ;; Asigna aleatoriamente a que máquina pertenece la transacción.
                                                      (str (first (shuffle lista-letras)) (rand-int 6)) ;;
Genera aleatoriamente el código de producto de la máquina.
                                                      (repeatedly (first (shuffle (range 1 11))) #(first
(shuffle lista-monedas)))))) ;; Asinga aleatoriamente las monedas que se usarán y cuántas monedas en total serán,
                            arch-Maq (dec vez)))))



;; Función que obtiene el top10% de las máquinas con más ganancias obtenidas
(defn top-10%-ganancias [lst-og lst r]
  (if (= (inc (quot (count lst) 10)) (count r))
    r ;; Despliega el resultado.
    (top-10%-ganancias lst-og (remove #{(ACCESA-N (INDICE (some ;; Remueve las máquinas que ya fueron consideradas
para seguir buscando a las demás.
                                                          #{(apply max (map second lst))}
                                                          (map second lst))
                                                         (map second lst))
                                                  lst)} lst) (concat r (list (ACCESA-N (INDICE (some ;; Adjunta a la
lista resultante la máquina que es considerada como la de mayor valor
                                                                                #{(apply max (map
second lst))}
                                                                                (map second lst))
                                                                               (map second lst))
                                                                        lst))))))
```

```clojure
;; Función que obtiene las ganancias totales después de ejecutar todas las transacciones.
(defn ganancia-obtenida [original-inventario-productos final-inventario-productos r]
  (if (and (empty? original-inventario-productos) (empty? final-inventario-productos)) ;; Si ya se completó el
recorrido de ambas listas.
    (apply + r) ;; Regresa sumatoria
    (if (< (second (rest (nfirst final-inventario-productos))) (second (rest (nfirst original-inventario-
productos)))) ;; Compara si es menor la cantidad de productos depsués de finalizar las transacciones a cómo eran
antes de ello.
      (ganancia-obtenida (rest original-inventario-productos) (rest final-inventario-productos) (concat r (list (*
(second (nfirst original-inventario-productos)) ;; Si es menor la cantidad se suma la multiplicación de la cantidad
de unidades de determinado producto por la diferencia entre la cantidad al inicio y la actual.
                                                                                                    (-
(second (rest (nfirst original-inventario-productos))) (second (rest (nfirst final-inventario-productos)))))))))
      (ganancia-obtenida (rest original-inventario-productos) (rest final-inventario-productos) r)))) ;; Ejecuta la
función recursivamente si no se cumple con la condición.

;; Función que obtiene las productos que tienen pocas unidades restantes en el inventario.
(defn productos-poco-inventario [inventario-productos]
  (if (empty? inventario-productos) ;; Si ya se completó el recorrido de la lista...
    '() ;; Regesa nulo
    (if (<= (second (rest (nfirst inventario-productos))) 5) ;; Hace la comparación si hay igual o menos de 5
unidades.
      (cons (concat (list (ffirst inventario-productos)) (list (first (nfirst inventario-productos)))) (productos-
poco-inventario (rest inventario-productos))) ;; Agrupa en una lista los coincidentes.
      (productos-poco-inventario (rest inventario-productos))))) ;; Sigue buscando recursivamente.

;; Función que obtiene cuales son las monedas que ya están llenos o casi llenos su repositorio.
(defn monedas-casi-llenas-repositorio [repositorio-monedas]
  (if (empty? repositorio-monedas) ;; Si ya se completó el recorrido de la lista...
    '() ;; Regesa nulo
```

```
    (if (>= (second (nfirst repositorio-monedas)) (* 0.8 (second (rest (nfirst repositorio-monedas))))) ;; Hace la
comparación si hay igual o más del 80% de la capacidad maxima.
      (cons (first (nfirst repositorio-monedas)) (monedas-casi-llenas-repositorio (rest repositorio-monedas))) ;;
Agrupa en una lista los coincidentes.
      (monedas-casi-llenas-repositorio (rest repositorio-monedas)))))) ;; Sigue buscando recursivamente.

