Guillermo Cortés y Enmanuel Amaya

Programación lógica

UCA Editores

Índice

Presentación de la Materia 1
La lógica sentencial y la lógica cuantificacional 7
Conceptos introductorios 17
Las variables en Prolog 27
Los tipos de datos en Prolog 31
Construcción de cláusulas en Prolog 37
Operadores y expresiones 43
Comandos para entrada y salida de datos en consola 51
Operadores de control 57
La resolución de objetivos 67
Árboles SLD 73
Implementación de recursión en la solución de problemas 81
Aspectos adicionales sobre la evaluación de funciones 93
Manejo de listas en Prolog 97
Listas de listas (*Multilistas*) 107
Árboles 111
Manejo de archivos 119
Bibliografía 133

Presentación de la Materia

¿Qué se entiende por Programación Lógica?

Cuando se habla de Programación Lógica se hace referencia a un paradigma de programación. Esto es, una forma de escribir programas de computadora. A esta forma de programar también se le conoce como Programación Declarativa.

Existen varios paradigmas de programación, éstos han venido surgiendo a lo largo de la historia de la computación, de acuerdo al enfoque o a la necesidad. Así pueden mencionarse diversos paradigmas como:

- El paradigma de programación estructurada.
- El paradigma de programación orientada a objetos.
- El paradigma de programación funcional.

Además, existen lenguajes en los que se puede escribir utilizando varios paradigmas, a este tipo de programación se le conoce como *programación multiparadigma*.

En la mayoría de paradigmas, o formas de programar, cuando se escriben las instrucciones de un programa, se le está indicando a la computadora qué hacer para resolver un tipo de problema determinado, es decir, cómo se resuelve el problema paso a paso. Esto es lo que se hace, por ejemplo, en programación estructurada o en programación orientada a objetos.

En cambio, cuando se utiliza el paradigma de programación declarativa, se le indica a la computadora cuál es el problema a resolver, o el objetivo del cálculo, y no cómo resolverlo. Se deja la búsqueda de la solución en manos de la herramienta que se está utilizando. Las especificaciones para la solución se establecen en forma de expresiones de lógica matemática.

El lenguaje de programación PROLOG

El lenguaje que se utilizará en este curso se llama Prolog, un lenguaje que fue creado en la Universidad de Marsella, Francia, por Alain Colmerauer en los años 1970 - 1972. Su nombre viene del término francés *Programation et Logique*. Surgió por la motivación de las investigaciones de procesamiento de lenguaje natural y demostración automática de teoremas. Tiene, además, aplicaciones en inteligencia artificial, pues contiene un motor interno de búsqueda que pretende imitar la forma en que el ser humano realiza una búsqueda de soluciones. Algunas otras aplicaciones del lenguaje son:

- Los sistemas expertos.
- Las bases de datos deductivas.
- Los sistemas de procesamiento de lenguaje natural.

Como ejemplos de lenguajes que resuelven problemas por medio del uso de otros paradigmas pueden mencionarse:

- Fortran, desarrollado entre 1954 – 1958.
- Cobol, desarrollado entre 1959 – 1961.
- Pascal, desarrollado entre 1967 – 1971.
- C, desarrollado en 1972.
- SQL, desarrollado en 1974.
- Visual Basic, desarrollado en 1991.
- PHP, desarrollado en 1995.
- Java, HTML y XML, desarrollados en los años 90.

Hay cientos de lenguajes de programación de computadora, estos son sólo algunos, que se consideran los más representativos en términos históricos. Todos ellos pueden clasificarse como imperativos, porque al ejecutarlos, la computadora hace lo que las instrucciones le dicen, y nada más. En cambio, Prolog es un lenguaje declarativo, como se dijo anteriormente, y tiene unas características bien particulares, que a los que están acostumbrados a programar en lenguajes como C, Visual Basic, Java, PHP, etc., les parecerán extrañas. Estas características son las siguientes:

- No cuenta con instrucciones de control, como las de decisión e iteración.
- No cuenta con variables como depósitos de datos, así como se conocen en otros lenguajes. El concepto y el manejo de variables es diferente.
- Es un lenguaje de programación que es usado para resolver problemas que envuelven objetos y las relaciones entre ellos.

