

UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Departamento de Ingeniería Informática



PROYECTO DE LABORATORIO 2 – PARADIGMA LÓGICO

Paradigmas de Programación

Isaac Alejandro Espinoza Barría 21.278.828-7

Sección:

13204-0-A-1 E.L

Profesor:

Víctor Flores Sánchez

Fecha:

27 de mayo de 2024

TABLA DE CONTENIDO

TABLA DE CONTENIDO	2
1. INTRODUCCIÓN	3
2. DESCRIPCIÓN DE LA PROBLEMÁTICA	3
2.1. PROBLEMA.....	3
2.2. DESCRIPCIÓN DEL PARADGIMA.....	3
3. ANÁLISIS DEL PROBLEMA.....	5
4. DISEÑO DE LA SOLUCIÓN.....	5
5. CONSIDERACIONES DE IMPLEMENTACIÓN	6
5.1. ESTRUCTURA DEL PROYECTO	6
5.2. BIBLIOTECAS EMPLEADAS	6
5.3. INTERPRETE USADO.....	6
6. INSTRUCCIONES DE USO	6
6.1. PASOS GENERALES	6
6.2. EJEMPLOS.....	6
6.3. RESULTADOS ESPERADOS	7
6.4. POSIBLES ERRORES.....	7
7. RESULTADOS Y AUTOEVALUACIÓN.....	7
8. CONCLUSIONES.....	7
REFERENCIAS	8
ANEXO.....	9

1. INTRODUCCIÓN

Conocer diferentes paradigmas de programación es esencial para que los programadores desarrollen soluciones efectivas ante problemáticas considerando todas las perspectivas posibles y así, desde una visión holista, elegir los mejores métodos al escribir sus programas.

En ciudades altamente densas de población, la necesidad de transporte es indispensable; las personas requieren moverse por diferentes puntos críticos, donde muchas veces la locomoción terrestre no da abasto con la demanda de transporte. Por este motivo, una red de metro se vuelve un satisfactor primordial a la hora de abordar esta problemática de forma eficiente.

El presente laboratorio plantea el problema de la demanda de movilidad urbana y propone emplear la solución de un sistema de administración de una red de metro, mediante un programa en el lenguaje PROLOG bajo el paradigma lógico 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 dificultades que 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 conflictos de sobrepoblación y crisis migratoria, lo que produce un crecimiento exponencial de la cantidad de habitantes y, por ende, una gran cantidad de autos en las calles.

La locomoción tradicional no abarca la 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 (Departamento de Ingeniería Informática [DIINF], 2024b).

4.- Recursión: Se refiere a una forma de definir procesos, de modo que un proceso es recursivo si se contiene a él mismo como subproceso (Moya, 1972). Es un método de resolución de problemas donde la solución depende de soluciones más pequeñas o parciales del mismo problema (DIINF, 2024c).

El paradigma lógico es un paradigma de programación de modo declarativo, esto quiere decir que se basa en el hecho de que un programa implementa una relación antes que una correspondencia o asignación (Ruíz, 2001). Utiliza mecanismos básicos como la unificación, el backtracking automático y las estructuras de datos basadas en árboles (DIINF, 2024a). Un programa en el paradigma lógico se describe como una base de conocimientos, la cual debe ser cargada en un intérprete donde mediante comandos se realicen consultas sobre esta.

El dialecto más popular de este paradigma es PROLOG, dado que “es un lenguaje de programación especialmente indicado para modelar problemas que impliquen objetos y las relaciones entre ellos . . . La sintaxis del lenguaje incluye la declaración de hechos, preguntas y reglas” (Lobo, Aparicio, & Monferrer, 2001). Según el material de clases de (DIINF, 2024a), estos conceptos se reducen a tipos de cláusulas que definen la relación entre términos; donde los hechos son siempre verdad, y la veracidad de las reglas depende de su conjunción de objetivos, es decir, que todas sus metas se cumplan. La estructura de los hechos es “nombreRelacion(termino1, termino2, ..., terminoN)”; mientras que la estructura de las reglas es “nombreConsecuente(termino1, termino2, ..., terminoN):-antecedente1(...), ..., antecedenteM(...).”.

PROLOG posee tipos de datos como los átomos, que son el dato constante básico y comienzan con minúscula; los números, enteros o reales; las variables, que representa un dato sin un valor aun establecido y comienzan con mayúsculas o guion bajo; y las listas, que son estructuras que permiten almacenar una cantidad variable de elementos heterogéneos (DIINF, 2024a).

Las consultas que se realizan sobre la base de conocimientos constan de respuestas true o false, o de valores que responden a la unificación de variables con hechos y reglas válidas encontradas.

