Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Informe laboratorio 2 (Lógico - Prolog)

Alumno: Cristóbal Torres Undurraga
Profesor: Roberto González I.
Fecha: 12 de Junio de 2023
Asignatura: Paradigmas de Programación

1. Introducción

El informe consiste en 11 partes: Introducción, Descripción del Problema, Descripción del Paradigma, Análisis del Problema, Diseño de la Solución, Aspectos de la Implementación, Instrucciones de Uso, Resultados, Conclusiones, Referencias y Anexos.

En el siguiente informe se explicará la implementación de un sistema operativo con el paradigma lógico, en específico, el lenguaje de programación Prolog. El paradigma lógico corresponde a un tipo de programación que se basa en una base de conocimientos y consultas a esta.

2. Descripción del Problema

El problema consiste en la simulación de un sistema operativo (Unix-like) simplificado. Debe tener un sistema de archivos (file system), el cual contiene procesos, métodos y reglas. El sistema operativo debe ser capaz de realizar operaciones a través de comandos, los cuales crean, modifican o eliminan elementos del sistema de archivos. Debe tener 5 partes esenciales: sistema, drivers, usuarios, carpetas y archivos. El desafío está en cómo abstraerse del problema y poder organizar el sistema para que se implementen todas las partes y funcionen entre sí por medio de operaciones (register, login, mkdir, add-file, switch-drive, etc).

3. Descripción del Paradigma

El paradigma utilizado en esta instancia es el paradigma Lógico, este se basa en formalismos abstractos, en particular, en la lógica de primer orden. Este es un paradigma que pertenece a la familia declarativa, es decir, se enfoca en resolver un problema, el ***que***, no en ***cómo*** se resuelve. El paradigma se basa en 2 conceptos fundamentales: la base de conocimientos en la cual se pueden declarar hecho sobre términos y sus relaciones o también conocidas como reglas, que se utilizan para definir cómo se acepta una consulta, y las consultas, que son preguntas que se realizan al intérprete de Prolog, el cual se responde según lo existente que se cargó en la base de conocimientos. Solo se puede responder *true* o *false*.

El paradigma lógico, o Prolog para ser más específicos, utiliza ciertas herramientas y estrategias para poder resolver problemas: *Sustitución*, consiste en reemplazar los valores de una consulta hasta que la consulta se haga verdad. *Unificación*, es el proceso por el cual se sustituye variables por términos para intentar igualarlos, mediante un proceso recursivo. *Backtracking automático*, o *vuelta atrás*, es una proceso recursivo, en el cual se prueba con todos los términos hasta lograr la *unificación*. *Estructuras basadas en árboles*, al realizar el proceso de *Backtracking*, se empieza a generar un árbol, que representa la búsqueda de la solución en el espacio. En el caso de que se termine de buscar en el árbol y no se encuentre una solución, se retorna falso. (Flores, 2023)

4. Análisis del Problema

Para crear y generar el sistema operativo y sistema de archivos con el paradigma lógico, se elige como mejor representación usar listas, además del sistema, todos los TDAs se representan por listas. Las listas son un tipo de dato que está incluído de forma nativa en Prolog. Sobre esta lista que será el sistema operativo, hay que realizar ciertas operaciones que modifiquen, agreguen o eliminen elementos dentro del sistema. En el caso de Prolog, se deben realizar las reglas necesarias para moldear el sistema de forma apropiada. En está implementación, se modifican ciertos elementos del sistema principal y posteriormente se rearman como una lista nuevamente. El 1er punto más mencionado en los requisitos funcionales es consultar si es que un elemento ya existe en cierto directorio o en general, como ***add-drive***, ***register***, ***login***, ***switch-drive*** y ***format***. Por ejemplo, si ya existe un archivo con el mismo nombre en el directorio. Esto se puede hacer recursivamente por medio de ***member*** (o buscar, que es como se expresa para buscar cierto tipo de datos en el laboratorio, por ejemplo, ***buscar carpeta***) y se utiliza en varios predicados como ***add-drive***, ***register***, ***login***, ***switch-drive***, ***mkdir***, ***cd***, ***add-file***, ***del copy***, ***move***, ***ren*** y ***format***. Un 2do punto importante es agregar un tipo de dato a las listas, principalmente con las operaciones para crear y a veces modificar elementos del sistema. Para esto se hace uso de ***insertarCola***, que permite agregar un elemento al final de la lista, y como cada tipo de dato está en una lista distinta, se logra obtener un orden, todo de forma recursiva. El 3er punto importante es eliminar un elemento del sistema, o de su lista específica. Lo utilizan predicados como ***del***, ***move*** y ***format***, que eliminan los archivos originales, con la particularidad de que del los mueve a la papelera y ***move*** los mueve a otra ruta. Para este caso, es útil utilizar ***eliminar***, que es un predicado recursivo que en conjunto con ***insertarCola***, rearman la lista sin el elemento indicado.