;; Función que obtiene cuales son las monedas que ya están vacíos o casi vacíos su repositorio.
(defn pocas-monedas-repositorio [repositorio-monedas]
  (if (empty? repositorio-monedas) ;; Si ya se completó el recorrido de la lista...
    '() ;; Regesa nulo
    (if (<= (second (nfirst repositorio-monedas)) (* 0.2 (second (rest (nfirst repositorio-monedas))))) ;; Hace la
comparación si hay igual o menos del 20% de la capacidad maxima.
      (cons (first (nfirst repositorio-monedas)) (pocas-monedas-repositorio (rest repositorio-monedas))) ;; Agrupa en
una lista los coincidentes.
      (pocas-monedas-repositorio (rest repositorio-monedas))))) ;; Sigue buscando recursivamente.



;; Función que etiqueta con el id de cada máquina el resultado de sus ganancias obtenidas.
(defn etiqueta-ganancias [id lst r]
  (if (empty? lst) ;; Si ya terminó de recorrer la lista...
    r ;; Entrega el resultado.
    (etiqueta-ganancias (inc id) ;; Incrementa el número del id para etiquetarlo en la siguiente pasada.
                        (rest lst) ;; Avanza en la lista.
                        (concat r ;; Concatena la lista etiquetada con los siguientes resultados.
                                (list (concat
                                       (list (str "La ganancia de la maquina " (str "M" id) " fue de:")) ;; Mensaje
con el id de la máquina.
                                       (list (first lst)) ;; Valor de la ganancia obtenida.
                                       (list "pesos."))))))) ;; Mensaje que indica el tipo de moneda de la ganancia.

;; Función que etiqueta con el id de cada máquina el resultado de los productos con poco inventario.
```

```
(defn etiqueta-lista-productos [id lst r]
  (if (empty? lst) ;; Si ya terminó de recorrer la lista...
    r ;; Entrega el resultado.
    (etiqueta-lista-productos (inc id) ;; Incrementa el número del id para etiquetarlo en la siguiente pasada.
                              (rest lst) ;; Avanza en la lista.
                              (concat r ;; Concatena la lista etiquetada con los siguientes resultados.
                                      (list (concat
                                             (list (str "M" id)) ;; Id de la máquina.
                                             (list "Productos:") ;; Mensaje de Productos.
                                             (list (first lst))))))))) ;; Lista de productos con inventario bajo.

;; Función que etiqueta con el id de cada máquina el resultado de las monedas con mucha o poca cantidad en el
repositorio.
(defn etiqueta-lista-monedas [id lst r]
  (if (empty? lst) ;; Si ya terminó de recorrer la lista...
    r ;; Entrega el resultado.
    (etiqueta-lista-monedas (inc id) ;; Incrementa el número del id para etiquetarlo en la siguiente pasada.
                            (rest lst) ;; Avanza en la lista.
                            (concat r
                                    (list (concat
                                           (list (str "M" id)) ;; Id de la máquina.
                                           (list "Monedas de:") ;; Mensaje que señala las moendas.
                                           (list (first lst)) ;; Lista de monedas.
                                           (list "pesos."))))))) ;; Mensaje que indica el tipo de moneda.


;; Función que ejecuta las funciones necesarias para hacer el reporte final después de las transacciones.
(defn Reporte-final [tod-inv-og tod-inv-final tod-rep-final]
  (println "REPORTE FINAL")
  (println (str "Lista del top 10% de maquinas con más ganancia despues de las transacciones: "
                (pr-str (top-10%-ganancias (count tod-inv-og) (etiqueta-ganancias 1 ;; ID inicial.
```

```
                                                                    (map (fn [x] (ganancia-obtenida
(first x) (second x) '())) ;; Aplica la función de ganancia-obtenida recibiendo dos parámetros de la misma lista
combinada,
                                                                        (map vector tod-inv-og tod-
inv-final)) '()) '()))))) ;; Se hace merge de ambas listas para que el map pueda recorrer ambas listas de manera
simultánea.
  (println (str "Lista de maquinas cuyo(s) producto(s) su inventario es poco o nulo (menor que 5 unidades): " (pr-str
(remove (fn [x] (empty? (caddr x))) ;; Remueve de la lista las máquinas que no cumplieron con la condición de los
inventarios de bajo nivel.

