UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

PROYECTO DE LABORATORIO 1 – PARADIGMA FUNCIONAL

Paradigmas de Programación

Isaac Alejandro Espinoza Barría 21.278.828-7

Sección:

13204-0-A-1 E.L

Profesor:

Roberto González-Ibáñez

Fecha:

22 de abril de 2024

TABLA DE CONTENIDO

1. INTRODUCCIÓN

Conocer diferentes paradigmas de programación es esencial para que los programadores desarrollen soluciones efectivas ante problemáticas considerando todos los puntos de vista posibles y eligiendo los mejores métodos para escribir sus programas.

En el presente laboratorio plantea el problema de la demanda de movilidad urbana y cómo emplear la solución de un sistema de administración de una red de metro mediante un programa en el lenguaje Racket/Scheme bajo el paradigma funcional de programación.

El documento consta de introducción, descripción del problema, análisis del problema, diseño de la solución, consideraciones de implementación, instrucciones de uso, resultados y autoevaluación, y conclusiones.

2. DESCRIPCIÓN DE LA PROBLEMÁTICA

2.1. PROBLEMA

En el mundo actual, y con miras al futuro, la humanidad enfrenta grandes desafíos en torno a las condiciones de vida; la contaminación y mala calidad del aire son problemas que progresivamente se han vuelto importantes de combatir.

Asimismo, en las grandes urbes donde millones de personas confluyen en el diario vivir, los problemas de contaminación son evidentes; particularmente la contaminación que produce la locomoción de vehículos particulares produciendo gases de invernadero. Además, las grandes ciudades enfrentan dificultades en torno a la sobrepoblación y crisis migratoria que producen un crecimiento exponencial de la cantidad de habitantes y, por ende, gran cantidad de autos en las calles.

La locomoción tradicional, no puede abarcar la gran demanda de transporte existente. El aumento de habitantes produce que sea muy difícil el transportarse en la ciudad, por lo que las personas optan por usar sus vehículos particulares a pesar del costo monetario que esto implica y lo contaminantes que son.

Ante esto, es indispensable que las ciudades se expandan para recibir nuevos habitantes. No basta con la construcción de viviendas y edificios de servicios públicos, es necesario la creación de un sistema de locomoción que sea rápido, sustentable y amigable con el medio ambiente; dado que ayude a todos los habitantes a desplazarse a sus lugares de trabajo, estudio u hogares.

La creación de una red de metro es una beneficiosa solución al problema de contaminación y la sobrepoblación, ya que, es un sistema de locomoción sustentable y poco contaminante, alivia la congestión de tráfico automotriz en la superficie, además de lograr conectar diversos puntos de la ciudad en poco tiempo.

2.2. DESCRIPCIÓN DEL PARADGIMA

Para entender los procedimientos a emplear en la solución del problema, es necesario conocer algunos conceptos previos relevantes que logran facilitar su comprensión. A continuación, se enumeran y definen:

1.- Paradigma: Forma o manera de percibir el mundo; modo de hacer las cosas para abordar situaciones. “Realizaciones científicas universalmente reconocidas que, durante cierto tiempo, proporcionan modelos de problemas y soluciones a una comunidad científica” (Kuhn, 1962).

2.- Paradigma de programación: Modelo básico de diseño e implementación de programas en un computador. Donde se aceptan válidos ciertos criterios y principios que modelan el proceso de diseño de programa, orientan la forma de pensar y solucionar problemas (Spigariol, 2005).

3.- Tipo de dato abstracto (TDA): Estructura de datos conceptual creada por un programador en un computador; busca representar entidades de la realidad rescatando los elementos esenciales y omitiendo lo superfluo o poco necesario para la representación de estos. La especificación de un tipo de dato abstracto se basa en capas que facilitan la implementación propia del TDA y sus operaciones, las que son: constructores, funciones de pertenencia, selectores, modificadores y otras implementaciones.

4.- Función: Procedimiento o relación que toma elementos de algún conjunto de partida (dominio), y retorna elementos de un conjunto de llegada (recorrido); de modo que para cada elemento del dominio sólo exista un único elemento del recorrido.

