UNIVERSIDAD DE SANTIAGO DE CHILE

Facultad de Ingeniería
Departamento de Ingeniería Informática

Informe Laboratorio 2 - Paradigma Lógico

Paradigmas de Programación

Nombre: Felipe Cubillos Arredondo
Profesor: Gonzalo Martinez

Sección: B-2
Fecha: 27/05/2024

Índice

Figura 1: Logo De SWI-Prolog

1. Introducción

1.1. Descripción del Problema

Nuestro trabajo, cómo el Laboratorio anterior, consiste en representar un sistema de redes de metro, siendo una red de metro, por ejemplo, el famoso Metro de Santiago. La red está conformada por una o más líneas, estas conformadas por secciones, que determinan el costo y distancia del trayecto. Las secciones se definen cómo el tramo entre dos estaciones adyacentes. La complejidad de esto, yace en **cómo podemos entregar al usuario de este sistema informático las consultas deseadas** bajo el uso del Paradigma Lógico, cual será descrito y desarrollado en el siguiente ítem.

1.2. Descripción del Paradigma

Paradigma Lógico: El paradigma lógico en grandes rasgos se diferencia del imperativo, funcional y orientado a objetos gracias a su enfoque en el que se necesita y no en la implementación en cuestión; osease, este se destaca en su uso de sentencias y reglas lógicas, acercándose más al lenguaje humano. **No nos preocupamos tanto del ¿cómo? Si no mas bien del ¿qúe?.** La programación lógica es de tipo declarativo, por lo que a diferencia del paradigma funcional, este tiene manejo de variables.

> **Motivación:** *"Programming systems in this way has multiple benefits. In writing or reading or modifying logic programs, programmers can get by with little or no knowledge of the capabilities and limitations of the systems executing those programs. As a result, logic programs are easier to produce than traditional programs; they are easier to understand; and they are easier to modify. Logic programs are more versatile than traditional programs - they can be used in multiple ways for multiple purposes without modification. And they are more amenable to programmatic analysis and optimization."* [1]

Algunos conceptos claves que debemos de tomar en cuenta son los siguientes:

- Átomos: Se denominan átomos por que son cosas que se asumen verdad, es la únidad más pequeña en el archivo de sentencias.
- Variables: Las variables en este paradigma se inician con una letra mayúscula, por ejemplo `Train, Station, Section`, etc... Además otro tipo de variable clave son las **Variables Anónimas**, cuyo propósito es al momento de hacerse una consulta y no nos interesa uno de los parámetros de la cláusula se rellena con el _.
- Clausulas de Horn (Reglas): Análogo a las funciones, son una serie de reglas asignadas respecto de ciertos parámetros (átomos). Estas se denominan con primera letra minúscula, por ejemplo: `lineLength, lineSectionLength, trainCapacity, subwayAddDriver`, etc...
- Recursión: Al igual que el laboratorio anterior, se debe de utilizar la recursión a falta de métodos imperativos. Apoyado enormemente de una técnica de diseño de algoritmos ya vista conocida cómo **Backtracking** cual busca todas las soluciones posibles dado un problema.
- Manejo de Listas: Al igual que los lenguajes previamente vistos, uno de los tipos de datos más cruciales son las listas, en este lenguaje se utiliza junto a la recursión y el operador corte `[H|T]`.

Lenguaje SWI-Prolog: Abreviación de **PROgrammation en LOGique**, ideado en Francia en los 70's, busca emplear los conceptos previamente mostrados en un lenguaje de programación. Para este laboratorio elegiremos la implementación de **SWI-Prolog**, ideada en 1987 en la Universidad de Amsterdam. Esta implementación se utilizará por su facilidad pedagógica y facilidad, independientemente de estos factores, es un lenguaje capáz de abordar la problemática previamente estipulada.