(etiqueta-lista-productos 1 ;; ID inicial de la máquina.

(map productos-poco-inventario tod-inv-final) '()))))) ;; Mapea la función con la lista que contiene a todos los
inventarios.
  (println (str "Lista de maquinas cuya(s) moneda(s) su respositorio esta lleno o casi lleno (80% de capacidad o
mas): " (pr-str (remove (fn [x] (empty? (caddr x))) ;; Remueve de la lista las máquinas que no cumplieron con la
condición de los repositorios de (casi) llenos.

(etiqueta-lista-monedas 1 ;; ID inicial de la máquina.

(map monedas-casi-llenas-repositorio tod-rep-final) '()))))) ;; Mapea la función con la lista que contiene a todos
los repositorios.
  (println (str "Lista de maquinas cuya(s) moneda(s) su respositorio esta vacio o casi vacio (20% de capacidad o
menos): " (pr-str (remove (fn [x] (empty? (caddr x))) ;; Remueve de la lista las máquinas que no cumplieron con la
condición de los repositorios de (casi) vacíos.

(etiqueta-lista-monedas 1 ;; ID inicial de la máquina.

(map pocas-monedas-repositorio tod-rep-final) '()))))) ;; Mapea la función con la lista que contiene a todos los
repositorios.
;; Función que automatiza la generación de máquinas, inventarios, repositorios, y transacciones.
(when true
```

```
  (println "Introduzca la cantidad de maquinas que desea generar.")
  (def numMaquinas (parse-int (read-line))) ;; Se lee la entrada del usuario para saber cuantas máquinas desea
generar.
  (def arch-Maquinas (read-string (slurp nombre-archivo-Registro-Maquinas))) ;; Lee el archivo de las máquinas
  (println "Introduzca la cantidad de productos que habra por maquina.")
  (def cant-productos-maquina (parse-int (read-line)))
  (println "Introduzca la cantidad de transacciones que desea generar en total por todas las maquinas.")
  (def cant-transacciones (parse-int (read-line)))
  (genera-maquinas numMaquinas 1 numMaquinas '()) ;; Llamado de la función que genera las máquinas.
  (inicia-generacion-BD (count arch-Maquinas) cant-productos-maquina arch-Maquinas 1) ;; LLama a la función que
iniciaiza la generación de inventarios y repositiorios.
  (genera-transacciones cant-transacciones '() arch-Maquinas 1)) ;; Llama a la función que genera las transacciones
de todas las máquinas.

;; Aquí se leen las transacciones de cada máquina individualmente mediante recursividad tomando en cuenta la carpeta
en donde se encuentran.
(def lectura-todas-transacciones
  (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre de los
archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
                                                          (filter (comp not #(.isDirectory %))
                                                                  (tree-seq #(.isDirectory %) #(.listFiles %)
(io/file nombre-carpeta-Transacciones-Maquinas)))))))))

; Aquí se leen los inventarios de cada máquina individualmente mediante recursividad tomando en cuenta la carpeta en
donde se encuentran.
(def todos-original-inventario-productos (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %)
;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la
cantidad de carácteres.
                                                                                          (filter (comp
not #(.isDirectory %))

(tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))
```

```
;; Función que ejecuta todas las funciones del codigo para que el proceso de las transacciones sea automático y
secuencial.
(defn inicia-operaciones-secuencial [tod-transaccs]
  (let [todos-nuevo-inventario-productos (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %)
;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la
cantidad de carácteres.
                                                                                                    (filter (comp
not #(.isDirectory %))

(tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))
        todos-nuevo-repositorio-monedas (map (fn [x] (with-open [rdr (io/reader x)]
                                                       (read-string (second (doall (line-seq rdr)))))) ;; Para que
lea solamente la segunda línea del archivo.
                                                (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre
de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
                                                                                  (filter (comp not #(.isDirectory %))
                                                                                          (tree-seq #(.isDirectory %)
#(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))]
    (if (not (every? empty? tod-transaccs)) ;; Si no se han procesado todas las transacciones correspondientes a cada
máquina...
      (when true
        (map (fn [x] (verif-exis-prod (first x) (second x) (caddr x) (cadddr x) (first x))) (map vector todos-nuevo-
inventario-productos
                                                                                              tod-transaccs
                                                                                              todos-nuevo-
repositorio-monedas

