Campus Guadalajara

Evidencia #2: Implementación de un simulador de una máquina expendedora

Primera parte

Fernando López Gómez | A01639715

Implementación de métodos computacionales

TC2037.850

Profesor Román Martínez Martínez

Fecha de entrega: 23/10/2022

Sobre la Situación Problema:

El programa por realizar en Scheme es un simulador de una máquina expendedora que puede tener diversos productos de diferentes precios y que sólo acepta pagos con monedas, teniendo la capacidad de dar cambio si un pago lo requiere.

La simulación consistirá en procesar las transacciones de venta que estarán en otro archivo de texto con los datos de estas. Cada transacción de venta se identifica con la letra del producto a comprar y la secuencia de monedas que se depositan para la compra. El programa al procesar una transacción indicará con letreros en pantalla si la compra se realiza y las monedas sobrantes en caso de que las haya. Obviamente, cada transacción deberá actualizar los inventarios correspondientes. Cuando una transacción no sea posible, se deberá indicar con un mensaje la situación, por ejemplo, una venta no posible por falta de producto, o un sobrante de monedas no factible de entregar.

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

- Ganancia obtenida después de todas las transacciones de venta.
- Productos con poco inventario o inventario nulo.
- Denominaciones de monedas que tienen el repositorio lleno o casi lleno.
- Denominaciones de monedas que tienen poco inventario o nulo.

Por lo pronto, no habrá una interfase para modificar inventarios a través del programa. Cualquier ajuste de inventario para las pruebas de simulación se hará directamente en la lista de datos de la máquina.

Estructuras de datos implementadas

Para la solución de esta SP se implementaron listas imbricadas (sinónimo de listas anidadas). En racket existen 2 tipos de listas, propia e impropia; para el manejo del inventario de monedas se utilizaron listas impropias para almacenar la información de cada moneda con el propósito de reducir el espacio en memoria que ocupan almacenarlas gracias a que el segundo valor de la celda se convierte en un átomo en lugar de una nueva lista, además de que nos permite acceder al segundo valor como átomo y no como lista, lo cual puede llegar a hace el código un poco más fácil de leer. Sin embargo se utilizaron listas propias para el manejo del inventario de productos debido a que cada producto tiene más de 2 datos de información, por lo que ocupamos más casillas de memoria, además de que los datos de cada producto quedan en listas planas, lo que facilita su manipulación.

Para las transacciones se utilizó una lista propia para almacenar los datos de cada transacción, sin embargo el segundo dato es una lista propia con las monedas que el usuario ingresó para poder realizar la compra. Esto ayuda a la manipulación de los datos debido a que el segundo dato de la lista impropia, al ser una lista, se regresa una única vez como lista. Si se hubiera almacenado dentro de una lista propia se regresaría una lista doble y sería un tanto más complicado de acceder a los datos.

Diseño del autómata

Para evaluar que el ingreso de las monedas del usuario sea válido, se diseñó un autómata que estará evaluando la validez de las entradas

El formato matemático para la modelación del autómata es el siguiente:

A = ({p+1} , m, {δ(x, y, x+y) }, 0, {>=p})

donde :

- *p* es el precio del producto
- El autómata tiene *p+1* estados, el estado 0, que es el estado inicial, cuando no se tiene ninguna moneda, y cada estado está definido por el estado anterior +1 hasta llegar al valor de p
- *m* son las monedas disponibles mediante las cuales se pueden hacer transiciones
- *δ(x, y, x+y)* es un conjunto de transiciones donde "*x"* es el estado inicial, *"y"* es el valor de la moneda ingresada y el resultado de la suma de estos 2 números será el estado final de la transición.
- *0* es el estado inicial.
- *{>=p}* son los estados finales. Esto define que un estado terminal es aquel que termina en un estado mayor o igual al precio del producto

Algoritmo para la implementación del autómata

Primeramente empieza en el estado inicial, el cual recibe la lista de transacciones productos y monedas. Evalúa la lista de transacciones y si es que aún hay transacciones posibles entonces evalúa la primera transacción. Se pasa como parámetro la lista completa debido a que si es que el programa arroja un error, el poder continuar evaluando las demás transacciones a partir de ese punto.

