



UNIVERSIDAD
DE SANTIAGO
DE CHILE

PARADIGMAS DE PROGRAMACIÓN.

2023

Laboratorio N°2: Paradigma Lógico usando Prolog.



Nicolas Espina Valenzuela

19.973.961-1

INDICE

- INTRODUCCION
 - Descripción del problema.
 - Descripción de paradigma: ¿Qué es el Paradigma Funcional?
- ANALISIS DEL PROBLEMA
- MODELAMIENTO Y DISEÑO DE SOLUCIÓN
 - Diseño de la solución (¿Qué se realizó?)
 - Aspectos de la implementación
- RESULTADOS Y AUTOEVALUACIÓN
- CONCLUSIONES

INTRODUCCIÓN

Una de las funciones fundamentales de una computadora es, más allá del cálculo operacional, el manejo de memoria para el almacenamiento y borrado de datos. Como es evidente, a nivel usuario es requerida una aplicación software que gestione y organice estos datos con el fin de poder manejarlos, y establecer operaciones sobre ellos. Así nace lo que es llamado un “Sistema de Archivos”.

Dicho sistema gestiona, como una biblioteca, los datos presentes en la memoria de una computadora, haciendo posible su uso.

El presente trabajo consiste en modelar un “Sistema de Archivos” usando el lenguaje de programación “Prolog”, el cual está clasificado como un lenguaje que hace “uso” del “Paradigma Lógico”. Para ello, se realizará una implementación que realice hechos y reglas sobre carpetas y archivos de manera algo superficial, es decir, omitiendo detalles más profundos acerca del funcionamiento de estos sistemas.

Así, es imperativo definir que es en sí un paradigma y más específicamente definir que es el “Paradigma Lógico” para dar contexto al resto del trabajo.

Un “paradigma” es básicamente una “forma de ver el mundo”, una forma de afrontar alguna problemática. En el contexto presente, “Paradigma Lógico” hace referencia establecer reglas lógicas sobre objetos, relacionándolos y realizar consultas sobre ellos.

Como tal, el paradigma está basado en la lógica matemática, y las "operaciones" que se pueden realizar sobre el programa son las llamadas "reglas", las cuales poseen antecedente y consecuente (una implicación lógica).

En Prolog ocurre que el "cuerpo" de una regla (el equivalente al cuerpo de una función) es el antecedente, y el encabezado de la regla es el consecuente.

así, si el cuerpo (Antecedente) de una regla es "verdadero", entonces el encabezado (lo que quiero conseguir) será, por implicación lógica, "verdadero".

Ahora bien, El manejo de "variables" en Prolog, y la construcción del cuerpo de una regla lógica (eq. a cuerpo de función) es muy distinto a los demás paradigmas.

para construir las "sentencias" de una regla, se usa el operador lógico "y" (con el símbolo ","), así, un antecedente se construye a partir de muchas conjunciones lógicas.

Luego, las variables adoptan valor mediante la "unificación": mecanismo de Prolog para asignar valor a una variable para que una regla o meta se cumpla usando "backtracking".

ANÁLISIS DEL PROBLEMA

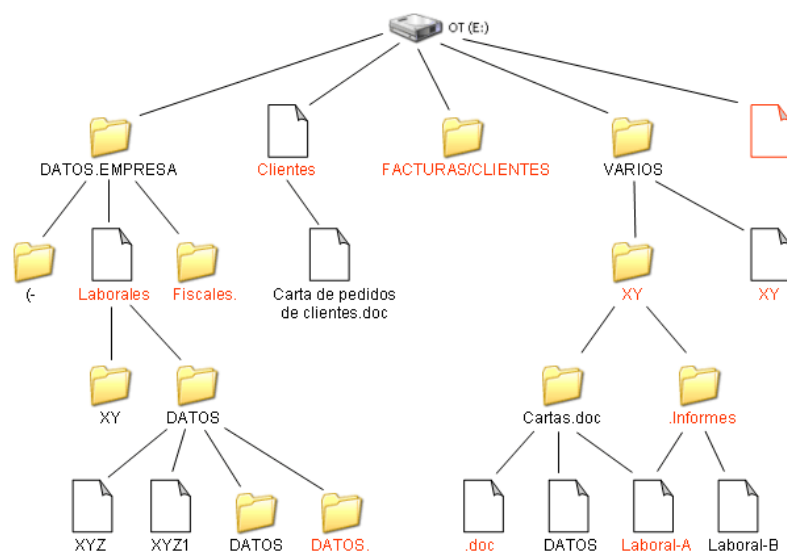
Entrando ya en lo que es el problema en sí, se comienza dando un breve análisis del problema de crear un sistema de archivos.

