

# Informe Laboratorio 1 - Paradigma Funcional

Paradigmas de Programación

**Nombre:** Felipe Cubillos Arredondo

**Profesor:** Gonzalo Martinez

**Sección:** B-2

**Fecha:** 22/04/2024

## Índice

<b>1. Introducción</b>	<b>2</b>
1.1. Descripción del Problema . . . . .	2
1.2. Descripción del Paradigma . . . . .	2
<b>2. Desarrollo</b>	<b>3</b>
2.1. Análisis del Problema . . . . .	3
2.2. Diseño de la Solución . . . . .	3
2.3. Aspectos de Implementación . . . . .	4
2.4. Instrucciones de uso . . . . .	4
2.5. Resultados y Autoevaluación . . . . .	4
<b>3. Conclusiones</b>	<b>5</b>
3.1. Referencias . . . . .	5
3.2. Anexos . . . . .	5

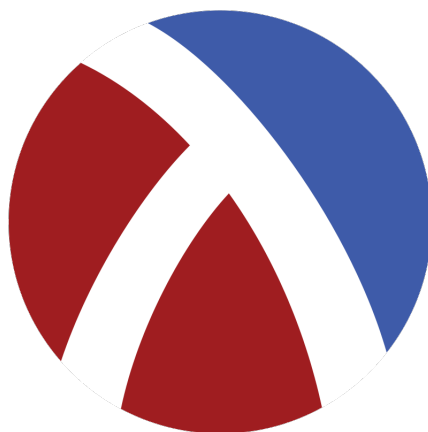


Figura 1: Logo De Lenguaje Racket

# 1. Introducción

## 1.1. Descripción del Problema

A groso modo, para este laboratorio se nos a pedido implementar un sistema de lineas de metro-tren, tal que se puedan representar **estaciones, secciones, líneas, carros, conductores, trenes y redes**. Cada una de estas con distintivos requisitos funcionales que permiten la interacción y administración del sistema de manera cohesionada. Debemos tomar en consideración las restricciones que el metro considera, cómo la coherencia que deben de tener los carros dentro de un tren y análogamente las secciones dentro de una línea, al igual de casos excepcionales cómo líneas circulares y otros posibles detalles. Además de los posibles datos que pida el usuario para obtener información respecto de cualquier elemento del sistema en general, tales cómo las distancias, horas, ubicaciones, etc...

## 1.2. Descripción del Paradigma

**Paradigma Funcional:** Apoyado del concepto del cálculo Lambda, acuñado por Alonzo Church, el paradigma funcional, cómo indicado por su nombre, se desenvuelve al rededor de funciones; esto implica que, a diferencia del imperativo, no existen los métodos mutables, Además **la salida depende únicamente de la entrada**. Otros conceptos importantes que hay que tener en cuenta son:

- High Order Functions: Funciones de orden superior, funciones que tienen cómo input una función (junto a otros posibles parámetros), cuales devuelven otra función, por ejemplo, la derivada.
- Currificación: La currificación es una técnica utilizada para emplear varias variables en una función por medio de lambda's anidados.
- Recursión: Cómo fué especificado antes, al no existir variables, no se pueden recorrer listas de manera tradicional, por lo que se emplean métodos donde se llama a la misma función dentro de sí misma, siguiendo la estructura de caso base y paso recursivo.

**Lenguaje Racket:** Un dialecto (modernización) de Lisp, en teoría es multiparadigma pero para efectos del curso solo se empleara de manera funcional / declarativa. Apoyado por el Software *Dr.Racket* se desarrollará el laboratorio.

Racket provides building blocks for strong protection mechanisms. If programming is about solving problems in the correct language, systems will necessarily consist of interconnected components in several different languages. Due to the connections, values flow from one linguistic context into another. Since languages are charged with providing and preserving invariants, the creators of languages must have the power to protect the languages' invariants. By implication, Racket must come with mechanisms that enable programmers to protect individual components from their clients. For this reason, Racket comes with the proper building blocks to set up or construct protection mechanisms at any level, all the way from C to languages with sound, higher-order type systems, and any mixture in between."

Este párrafo resume de manera excelente el por que se utiliza en este proyecto, puesto que es un idioma inmutable, sólido y posee la capacidad de generar estructuras, por lo que es idóneo para la representación de sistemas.

Otro aspecto notable de Racket es que este no posee muchos tipos de datos, se trabajará en su mayor parte con números (enteros, decimales, etc...), cadenas de texto, booleanos y más importante, los pares y listas (pares de pares). Además de su enfoque en la recursión, como no se manejan variables, por lo que permite ir operando respecto de estas listas.