5. Diseño de la Solución

Para formar y operar dentro del sistema operativo, se optó por usar una representación de listas, como se mencionó anteriormente. El principal es cómo organizar y comunicar entre sí carpetas y archivos. En este caso, se utilizó una lista para representar el sistema, y dentro de esta se encuentran 10 elementos: Las primeras 5 posiciones contienen datos relativos al sistema y lo que se está utilizando (nombre, fecha de creación, letra unidad actual, usuario actual y ruta actual). Las últimas 5 corresponden a la parte del sistema de archivos, todas como listas (unidades, carpetas, archivos, usuarios y papelera).

Las unidades, carpetas, archivos y papelera son listas que contienen listas, siendo cada una de estas un elemento, las cuales poseen un identificador ya sea de ruta y/o nombre. Las unidades tienen una letra, nombre y capacidad. Las carpetas poseen una ruta, creador y otros datos relacionados. Los archivos son una lista dentro de otra, siendo la 1era posición la ruta en que se encuentra, y la lista de al lado contiene los datos respectivos al archivo. Finalmente, la papelera contiene carpetas y archivos como ya se almacenan en sus respectivas listas. [Diagrama 1]

Para saber donde se encuentra cada elemento, se verifican ciertos datos: las unidades usan la letra, las carpetas utilizan la ruta, los archivos la ruta y nombre, y la papelera una combinación de ambas. Por lo que se logra verificar a qué directorio

corresponde un archivo, debido a que tiene la misma ruta que un directorio, o como una carpeta pertenece a un drive, mediante a cómo empieza su ruta, entre otros.

Para realizar operaciones dentro del sistema se utilizan comandos, que pueden crear, modificar, seleccionar dentro del sistema. Todas las cláusulas tienen un “SB” (SystemBefore) que contiene al sistema inicial y un “SA” (SystemAfter) para guardar el resultado de la operación en el sistema (a excepción de “dir” que solo tiene un “SA”). Para la creación de elementos (comandos como add-drive, register, mkdir, add-file) se debe verificar que no se incumplan los requisitos, como que el nombre del elemento no exista, o no exista en el directorio en que se está trabajando. Para esto se hace uso de cláusulas recursivas tipo *buscar* (member), en la cual se busca elemento a elemento en una lista hasta encontrar el elemento o se acabe la lista. Después se crea el tipo de elemento mediante constructores, y se agregan a la cola de la lista que contiene ese tipo de datos. En caso de que las condiciones no se cumplan, la cláusula entrega *falso* (o SA es igual a SB, en el caso de “register” y “login”). Después de esto se rearma el sistema con una lista.

Para la modificación de datos actuales o de un elemento (login, logout, switch-drive, cd, del, copy, move, ren, format) primero se comprueba mediante los identificadores que el elemento exista en la ruta o la lista en general. En el caso de “copy” y “move”, además se debe comprobar que no exista el/los elemento/s en la ruta de destino. En el caso de “del”, todo lo eliminado de las listas originales, va a parar a la papelera, a diferencia de “format”, donde todo lo eliminado no se guarda en ninguna parte.

Algunas cláusulas pueden aceptar tanto archivos como carpetas (copy, move, ren). La forma en como saber que elemento se ingresó, es mediante “.”, ya que, en el caso de ser un archivo, el carácter se encuentra siempre a mitad del nombre, y las carpetas no. Al utilizar cualquiera de estás cláusulas, es necesario determinar cuales son los subdirectorios y sus archivos para modificar las rutas en el destino, esto se determina verificando cuando una ruta es una subruta de otra.