“Para obtener la o las respuestas, PROLOG recorre la base de datos hasta encontrar el primer hecho que coincide con el nombre de la relación y su aridad y con los argumentos que no son variables” (Lobo et al., 2001), así entrega respuestas true o false; sin embargo, al realizar consultas sobre variables, de modo similar se recorre la base y se va unificando las variables con los datos que las satisfacen las premisas; en el caso de encontrar

solución a una consulta, y que se requiera más repuestas, PROLOG mediante backtracking “vuelve atrás” a explorar más alternativas. “Para buscar las soluciones a una conjunción de objetivos, PROLOG establece una marca de posición para cada objetivo, y recorre toda la base de datos en búsqueda de cada objetivo” (Lobo et al., 2001), esto se refiere a las consultas sobre reglas y el uso de variables.

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 PROLOG, bajo el paradigma lógico declarativo. Esto quiere decir que se debe crear un programa que, mediante predicados, 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 funcionalidades que puedan operar la red de metro; y después en una consola externa, mediante comandos, se puedan realizar consultas sobre la base de conocimiento del sistema de metro.

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.

Para la construcción de estructuras se utilizaron principalmente listas, por ejemplo, un elemento del TDA station es una lista de 4 elementos, de la forma [ID station, nombre station, tipo de station, tiempo de parada en segundos]; en el Anexo 1 se exponen todas las estructuras de los TDAs.

La creación de estos tipos de datos abstractos no conlleva grandes dificultades si se siguen los pasos de implementación. No obstante, la definición de las otras operaciones con los TDAs, requiere mayor trabajo. En el Anexo 2 se presenta una tabla con estas funcionalidades y la explicación de sus procedimientos.

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 la naturaleza del lenguaje PROLOG, su relación con la programación declarativa e inexistencia de retornos; los TDAs creados utilizan listas como estructura de datos en el último término de cada cláusula correspondiente. A continuación, se mencionan los tipos de datos abstractos usados, y en el Anexo 1 se expone la estructura sobre cómo están constituidos: Station, Section, Line, Pcar, Train, Driver, Hora y Subway.

Algunos elementos importantes para destacar son que para que un tren sea válido debe cumplir con que sus carros extremos sean del tipo terminal y que sus carros internos sean del tipo central, dado que 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, pudiendo también ser del tipo mantención sólo si la estación anterior es 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 al realizar consultas, puede producir un error de ejecución. Como elementos de programación, se utilizó la recursión tanto natural como de cola para trabajar la construcción y corrimiento de listas. Procurando siempre respetar los principios de la programación lógica.

5. CONSIDERACIONES DE IMPLEMENTACIÓN

5.1. ESTRUCTURA DEL PROYECTO

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

5.2. BIBLIOTECAS EMPLEADAS

Fue utilizada sólo la biblioteca primitiva del dialecto. En particular, “The SWI-Prolog library”.

5.3. INTERPRETE USADO

El programa está desarrollado totalmente en SWI-Prolog (implementación del lenguaje PROLOG), y escrito en el editor de código fuente Visual Studio Code 1.89.1. Como interprete, SWI-Prolog 9.2.4 se utilizó para cargar la base de conocimientos y realizar consultas.

6. INSTRUCCIONES DE USO

6.1. PASOS GENERALES

- Instalar SWI-Prolog 8.4 o superior en el computador.
- En caso de no poseerla, descargar la carpeta “Lab1_Espinoza_Barria_212788287”.
- Abrir la carpeta y verificar la existencia 9 archivos “.pl” necesarios para la ejecución (ver Anexo 3).
- Abrir el software SWI-Prolog y consultar el archivo “main_212788287_EspinozaBarria.pl” como base de conocimientos (ver Anexo 4, 5 y 6).
- Realizar las consultas en la consola de SWI-Prolog y obtener resultados (ver Anexo 7).

Opcional: Es posible realizar las consultas de un archivo de pruebas existente en la carpeta principal. Para esto se debe copiar todo el texto del archivo “pruebas_212788287_EspinozaBarria.txt”, pegarlo en la consola de SWI-Prolog con la base ya cargada, y así obtener los resultados.

6.2. EJEMPLOS

Se definen 3 ejemplos de ejecución de funcionalidades: crear estación (ver Anexo 7), crear sección (ver Anexo 8) y crear línea (ver Anexo 9).

6.3. RESULTADOS ESPERADOS

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

6.4. POSIBLES ERRORES

Considerando que siempre se ingresen parámetros validos correspondiente a los dominios de los predicados. Sólo puede haber errores en dos funcionalidades: en la 24, al consultar la ubicación de un tren que tiene más de un recorrido, el predicado no fallará pero considerará sólo la primera asignación del tren a un horario de recorrido; y en la funcionalidad 25, al solicitar el recorrido de un tren que tiene más de un trayecto definido anteriormente, de forma similar al caso anterior, sólo considera la primera asignación tren-recorrido.