Si no hay impedimentos para poder hacer la transacción, entonces se actualizan los datos del inventario de productos y monedas y se vuelve a ejecutar la lectura de las transacciones con el resto de la lista de transacciones. Aquí es donde cobra sentido el pasar una lista de transacciones, así no regresamos a los valores originales, sino que trabajamos con los valores actualizados; lo mismo ocurre para los datos de productos y monedas, se vuelven a pasar a la función de leer transacciones para poder trabajar con valores actualizados sin necesidad de cerrar y volver a abrir archivos para su actualización.

Primeramente se recibe en una función las monedas que ingresó el usuario y las monedas que acepta la máquina. Para comenzar el proceso evaluamos si es que la lista de monedas ingresadas está vacía, lo cual significa que ya hemos recorrido toda la lista de monedas sin error alguno y regresaría el valor booleano True.

Retomando el proceso de validación de monedas, mientras haya datos en la lista va a estar evaluando cada moneda mediante la comparación de esta moneda con la lista de monedas disponibles.

```
;Verificar que todas las monedas se encuentren en el inventario
(define (monedas-aceptadas? monedas-ingresadas monedas-disponibles)
  (if (not(null? monedas-ingresadas))
      (if (not (equal? (member (car monedas-ingresadas) monedas-disponibles) #f))
          (monedas-aceptadas? (cdr monedas-ingresadas) monedas-disponibles)
          #f); Si member retorna false la moneda no es válida
      #t) ;si llega al final sin salirse retorna true
  )
```

Una vez que todas las monedas pasen la prueba, quiere decir que todas las monedas fueron válidas.

Como sabemos que todas las monedas son válidas, en el autómata se podría interpretar como que todas las posibles transiciones son válidas.

Ahora pasamos a validar el estado final. Para esto, debemos de encontrar el precio del producto que el usuario quiere comprar y compararlo con la suma de las monedas que ingresó, lo cual se interpreta como evaluar la secuencia de monedas y comparar si el estado en el que terminaron las transiciones concuerda con un estado terminal, el cual sería el precio del producto o el estado siguiente en caso de que la cantidad sobrepase dicho precio. En caso de que el estado en el que terminó la secuencia no sea el estado terminal, entonces arroja el error que establece una falta de monedas por ingresar para poder comprar el producto.

```
;Si la suma de las monedas ingresadas no es suficiente
[(< (apply + (cdar transacciones)) (find-product-price (caar transacciones) productos))
    (error-handler 2 transacciones productos monedas)]
```

Seguido a esto, se corrobora de que exista producto disponible dentro de la máquina

```
; Si el stock está vacío
[(equal? (find-product-stock (caar transacciones) productos) 0)
    (error-handler 3 transacciones productos monedas)]
```

Y por último, se evalúa que la máquina tenga la capacidad de dar cambio al usuario. Si no le es posible, la transacción no se realiza.

```
;Si la máquina no cuenta con cambio
[(not (cambio? (-(apply + (cdar transacciones))
                 (find-product-price (caar transacciones) productos))
               monedas))
```

```
;Esta función se encarga únicamente de ssaber si es posible dar cambio.
(define (cambio? cambio monedas)
  (cond
    ;Si ya no hay monedas por evaluar. Si no hay cambio restante entonces
    ;si se puede dar cambio
    [(null? monedas)
     (if (equal? cambio 0)
         #t
         #f)]
    ; Si es que si puede haber cambio con esa moneda y si hay
    ; monedas disponibles
    ;(quotient cambio (caar monedas)) --> Número de monedas posibles
    [(and (not(equal? (quotient cambio (caar monedas)) 0))
          (>= (cdar monedas) (quotient cambio (caar monedas))))
        (cambio? (- cambio (* (quotient cambio (caar monedas)) (caar monedas))) (cdr monedas))]

    ;Si es que no hay monedas suficientes o la moneda excede al cambio
    ;continúa con la siguiente moneda
    [else (cambio? cambio (cdr monedas))]
    )
  )
```