Para tener una referencia, dicho sistema a crear ha de tener funciones similares a los sistemas de archivos presentes en las computadoras. Como usuario, quizá ha usado por ejemplo la aplicación por defecto en Windows llamada “explorador de archivos”.

Con dicha aplicación usted puede organizar sus documentos, guardar archivos de texto, imágenes, canciones, así como también puede borrar de la computadora estos datos.

También puede crear alguna carpeta con algún nombre a elección, por ejemplo, “fotos del año pasado” y guardar en ella imágenes de las vacaciones, cumpleaños o salidas del año pasado.

A continuación, un ejemplo de organización de un sistema de archivos:



Esta aplicación, como es evidente, tiene una interfaz visual que hace posible el manejo mediante “arrastrés con el ratón”, ventanas visuales, etc. Sin embargo, el sistema de archivos que se creará usando “Prolog” carece de estas cosas, aunque si cumplirá con los aspectos funcionales, es decir, si realizará acciones como copiado, movimiento y borrado de archivos, entre otros.

Ahora, en términos de implementación, como esta trabajará con un lenguaje lógico, todas las funcionalidades del sistema de archivos, tales como borrar, mover, copiar, renombrar, crear, etc. Serán creadas a partir de reglas lógicas y hechos.

Por ejemplo, para realizar el copiado de un archivo o directorio se creará una regla llamada “copy” que recibirá argumentos, los procesará y dará un resultado, mediante unificación, que en este caso será la copia exitosa de un archivo o directorio.

Así, cada acción del sistema será una regla que recibirá al sistema en cuestión y realizará los cambios en el, que pueden ser copiados, movimientos o creaciones de archivos o directorios.

Cada una de estas reglas seguirá un “orden progresivo” siguiendo la forma que se dio para la creación del proyecto, es decir, cada regla n dependerá de que las otras $n-1$ funciones estén ya implementadas, esto debido a la estructura de Prolog y la conjunción de objetivos.

Si se cumple A y B y C entonces implica D.

DISEÑO DE LA SOLUCIÓN

En esta sección se abordará el diseño de la solución, es decir, la forma en cómo se empleó el sistema de archivos usando el lenguaje de programación “Prolog”.

Según lo pedido, y como ya se expresó anteriormente, la idea fundamental para crear el sistema de archivos es “dividir” todas las funcionalidades de este en reglas lógicas que harán uso de la unificación y backtracking.

Es decir, si el sistema realiza la acción de copiar un archivo, ha de crearse una regla que realice dicha actividad. Si el sistema también realiza la acción de mover o crear archivos o directorios, han de crearse reglas individuales para cada acción.

A diferencia de un lenguaje imperativo como C, donde existe un “main” donde se ejecuta el programa escrito de forma secuencial (sus instrucciones), en Prolog no existe un main, y, para el desarrollo de diversas funcionalidades, se realiza la conjunción de las distintas reglas que conforman a las diversas funcionalidades del sistema.

Aquí, es similar a “Racket”, sin embargo, no existe una anidación de reglas.

Ahora bien, para poder modelar los elementos que forman parte del sistema, como lo son las carpetas y archivos, es necesario usar el concepto de TDA (Tipo de dato abstracto) visto en clases.

Gracias a que “Prolog” es un lenguaje débilmente tipado (no es necesario declarar con anterioridad el tipo de dato a usar, como en C), se tiene que la “fabricación” de los

elementos del sistema, e incluso el sistema mismo, es bastante sencilla, ya que el lenguaje usa principalmente “listas” para el modelamiento de los datos.

Así, una carpeta (directorío), se puede definir como una lista de elementos que contenga: nombre, contenido, fecha de creación, etc. Y al ser “Prolog” débilmente tipado, en sí las listas pueden recibir “cualquier cosa”, sin especificar si son números, cadenas, o incluso otras listas.

Igualmente, para el modelamiento de las demás capas de estos TDA, se realizaron reglas que seleccionan un elemento de la lista que representa un TDA (como por ej. Un directorío), y otras reglas que modifiquen estos elementos.

En general, las reglas creadas para el sistema son bastante similares en la construcción, sin embargo, por obviedad, realizan acciones distintas.