Historia de la programación lógica

Una de las principales preocupaciones que afectaron a los profesionales de la computación durante los años cincuenta fue la posibilidad de hacer programas que llevaran a cabo demostraciones de teoremas de forma automática. Fue así como comenzaron los primeros trabajos en inteligencia artificial, que luego de más de veinte años provocaron la creación del primer lenguaje de programación que contempla los mecanismos de inferencia necesarios para la demostración automática, este lenguaje se nombró LISP. Este primer lenguaje se basó en el formalismo matemático de la Lógica de Primer Orden y dio inicio al campo de investigación entre las matemáticas y la computación, campo que ha sido denominado Programación Lógica.

Estos mecanismos iniciales fueron utilizados durante muchos años con bastante frecuencia y con un alto grado de ineficiencia; no fue sino hasta el surgimiento de Prolog que estos mecanismos fueron retomados en el seno de un grupo de investigación en el campo de la Inteligencia Artificial.

La Programación Lógica tiene sus orígenes en una propuesta realizada en 1965 por J.A. Robinson, cientista de la computación y matemático inglés; dicha propuesta consistió en

una regla de inferencia a la que Robinson llamó *resolución*, mediante la cual la demostración de un teorema puede ser llevada a cabo de manera automática.

La resolución es una regla que se aplica sobre cierto tipo de fórmulas del Cálculo de Predicados de Primer Orden llamadas *cláusulas*, y la demostración de teoremas bajo esta regla de inferencia se lleva a cabo por reducción al absurdo.

Por lo tanto, la programación lógica tiene sus raíces en el cálculo de predicados, que es una teoría matemática que permite lograr que un computador pueda realizar inferencias, capacidad que es requisito para que un computador pueda ser considerado como una "máquina inteligente". La manifestación de las técnicas y modelos que describen a la programación lógica es el lenguaje de programación Prolog.

El lenguaje de programación Prolog estuvo un tiempo diseñado para ejecutarse en minicomputadoras o estaciones de trabajo, cuando actualmente hay versiones de Prolog que pueden instalarse en cualquier tipo de computadora (computadoras de escritorio, laptops, servidores, supercomputadoras, celulares, etc.).

Características peculiares de Prolog

Estrictamente hablando, un programa escrito en Prolog puro es simplemente un conjunto de cláusulas. Sin embargo, Prolog, como lenguaje de programación moderno, incorpora otros elementos que son más usuales en programas que utilizan otros paradigmas de programación, como por ejemplo instrucciones para la entrada y salida de datos.

Dado que es un lenguaje que se basa en el Cálculo de Predicados de Primer Orden, la estructura de las cláusulas de Prolog es fiel a sus representaciones en el campo de la Matemática Discreta, a excepción de pocos aspectos.

Las diferencias sintácticas entre las representaciones lógicas y las representaciones Prolog son las siguientes:

- En Prolog todas las variables están implícitamente cuantificadas universalmente.

- En Prolog existe un símbolo explícito para la conjunción "y" (,), pero usualmente no existe uno para la disyunción "o", que se expresa como una lista de sentencias alternativas. Algunas versiones de Prolog permiten el uso del punto y coma (';') como operador de disyunción.

- En Prolog, las implicaciones **p -> q** se escriben al revés **q :- p**, ya que el intérprete siempre trabaja hacia atrás sobre un objetivo.

Desarrollo de versiones de Prolog

Prolog ha sido uno de los lenguajes para la creación de sistemas expertos con mayor éxito, lenguajes que son capaces de procesar símbolos de todo tipo y capaces de hacer inferencias asociadas con elementos de su sintaxis, lenguajes que cuentan con un ambiente flexible que permita el desarrollo de sistemas expertos de forma interactiva.

Por tal razón, son muchas las compañías de software que han creado sus propias versiones de Prolog. La diferencia entre estas versiones es mínima, ya que su sintaxis y semántica es básicamente la misma, la variación más relevante es el cambio de plataforma para el cual fueron desarrollados.