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, las 25 funcionalidades se ejecutan correctamente, esto considerando que los elementos de entrada pertenezcan a los conjuntos de dominio de cada predicado. Además, en el Anexo 11 se presenta la tabla autoevaluación de los requerimientos no funcionales.

8. CONCLUSIONES

En este segundo laboratorio del proyecto semestral, se logró implementar 100% de requerimientos funcionales solicitados, con lo que estoy bastante satisfecho considerando que en el laboratorio anterior no fue posible. También, en comparación que el informe 1, se corrigió la introducción, se explicaron conceptos claves del paradigma correspondiente, se mejoró al análisis del problema y utilizaron más referencias dentro del texto.

Elementos a destacar del paradigma lógico son su concepción de él “qué” por sobre él “cómo” se desarrolla el código, lo que facilita que cualquier persona pueda ejecutarlo sin mayores inconvenientes. Además de que los predicados son fáciles de leer y entender, ya que las reglas sólo requieren la conjunción de objetivos y los hechos son verdades absolutas.

Como elementos negativos, me fue complicado trabajar con SWI-Prolog por la existencia de errores de ejecución para los que aún no encuentro explicación; a veces, al realizar consultas, no se encontraban predicados que estaban bien definidos, y para solucionarlo tenía que volver a escribir la cláusula exacta.

A modo personal, siempre es difícil cambiar el paradigma y entrar a un mundo nuevo del cual no se está familiarizado. Pero sin duda, ha sido una experiencia enriquecedora desarrollar conocimientos en un paradigma tan distinto, ayudándome a salir de mi zona de confort.

Como informáticos y programadores, debemos conocer los diferentes paradigmas y modos de 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 desde una perspectiva puede ser fácil desde otro punto de vista.

REFERENCIAS

- Departamento de Ingeniería Informática [DIINF]. (2024a). Paradigma Lógico - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.
- Departamento de Ingeniería Informática [DIINF]. (2024b). Tipo de Dato Abstracto (TDA) - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.
- Departamento de Ingeniería Informática [DIINF]. (2024c). Programación Funcional - Material de Clases. *Paradigmas de Programación*. Santiago, Chile: USACH.
- Kuhn, T. S. (1962). *La estructura de las revoluciones científicas*.
- Lobo, F. T., Aparicio, J. P., & Monferrer, M. T. (2001). *El Lenguaje de Programación PROLOG*. Castellón de la Plana, España.
- Moya, E. C. (1972). *Recursive techniques in programming*, de DW Barron. Teorema: Revista internacional de filosofía.
- Ruíz, E. (2001). Lenguajes de programación: conceptos y paradigmas. *Industrial Data*, 71-74.
- 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
Station	Estación de metro.	Lista (id – nombre – type station – tiempo en segundos de parada por tren). Donde type station es un string que puede ser “r”, “m”, “c” o “t”.	Crear station.
Section	Sección entre estaciones.	Lista (TDA station 1 – TDA station 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 – 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.
Hora	Horario del día	String de 8 caracteres con la siguiente estructura: “HH:MM:SS”; donde HH es la hora del día, MM los minutos y SS los segundos.	Crear hora, transformar horario a segundos, ver diferencia de horarios en segundos, selectores.

Tabla: Especificación TDAs usados.

2.

Nombre Funcionalidad	Procedimientos
lineLength	Recorre una lista de Sections de una Line, acumulando datos de la cantidad de estaciones, distancia y costo en cada llamada recursiva hasta llegar a la lista vacía.
lineSectionLength	Similar a la anterior, pero primero se reduce la lista de Sections entre dos Stations específicas buscadas por su nombre.
lineAddSection	Antes de agregar una nueva Section a la Line, se verifica que no esté repetida y que sea consecutiva con la anterior.
isLine	Analiza la validez de las Lines circulares o normales, considerando la compatibilidad de las estaciones extremas.
trainAddCar	Recorre la lista de Pcars para insertar uno nuevo en una posición determinada.
trainRemoveCar	Recorre la lista de carros para eliminar uno en una posición determinada.
isTrain	Verifica la validez de un Train, considerando los tipos de carros y su compatibilidad de modelos.
trainCapacity	Recorre la lista de Pcars acumulando la capacidad de pasajeros de cada uno.
subwayAddTrain, subwayAddLine, subwayAddDriver	Verifican que los elementos a agregar no existan previamente en el Subway y luego los ingresan.
subwayToString	Recorre todos los elementos del Subway y los añade a un string en un formato legible.
subwaySetStationStoptime	Recorre todas las Stations de todas las Lines para encontrar una Station específica y cambiar su tiempo de parada.
subwayAssignTrainToLine	Busca un Train y una Line mediante la ID, verifica su compatibilidad de ambos y asigna el Train a la Line.
subwayAssignDriverToTrain	Similar al anterior, pero se considera la compatibilidad del Driver con el Train y la Line asignada con las Stations de inicio y fin.
whereIsTrain	Encuentra el recorrido de un Train, suma los tiempos de parada por Station y calcula el tiempo de viaje entre Sections dada la rapidez media del Tren; para determinar la última Station donde estuvo el Train en un horario específico, considerando ambos sentidos.
subwayTrainPath	Similar al anterior, pero recorre las Sections considerando las Stations que faltan por recorrer en un horario determinado.