5.- Paradigma funcional: Método, marco de referencia, modo de programar en un computador tal que percibe toda situación y proceso a través de funciones y la composición de estas, sin la asignación de variables. Es un paradigma de programación declarativo, esto implica que se detalla él qué debe hacer el programa por sobre él cómo se hace.

Para desarrollar la solución, en este primer laboratorio se utiliza el paradigma funcional.

3. ANÁLISIS DEL PROBLEMA

Para solucionar el problema, se desea crear un sistema de administración de una red de metro en el lenguaje de programación Scheme, bajo el paradigma funcional declarativo. Esto quiere decir que se debe crear un programa que, mediante funciones, consiga realizar todas las operaciones básicas para operar un sistema de metro.

Por consiguiente, se deben considerar diferentes TDAs que logran abstraer los elementos importantes de la red metro para poderlos trabajar dentro del programa. Donde se declaren todas las funciones que puedan operar la red de metro; y después en una consola externa, mediante comandos, se puedan ejecutar las funciones previas que el usuario estime conveniente.

La red de metro está constituida por una red líneas, estaciones, trenes, carros, vías, túneles, sistemas de control y mantenimiento, etc. Como requisitos, el programa de debe cumplir con las

siguientes funcionalidades: creación y gestión de líneas de metro, añadir estaciones, planificación de rutas y conexiones, gestión de trenes, control de tarifas, gestión de personal trabajador, mantenimiento de infraestructura, monitoreo y control, e interfaz de usuario.
Cabe destacar que la creación de esta red de metro consta de tres partes, siendo la presente la primera. Por lo que se espera que en la tercera parte estén completamente cubiertos estos requisitos.

4. DISEÑO DE LA SOLUCIÓN

Para implementar la red de metro, surge la necesidad de crear diferentes abstracciones de elementos mediante TDAs. De modo de ir conteniendo tipos de datos abstractos dentro de un gran TDA subway que represente el metro.
Debido a que el lenguaje Scheme está relacionado con las listas de elementos, los TDAs creados utilizan listas como estructura de datos. A continuación, se mencionan los tipos de datos abstractos usados, y en el (Anexo 1) se expone la explicación sobre cómo están constituidos: Type station, Station, Section, Line, Pcar, Train, Driver y Subway.
Algunos elementos importantes para destacar son que para que un tren sea válido debe tener cumplir con que sus carros extremos sean del tipo terminal y que sus carros internos sean del tipo central; la estructura mínima de un tren válido son dos carros del tipo terminal unidos. Para que una línea sea válida, puede ser una línea normal donde sus estaciones extremas sean del tipo terminal u combinación y que desde una estación se pueda recorrer todas las otras estaciones de la línea; o puede ser una línea circular donde no tiene estaciones terminales, pero se puede regresar a la misma estación inicial al recorrer todas las estaciones sin cambiar de sentido. Y, por último, cualquier dato de entrada incorrecto en las funciones puede producir un error de ejecución.
Como elementos de programación, se utilizó la currificación de funciones, además de la recursión tanto natural como de cola para trabajar con las listas y así también cumplir con los requerimientos funcionales que pedían emplear algún tipo de recursión en particular, o por el contrario resolver procedimientos de forma declarativa; por lo que para esto se usó funciones como map o filter de la librería estándar. Procurando siempre respetar los principios de la programación funcional.

5. CONSIDERACIONES DE IMPLEMENTACIÓN

5.1. ESTRUCTURA DEL PROYECTO

El proyecto se basa en la encapsulación de diferentes TDAs dado que todos están contenidos bajo el TDA principal llamado “subway”. Donda cada sub TDA tiene sus funciones propias que facilitan la operación de estos en TDAs externos, sin preocuparnos del cómo funcionan sino él lo qué hacen.

5.2. BIBLIOTECAS EMPLEADAS

El lenguaje Racket/Scheme posee varias librerías con funciones disponibles para el usuario, sin embargo, fueron utilizadas sólo las primitivas del dialecto; en particular la librería estándar del lenguaje Scheme (R6RS).