Algunas de las versiones más utilizadas y conocidas en el ámbito informático son las siguientes:

- PROLOG1 - Esta versión la ofrece Expert Systems International, se utiliza en máquinas que trabajan con los sistemas operativos MS-DOS, CP/M-86, RSX-11M y RT-11. Su mayor fortaleza radica en su intérprete PROLOG86, que contiene un editor de cláusulas del sistema. Tiene un mejor manejo de los tipos de datos entero y real, y además posee más predicados integrados por lo cual el programador ya no necesita redefinirlos.

- PROLOG QUINTUS - Es una versión avanzada del lenguaje. El objetivo de sus diseñadores era producir una versión que pudiera ofrecer velocidades rápidas de ejecución, así como la capacidad de comunicación con otros programas. Esta versión funciona en los sistemas operativos UNIX y VMS. Una de las características interesantes es su interface dividida, donde el código fuente de los archivos se edita en la parte superior, mientras Prolog ejecuta el programa en la parte inferior. Esto brinda una ayuda ya que cualquier cambio en las cláusulas del archivo, podría ser probada inmediatamente, solamente interactuando entre las secciones de la ventana.

- MACPROLOG - Esta versión está diseñada totalmente para correr en máquinas MAC. Esta versión combina sofisticadas técnicas de programación de inteligencia artificial en ambientes Macintosh. Al integrar el compilador Prolog con las herramientas de MAC (ratón, menú, ventanas y gráficos), Prolog ofrece un entorno de programación sofisticado que antes sólo se podía obtener con hardware costoso y muy especializado.

- SWI-PROLOG - Es una implementación de Prolog basada en un subconjunto del WAM (Warren Abstract Machine), la máquina virtual que interpreta los programas escritos en Prolog.

SWI-Prolog fue creado en la Universidad de Amsterdam, Holanda, por Jan Wielemaker, y fue diseñado e implementado de tal modo que puede ser empleado fácilmente para experimentar con la lógica de la programación y con las relaciones que esta mantiene con otros paradigmas de la programación. Posee además un rico conjunto de predicados

incorporados que hacen posible el desarrollo de aplicaciones robustas y es totalmente gratuito.

La versión de Prolog que se utilizará para la materia será SWI-Prolog.

Ejercicios:

1. ¿Qué es la programación declarativa? Explique utilizando sus propias palabras.

2. ¿Cuáles fueron las necesidades que provocaron el surgimiento de la programación lógica?

3. ¿Cuáles son las principales áreas de aplicación de la programación lógica?

4. ¿Cuáles son las principales diferencias entre el lenguaje de programación Prolog y otros lenguajes de programación?

5. ¿En qué consiste el proceso conocido como *resolución*?

6. ¿Qué es una cláusula? Explique utilizando sus propias palabras.

7. Reflexione sobre lo expuesto en esta sección acerca de los alcances del paradigma de programación lógica, y luego responda: ¿Es la inteligencia un aspecto programable?

8. Haga un resumen explicativo de las características peculiares con las que cuenta el lenguaje de programación Prolog.

9. Haga un resumen explicativo de las versiones de Prolog más conocidas, y para cada una de ellas explique sus principales características y funcionalidades.

10. Investigue 5 proyectos de la vida real en los que se ha aplicado la programación lógica. Indique la finalidad del proyecto, las dificultades encontradas que fueron resueltas mediante el uso de la programación lógica, las herramientas de programación lógica utilizadas, y los resultados obtenidos.

La lógica sentencial y la lógica cuantificacional

Se mencionó anteriormente que Prolog basa su definición en el Cálculo de Predicados de Primer Orden, y que por lo tanto la estructura de sus cláusulas es fiel a sus representaciones en matemática formal. Se expondrá ahora un poco sobre estos aspectos matemáticos que constituyen la base de la programación lógica: la *lógica sentencial* y la *lógica cuantificacional*.

Para definir la lógica sentencial, en primer lugar, es necesario definir algunos aspectos:

El pensamiento

Se entenderá como pensamiento al conjunto de actividades mentales. Es decir, el conjunto de procesos que la mente humana es capaz de realizar ante la percepción de la realidad:

- El razonamiento.
- La abstracción.
- La generalización, etc.