Tabla: Explicación de procedimientos de requerimientos funcionales.

3.

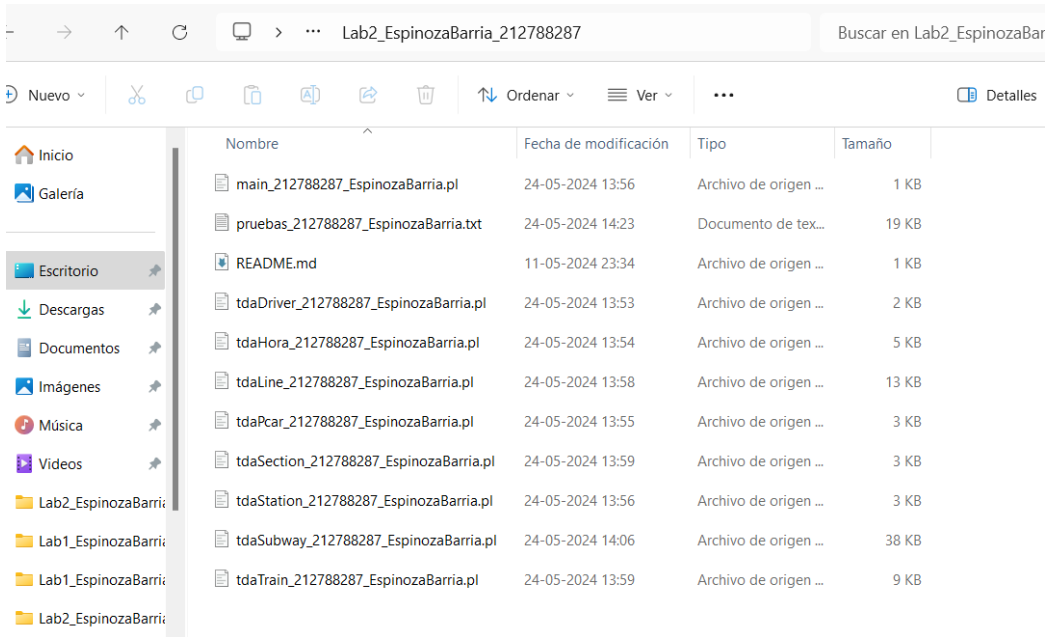


Imagen: Carpeta contenedora de los archivos.

4.

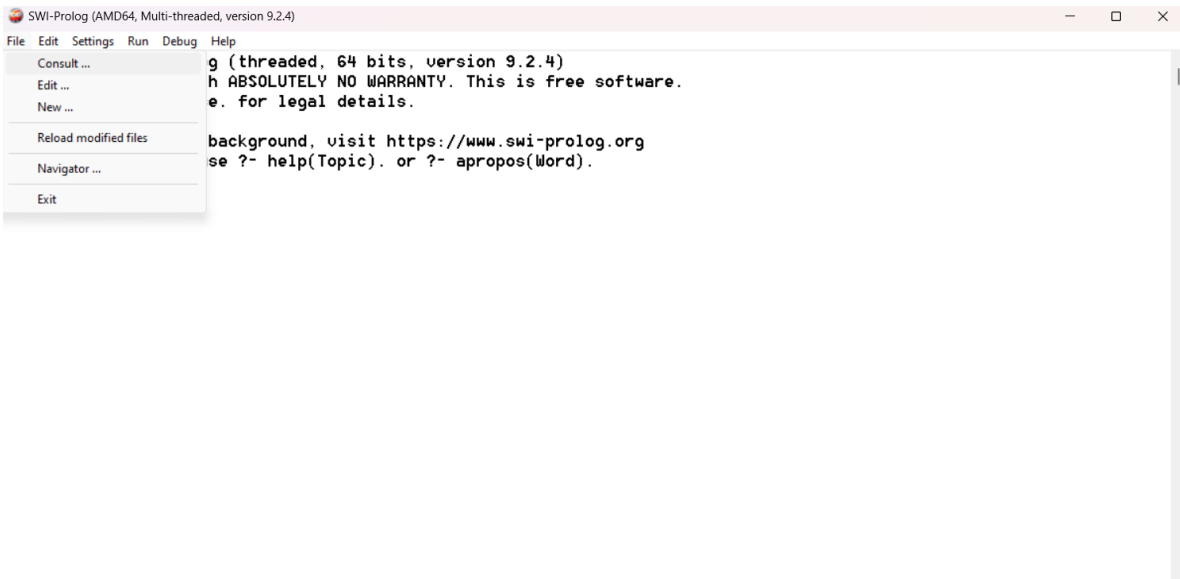


Imagen: Botón consultar base de conocimiento en SWI-Prolog.