En el caso de usar “dir”, se forma un string que lista los contenidos del directorio, tomando todos los nombres de carpetas y archivos dentro de este.

6. Aspectos de la Implementación

Para la implementación del proyecto se estructuró en 5 TDA como archivos: system, drive, folder, file y user. Se utilizaron estos TDA ya que se pueden realizar todas las operaciones sobre ellas, con un alto grado de organización. Cada TDA posee los tipos de operaciones: constructores, selectores, modificadores y otras operaciones, con sus respectivas documentaciones y explicación de la representación de cada TDA.

Para el desarrollo del laboratorio, solo se utilizó la librería base de Prolog (built-in), y se ocupó la versión 9.0.4 de SWI-Prolog para x86_64-linux.

7. Instrucciones de uso

Para ejecutar los comandos, es necesario tener un sistema sobre el cual trabajar, por medio de la cláusula ***system.*** [Figura 1] Ya con un sistema, para modificarlo de cualquier forma, se requiere señalar ***SB***, ***SA*** y los datos pertinentes, si es que los requiere. Un ejemplo de esto es ***add-drive.*** [Figura 2] Una cláusula que funciona distinto es ***add-file***, la cual debe definir por fuera un archivo, y después incluir el resultado del constructor ***file*** en ***add-file***. [Figura 3 y 4] Otra cláusula distinta es ***dir***, la cual utiliza ***SA***, los parámetros de entrada como una lista, y un ***String*** de salida, en lugar de un sistema. Como ejemplo tenemos el parámetro ***["/o N"]***. [Figura 5 y 6]

8. Resultados

Los requerimientos funcionales abordados lograron un alcance completo (a excepción de algunas funcionalidades ***cd***, ***del***, ***copy*** y ***dir***). [Tabla 1] Se realizaron pruebas en todos los predicados, las cuales se encuentran en el script de pruebas, en las que se comprueba que cumplan los objetivos, no sean case-sensitive, no realicen modificaciones si no existe cierto elemento, que no creen y agreguen tipos de datos si ya existe uno con el mismo nombre y directorio, y que se logre trabajar con varias unidades y carpetas entre ellos.

De los requerimientos que se implementaron, los no se completaron fueron: ***cd***, se implementó la versión simple del predicado. ***del***, se implementó la versión simple del predicado. ***copy***, se consiguió implementar una de las funcionalidades del comodín "*", que permite copiar todos los archivos que sean de cierto tipo (por ejemplo: "txt"). ***dir***, no se implementaron los parámetros "/s" y "/o [-]D". No se completaron estas funcionalidades debido a problemas de tiempo.

9. Conclusiones

A la hora de trabajar el problema, las principales ventajas que posee el paradigma lógico es a la hora de trabajar con subproblemas que tengan muchas reglas y condiciones especiales. En especial cuando esas reglas y condiciones provienen de muchas partes que están sufriendo constantes cambios. (*Chapter 1 - Introduction*, 2023) Al momento de tener que modificar cómo funciona una regla, solo hay que analizar cómo se debe realizar la consulta y bajo qué parámetros, permitiendo una maleabilidad y preocuparse menos el ***cómo*** se realizan los problemas, y más el ***que*** consultar.

Por otro lado, la forma en que se realiza la recursión, lo cual es uno de los puntos fundamentales del paradigma, puede llegar a ser difícil de entender. La no existencia de recursos como "if" puede ser intimidante en un principio, pero basta con saber definir hechos que permitan funcionar como condicionales. Otro punto en contra es la forma en que se obtienen los resultados. El problema del sistema operativo no tiene un gran volumen de datos y reglas, pero para problemas que tienen un volumen muy grande de datos, el uso de backtracking puede resultar algo lento, debido a que se revisa todos los casos hasta que sea verdadero.

En comparación con el paradigma funcional, como ambos son de la familia declarativa, comparten similitudes en cuanto a cosas que no se pueden hacer como el uso de variables al estilo del paradigma procedural, o que cualquier tipo de ciclo

se debe realizar obligatoriamente de forma recursiva. Ambos paradigmas tienen pocos conceptos que son muy poderosos. El paradigma funcional, en específico Scheme, se puede ver un poco limitado a la hora de resolver problemas, haciendo que realice más operaciones para llegar a una solución, en cambio el paradigma lógico, en específico Prolog, debe implementar menos conceptos ya que una parte considerable del trabajo de búsqueda lo realiza el backtracking del propio lenguaje.