## 2. Desarrollo

### 2.1. Análisis del Problema

Analizando meticulosamente el problema, tenemos que tener en cuenta un par de conceptos que se pueden emplear en este paradigma, principalmente el concepto de TDA. Un TDA (Tipo de Dato Abstracto) es una representación de alguna entidad para poder operar sobre esta:

Un Tipo de dato abstracto (en adelante TDA) es un conjunto de datos u objetos al cual se le asocian operaciones. El TDA provee de una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en como estén implementadas dichas operaciones. Esto quiere decir que un mismo TDA puede ser implementado utilizando distintas estructuras de datos y proveer la misma funcionalidad. [2]

Cabe destacar, que se utilizarán capas de los TDA's, osea funciones que se emplean a los TDA's en función de sacar información de estos, por ejemplo, si tengo el TDA avión y quiero saber cuanta capacidad tiene este avión debería emplear el getter `get-capacity` y si quisiera, por algún motivo, cambiar el nombre del avión tendría que emplear el método `set-name`.

En el fondo, **lo que se busca lograr es ir armando desde lo más pequeño**, osea estaciones, carros y secciones, **hasta llegar a algo de mayor escala**, cómo lo serían las líneas y los trenes. Todo esto para sintetizarlo en el **TDA Subway**, cual representaría una **red completa de metro**, cómo lo sería el metro de Santiago, el metro de Nueva York o entre otros.

### 2.2. Diseño de la Solución

El enfoque que tengo al momento de solucionar los problemas planteados es en cierta manera la ingeniería inversa, puesto que ya tengo una idea del resultado que tengo que obtener, por lo tanto se que tengo que descomponer el problema en varios archivos (TDA's) cuales poseen sus respectivos constructores, getters y requerimientos funcionales asociados.

Ya fué mencionado en la descripción del paradigma que uno de los tipos de datos mas utilizados son las listas, y es así como representaré los distintos TDA's, de manera de tener acceso a sus atributos mediante las funciones `car` (Contents of the Address part of Register number), que devuelve el primer elemento de la lista y `cdr` (Contents of the Decrement part of Register number), que devuelve el resto de la lista sin su primer elemento.

De ahí, notamos inmediatamente que uno de los TDA base es el de `tda-station` cual representa una estación con su id, nombre, tipo y tiempo de parada. Otro TDA base sería el `tda-pcar` cual contiene un id, capacidad, modelo y tipo.

Ahora, un TDA que no es base sería el `tda-section`, que comprende dos estaciones, estas secciones se representan como las dos estaciones en una lista junto a la distancia y costo entre ellas. Lo mismo ocurre con `tda-train` y `tda-line`, el primero sería una colección de datos cómo lo son el id del tren, su creador, el tipo de riel, su rapidez, tiempo de espera de estación, y una lista de 0 o más `pcar`'s, mientras que `line` posee un id, nombre, tipo de riel y una lista de 0 o más secciones.

Una vez establecida la jerarquía entre los elementos, nos enfocamos en los **requisitos funcionales** cuales piden distintos atributos entre los TDA's, por ejemplo `line-length` debería devolver la longitud total de una línea, esto se puede resolver de muchas maneras, pero se estima conveniente usar una resolución declarativa, osea, utilizando funciones ya establecidas en racket, cómo lo son el `map`, `filter`, `apply`, entre otras... En este caso conviene utilizar un `map`, que interviene en cada elemento de la lista respecto de una función y un `apply`, o reducción, cuyo objetivo es condensar la lista en un elemento, entonces la estrategia a seguir sería:

1. Obtener la lista de secciones de una línea
2. Obtener la distancia de cada sección
3. Finalmente, sumar todas esas distancias

Ahora, para otro requerimiento cómo lo sería `train-capacity`, se estima utilizar recursión natural, por lo que hay que pensar un poco más en la implementación de este. Primero lo primero, hay que identificar el caso base, para este caso es cuando no queden más elementos en la lista de `pcars`, por lo que la idea es ir llamando a la función, cuyo parámetro es la lista de `pcars`, pero aplicando `(train-capacity (cdr (get-pcar-list train)))`, y sumarle la capacidad del carro actual que esté trayendo, de manera que vayan disminuyendo la cantidad de elementos hasta llegar al caso base, recorriendo la lista en el proceso. Ahora, si este llega a estar vacío, debe retornar 0, puesto que al ser natural deja estados pendientes.


1. Caso Base: ¿Lista vacía? Devuelve 0
2. Si no: Función (lista sin primer elemento) + capacidad de primer elemento

En el caso anterior, y en varias funciones se emplea la técnica del `wrapper`, cuya función es ir definir una función dentro, o de manera separada a la función, cual posee más parámetros que la función original, estos parámetros son auxiliares, cómo lo sería un contador, con el propósito de no alterar el dominio original de la función, facilitando así la lectura del algoritmo. Este recurso se usa de manera extensiva a lo largo del código.