5.3. INTERPRETE USADO

El programa fue totalmente desarrollado en el lenguaje Racket (derivado de Scheme), escrito en el software DrRacket 8.10 como IDE; el que también puede compilar el programa y ejecutar las funciones en su propia consola de comando.

6. INSTRUCCIONES DE USO

6.1. PASOS GENERALES

- Instalar DrRacket 8.10 o superior en el computador.
- Abrir la carpeta “Lab1_Espinoza_Barria_212788287” y verificar la existencia 9 archivos rkt necesarios para la ejecución (ver Anexo 2).
- Abrir el archivo “main_212788287_EspinozaBarria” en el IDE DrRacket (ver Anexo 3).
- Presionar el botón “Run” en la esquina superior derecha del IDE (ver Anexo 4).
- Ejecutar funciones en la consola del IDE, generalmente en la parte inferior o derecha (ver Anexo 5).

6.2. EJEMPLOS

Se definen 3 ejemplos de ejecución de las funciones: crear estación (ver Anexo 6), crear sección (ver Anexo 7) y crear línea (ver Anexo 8 y 9); los que se muestran en imágenes.

6.3. RESULTADOS ESPERADOS

Se espera que los primeros 25 requerimientos funcionales se ejecuten correctamente, sin errores de compatibilidad y de análisis en las estructuras que generan.

6.4. POSIBLES ERRORES

Considerando que siempre se ingresen parámetros validos correspondiente a los dominios de las funciones. Sólo puede haber errores en la función 26, debido a la falta de condiciones para corroborar que los elementos ingresados coincidan con otros datos del TDA subway; como si el tren está

disponible, si el conductor está disponible, etc. Esto sin considerar a las funciones 27 y 28 que no fueron implementadas.

7. RESULTADOS Y AUTOEVALUACIÓN

En el Anexo 10 se presenta una tabla detallada donde se listan todas los requerimientos funcionales y su respectivo nivel de implementación. A modo general, 25 funcionalidades de las 28 se ejecutan correctamente, esto considerando que los elementos de entrada pertenezcan a los conjuntos de dominio de cada función; y la función 26 se ejecuta de forma parcial. También, en el Anexo 11 se presenta la tabla autoevaluación de los requerimientos no funcionales.

8. CONCLUSIONES

En este primer laboratorio del proyecto semestral, se logró implementar más del 92% del total de requerimientos funcionales solicitados, con lo que estoy bastante satisfecho. Sin embargo, espero que para próximos laboratorios logre entregar el 100% de las funcionalidades.

Elementos que me gustaría destacar del paradigma funcional es su concepción de él “qué” por sobre él “cómo” se desarrolla el código, por lo que facilita que cualquier persona pueda ejecutar el programa sin mayores inconvenientes. Además, destaco la afinidad entre el lenguaje y la recursión, debido a que una vez que entiendes su funcionamiento, es fácil su uso. Y como elementos negativos, me fue complicado el uso de listas dentro de listas y trabajar con los elementos anidados en ellas.

A modo personal, en un inicio fue difícil dejar de pensar en el paradigma procedural imperativo en el que estaba acostumbrado. Pero sin duda, ha sido una experiencia enriquecedora desarrollar conocimientos en un paradigma tan distinto, ayudándome a salir de mi zona de confort, y a darme cuenta de que no todo tiene asignarse en variables.

Como informáticos y programadores, debemos conocer los diferentes paradigmas y modos sobre cómo abordar problemas. Trabajar con variados paradigmas no sólo es importante para solucionar problemáticas con los mejores métodos posibles, sino también para abrir nuestro entendimiento y comprender que lo que es difícil de abordar con una mirada puede ser fácil de enfrentar desde otro punto de vista.

REFERENCIAS

Kuhn, T. S. (1962). *La estructura de las revoluciones científicas.*

Spigariol, L. (2005). *Fundamentos teóricos de los Paradigmas de Programación.* Buenos Aires: Facultad Regional Buenos Aires Universidad Tecnológica Nacional.

ANEXO

1.