5.

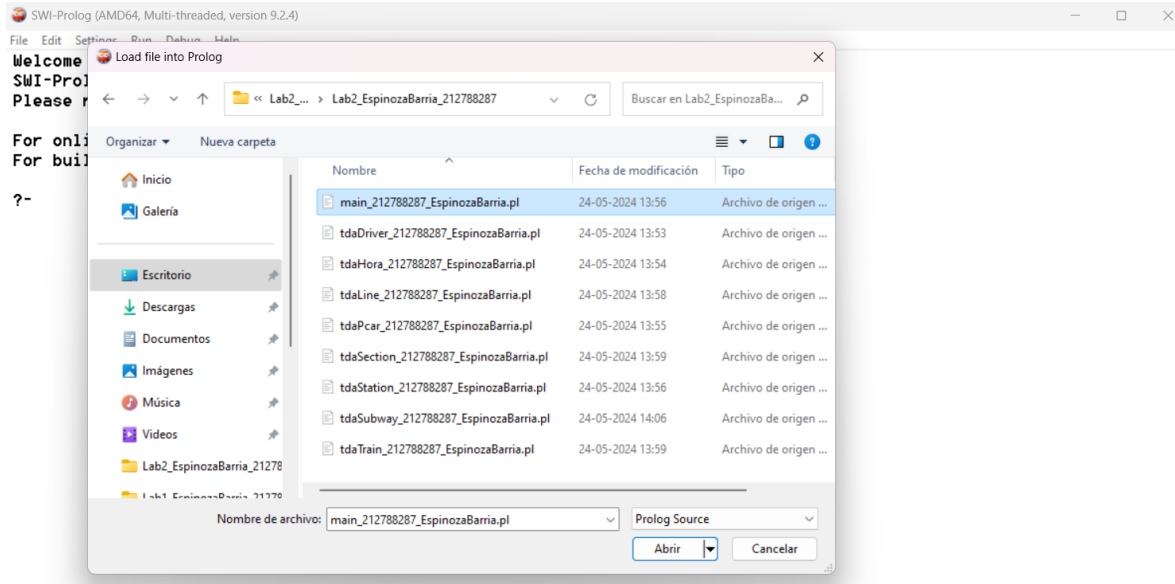


Imagen: Archivo “main_212788287_EspinozaBarria.pl” para cargar en SWI-Prolog.

6.

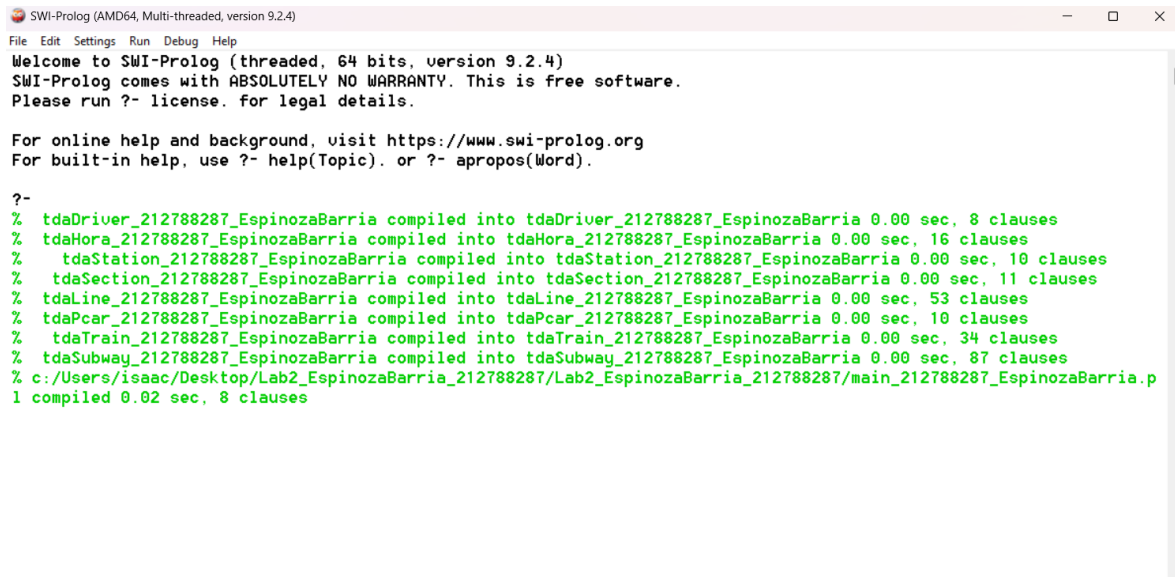
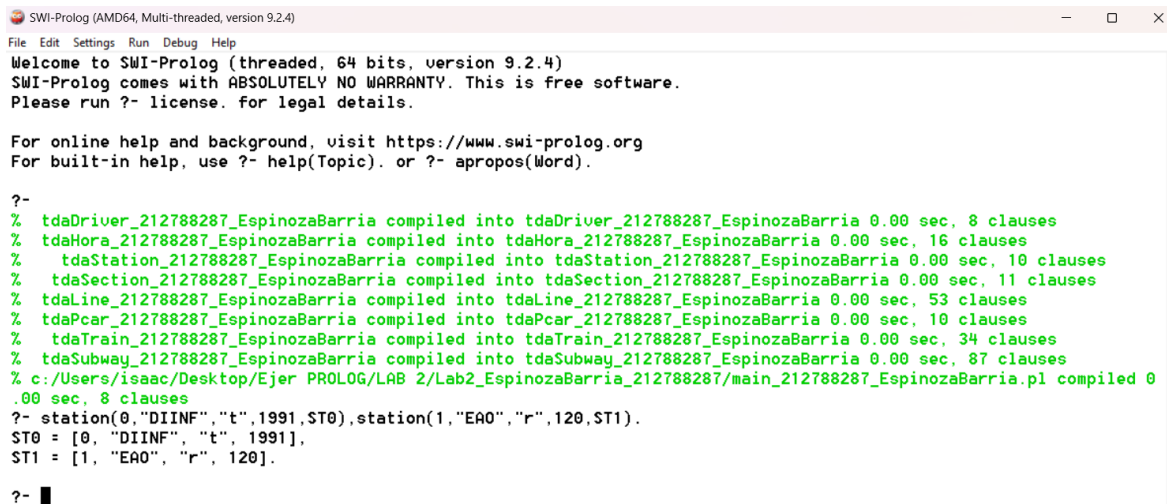