En caso de que todas estas pruebas hayan pasado entonces se actualizan los datos de inventario de la máquina

```
[else (success (caar transacciones);Producto
               (find-product-price (caar transacciones) productos);Precio
               (cdar transacciones);Monedas ingresadas
               transacciones);lista de transacciones
      ])
```

```
(leerTransacciones (cdr transacciones-nueva)
                   (update-stock producto productos)
                   (update-coins (-(apply + monedas-ingresadas)  precio-producto);cambio
                     (add-coins monedas (sort monedas-ingresadas >))); monedas
                   )
)
```

Como podemos observar, tenemos a la función de add coins dentro de la de update coins. Esto es debido a que la función de add coins agrega al inventario las monedas ingresadas por el usuario antes de evaluar si existe cambio que dar.

La función que se encarga de dar el cambio es la de update coins, la cual va restando inventario de las monedas con las que puede dar cambio hasta no poder continuar dando cambio con dicha moneda.

```
;Actualizamos inventario de monedas para el cambio
(define (update-coins cambio monedas)
  (cond
    [(null? monedas) '()]
    ; Si es que si puede haber cambio con esa moneda y si hay
    ; monedas disponibles
    ;(quotient cambio (caar monedas)) --> Número de billetes posibles
    [(and (not(equal? (quotient cambio (caar monedas)) 0))
          (>= (cdar monedas) (quotient cambio (caar monedas))))
    (append (list(cons (caar monedas) (- (cdar monedas) (quotient cambio (caar monedas)))))
            (update-coins (- cambio (* (quotient cambio (caar monedas)) (caar monedas))) (cdr monedas)))]

    ;Si es que no hay monedas suficientes o la moneda excede al cambio
    ;continúa con la siguiente moneda
    [else (append (list(car monedas)) (update-coins cambio (cdr monedas)))]
    )

  )
```

Mutabilidad de los datos

El programa maneja una estructura de datos muy amigable, que son las listas imbricadas, esto le permite al usuario el ser capaz de agregar un producto al inventario simplemente agregando la información de dicho producto a la lista, lo cual hace el inventario de la máquina escalable. Asimismo, si se quiere modificar el precio o la información de un producto, se puede realizar modificando la lista del inventario en donde se encuentra dicho producto.

Lo mismo ocurre para el inventario de monedas, en este caso, la máquina utiliza 6 tipos de monedas y si se ingresa una que no reconoce entonces no se realiza la transacción. Como el inventario se comporta de la misma manera que el inventario de productos, eso permite agregar diferentes denominaciones de monedas a la máquina, por lo que el código puede permanecer intacto y únicamente habría que cambiar la lista del inventario de monedas para que la máquina ya pueda reconocer otras monedas.

Ventajas/desventajas al desarrollar el simulador en un lenguaje funcional

Ventajas:

- Una de las ventajas de desarrollar el simulador en un lenguaje funcional es el poder reducir líneas de código gracias a la implementación de métodos de programación como la recursividad y las funciones lambda.
- Las funciones son un poco más fáciles de entender, debido a que siempre se busca que la entrada sea el mismo tipo de dato que la salida.
- El uso de la lazy evaluation en racket me ha ayudado a definir condicionales que en en otro programa escrito en lenguaje imperativo hubiera arrojado errores.

Desventajas:

- Uso de la recursividad, pero en este caso viéndolo desde el enfoque del uso de memoria utilizada por el stack, ya que puede llegar a provocar problemas con la invasión de memoria.
- Los datos son inmutables, por lo que no se pueden usar variables que se están modificando a lo largo del programa.
- Al momento de trabajar con bases de datos, o programas que requieren de mucho análisis computacional, el tiempo de respuesta puede llegar a ser mucho mayor que en un programa de tipo imperativo.