El pensamiento tiene como finalidad que el ser humano sea capaz de realizar de forma intuitiva:

- La resolución de problemas.
- La adopción de decisiones.
- La representación de la realidad externa.

Entonces, básicamente puede decirse que el pensamiento es una modalidad del instinto de supervivencia. De hecho, una de las corrientes más aceptadas actualmente, plantea que el pensamiento es posible en diversas especies de seres vivos, y que dicha función está directamente relacionada con la complejidad del sistema nervioso en dichas especies.

En general, se sabe que los delfines, los perros, los caballos, los primates superiores (antropoides) y el homo sapiens, por supuesto, son poseedores de las habilidades típicas del pensamiento.

El juicio lógico

Se le llama juicio a la parte mental del pensamiento lógico. Un juicio es parte de una actividad más general que anteriormente hemos definido como pensamiento.

Al decir que es la parte mental del pensamiento, se está expresando que el juicio todavía no ha sido expresado en ninguna forma perceptible por los demás individuos, es decir, no ha sido hablado, escrito o representado de ninguna forma.

La proposición lógica

Se le llama así a una parte estructural del juicio, esencialmente a un razonamiento.

Dicho razonamiento puede aislarse y estudiarse, tanto en su forma (que es lo que pretende en general la rama matemática de la Lógica) como en su sintaxis, dentro del proceso más general del juicio.

En general puede decirse que un conjunto de proposiciones lógicas constituyen un juicio lógico.

El enunciado lógico

Se le llama así, a una proposición lógica que ha sido simbolizada e incorporada para su análisis dentro de un todo lógico. Un enunciado lógico puede tener solamente dos valores: VERDADERO o FALSO.

Dicho valor debe de ser independiente de su forma y de su relación con los demás enunciados. A la inversa, un enunciado lógico debe tener una forma tal, que pueda ser reducido a dos clases de elementos constituyentes: sus *partículas lógicas* (conectivas, cuantificadores) y sus *partículas fácticas* (hechos, el cuerpo de las proposiciones).

En base a lo anterior, puede definirse entonces a la lógica sentencial como: *la rama de la lógica que considera a los enunciados (o sentencias) como unidades, y estudia sus combinaciones, las cuales vienen a ser representaciones de los juicios lógicos.*

La lógica sentencial utiliza un lenguaje lógico que contiene:

- Símbolos para representar enunciados.
- Símbolos para representar funciones.
- Símbolos para representar relaciones entre enunciados.
- Tablas para el cálculo de los valores de verdad de las combinaciones de enunciados.
- Partículas lógicas que permiten la combinación de enunciados.
- Una jerarquía mínima de dichas combinaciones o fórmulas lógicas.

Esta jerarquía mínima aplicada sobre las fórmulas lógicas tiene tres categorías:

- *Tautologías*: fórmulas que son verdaderas en todas sus posibilidades.
- *Contingencias*: fórmulas que pueden ser verdaderas o falsas en todas sus posibilidades.
- *Contradicciones*: fórmulas que son falsas en todas sus combinaciones.

El propósito de la lógica científica es el hallazgo primario de tautologías dentro de los discursos.

La lógica cuantificacional es la rama de la lógica que brinda todos los símbolos, conectivas y cuantificadores que hacen posible el trabajar con los elementos de la lógica sentencial. La lógica cuantificacional es la que permite la simbolización de predicados lógicos (F, G, etc.), la construcción de esquemas cuantificacionales para el análisis de elementos aislados (*esquemas cuantificacionales monádicos*), la construcción de esquemas cuantificacionales para el análisis de grupos de elementos (*esquemas cuantificacionales poliádicos*).

En la matemática es poco lo que puede hacerse si no existe una lógica de la cuantificación, y es por esto que todos los enunciados de las ciencias naturales son sujetos de estudio de la lógica cuantificacional.