                                                                                              (sort-by count (sort
(map str (map #(.getPath %)

(filter (comp not #(.isDirectory %))
```

```
(tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-Transacciones-Maquinas)))))))))
        (inicia-operaciones-secuencial (map rest tod-transaccs)))
      (Reporte-final todos-original-inventario-productos todos-nuevo-inventario-productos todos-nuevo-repositorio-
monedas))))  ;; Llamada para ejecutar el reporte final.


;; Función que ejecuta todas las funciones del codigo para que el proceso de las transacciones sea automático y
paralela.
(defn inicia-operaciones-paralelo [tod-transaccs]
  (let [todos-nuevo-inventario-productos (map read-string (map slurp (sort-by count (sort (map str (map #(.getPath %)
;; Se ordenan el nombre de los archivos por el id de su máquina; primero por la longitud del string, y después por la
cantidad de carácteres.
                                                                                                    (filter (comp
not #(.isDirectory %))

(tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))
        todos-nuevo-repositorio-monedas (map (fn [x] (with-open [rdr (io/reader x)]
                                                       (read-string (second (doall (line-seq rdr)))))) ;; Para que
lea solamente la segunda línea del archivo.
                                             (sort-by count (sort (map str (map #(.getPath %) ;; Se ordenan el nombre
de los archivos por el id de su máquina; primero por la longitud del string, y después por la cantidad de carácteres.
                                                                                  (filter (comp not #(.isDirectory %))
                                                                                          (tree-seq #(.isDirectory %)
#(.listFiles %) (io/file nombre-carpeta-BD-Maquinas)))))))))]
    (if (not (every? empty? tod-transaccs)) ;; Si no se han procesado todas las transacciones correspondientes a cada
máquina...
      (when true (doall (map (fn [y] (doall
                                      (pmap (fn [x] (verif-exis-prod (first x) (second x) (caddr x) (cadddr x) (first
x))) y)))
                             (partition-all (quot (count todos-nuevo-inventario-productos) 10) (map vector todos-
nuevo-inventario-productos
```

```
                                                                                tod-transaccs
                                                                                todos-nuevo-
repositorio-monedas
                                                                                (sort-by count
(sort (map str (map #(.getPath %)

(filter (comp not #(.isDirectory %))

(tree-seq #(.isDirectory %) #(.listFiles %) (io/file nombre-carpeta-Transacciones-Maquinas)))))))))))
           (inicia-operaciones-paralelo (map rest tod-transaccs)))
     (Reporte-final todos-original-inventario-productos todos-nuevo-inventario-productos todos-nuevo-repositorio-
monedas))))  ;; Llamada para ejecutar el reporte final.