Código en racket

```
#|

Evidencia #2: Implementación de un simulador de una máquina expendedora (primera parte)

Fernando López Gómez | A01639715

23/10/2022

|#


#lang racket


;-----------------------------------------------------------------------------------------------
;************** LEEMOS Y CERRAMOS LOS ARCHIVOS ****************
(define transacciones(read (open-input-file "transacciones.txt")))
(define monedas (read (open-input-file "monedas.txt")))
(define productos (read (open-input-file "productos.txt")))
(close-input-port (open-input-file "transacciones.txt"))
(close-input-port (open-input-file "monedas.txt"))
(close-input-port (open-input-file "productos.txt"))


;-----------------------------------------------------------------------------------------------
  ;******************** LIST RELATED FUNCTIONS *******************************
(define (find-product-price product list)
  ;Regresa el precio de un producto determinado
  (if (null? list)
      '()
      (if (equal? product (caar list))
          (cadar list)
          (find-product-price product (cdr list))
          )
      )
  )

(define (find-product-stock product list)
  ;Regresa el precio de un producto determinado
  (if (null? list)
      '()
      (if (equal? product (caar list))
          (caddar list)
          (find-product-stock product (cdr list))
          )
      )
  )

;Contar elementos dentro de una lista plana
(define (count list target)
  (if (null? list)
      0
      (if (equal?(car list) target)
          (+ 1 (count (cdr list) target))
          (count (cdr list) target)
          )
      )
  )

;Cortar n elementos de una lista
(define (cut-list list spaces)
  (if (equal? spaces 1)
      (cdr list)
      (cut-list (cdr list) (- spaces 1))
      )
  )


;-----------------------------------------------------------------------------------------------
;************** UPDATE FUNCTIONS ****************

(define (update-stock producto lista-productos)
  ;Encontramos el producto que se vendió
  ;y le restamos 1 al stock
```

```
  (cond
    [(null? lista-productos) '()]
    [(equal? producto (caar lista-productos))
    (append (list (list(caar lista-productos) (cadar lista-productos) (- (caddar lista-productos) 1)))
          (update-stock producto (cdr lista-productos)))]
    [else (append (list(car lista-productos)) (update-stock producto (cdr lista-productos)))]
  )
)
;---------------------------
;Las monedas ingresadas son una lista ordenada de las monedas que se utilizaron
(define (add-coins monedas monedas-ingresadas)
  (cond
    [(null? monedas) '()]
    ;Si no hay monedas ingresadas o no coinciden con el valor buscado, pasa a la siguiente moneda
    [(or (null? monedas-ingresadas) (not(equal? (car monedas-ingresadas) (caar monedas))))
     (append (list(car monedas)) (add-coins (cdr monedas) monedas-ingresadas))
        ]
    ;Agrega 1 a la cantidad de monedas actuales y corta la lista de monedas ingresadas
    ;hasta que dejen de repetirse los valores
    [else (append (list(cons (caar monedas)
                             (+(cdar monedas)(count monedas-ingresadas (car monedas-ingresadas)))))
                (add-coins (cdr monedas) (cut-list monedas-ingresadas
                                          (count monedas-ingresadas (car monedas-ingresadas)))))]
    )
  )
;---------------------------

;Actualizamos inventario de monedas para el cambio
(define (update-coins cambio monedas)
  (cond
    [(null? monedas) '()]
    ; Si es que si puede haber cambio con esa moneda y si hay
    ; monedas disponibles
    ;(quotient cambio (caar monedas)) --> Número de monedas posibles
    [(and (not(equal? (quotient cambio (caar monedas)) 0))
          (>= (cdar monedas) (quotient cambio (caar monedas))))
    (append (list(cons (caar monedas) (- (cdar monedas) (quotient cambio (caar monedas)))))
           (update-coins (- cambio (* (quotient cambio (caar monedas)) (caar monedas))) (cdr monedas)))]

    ;Si es que no hay monedas suficientes o la moneda excede al cambio
    ;continúa con la siguiente moneda
    [else (append (list(car monedas)) (update-coins cambio (cdr monedas)))]
    )

  )

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


;Esta función se encarga únicamente de ssaber si es posible dar cambio.
(define (cambio? cambio monedas)
  (cond
    ;Si ya no hay monedas por evaluar. Si no hay cambio restante entonces
    ;si se puede dar cambio
    [(null? monedas)
     (if (equal? cambio 0)
         #t
         #f)]
    ; Si es que si puede haber cambio con esa moneda y si hay
    ; monedas disponibles
    ;(quotient cambio (caar monedas)) --> Número de monedas posibles
    [(and (not(equal? (quotient cambio (caar monedas)) 0))
          (>= (cdar monedas) (quotient cambio (caar monedas))))
        (cambio? (- cambio (* (quotient cambio (caar monedas)) (caar monedas))) (cdr monedas))]

    ;Si es que no hay monedas suficientes o la moneda excede al cambio
    ;continúa con la siguiente moneda
    [else (cambio? cambio (cdr monedas))]
    )
  )


;-------------------------------------------------------------------------------------------
  ;****************** PROCESS STATE FUNCTIONS ******************
```

```
(define (success producto precio-producto monedas-actuales monedas-ingresadas transacciones-nueva)
  ;Despliegue de información de la transacción
  (display "TRANSACCIÓN EXITOSA\n")
  (display "Producto: ")
  (display producto)
  (display "\n")
  (display "Precio: ")