Los símbolos de la lógica cuantificacional extendida son:

- Símbolos primitivos

    1. Símbolos para denotar enunciados.
    2. Símbolos para denotar individuos, que se llamarán *variables*.
    3. Símbolos para denotar propiedades, que se llamarán *variables predicados*.
    4. La conectiva de *negación lógica.*
    5. La conectiva de *suma lógica, o disyunción inclusiva.*

- Símbolos definidos

    1. La conectiva de *producto lógico, o conjunción.*
    2. La conectiva del *condicional*.
    3. La conectiva de la *anti valencia, o disyunción exclusiva.*
    4. La conectiva de la *bicondicional.*
    5. La conectiva del *producto de relaciones.*
    6. La conectiva del *producto de propiedades de relaciones.*
    7. El cuantificador existencial *E*.

Razonamientos con proposiciones simples

Como se mencionó anteriormente, la lógica cuantificacional permite la representación de los razonamientos. El tipo de razonamiento básico con el que puede trabajarse es el razonamiento con proposiciones simples.

Un razonamiento con dos proposiciones simples es un razonamiento de la siguiente forma:

> Todos los hombres son mortales
> Sócrates es hombre
> Luego Sócrates es mortal

Este es un ejemplo de un silogismo clásico válido, porque la conclusión es una consecuencia lógica de las premisas. Sin embargo, este razonamiento en la lógica proposicional no es válido, porque si se simbolizara y se le aplicara, por ejemplo, lo que se conoce como el *método de la implicación tautológica*, el cual consiste en conjuncionar las premisas e implicarlas con la conclusión, para luego aplicar las tablas de verdad, podría verificarse sin ningún problema que se trata de una proposición contingente, lo que provocaría que el razonamiento sea inválido.

Es decir, no existe ninguna regla en la lógica proposicional que justifique la deducción de la conclusión. La validez de este tipo de razonamiento depende de la estructura interna de las proposiciones simples que la componen. La lógica cuantificacional estudia la composición íntima de las proposiciones, utiliza nuevos símbolos, leyes y métodos para establecer la validez de los razonamientos.

Proposiciones simples

Una proposición simple se compone de dos elementos: un argumento y un predicado.

El argumento es el nombre propio o frase que designa a los objetos individuales:

Juana es bailarina
Pedro constuye
Este libro es bueno
Sesenta es mayor que **veinte**
José es menor que **Francisco**
Jessica y **Estrella** son mejores que **Sonia** y **Blanca**

El Predicado es la parte de la proposición que expresa las propiedades o relaciones entre los objetos, las relaciones entre los argumentos. Normalmente, el predicado está formado por el verbo que indica la relación, ya sea con o sin complementos:

Roger **es mortal**
Hugo **canta**
Julia **está alegre**
La Luna **es un satélite**
Diez **es mayor que** tres
Roberto **es el jefe directo de** Raquel

Simbolización de las proposiciones simples

Los argumentos son llamados *constantes de individuo*, y se simbolizan mediante las letras minúsculas:

“a”, “b”, “c”, … “f”, “g”, “h”, …

Los predicados son llamados *letras de predicado*, y se simbolizan mediante las letras mayúsculas:

"A", "B", "C",… "F", "G", "H",…

Para simbolizar las proposiciones simples, primero se escribe la letra de predicado y luego la constante individual. En una proposición simple como "Hoy es martes", la constante de individuo "Hoy", y la letra de predicado "es martes", pueden ser sustituidos por los símbolos 'h' y 'M' respectivamente. La proposición quedaría simbolizada como:

"Hoy es martes"
"es martes" "Hoy"
Mh

Nótese como se han utilizado las primeras letras de cada uno de los elementos para la selección de los símbolos, esto es una práctica común y será considerada como convención para el resto del texto. A continuación se presentan algunos ejemplos más:

"Juan trabaja"
"trabaja" "Juan"
Tj

"Luis compone"
"compone" "Luis"
Cl

"Saturno es un planeta"
"es un planeta" "Saturno"
Ps

Las proposiciones simples pueden ser negadas; ésto se simboliza anteponiendo la negación a la fórmula por medio de un guión ('-'):