> SWI-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real world applications. SWI-Prolog is widely used in research and education as well as commercial applications. Join over a million users who have downloaded SWI-Prolog. [2]

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. [3]

Cabe destacar, que se utilizarán capas de los TDA's, osease 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 qusiera, por algún motivo, cambiar el nombre del avión tendría que emplear el método `set-name`. Para el caso de ProLog, cómo tenemos la implementación de TDA por listas, sabemos que los getters y setters se obtienen prácticamente del mismo constructor.

En el fondo, **lo que se busca lograr es ir armando desde lo más pequeño**, osease 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

Al igual que en laboratorio anterior, se busca resolver el problema desde el punto de vista de la ingeniería inversa puesto que ya conocemos, o podemos calcular facilmente, los resultados que buscamos. Por lo que principalmente nos vamos a enfocar en la representación de los TDA's y los requisitos funcionales pedidos.

Cómo vimos en la descripción del lenguaje, tenemos a nuestra disposición los átomos, relaciones y predicados. En este caso, al igual que el laboratorio anterior, tenemos representación y manejo de listas para los TDA's, por lo cúal se vuelve de suma importancía el operador corte y las variables anónimas, osease, dada una representación de lista, en caso de que se necesite que una variable quede cómo "vacía"se representa con _, por ejemplo: `train(_,Nombre,_,_, Train)`, que devolvería dado un Train, este devuelve su nombre.

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 StopTime. 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.

2.3. Aspectos de Implementación

Un aspecto de implementación importante es, cómo fué descrito con anterioridad, `tda_subway.pl` es el archivo que posee todas las dependencias y TDA's anteriormente descritos, por lo que **este actuará de main**. Dicho esto, debemos de utilizar la representación de listas, cómo fué mencionado y utilizado en el lab anterior, así distribuimos en cada archivo un respectivo TDA, osease tendríamos los siguientes archivos:

- `tda_station_21461391_CubillosArredondo.pl`: Representa las estaciones
- `tda_section_21461391_CubillosArredondo.pl`: Representa las secciones
- `tda_line_21461391_CubillosArredondo.pl`: Representa las líneas y posee requirimientos funcionales

- `tda_pcar_21461391_CubillosArredondo.pl`: Representa los carros
- `tda_train_21461391_CubillosArredondo.pl`: Representa los trenes y posee requerimientos funcionales
- `tda_driver_21461391_CubillosArredondo.pl`: Representa a los conductores
- `tda_subway_21461391_CubillosArredondo.pl`: Representa un sistema completo de metro y posee requerimientos funcionales.

  Es importante destacar que el script de pruebas se encuentra en un archivo txt, este contiene todas las consultas pedidas poniendo a prueba los requerimientos funcionales. En el apartado de instrucciones de uso. Además se encuentra en el repositorio el script de prueba determinado por el profesor.

Otra consideración importante es el tema de los **getters** es importante mencionar que para esta ocasión **NO SON ESTRICTAMENTE NECESARIOS** ya que las implementaciones de constructores y la naturaleza del lenguaje que facilita la consulta de los átomos pedidos resulta más efectiva que la realización de getters individuales. Por ejemplo, en el caso de querer obtener el ID de alguna estación en vez de construir la claúsula de `getStationId(TargetStation, Id)`, se puede directamente hacer `station(Id,_,_,_,TargetStation).` Acá utilizamos el concepto de las **variables anónimas,** que fueron explicadas en la descripción del paradigma y diseño de la solución.

Ahora, respecto a los requisitos funcionales, la mayoría eran resolvibles con métodos

2.4. Instrucciones de uso

Cómo descrito en la sección anterior, para correr el programa abriendo la consola de SWI-Prolog se debe consultar el archivo `tda_subway.pl` *(File → Consult)*. Hecho esto, se debe copiar y pegar las sentencias propuestas por mí y el profesor encargado del Laboratorio. Hecho esto, se mostrarán todas las consultas. Es importante mencionar que primero se debe consultar el archivo y luego copiar las sentencias, esto debido a que por alguna razón SWI-Prolog deja en el portapapeles el mensaje de bienvenida. **Para dejar el procedimiento lo más entendible posible hay que hacer lo siguiente:**