  (display precio-producto)
  (display "\n")
  (display "Monto ingresado: ")
  (display (apply + monedas-ingresadas))
  (display "\n")
  (display "Cambio: ")
  (display (-(apply + monedas-ingresadas)  precio-producto))
  (display "\n")
  (display "------------------")
  (display "\n")


   #|

  Para poder actualizar productos y monedas, mandamos llamar a la función update-stock,
    que regresa una lista con los productos actualizados

    Para el caso de las monedas, se utiliza la función update-coins, la cual
    tiene implicita la función add-coins, que se encarga de actualizar las monedas
    con la cantidad que ingresó el usuario.
 Una vez recibido esta nueva lista, genera otra lista que resta las monedas necesarias
 para proporcionarle el cambio al usuario.
  |#

  ;Leemos la siguiente transacción
  (leerTransacciones (cdr transacciones-nueva)
                     (update-stock producto productos)
                     (update-coins (-(apply + monedas-ingresadas)  precio-producto);cambio
                       (add-coins monedas-actuales
                                  (sort monedas-ingresadas >))); monedas
                     )
  )

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

;Verificar que todas las monedas se encuentren en el inventario
(define (monedas-aceptadas? monedas-ingresadas monedas-disponibles)
  (if (not(null? monedas-ingresadas))
      (if (not (equal? (member (car monedas-ingresadas) monedas-disponibles) #f))
          (monedas-aceptadas? (cdr monedas-ingresadas) monedas-disponibles)
          #f); Si member retorna false la moneda no es válida
      #t) ;si llega al final sin salirse retorna true
  )

;----------------------------
;Desplegar errores y continuar con la lista de transacciones
(define (error-handler id-error transacciones productos monedas)
  (display "\n")
  (cond
    [(equal? id-error 1)
     (display "ERROR: No se ha aceptado alguna de las monedas\n")]
    [(equal? id-error 2)
     (display "ERROR: No se han ingresado monedas suficientes\n")]
    [(equal? id-error 3)
     (display "ERROR: Producto no disponible\n")]
    [(equal? id-error 4)
     (display "ERROR: No hay cambio suficiente\n")]
    )
  (display "------------------")
  (display "\n")
  (leerTransacciones (cdr transacciones)
                     productos
                     monedas
                     )
  )


;----------------------------
```

```
(define (evalua transacciones productos monedas)
  ;Evaluamos si la transacción es posible
  (cond
    ;Si alguna moneda no es aceptada
    [(not(monedas-aceptadas? (cdar transacciones) (map car monedas)))
        (error-handler 1 transacciones productos monedas)]
    ;Si la suma de las monedas ingresadas no es suficiente
    [(< (apply + (cdar transacciones)) (find-product-price (caar transacciones) productos))
       (error-handler 2 transacciones productos monedas)]
    ; Si el stock está vacío
    [(equal? (find-product-stock (caar transacciones) productos) 0)
       (error-handler 3 transacciones productos monedas)]
    ;Si la máquina no cuenta con cambio
    [(not (cambio? (-(apply + (cdar transacciones))
                     (find-product-price (caar transacciones) productos))
                   monedas))
       (error-handler 4 transacciones productos monedas)]
    [else (success (caar transacciones);Producto
                   (find-product-price (caar transacciones) productos);Precio
                   monedas; lista de monedas actuales
                   (cdar transacciones);Monedas ingresadas
                   transacciones);lista de transacciones
          ])

 )

;----------------------------
(define (terminar-procesos productos monedas total-inicial)

  ;Mostramos los datos finales al usuario
  (display "\n")
  (display "---------- FIN DE PROCESOS ---------- \n")
  (display "GANANCIA OBTENIDA: ")
  (display (- (apply + (map (lambda (x) (* (car x) (cdr x))) monedas)) total-inicial))
  (display "\n")
  (display "PRODUCTOS CON POCO INVENTARIO: ")
  (display (map car (filter (lambda (x) (<= (caddr x) 3)) productos)))
  (display "\n")
  (display "MONEDAS CON MUCHO INVENTARIO: ")
  (display (map car (filter (lambda (x) (>= (cdr x) 35)) monedas)))
  (display "\n")
  (display "MONEDAS CON POCO INVENTARIO: ")
  (display (map car (filter (lambda (x) (<= (cdr x) 5)) monedas)))

  ;Modificamos los archivos con los valores que resultaron
  (define archivo-productos (open-output-file "productos.txt" #:exists `replace))
  (write productos archivo-productos)
  (close-output-port archivo-productos)

  (define archivo-monedas (open-output-file "monedas.txt" #:exists `replace))
  (write monedas archivo-monedas)
  (close-output-port archivo-monedas)


)

;-----------------------------------
;Si la lista de transacciones está vacía, termina el programan de lo contrario evalúa
;la lista
(define (leerTransacciones transacciones-nueva productos-nuevos monedas-nuevas)
  (if (not(null? transacciones-nueva))
      (evalua transacciones-nueva productos-nuevos monedas-nuevas)
      (terminar-procesos productos-nuevos
                         monedas-nuevas
                        (apply + (map (lambda (x) (* (car x) (cdr x))) monedas)))
   )
 )


;-------------------------------------------------------------------------------------------
; ************** PROGRAMA PRINCIPAL *********************************

(display "----- SISTEMA DE EVALUACIÓN DE TRANSACCIONES -----\n")

(leerTransacciones transacciones productos monedas)
```