;; Ejecución de las transacciones.
(when true
  (time (inicia-operaciones-secuencial lectura-todas-transacciones))
  (println "Ejecutado de forma Secuencial"))
(when true
  (time (inicia-operaciones-paralelo lectura-todas-transacciones))
  (println "Ejecutado de forma Paralela"))
```

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

   **Ejemplo Prueba Individual**

   Este ejemplo en específico es para mostrar la estructura y el resultado de una prueba individual.

   **Bases de Datos**

   **Inventario de Productos:**

   *((A1 Lata-Coca-Cola 10 25) (A2 Botella-Coca-Cola-500ml 20 15) (A3 Botella-Ciel-600ml 22 22) (B1 Sabritas 13 41) (B2 Ruffles-Originales 11 27) (B3 Ruffles-Queso 13 16) (B4 Cheetos 17 30) (A4 JUMEX 15 3) (C1 Hamburguesa 50 2) (C2 Palomitas 5 31))*

   **Repositorio de Monedas:**

   ((1 1 49 50) (2 2 13 45) (3 5 36 40) (4 10 10 25) (5 20 15 20) (6 50 1 5))

   **Transiciones**

   **Pruebas**

   **Supera la cantidad de monedas de $1:** (M1 A1 (5 2 1 1))

   **Realizada con éxito. Pago exacto:** (M1 A2 (5 5 5 5))

   **Realizada con éxito. Cambio de $2:** (M1 B3 (10 5))

   **Realizada con éxito. Cambio de $3:** (M1 B4 (20))

   **Supera la cantidad de monedas de $1:** (M1 C2 (1 1 2 1))

   **Realizada con éxito. Cambio de $37:** (M1 B1 (50))

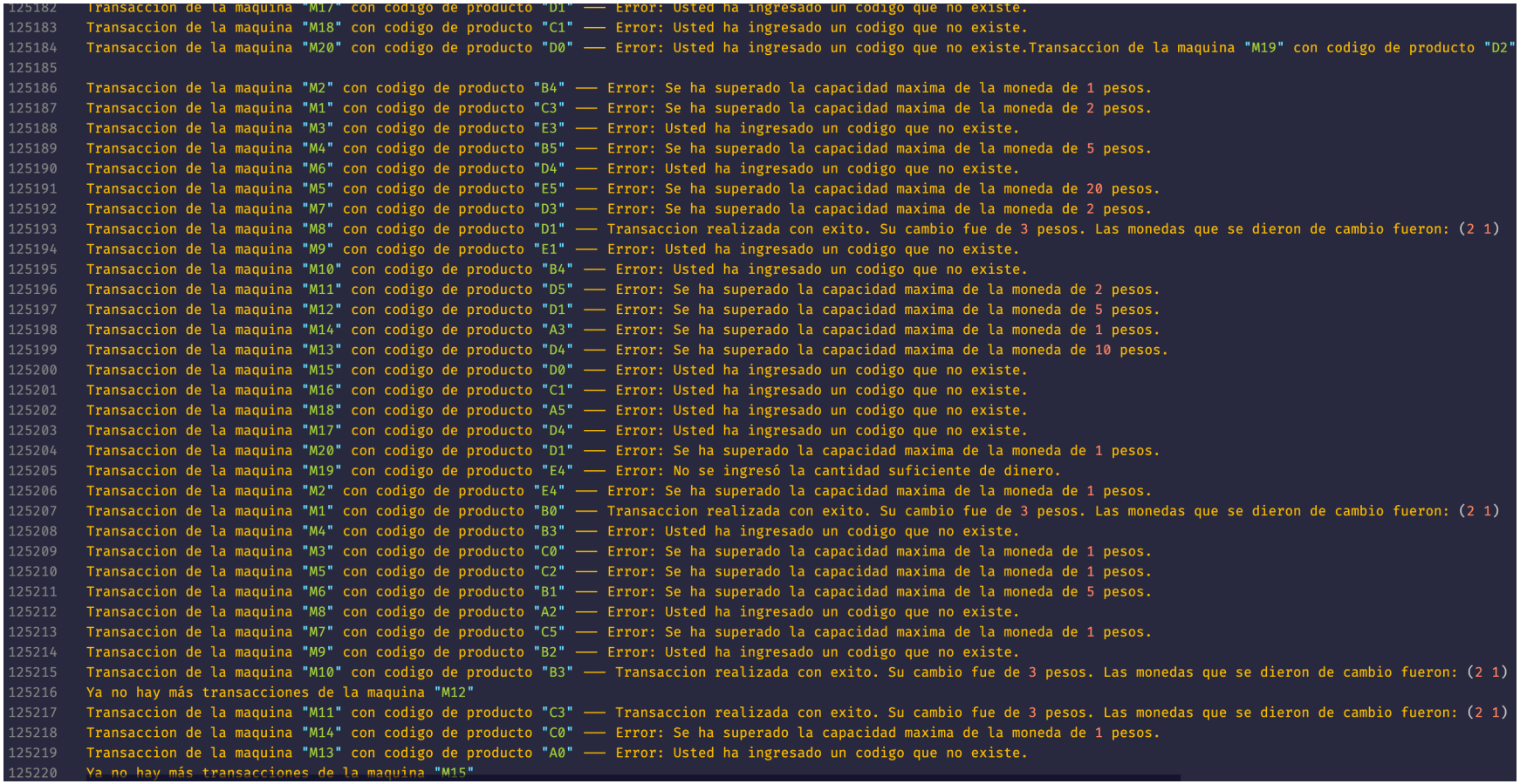
   **Código introducido es erróneo:** (M1 E5 (50))

   **Ejemplos con respecto a la paralelización**

   Los ejemplos con respecto a la comparación de forma secuencial y de forma paralela ya se abordaron en el punto #5. Aquí se concentrará en los resultados de la función de reporte final y otros aspectos de la forma paralela.

En esta ocasión voy a analizar de manera individual las pruebas paralelas que se ejecutaron en el punto #5.

**Impresión de los resultados de las transacciones que se van evaluando:**




Como se ve, al momento de imprimir los resultados podemos notar que no va en orden. Si hubiese sido de manera secuencial hubiera analizado la transacción de la máquina M1, M2 M3, …, y aquí de forma paralela esto no

siempre sucede porque está analizando más de una transacción a la vez y a veces puede terminar de procesar una transacción antes que otra y así lo despliega en el programa.

**Reporte Final – Prueba 40 Máquinas, 40 Productos en cada Inventario, 500 Transacciones.**