Algo en particular en la fabricación de las reglas es que como no existen en sí los saltos condicionales en Prolog (esto a diferencia de Racket), la forma en que una regla evalúa una condición para posteriormente tomar una decisión es crear muchas copias de una regla, la cual cada una hará algo distinto en función de lo que reciba como su dominio.

Por ejemplo, si tengo la regla “systemDel” y esta tiene una “lista de comandos”, entonces cada comando tendrá su copia de la regla que efectúe las operaciones solo para este él.

Luego, para comenzar todo, es necesario establecer como se representa un sistema.

Como todo en general y como se mencionó, el sistema se representa como una “lista” de elementos, los cuales son un nombre, una lista de usuarios, lista unidades físicas o lógicas, lista de rutas, una papelera, entre otros.

Las reglas principales son aquellas que se piden explícitamente en el trabajo, como copy, move, login, register, etc. Todas estas reglas reciben como input principal un TDAsystem (un sistema, una lista) y realizan los cambios sobre este TDA, es decir, el sistema, mediante la conjunción de diversas reglas para luego “unificar” todo lo nuevo con una nueva variable.

Ahora bien, para cada una de estas reglas principales se definieron otras reglas que no son explícitamente pedidas en el trabajo, pero si son indispensables para la elaboración de las principales.

Por ejemplo, para la regla “systemLogin”, la cual inicia sesión de un usuario ya registrado, es necesario crear reglas auxiliares que realicen, por ejemplo, la comprobación de la existencia de un usuario.

Existen múltiples reglas auxiliares para cada regla principal, y que están definidas en el código fuente.

En cuanto a los aspectos de la implementación en términos de lenguaje, se tiene que, debido a las restricciones dadas por el proyecto de no usar algo que efectué los condicionales, solo se usa los predicados que aporta el lenguaje para el manejo de listas, además de otras reglas para realizar operaciones más complejas

Uno de los puntos más importantes de la implementación es el cómo establecer el “movimiento” entre directorios del sistema. borrando, moviendo o copiando archivos o directorios. Aquí, y en contraste con la primera entrega, se definió un nuevo TDA para

representar una ruta mediante una lista de elementos. Así, cada “ruta” posee elementos que permiten establecer una relación entre los diversos componentes del sistema.

Por ejemplo, cada ruta posee un “id” que corresponde a un id de ese elemento en cuestión, y un “idpadre” que establece que elemento lo contiene en el sistema.

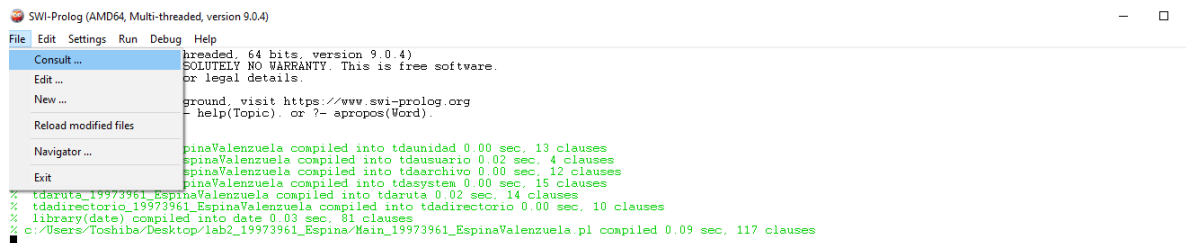
Mediante un selector de “idpadre”, fue posible establecer una conexión entre los elementos que componen el sistema.

INSTRUCCIONES DE USO

Primero que todo, el archivo comprimido que contiene los ficheros de extensión “.pl” ha de descomprimirse completamente en solo una carpeta contenedora.