Nombre TDA	Dato Abstracto	Estructura	Operaciones Relevantes
Type Station	Tipo de estación. Regular, mantención, combinación o terminal.	String de un carácter que representa el tipo, puede ser “r”, “m”, “c” o “t”.	Crear dato type-station.
Station	Estación de metro.	Lista (id – nombre – type station – tiempo en segundos de parada por tren).	Crear station.
Section	Sección entre estaciones.	Lista (estación 1 – estación 2 – distancia en kilómetros entre estaciones – costo monetario).	Crear section.
Line	Línea de metro.	Lista (id – nombre – tipo de riel – lista TDAs section).	Crear line, agregar sections a un line, obtener datos de un line.
Pcar	Carro de metro.	Lista (id - capacidad pasajeros – modelo – tipo de carro).	Crear pcar.
Train	Tren de metro. Conjunto de carros.	Lista (id – fabricante – tipo de riel – rapidez km/h – tiempo de espera en minutos por estación – lista de carros en orden).	Crear tren, verificar validez, añadir o eliminar carros, obtener datos de tren.
Driver	Conductor de tren.	Lista (id – nombre – fabricante de trenes que conduce).	Crear driver.
Subway	Metro.	Lista (id – nombre – lista de trenes – lista de líneas – lista de conductores – lista de pares (id línea – ids trenes) – lista de rutas (id conductor – id tren – hora partida – estación inicio – estación llegada))	Crear subway, añadir y asignar elementos, obtener datos.

Tabla: Especificación TDAs usados.

2.

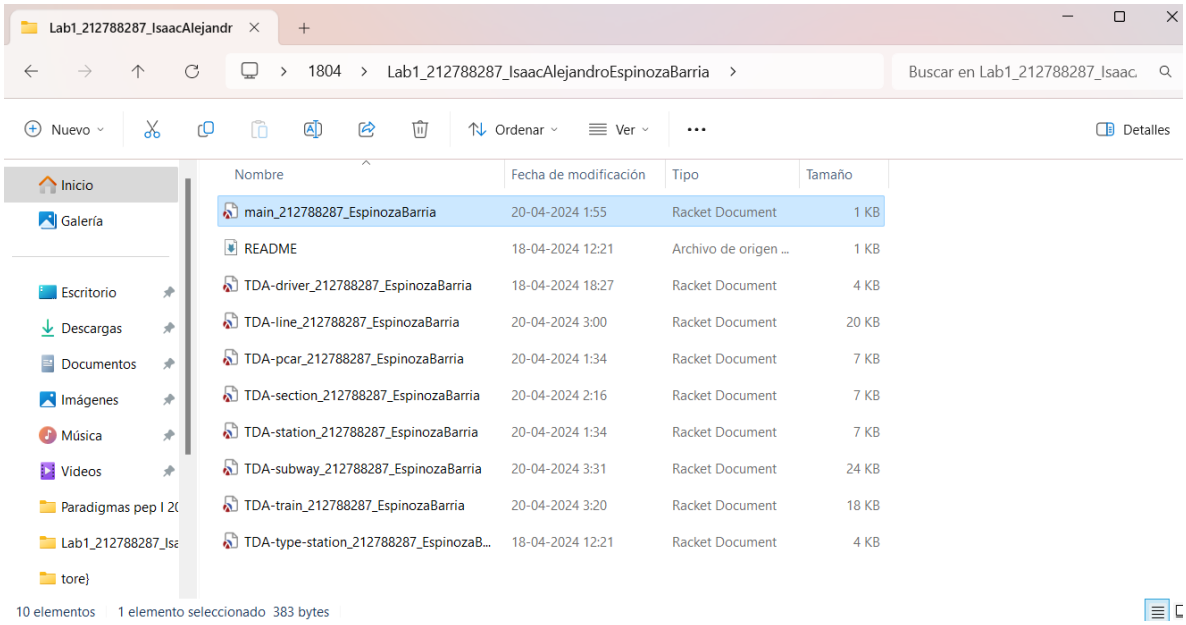

Imagen: Carpeta contenedora de los archivos.

3.