En conclusión, se logró implementar de manera mayormente exitosa el paradigma lógico. Este paradigma es una poderosa forma de resolver casi cualquier problema, debido a que la solución la busca el propio paradigma y no el usuario. Al igual que el paradigma funcional, la dificultad recae en adaptarse a la familia declarativa para lograr abstraerse adecuadamente. El problema se logró resolver de forma satisfactoria.

10. Referencias

Chapter 1 - Introduction. (2023). Stanford.edu. Extraido desde https://album.stanford.edu/logicprogramming/notes/chapter_01.html1.

Flores, V. (2023). *Paradigmas de programación. Programación Lógica*. [Diapositivas de PowerPoint]. Extraído desde Moodle Usach Paradigmas de Programación (13204 y 13310) 1-2023

11. Anexos

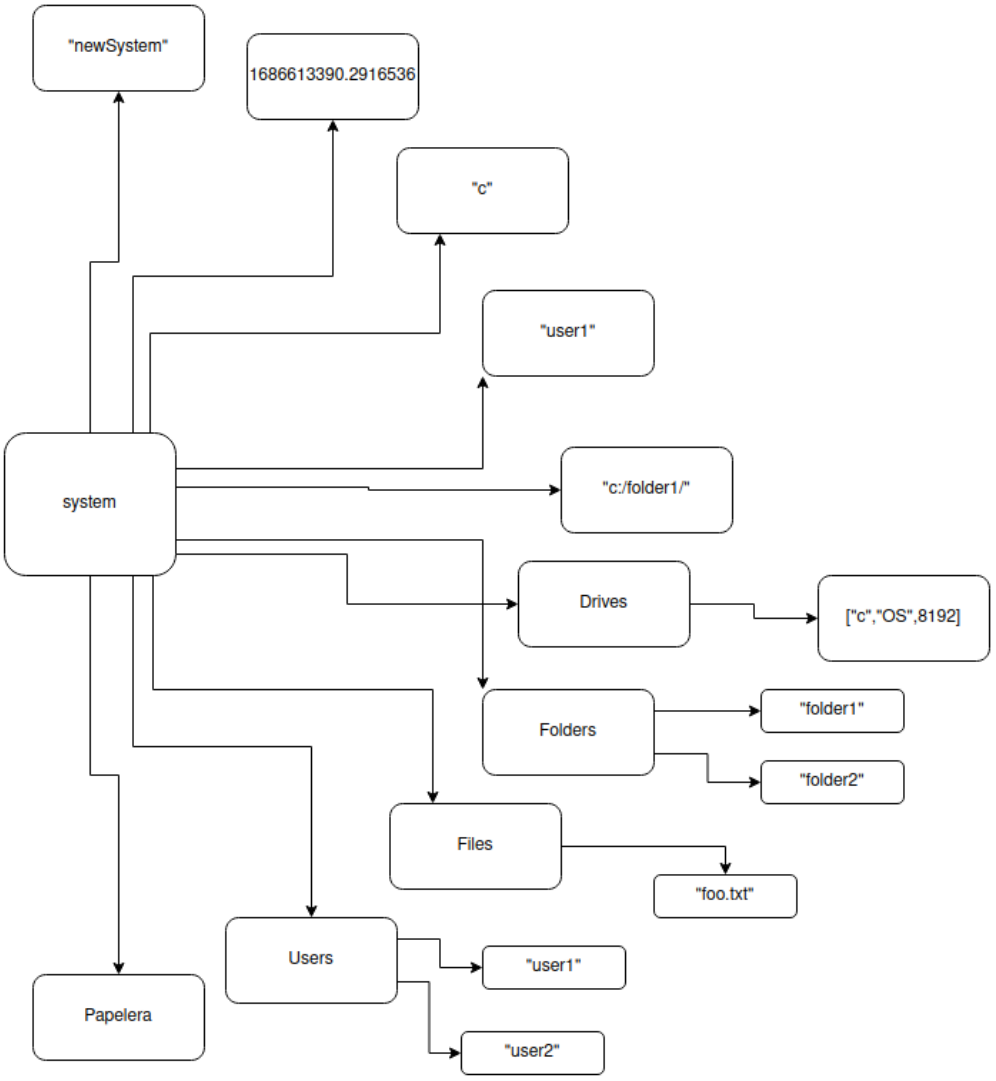

Diagrama 1, Representación del sistema.

```
?- system("Slackware Linux",Sistema).
Sistema = [Slackware Linux,1686616335.3369396,,,,[],[],[],[],[]].
```