Para ejecutar como tal el programa, es necesario abrir el archivo

“Main_19973961_EspinaValenzuela.pl” usando el intérprete de Prolog previamente instalado. Así, para poder “ejecutar” el programa, ha de seleccionarse el archivo usando la opción “consult”, como se muestra a continuación:



```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)
File Edit Settings Run Debug Help
Consult ... hreaded, 64 bits, version 9.0.4)
Edit ... SOLUTELY NO WARRANTY. This is free software.
New ... or legal details.
Reload modified files ground, visit https://www.swi-prolog.org
Navigator ... help(Topic), or ?- apropos(Word).
Exit pinaValenzuela compiled into tdaunidad 0.00 sec, 13 clauses
% tdaruta_19973961_EspinaValenzuela compiled into tdausuario 0.02 sec, 4 clauses
% tdaunidad_19973961_EspinaValenzuela compiled into tdaarchivo 0.00 sec, 12 clauses
% tdaarchivo_19973961_EspinaValenzuela compiled into tdausuario 0.00 sec, 15 clauses
% tdausuario_19973961_EspinaValenzuela compiled into tdaunidad 0.02 sec, 14 clauses
% tdaunidad_19973961_EspinaValenzuela compiled into tdaarchivo 0.00 sec, 10 clauses
% tdaarchivo_19973961_EspinaValenzuela compiled into tdausuario 0.03 sec, 81 clauses
% library(date) compiled into date 0.03 sec, 81 clauses
% c:/Users/Toshiba/Desktop/lab2_19973961_Espina/Main_19973961_EspinaValenzuela.pl compiled 0.09 sec, 117 clauses
```

Este archivo contiene la definición de reglas correspondiente al TDAsystem (sistema) y todas las reglas principales y secundarias (como systemDel y las reglas auxiliares de systemDel).

El archivo “Main...” está modulado, todos los demás TDA(en archivos .pl) son llamados por este para poder ejecutar las funcionalidades del sistema.

Para poder ejecutar la simulación del sistema, se siguen básicamente las instrucciones dadas con anterioridad, sin cambios.

Primeramente, ha de crearse un sistema usando el constructor de este definido en “tdasystem_19973961_EspinaValenzuela.pl”, de la siguiente forma:

$$\text{system}(\text{“nombreSystem”}, S1)$$

se crea un sistema llamado “Sistema1”

Luego, todas las demás reglas han de ejecutarse usando el operador de conjunción sobre la variable unificada de alguna regla anterior, por ejemplo:

$$\text{system}(\text{“nombreSystem”}, S1), \text{systemAddDrive}(S1, \text{“C”}, S2).$$

Cada regla principal efectúa cambios en el sistema; estos cambios son representados por un nuevo sistema en otra variable de “salida” mediante la unificación.

En el ejemplo anterior, “system” crea un sistema con nombre “nombreSystem” en una variable S1, luego, “systemAddDrive” toma S1 como argumento, añade una unidad “C” a S1 creando un nuevo sistema (ya que se modificó) el cual “guarda” (unifica) con la variable S2.

Importante es que todas las ejecuciones de las reglas sigan la estructura antes descrita,

Es decir: R1 , R2, R3.... , Rn.

RESULTADOS

Aproximadamente la mitad de las reglas que realizan las actividades del sistema fueron implementadas.

Esto representa, en comparación con la primera entrega, un retroceso evidente.

La dificultad de realizar mas funcionalidades en comparación con Racket radicó principalmente en la naturaleza poco intuitiva de Prolog además de la inexistencia de los condicionales IF y ELSE.

Otra causa fue que se creo otro TDA además de los presentados en la primera entrega; Un TDAruta, que si bien representó de mejor manera lo que una ruta representa (Hijos, conexiones entre directorios y archivos, etc.), también trajo consigo una mayor dificultad en la creación de las reglas que involucran la operación de rutas del sistema.

Por ejemplo, en la primera entrega solo fueron representadas con strings, así las operaciones solo eran sobre strings. En cambio, en Prolog, al definir un TDAruta se provocó que debieron existir mas operaciones para manipular este TDA que, junto con la naturaleza del paradigma, dificultaron en demasía su implementación (no se pudo implementar completamente ni siquiera la funcionalidad COPY”).

CONCLUSIONES

Para finalizar, se considera que el objetivo de esta entrega se cumplió a medias.

Debido a la naturaleza del paradigma lógico, la sintaxis de prolog y la inexistencia del IF/ELSE que si presenta Racket, muchas funcionalidades importantes no fueron implementadas.

Sumado a lo anterior, la creación de un nuevo TDAruta aportó una mejor implementación de esta característica en comparación a la primera entrega, ya que representaba mucho mejor lo que es una ruta: indica el nombre del directorio, su id, sus hijos, y el id del elemento que lo contiene.

Sin embargo, este mismo hecho provocó que la implementación de muchas reglas principales no se logrará debido al aumento de complejidad en las operaciones requeridas para el manejo de rutas en comparación a la primera entrega.

Se espera que para la entrega final usando POO, se logre solucionar estos problemas de modelamiento e implementación.