Imagen: Base de conocimiento cargada en SWI-Prolog.

7.



```

SWI-Prolog (AMD64, Multi-threaded, version 9.2.4)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

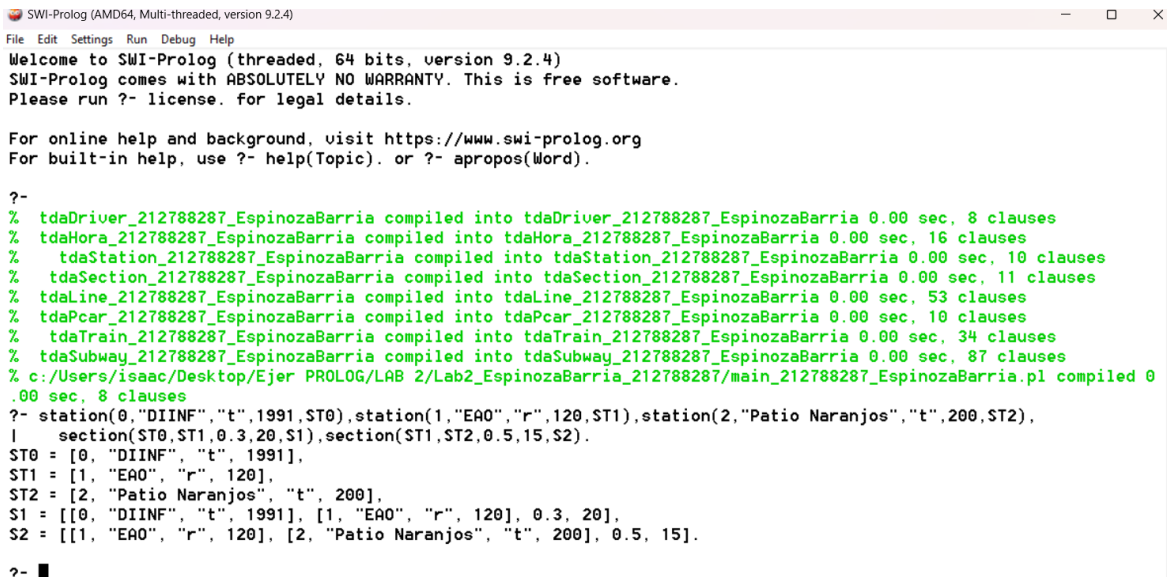
?-
% tdaDriver_212788287_EspinozaBarria compiled into tdaDriver_212788287_EspinozaBarria 0.00 sec, 8 clauses
% tdaHora_212788287_EspinozaBarria compiled into tdaHora_212788287_EspinozaBarria 0.00 sec, 16 clauses
% tdaStation_212788287_EspinozaBarria compiled into tdaStation_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaSection_212788287_EspinozaBarria compiled into tdaSection_212788287_EspinozaBarria 0.00 sec, 11 clauses
% tdaLine_212788287_EspinozaBarria compiled into tdaLine_212788287_EspinozaBarria 0.00 sec, 53 clauses
% tdaPcar_212788287_EspinozaBarria compiled into tdaPcar_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaTrain_212788287_EspinozaBarria compiled into tdaTrain_212788287_EspinozaBarria 0.00 sec, 34 clauses
% tdaSubway_212788287_EspinozaBarria compiled into tdaSubway_212788287_EspinozaBarria 0.00 sec, 87 clauses
% c:/Users/isaac/Desktop/Ejer PROLOG/LAB 2/Lab2_EspinozaBarria_212788287/main_212788287_EspinozaBarria.pl compiled 0
.00 sec, 8 clauses
?- station(0,"DIINF","t",1991,ST0),station(1,"EA0","r",120,ST1).
ST0 = [0, "DIINF", "t", 1991].
ST1 = [1, "EA0", "r", 120].

?-