Figura 1, Creación de un sistema con "system".

```
?- system("Slackware Linux",S1),systemAddDrive(S1,"c","Linux Filesystem",
100000,S2).
S1 = [Slackware Linux,1686616582.9395146,,,,[],[],[],[],[]],
S2 = [Slackware Linux,1686616582.9395146,,,,[[c,Linux Filesystem,100000]]
,[[c:/,,1686616582.939541,1686616582.939541]],[],[],[]].
```

Figura 2, Comando y resultado de agregar un drive con "systemAddDrive".

```
?- system("newSystem",S1),systemAddDrive(S1,"C","OS",10000000000,S2),syst
emRegister(S2,"user1",S3),systemRegister(S3,"user2",S4),systemLogin(S4,"u
ser1",S5),systemSwitchDrive(S5,"C",S6),systemMkdir(S6,"folder1",S7),syste
mMkdir(S7,"folder2",S8),systemCd(S8,"folder1",S9),systemMkdir(S9,"folder1
1",S10),systemLogout(S10,S11),systemLogin(S11,"user2",S12),file("foo.txt"
,"hello world",F1),systemAddFile(S12,F1,S13).
```

Figura 3, Comando para crear y agregar un archivo con "file" y "systemAddFile".

```
S13 = [newSystem,1686616804.1692147,c,user2,c:/folder1/,[[c,OS,1000000000
0]],[[c:/,,1686616804.1692314,1686616804.1692314],[c:/folder1/,user1,1686
616804.1692977,1686616804.1692977],[c:/folder2/,user1,1686616804.169305,1
686616804.169305],[c:/folder1/folder11/,user1,1686616804.1693313,16866168
04.1693313]],[[c:/folder1/,[foo.txt,txt,hello world,1686616804.1693423]]]
,[[user1,1686616804.1692421],[user2,1686616804.1692712]],[]].
```

Figura 4, Resultado de "systemAddFile".

```
?- system("newSystem", S1), systemAddDrive(S1, "C", "OS", 10000000000, S2
), systemRegister(S2, "user1", S3), systemRegister(S3, "user2", S4), syst
emLogin(S4, "user1", S5), systemSwitchDrive(S5, "C", S6), systemMkdir(S6,
 "folder1", S7), systemMkdir(S7, "folder2", S8), systemCd(S8, "folder1",
S9), systemMkdir(S9, "folder11", S10), systemLogout(S10, S11), systemLogi
n(S11, "user2", S12), file( "foo.txt", "hello world", F1), systemAddFile(
S12, F1,S13),systemCd(S13, "/folder2", S14),  file("ejemplo.txt","456",F2
),systemAddFile(S14,F2,S15),systemCd(S15,"/folder1",S16),systemMkdir(S16,
"carpeta",S17),systemMkdir(S17,"Botellas",S18),file("ayudas.csv","7,7,7,1
",F3),systemAddFile(S18,F3,S19),systemMkdir(S19,".escondido",S20),systemD
ir(S20,["/o N"],Str).
```

Figura 5, Comando para usar "systemDir".

```
Str =
ayudas.csv
botellas
carpeta
folder11
foo.txt.
```

Figura 6, Resultado de "systemDir".

Función	Alcance Conseguido
TDA	Alcance Completo
system	Alcance Completo
add-drive	Alcance Completo
register	Alcance Completo
login	Alcance Completo
logout	Alcance Completo
switch-drive	Alcance Completo
md	Alcance Completo
cd	Se implementó una versión simple de la función
add-file	Alcance Completo
del	Se implementó una versión simple de la función
copy	Alcance incompleto
move	Alcance Completo
ren	Alcance Completo
dir	Alcance Completo exceptuando “/o [-]D”
format	Alcance Completo

Tabla 1, Alcances de los predicados implementados.