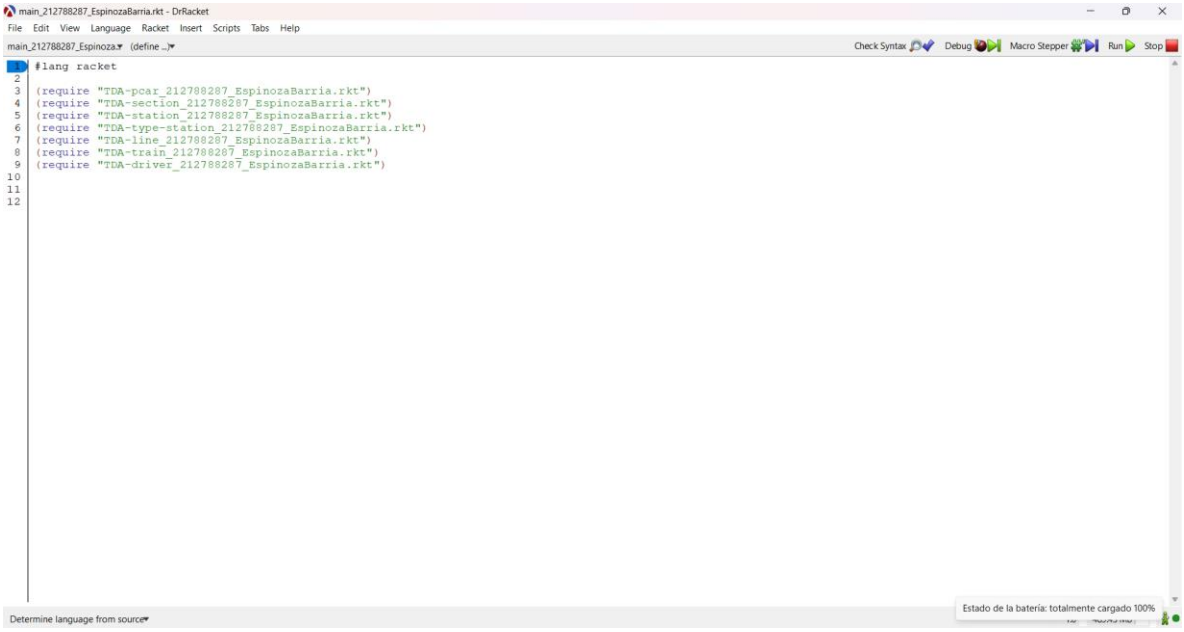

Imagen: Archivo “main_212788287_EspinozaBarria” en el IDE DrRacket.

4.-

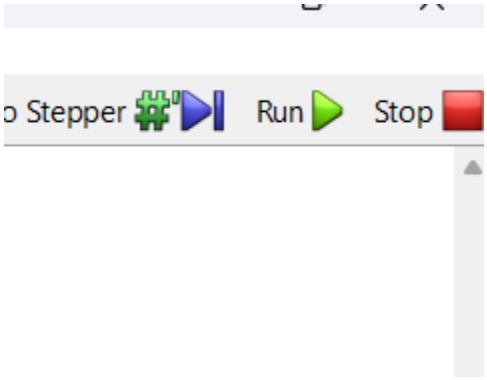

Imagen: Botón “Run” dentro del IDE.