## 2.3. Aspectos de Implementación

La estructura del proyecto es bastante sencilla, cómo fue explicado anteriormente, esta se divide en distintos archivos cada cual con su TDA y funciones asociadas respectivamente, sigue una estructura jerárquica utilizando la palabra clave de Racket (`require`) y (`provide`) cuales permite la exportación e importación de otros archivos respectivamente, así permitiendo la comunicación entre los TDA's. En temas de bibliotecas, no se utilizó más que la base de Racket, ocase `#lang racket`. Y cómo intérprete se utilizó el software Dr. Racket (versión 8.11), cual facilita la escritura de las funciones gracias a la completación de paréntesis y entre otros detalles que mejora la calidad de escritura. Cabe mencionar que el código, junto con otros requisitos se encuentran en Git, por lo que se puede ver el avance y versionamiento del proyecto en general.

## 2.4. Instrucciones de uso

**Antes de empezar:** Se espera que una vez descomprimido el `.zip` o clonado el repositorio de GitHub que quede todo en una misma carpeta, cómo mencionado anteriormente, depende del (`require`), (`provide`), si esto no se cumple, el script de pruebas no funcionará adecuadamente; de igual manera, si se llegase a cambiar cualquier nombre de algún archivo, lo más probable es que no funcionaría el script de pruebas. Una vez hecho esto, solo basta con abrir el script de pruebas con Dr. Racket y hacer click en , que está en la esquina superior derecha de la interfaz. En caso de que se quiera cambiar los parámetros de las funciones en el script, se tiene que hacer respetando el dominio y recorrido documentado en los TDA's respectivos.

## 2.5. Resultados y Autoevaluación

Para ir finalizando el informe, vamos a partir con los aspectos negativos y no logrados, por lo que es importante resaltar que funciones no cumplen con lo pedido:

- `line`: constructor no revisa caso de líneas circulares ni `id`'s repetidos
- `line?`: análogo a constructor, no revisa caso de líneas circulares ni `id`'s repetidos, además no revisa si se puede llegar a todas las líneas aledañas.
- `train?`: funciona para mayoría de los casos, excepto que arroja un error cuando el tren no tiene estaciones, por lo que se omiten ciertos casos en script de pruebas.

Para todo el TDA `train` la función de pertenencia era sumamente importante, puesto que era la que verificaba si siquiera se podía hacer un constructor, por lo que las funciones de constructor, `train-add-car` y `train-remove-car` se ven afectadas. Todo lo descrito anteriormente está en la sección de anexos.

Otra función que cabe destacar proviene del TDA `subway`, en este caso sería `subway->string`, si bien lo convierte a formato string, queda con todo el residuo de paréntesis y en general no es muy agradable a vista.

Respecto al resto de funciones que conciernen al TDA subway no pudieron ser finalizadas por temas de tiempo. En general, todo lo que no fué completado fué por temas de tiempo y por la alta complejidad del laboratorio.

Por otro lado, viendo el lado positivo, todo lo que respecta a station, section, line-length, line-section-length, line-cost, line-section-cost, line-add-section, pcar, train-capacity, driver, subway, subway-add-train, subway-add-line y subway-add-driver funcionan tal y cómo pedido, con sus respectivos dominios, recorridos e implementaciones estipuladas por el enunciado. Todo esto fué evaluado con varias pruebas utilizando mi script de pruebas y el script entregado por el profesor. Además, todo lo descrito anteriormente se vé reflejado en el archivo `autoevaluacion_21461391.CubillosArredondo`.

### 3. Conclusiones

En conclusión, se logró en su mayoría lo que se proponía en un inicio, se pudo representar un sistema de redes de metro utilizando el paradigma funcional, utilizando los recursos vistos en cátedra y por medios de autoestudio se implementa una manera de poder interactuar con el software de manera oportuna, a costa de las limitaciones presentadas en el ítem anterior.

En lo personal, siento que me faltó tiempo para poder perfeccionar el laboratorio y seriamente me hubiera gustado poder haberle dado más énfasis, no obstante, me siento satisfecho con lo logrado en este laboratorio, puesto que aprendí a utilizar el paradigma funcional de manera efectiva y divertida, ahora que tengo estos conocimientos y sabiendo los beneficios de la programación funcional, podré eventualmente aplicarlos a un entorno laboral.

#### 3.1. Referencias

1. Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., & Tobin-Hochstadt, S. (2015). The racket manifesto. In 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss-Dagstuhl-Leibniz Zentrum für Informatik. [1]
2. CC30A Algoritmos y Estructuras de Datos: Tipos de datos abstractos. (s. f.). <https://users.dcc.uchile.cl/~bustos/apuntes/cc30a/TDA/> [2]

#### 3.2. Anexos

	Line Pertenencia
<code>(line? l1) ;devuelve true</code>	<code>#t</code>
<code>(line? l2) ;devuelve false</code>	<code>#f</code>
<code>(line? l2e) ;devuelve false</code>	<code>#t</code>
<code>(line? l2h) ;devuelve true</code>	<code>#t</code>

(a) esperado

(b) obtenido

Figura 2: Error en verificación de line?