```

Imagen: Creación de dos estaciones en la consola de SWI-Prolog.

8.



```

SWI-Prolog (AMD64, Multi-threaded, version 9.2.4)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.2.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% tdaDriver_212788287_EspinozaBarria compiled into tdaDriver_212788287_EspinozaBarria 0.00 sec, 8 clauses
% tdaHora_212788287_EspinozaBarria compiled into tdaHora_212788287_EspinozaBarria 0.00 sec, 16 clauses
% tdaStation_212788287_EspinozaBarria compiled into tdaStation_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaSection_212788287_EspinozaBarria compiled into tdaSection_212788287_EspinozaBarria 0.00 sec, 11 clauses
% tdaLine_212788287_EspinozaBarria compiled into tdaLine_212788287_EspinozaBarria 0.00 sec, 53 clauses
% tdaPcar_212788287_EspinozaBarria compiled into tdaPcar_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaTrain_212788287_EspinozaBarria compiled into tdaTrain_212788287_EspinozaBarria 0.00 sec, 34 clauses
% tdaSubway_212788287_EspinozaBarria compiled into tdaSubway_212788287_EspinozaBarria 0.00 sec, 87 clauses
% c:/Users/isaac/Desktop/Ejer PROLOG/LAB 2/Lab2_EspinozaBarria_212788287/main_212788287_EspinozaBarria.pl compiled 0
.00 sec, 8 clauses
?- station(0,"DIINF","t",1991,ST0),station(1,"EA0","r",120,ST1),station(2,"Patio Naranjos","t",200,ST2),
| section(ST0,ST1,0.3,20,S1),section(ST1,ST2,0.5,15,S2).
ST0 = [0, "DIINF", "t", 1991].
ST1 = [1, "EA0", "r", 120].
ST2 = [2, "Patio Naranjos", "t", 200].
S1 = [[0, "DIINF", "t", 1991], [1, "EA0", "r", 120], 0.3, 20].
S2 = [[1, "EA0", "r", 120], [2, "Patio Naranjos", "t", 200], 0.5, 15].

?-

```

Imagen: Creación de dos secciones en la consola de SWI-Prolog.

9.

```

SWI-Prolog (AMD64, Multi-threaded, version 9.2.4)
File Edit Settings Run Debug Help

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% tdaDriver_212788287_EspinozaBarria compiled into tdaDriver_212788287_EspinozaBarria 0.00 sec, 8 clauses
% tdaHora_212788287_EspinozaBarria compiled into tdaHora_212788287_EspinozaBarria 0.00 sec, 16 clauses
% tdaStation_212788287_EspinozaBarria compiled into tdaStation_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaSection_212788287_EspinozaBarria compiled into tdaSection_212788287_EspinozaBarria 0.00 sec, 11 clauses
% tdaLine_212788287_EspinozaBarria compiled into tdaLine_212788287_EspinozaBarria 0.00 sec, 53 clauses
% tdaPcar_212788287_EspinozaBarria compiled into tdaPcar_212788287_EspinozaBarria 0.00 sec, 10 clauses
% tdaTrain_212788287_EspinozaBarria compiled into tdaTrain_212788287_EspinozaBarria 0.00 sec, 34 clauses
% tdaSubway_212788287_EspinozaBarria compiled into tdaSubway_212788287_EspinozaBarria 0.02 sec, 87 clauses
% c:/Users/isaac/Desktop/Ejer PROLOG/LAB 2/Lab2_EspinozaBarria_212788287/main_212788287_EspinozaBarria.pl compiled 0
.02 sec, 8 clauses
?- station(0,"DIINF","t",1991,ST0),station(1,"EAO","r",120,ST1),station(2,"Patio Naranjos","t",200,ST2),
section(ST0,ST1,0.3,20,S1),section(ST1,ST2,0.5,15,S2),
! line(1,"Linea 1","UIC 60 ASCE",[S1,S2],L1).
ST0 = [0,"DIINF","t",1991],
ST1 = [1,"EAO","r",120],
ST2 = [2,"Patio Naranjos","t",200],
S1 = [[0,"DIINF","t",1991],[1,"EAO","r",120],0.3,20],
S2 = [[1,"EAO","r",120],[2,"Patio Naranjos","t",200],0.5,15],
L1 = [1,"Linea 1","UIC 60 ASCE",[0,"DIINF","t",1991],1,"EAO",120],0.3,20,[1,"EAO",120],2,200],0.5,15],...]]].
?-

```

Imagen: Creación de una línea en la consola de SWI-Prolog.