5.-

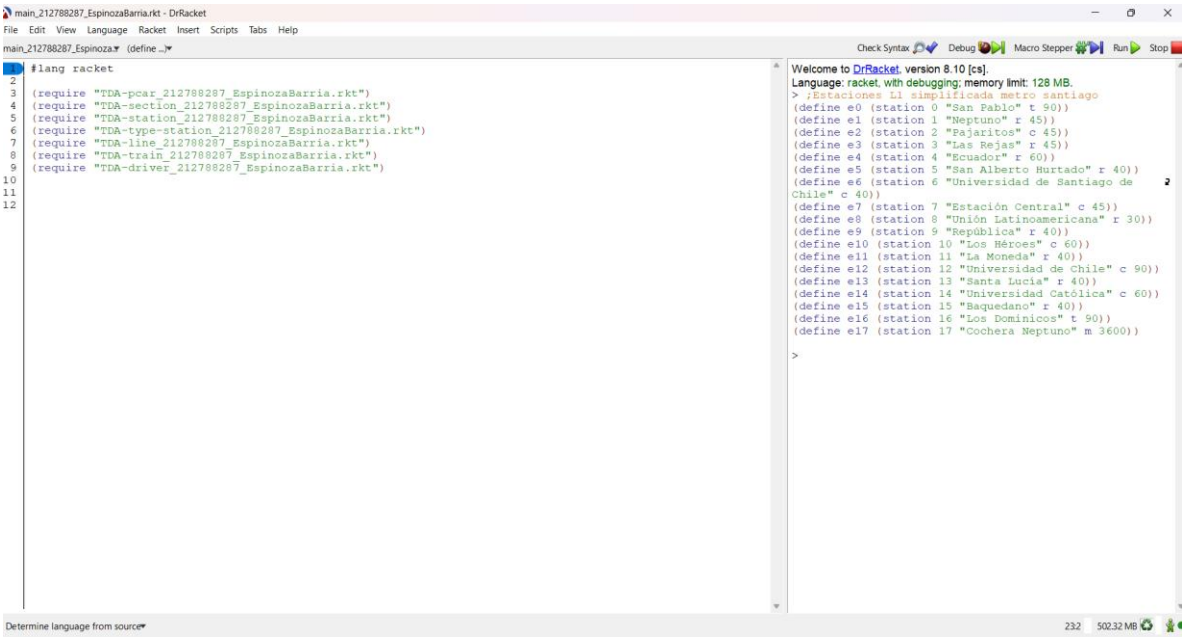

Imagen: Consola de comandos en IDE DrRacket.

6.-

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ;CREACION DE ESTACION
(define e0 (station 0 "San Pablo" t 90))
> e0
'(0 "San Pablo" "t" 90)
>
```

Imagen: Creación de una estación en la consola de DrRacket.

7.-

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with debugging; memory limit: 128 MB.
>
;CREACION DE ESTACIONES
(define e0 (station 0 "San Pablo" t 90))
(define e1 (station 1 "Neptuno" r 45))

;CREACION DE SECCION
(define s0 (section e0 e1 4 15))
> s0
'((0 "San Pablo" "t" 90) (1 "Neptuno" "r" 45) 4 15)
>
```

Imagen: Creación de una sección en la consola de DrRacket.

8.-

```
Welcome to DrRacket, version 8.10 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ;Estaciones L1 simplificada metro santiago
(define e0 (station 0 "San Pablo" t 90))
(define e1 (station 1 "Neptuno" r 45))
(define e2 (station 2 "Pajaritos" c 45))
(define e3 (station 3 "Las Rejas" r 45))
(define e4 (station 4 "Ecuador" r 60))
(define e5 (station 5 "San Alberto Hurtado" r 40))
(define e6 (station 6 "Universidad de Santiago de Chile" c 40))
(define e7 (station 7 "Estación Central" c 45))
(define e8 (station 8 "Unión Latinoamericana" r 30))
(define e9 (station 9 "República" r 40))
(define e10 (station 10 "Los Héroes" c 60))
(define e11 (station 11 "La Moneda" r 40))
(define e12 (station 12 "Universidad de Chile" c 90))
(define e13 (station 13 "Santa Lucía" r 40))
(define e14 (station 14 "Universidad Católica" c 60))
(define e15 (station 15 "Baquedano" r 40))
(define e16 (station 16 "Los Dominicos" t 90))
(define e17 (station 17 "Cochera Neptuno" m 3600))

;Tramos Línea 1
(define s0 (section e0 e1 4 15))
(define s1 (section e1 e2 3 14))
(define s2 (section e2 e3 2.5 10))
(define s3 (section e3 e4 4.5 17))
(define s4 (section e4 e5 4.7 18))
(define s5 (section e5 e6 4.3 17))
(define s6 (section e6 e7 3.8 12))
(define s7 (section e7 e8 2.5 10))
(define s8 (section e8 e9 4.5 17))
(define s9 (section e9 e10 4.7 18))
(define s10 (section e10 e11 4.3 17))
(define s11 (section  e11 e12 3.8 12))
(define s12 (section e12 e13 4.5 17))
(define s13 (section e13 e14 4.7 18))
(define s14 (section e14 e15 4.3 17))
(define s15 (section e15 e16 4.2 17))
;enlace cochera
(define s16 (section e1 e17 3.8 12))

;Creación de Línea 1 con todos los tramos
(define l1 (line 1 "Línea 1" "UIC 60 ASCE" s0 s1 s2 s3 s4 s5 s6 s7
s8 s9 s10 s11 s12 s13 s14 s15))
>
```