```
126798  REPORTE FINAL
126799  Lista del top 10% de maquinas con más ganancia despues de las transacciones: (("La ganancia de la maquina M1 fue de:" 0 "pesos.") ("La ganancia de la maquina M2 fue de:" 0 "pesos.")
126800  Lista de maquinas cuyo(s) producto(s) su inventario es poco o nulo (menor que 5 unidades): (("M1" "Productos:" (("E1" Botella-Ciel-600ml) ("D4" Lata-Coca-Cola) ("D1" Botella-JUMEX-Manzana)
126801  ("A2" Ruffles-Originales) ("E0" Sabritas) ("A5" Ruffles-Queso) ("E1" Ruffles-Queso))) ("M15" "Productos:" (("A4" Lata-Coca-Cola) ("E5" Botella-Coca-Cola-500ml) ("A4" Botella-Coca-Cola-500ml)
126802  oductos:" (("E4" Placa-Base) ("D4" Placa-Base))) ("M28" "Productos:" (("E2" Lata-Coca-Cola) ("D3" Botella-JUMEX-Manzana) ("A4" Lata-Coca-Cola) ("E3" Botella-JUMEX-Mango))) ("M29"
126803  X-Manzana) ("E1" Botella-JUMEX-Manzana) ("C3" Botella-Coca-Cola-500ml) ("B0" Botella-Coca-Cola-500ml) ("C2" Botella-JUMEX-Mango)))
126804  Lista de maquinas cuya(s) moneda(s) su respositorio esta lleno o casi lleno (80% de capacidad o mas): (("M1" "Monedas de:" (2 50) "pesos.") ("M2" "Monedas de:" (1 5 10) "pesos.")
126805  ("M23" "Monedas de:" (1 10 50) "pesos.") ("M24" "Monedas de:" (5 10 20 50) "pesos.") ("M25" "Monedas de:" (10 20) "pesos.") ("M26" "Monedas de:" (1 2 5 10 50) "pesos.") ("M27"
126806  Lista de maquinas cuya(s) moneda(s) su respositorio esta vacio o casi vacio (20% de capacidad o menos): (("M2" "Monedas de:" (2 20) "pesos.") ("M3" "Monedas de:" (5) "pesos.")
126807  "Elapsed time: 643.809084 msecs"
126808  Ejecutado de forma Paralela
```