Pruebas realizadas

1. Transacción exitosa sin cambio

Caso: ('R . (5 5 5))

Respuesta por consola:

```
----- SISTEMA DE EVALUACIÓN DE TRANSACCIONES -----
TRANSACCIÓN EXITOSA
Producto: 'R
Precio: 15
Monto ingresado: 15
Cambio: 0
------------------
```

2. Transacción exitosa con cambio

Caso: ('R . (5 5 10))

Respuesta por consola

```
------------------
TRANSACCIÓN EXITOSA
Producto: 'R
Precio: 15
Monto ingresado: 20
Cambio: 5
------------------
```

3. Un producto se termina a mitad de las transacciones

Caso: ('R . (5 5 5))('R . (5 5 10))

Producto: ((quote R) 15 1)

Respuesta por consola:

```
TRANSACCIÓN EXITOSA
Producto: 'R
Precio: 15
Monto ingresado: 15
Cambio: 0
------------------
ERROR: Producto no disponible
```

4. Monedas ingresadas no suficientes

('T . (5 1 2 2))

Respuesta por consola:

```
ERROR: No se han ingresado monedas suficientes
```

5. Moneda no aceptada

('T . (5 7 2 3))

Respuesta por consola

```
ERROR: No se ha aceptado alguna de las monedas
```

6. No hay stock suficiente de monedas

Caso:('S . (50))

Stock de monedas: ((50 . 0) (20 . 0) (10 . 0) (5 . 0) (2 . 0) (1 . 0))

Respuesta por consola:

```
------------------

ERROR: No hay cambio suficiente
------------------
```

7. No se ingresaron monedas

Caso: ('S . ())

Respuesta por consola:

```
ERROR: No se han ingresado monedas suficientes
------------------
```