Imagen: Creación de una linea en la consola de DrRacket. Parte 1.

9.-

```
> l1
'(1
  "Línea 1"
  "UIC 60 ASCE"
  (((0 "San Pablo" "t" 90) (1 "Neptuno" "r" 45) 4 15)
   ((1 "Neptuno" "r" 45) (2 "Pajaritos" "c" 45) 3 14)
   ((2 "Pajaritos" "c" 45) (3 "Las Rejas" "r" 45) 2.5 10)
   ((3 "Las Rejas" "r" 45) (4 "Ecuador" "r" 60) 4.5 17)
   ((4 "Ecuador" "r" 60) (5 "San Alberto Hurtado" "r" 40) 4.7 18)
   ((5 "San Alberto Hurtado" "r" 40)
    (6 "Universidad de Santiago de Chile" "c" 40)
    4.3
    17)
   ((6 "Universidad de Santiago de Chile" "c" 40)
    (7 "Estación Central" "c" 45)
    3.8
    12)
   ((7 "Estación Central" "c" 45)
    (8 "Unión Latinoamericana" "r" 30)
    2.5
    10)
   ((8 "Unión Latinoamericana" "r" 30)
    (9 "República" "r" 40)
    4.5
    17)
   ((9 "República" "r" 40) (10 "Los Héroes" "c" 60) 4.7 18)
   ((10 "Los Héroes" "c" 60) (11 "La Moneda" "r" 40) 4.3 17)
   ((11 "La Moneda" "r" 40)
    (12 "Universidad de Chile" "c" 90)
    3.8
    12)
   ((12 "Universidad de Chile" "c" 90)
    (13 "Santa Lucía" "r" 40)
    4.5
    17)
   ((13 "Santa Lucía" "r" 40)
    (14 "Universidad Católica" "c" 60)
    4.7
    18)
   ((14 "Universidad Católica" "c" 60)
    (15 "Baquedano" "r" 40)
    4.3
    17)
   ((15 "Baquedano" "r" 40) (16 "Los Dominicos" "t" 90) 4.2 17)))
>
```

Imagen: Creación de una linea en la consola de DrRacket. Parte 2.

10.-

Requerimientos	Grado de Implementación	Pruebas realizadas	% Pruebas exitosas	% Pruebas fracasadas	Anotación
TDAs	1	8 TDAs creados exitosamente.	100	0	
Station	1	(+10) casos de station exitosos.	100	0	
Section	1	(+10) casos de section exitosos.	100	0	
Line	1	(3) casos line exitosos agregando secciones al definir, y (3) casos line exitosos agregando secciones posteriormente.	100	0	Crea estructuras line pero no necesariamente válidas, esto se comprueba en line?.
Line-length	1	(+5) casos line-length exitosos.	100	0	
Line-section-length	1	(+5) casos line-section-length exitosos.	100	0	
Line-cost	1	(+5) casos line-cost exitosos.	100	0	
Line-section-cost	1	(+5) casos line-section-cost exitosos.	100	0	
Line-add-section	1	(+15) casos line-add-section exitosos, (+10) casos exitosos de secciones repetidas	100	0	
Line?	1	(+5) casos line? exitosos con líneas válidas e inválidas.	100	0	Comprueba líneas del tipo regular y circular, donde se recorran todas las estaciones.
Pcar	1	(+15) casos pcar exitosos.	100	0	
Train	1	(5) casos train exitosos.	100	0	Crea estructuras train pero no necesariamente válidas, esto se analiza en train?.
Train-add-car	1	(+10) casos train-add-car exitosos. (3) casos exitosos de carros incompatibles.	100	0	SI se intenta agregar carros incompatibles de modelo, se retorna el tren sin cambio.
Train-remove-car	1	(+10) casos train-remove-car exitosos, (2) casos exitosos donde la posición es invalida	100	0	Si se ingresa una posición inválida, se retorna el tren sin cambio
Train?	1	(+10) casos train? exitosos, donde los trenes eran válidos e inválidos	100	0	Comprueba compatibilidad carros y