1. Descomprimir los archivos en una misma carpeta teniendo cuidado de no cambiar el nombre de ningún archivo `.pl`.
2. Una vez abierto swipl (SWI-Prolog), consultar el archivo:
   `tda_subway_21461391_CubillosArredondo.pl` cómo fué descrito anteriormente [4]
3. Una vez mostrados los warning de Singleton Variables, se copia y pega las sentencias guardadas en `pruebas_21461391_CubillosArredondo.txt`, esto se puede hacer utilizando en la parte superior *(Edit → Copy)* [3]
4. Una vez copiado y pegado, Prolog preguntara si se debe corregir a su respectivo TDA, esto debido al uso de `use_module`, por lo que se debe escribir con el teclado `y` dando a entender que sí, se quiere corregir al TDA correspondiente. [2]
5. Hecho esto, se mostrará en pantalla todo lo pedido.

Es importante agregar que para el archivo de consultas dado por el profesor se debe hacer el mismo procedimiento. [6]

2.5. Resultados y Autoevaluación

Una vez mostrados los resultados en pantalla, vemos que se muestra todo hasta el predicado `subwayAddDriver/3` [5] Por lo tanto, los resultados son satisfactorios en su mayoría, ocurre la expcepción en los siguientes requisitos funcionales:

- `lineSectionLength`: Si bien entrega resultados, no son resultados correctos, por más que el resultado concuerde con las metas.

- `isLine`: No fué posible revisar líneas circulares ni ocurrencias repetidas de estaciones.
- `isTrain`: No fué posible revisar ocurrencias repetidas de estaciones.
- `constructores line, train y subway`: No fué posible revisar Id's repetidos.

Más allá de esto, los otros requisitos funcionales son logrados en su mayoría con resultados satisfactorios; en el archivo `autoevaluacion_21461391_CubillosArredondo.txt` se determinan los puntajes por requisito funcional.

3. Conclusiones

Ahora, concluyendo este informe, en lo personal opino que fué sumamente inspirador y formativo en mi carrera de informático, me enseñó que el paradigma lógico puede ser implementado en situaciones actuales y ser competente en un entorno laboral que requiera de un acercamiento más abstracto y acercado a la lengua natural sin necesidad de mecanizar toda la problemática en sus más ínfimos detalles; **el poder que tiene PROLOG es el de saber que es poder satisfacer una consulta sin saber cómo esta es hecha**.
Comparando con el laboratorio anterior me siento satisfecho con lo logrado, ya que se logró un desempeño similar en términos de órden, dedicación y progreso. Para el próximo laboratorio espero hacer aún más y espero tener la misma o mayor disposición para aprender.

3.1. Referencias

1. *A Brief Introduction to Basic Logic Programming, Michael Genesereth Computer Science Department, Stanford University (http://ggp.stanford.edu/notes/blp.html).* [1]
2. *https://www.swi-prolog.org/* [2]
3. *CC30A Algoritmos y Estructuras de Datos: Tipos de datos abstractos. (s. f.). https://users.dcc.uchile.cl/ bebustos/apuntes/cc30a/TDA/*[3]