```
126799  "La ganancia de la maquina M2 fue de:" 0 "pesos.") ("La ganancia de la maquina M3 fue de:" 0 "pesos.") ("La ganancia de la maquina M4 fue de:" 0 "pesos."))
126800  600ml) ("D4" Lata-Coca-Cola) ("D1" Botella-JUMEX-Manzana))) ("M2" "Productos:" (("A3" Sabritas) ("C5" Cheetos-Poffs) ("B3" Ruffles-Originales))) ("M3" "Productos:" (("A5" Samsung
126801  E5" Botella-Coca-Cola-500ml) ("A4" Botella-Coca-Cola-500ml) ("E3" Lata-Coca-Cola) ("E3" Botella-JUMEX-Manzana) ("B2" Botella-JUMEX-Manzana))) ("M16" "Productos:" (("D1" Shorts-Mezclilla)
126802  -Coca-Cola) ("E3" Botella-JUMEX-Mango))) ("M29" "Productos:" (("A3" Cheetos-Torciditos))) ("M30" "Productos:" (("E5" Shorts-Mezclilla) ("B1" Bermudas) ("B3" Bermudas))) ("M31"
126803
126804  pesos.") ("M2" "Monedas de:" (1 5 10) "pesos.") ("M3" "Monedas de:" (1 2 10 50) "pesos.") ("M4" "Monedas de:" (1 2 5 20 50) "pesos.") ("M5" "Monedas de:" (1 2 5 50) "pesos.")
126805  "Monedas de:" (1 2 5 10 50) "pesos.") ("M27" "Monedas de:" (1 5 20) "pesos.") ("M28" "Monedas de:" (1 2 5 10 20 50) "pesos.") ("M29" "Monedas de:" (20 50) "pesos.") ("M30" "Monedas
126806  "pesos.") ("M3" "Monedas de:" (5) "pesos.") ("M5" "Monedas de:" (20) "pesos.") ("M6" "Monedas de:" (2) "pesos.") ("M9" "Monedas de:" (2 50) "pesos.") ("M11" "Monedas de:" (1 2
126807
126808
```

Aquí se puede apreciar que el programa hace una lectura de todos los archivos de texto de Base de Datos (inventario y repositorio) para filtrar la información. En este caso, se puede observar que se generaron 40 máquinas, y solamente muestra 4 resultados en el top 10% de máquinas con más ganancias.

También se puede notar que hace una lectura correcta de los productos con poco inventario de cada máquina, así como los repositorios de moneda que tienen 80% capacidad o más, así como los que cuentan con 20% de capacidad o menos.

.

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

   Fue muy retadora esta segunda parte de la Situación Problema #2, primero que nada, el proceso de traducción de Scheme/Racket a Clojure fue más complicado de lo esperado, porque inicialmente no podía comprobar los resultados de los procesos debido a que no escribía la lista resultante en el archivo txt.

   Una vez solucionado eso, me aparecían otros errores de *Execution error* y de *StackOverFlow*, como no pude solucionarlos en su momento, decidí enfocarme en la generación de datos de manera automática, preparar la función reporte final, e incluso me dediqué a realizar la función que paraleliza las transacciones antes de que mi programa las realizara individualmente.

   También decidí convertir las funciones que había hecho en la Parte #1 a recursividad terminal para aumentar la eficiencia de mi programa.

   No fue hasta un día antes de la entrega de este mismo trabajo que por fin pude realizar transacciones en el programa. Fue una satisfacción enorme porque finalmente podía comparar resultados y tiempos de manera más adecuada.

   Durante la programación de este código aprendí muchísimas funciones de originales Clojure y otras que el lenguaje toma prestada de Java, las cuales me sirvieron para llevar a cabo la solución. Además, con esta evidencia hizo que le tomara mucha más importancia a lo que es la eficiencia, porque antes yo era de la idea que con que sólo funcionase el código era suficiente, pero ahora con esta experiencia con tantas veces que me aparecía el error de *StackOverFlow*, y además que el objetivo de esta Situación Problema es completar las transacciones en el mejor tiempo posible y tomando en cuenta que se ejecutarán hilos con la aplicación del *pmap*, mi modo de pensamiento cambió y ahora le tomo más importancia a la eficiencia.

**Link del video:**

https://drive.google.com/file/d/1q8TyqTGKOgcZOGLG-4HzSeVFPDkfED9G/view?usp=sharing