		considerando la estructura mínima.			de tipo de carro (terminales y centrales).
Train-capacity	1	(+10) casos train-capacity exitosos.	100	0	
Driver	1	(4) casos driver exitosos.	100	0	
Subway	1	(+10) casos subway exitosos.	100	0	
Subway-add-train	1	(+10) casos subway-add-train exitosos, (+10) casos exitosos de trenes repetidos.	100	0	Si se intenta agregar trenes repetidos se retorna el mismo subway de ingreso.
Subway-add-line	1	(+10) casos subway-add-line exitosos, (3) casos exitosos de lineas repetidos.	100	0	Si se intenta agregar lineas repetidas se retorna el mismo subway de ingreso.
Subway-add-driver	1	(+10) casos subway-add-train exitosos, (2) casos exitosos de trenes repetidos.	100	0	Si se intenta agregar conductores repetidos se retorna el mismo subway de ingreso.
Subway->string	1	(+ 5) casos subway->string exitosos	100	0	
Subway-rise-section-cost	1	(3) casos subway-rise-section-cost exitosos.	100	0	
Subway-set-station-stoptime	1	(+5) casos subway-set-station-stoptime exitosos.	100	0	
Subway-assign-train-to-line	1	(+10) casos subway-assign-train-to-line, (5) casos exitosos con trenes repetidos y/o incompatibles con el tipo de riel.	100	0	
Subway-assign-driver-to-train	0,5	(+10) casos subway-assign-driver-to-train exitosos, (4) casos sin éxitos al ingresar trenes ya asignados o conductores sin compatibilidad con el tren.	100	0	Función se ejecuta correctamente pero no verifica si efectivamente los elementos ingresados están presentes en el subway.
Where-is-train	0	No implementada.	-	-	No implementada principalmente por motivos de tiempo, y de que no se me ha ocurrido cómo abordarla.
Subway-train-path	0	No implementada.	-	-	No implementada principalmente por motivos de tiempo, y de

					que no se me ha ocurrido cómo abordarla.

Tabla: Autoevaluación requerimientos funcionales.

11.-

Requerimientos No Funcionales	Grado de Implementación	Anotaciones
Autoevaluación	1	Está presente en este documento como en un archivo txt presente el archivo zip contendor del laboratorio.
Lenguaje	1	Scheme/Racket
Versión	1	Se utilizó la versión 8.10, que es superior a la 6.11 dada como la versión mínima.
Standard	1	Se usaron funciones standard de Racket.
No variables	1	No se usaron en ningún momento funciones como set! o let.
Documentación	1	Todas las funciones indican dominio, recorrido, tipo de recursión si es que usa y lo que hace la función.
Dom->Rec	1	Se respetan los conjuntos de dominio y recorrido de las funciones en su definición.
Organización	1	El laboratorio consta con cada TDA en un archivo aparte, donde cada uno contiene los requerimientos funcionales asociados al propio TDA.
Historial	1	Se utiliza GitHub para guardar versiones del avance del proyecto, donde se hicieron más de 10 commits desde hace dos semanas.
Script de pruebas	1	Se entrega un archivo script de pruebas dentro de la carpeta contenedora de los códigos.
Prerrequisitos	1	Se cumplen los prerrequisitos de todas las funcionalidades que sí se emplearon.

Tabla: Autoevaluación requerimientos no funcionales.