3.2. Anexos

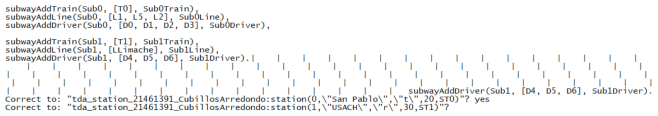

Figura 2: Ocurrencia a momento de consulta, poner y

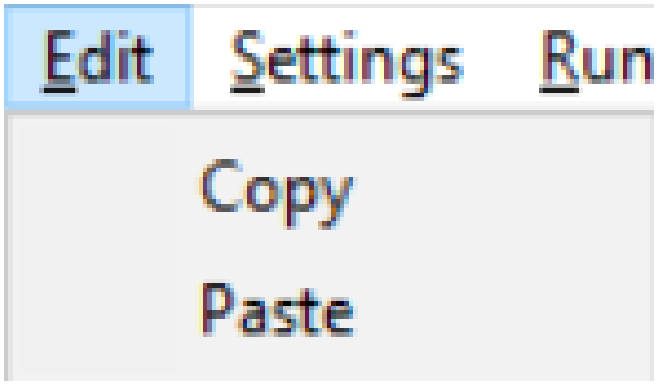

Figura 3: Ejemplo de Edit → Paste

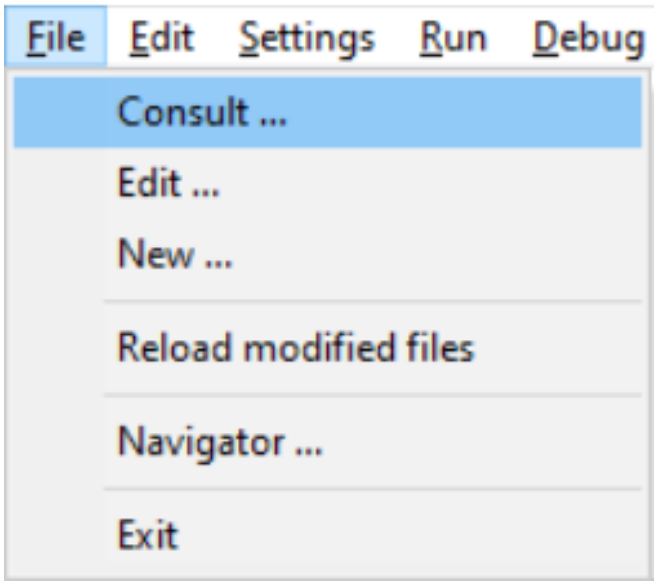

Figura 4: Ejemplo de File → Consult

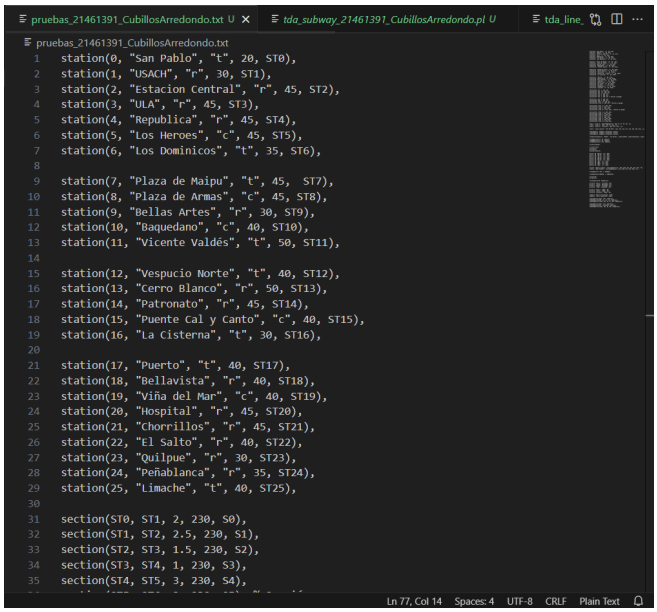

Figura 5: Ejemplo Script de Pruebas

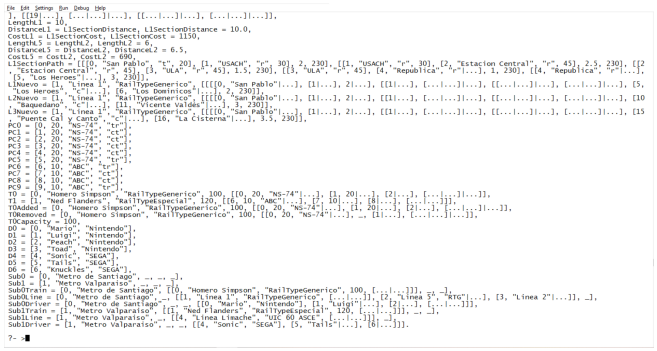

Figura 6: Ejemplo de Consulta