Requerimientos	Grado de Implementación	Pruebas realizadas	% Pruebas exitosas	% Pruebas fracasadas	Anotación
TDAs	1	9 TDAs creados exitosamente.	100	0	
station	1	(+10) casos.	100	0	
section	1	(+10) casos.	100	0	
line	1	(3) casos exitosos agregando secciones al definir, y (3) casos exitosos agregando secciones después.	100	0	Crea estructuras line pero no necesariamente válidas, esto se comprueba en isLine.
lineLength	1	(+5) casos.	100	0	
lineSectionLength	1	(+5) casos.	100	0	
lineAddSection	1	(+15) casos exitosos, (+10) casos exitosos de secciones repetidas	100	0	
isLine	1	(+5) casos exitosos con líneas válidas e inválidas.	100	0	Comprueba lines del tipo regular y circular, donde se recorran todas las estaciones, y verifica estaciones extremas.
pcar	1	(+15) casos exitosos.	100	0	
train	1	(5) casos exitosos.	100	0	Crea estructuras train pero no necesariamente válidas, esto se analiza en isTrain.
trainAddCar	1	(+10) casos exitosos, (3) casos exitosos de carros incompatibles.	100	0	Es false si se intenta agregar carros incompatibles de modelo.
trainRemoveCar	1	(+10) casos exitosos.	100	0	Es false si se ingresa una posición inválida.
isTrain	1	(+10) casos exitosos, donde los trenes eran válidos e inválidos considerando la estructura mínima tr-tr.	100	0	Comprueba compatibilidad carros y de tipo de carro (terminales y centrales).
trainCapacity	1	(+10) casos exitosos.	100	0	
driver	1	(5) casos exitosos.	100	0	
subway	1	(4) casos exitosos.	100	0	
subwayAddTrain	1	(+10) casos exitosos.	100	0	Es false si se intenta agrega un tren repetido.

subwayAddLine	1	(+10) casos exitosos.	100	0	Es false si se intenta agrega una línea repetida.
subwayAddDriver	1	(+10) casos exitosos.	100	0	Es false si se intenta agregar un conductor repetido.
subwayToString	1	(+ 3) casos exitosos.	100	0	
subwaySetStationStoptime	1	(+5) casos exitosos.	100	0	
subwayAssignTrainToLine	1	(+10) casos, (5) casos exitosos con trenes repetidos y/o incompatibles con el tipo de riel.	100	0	
subwayAssignDriverToTrain	1	(+10) casos exitosos, considerando compatibilidad de conductor y tren.	100	0	No hay condiciones que lo impidan asignar un tren en un horario que ya tiene de recorrido
whereIsTrain	1	(+10) casos exitosos, considerando recorridos en ambos sentidos de dirección.	100	0	En caso de que un tren tenga más de un recorrido asignado, la respuesta a la cláusula siempre considera la primera asignación tren-trayecto.
subwayTrainPath	1	(+10) casos exitosos, considerando recorridos en ambos sentidos de dirección.	100	0	En caso de que un tren tenga más de un recorrido asignado, la respuesta a la cláusula siempre considera la primera asignación tren-trayecto.

Tabla: Autoevaluación requerimientos funcionales.

11.-

Requerimientos No Funcionales	Grado de Implementación	Anotaciones
Autoevaluación	1	Está presente en este documento como también en un archivo txt presente el zip contenedor del laboratorio.
Lenguaje	1	PROLOG.
Versión	1	Se utilizó la versión 9.2.4, que es superior a la 8.4 dada como la versión mínima.
Standard	1	Se usaron predicados estándar de SWI-Prolog
Documentación	1	Todos los predicados están correctamente documentados. Indican dominio, recorrido y lo que hacen. Además, al inicio de cada archivo .pl, existe la documentación general de toda la base de conocimientos.
Organización	1	El laboratorio consta con cada TDA en un archivo aparte, donde se incluyen los requerimientos funcionales correspondientes a cada TDA.
Historial	1	Se utiliza GitHub para guardar versiones del avance del proyecto, donde se hicieron más de 10 commits desde hace más de 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.

Tabla: Autoevaluación requerimientos no funcionales.