8. Todas las transacciones arrojan error

Este caso de prueba es más para evaluar el contenido de la ganancia obtenida

```
----- SISTEMA DE EVALUACIÓN DE TRANSACCIONES -----

ERROR: No se han ingresado monedas suficientes
------------------

ERROR: Producto no disponible
------------------

ERROR: Producto no disponible
------------------

ERROR: No se han ingresado monedas suficientes
------------------

ERROR: No se ha aceptado alguna de las monedas
------------------

ERROR: No hay cambio suficiente
------------------

----------- FIN DE PROCESOS ----------
GANANCIA OBTENIDA: 0
PRODUCTOS CON POCO INVENTARIO: ('R)
MONEDAS CON MUCHO INVENTARIO: ()
MONEDAS CON POCO INVENTARIO: (50 20 2 1)
```

Experiencia de aprendizaje

Me divertí mucho haciendo esta evidencia. Verdaderamente fue una experiencia retadora que me ayudó a pensar de otra manera y buscar continuamente soluciones fuera de lo común que se pudieran adaptar a las metas que busco lograr. Fui capaz de aplicar los conocimientos adquiridos en clase para la resolución de esta evidencia integradora. Ojalá los demás proyectos de otras materias fueran igual de integradores y puedan aplicar valor como lo hizo esta evidencia.

Anexos

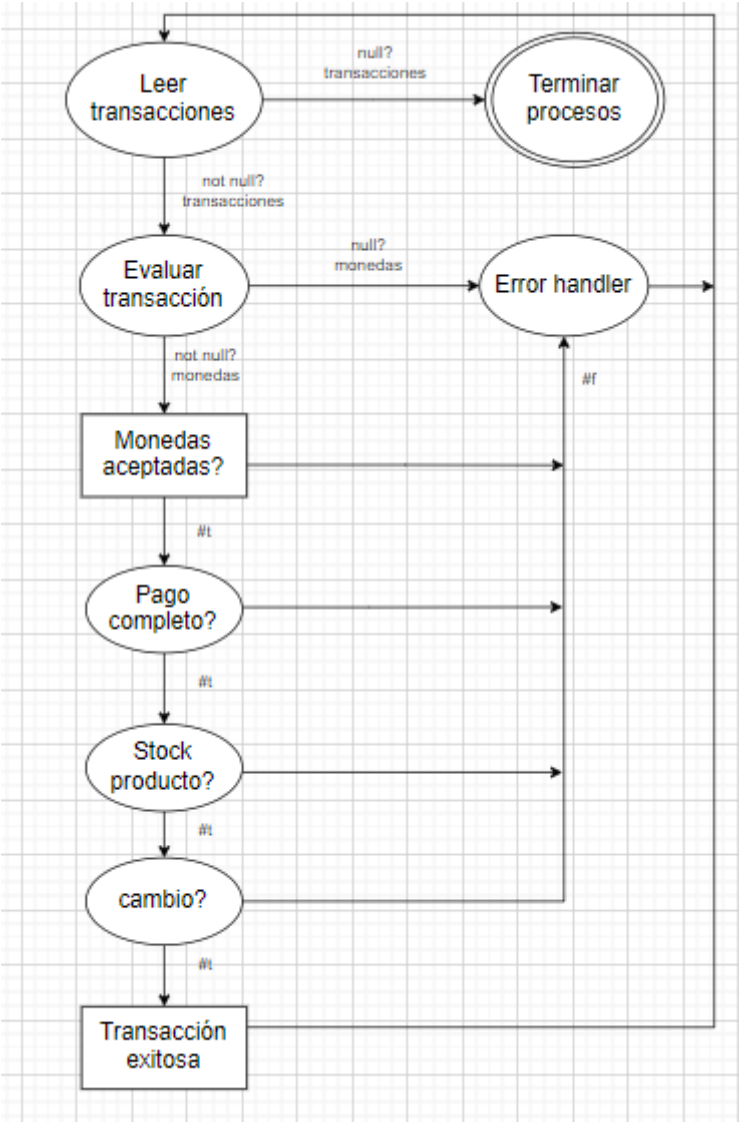

Diagrama de estados del programa