"Mañana no es jueves"
"no" "es jueves" "Mañana"
-Jm

"Daniel no trabaja"
"no" "trabaja" "Daniel"
-Td

"Lenin no es matemático"
"no" "es matemática" "Lenin"
-Ml

Las proposiciones que tienen un predicado y dos o más argumentos se simbolizan de la siguiente manera: se escribe primero la letra del predicado, y se escriben después las constantes individuales en el orden en que se presentan. Por ejemplo:

"Jorge ama a Cindy"
"ama a" Jorge" "Cindy"
Ajc

"Jorge viaja con Carlos"
"viaja con" "Jorge" "Carlos"
Vjc

"El novio era mayor que Claudia"
"era mayor que" "El novio" "Claudia"
Mnc

"Aristóteles nació en Estagira"
"nació en" "Aristóteles" "Estagira"
Nae

Estas proposiciones también pueden ser negadas y se simbolizan de igual forma, anteponiendo la negación a la fórmula:

"La Tierra no ilumina al Sol"
"no" "ilumina al" "La Tierra" "Sol"
-Its

"Jorge no ama a Cindy"
"no" "ama a" "Jorge" "Cindy"
-Ajc

Las proposiciones simples pueden formar proposiciones compuestas, es en este caso que las proposiciones son simbolizadas auxiliándose del uso de los conectivos lógicos:

- Para el conectivo lógico de conjunción se utiliza el símbolo "∧".
- Para el conectivo lógico de disyunción se utiliza el símbolo "v".
- Para el conectivo lógico condicional se utiliza el símbolo "!".

Utilizando lo anterior, éstos serían algunos ejemplos:

"Juan lee y Pedro escribe"
"lee" "Juan" "y" "escribe" "Pedro"
Lj ∧ Ep

"Einstein fue matematico o Kant fue filósofo"

"fue matematico" "Einstein" "o" "fue filosofo" "Kant"
Me ∨ **Fk**

"Si Jorge canta entonces Lizy baila"
"canta" "Jorge" "Si… entonces" "baila" "Lizy"
Cj ! **Bl**

"Si Carlos es primero, entonces Gabriel es segundo y Daniel es tercero"
"es primero" "Carlos" "Si… entonces" "es segundo" "Gabriel" "y" "es tercero" "Daniel"
Pc ! Sg ∧ **Td**

"Patricia es alegre y jovial"
"es alegre" "Patricia" "y" "es jovial" "Patricia"
Ap ∧ **Jp**

"Lorena canta y baila"
"canta" "Lorena" "y" "baila" "Lorena"
Cl ∧ **Bl**

A partir de los ejemplos puede concluirse lo siguiente: para argumentos y predicados diferentes se usan letras diferentes, para los mismos argumentos las mismas letras, y para los mismos predicados las mismas letras:

"Juan estudia y Carlos trabaja"
"estudia" "Juan" "y" "trabaja" "Carlos"
Ej ∧ **Tc**

"María estudia y trabaja"
"estudia" "María" "y" "trabaja" "María"
Em ∧ **Tm**

"Carmen y Hugo son músicos"
"es músico" "Carmen" "y" "es músico" "Hugo"
Mc ∧ **Mh**

Funciones proposicionales

Teniendo las proposiciones simples:

"Sócrates es filósofo" (**Fs**)

"Liszt es músico" (**Ml**)

Si sustituimos los nombres propios y las constantes de individuos por la variable individual "x", tendremos las siguientes expresiones:

"x es filósofo"
"es filósofo" "x"
Fx

"x es músico"
"es músico" "x"
Mx

Estas expresiones no son proposiciones, ya que no son verdaderas ni falsas, debido a que el argumento o individuo del cual se predica se encuentra indeterminado, por lo que se denominan *funciones proposicionales***.**

Las funciones proposicionales son formas que tienen una o más variables libres. Las funciones proposicionales se convierten en proposiciones, sustituyendo la variable por nombres propios o por constantes de individuo:

"x es filósofo" => **Fx**
Con instanciación: "Sócrates es filósofo" => **Fs**
x = Sócrates

"x no es matemático" => **-Mx**
Con instanciación: "Kant no es matemático" => **-Mk**
x = Kant

"x viajo a y" => **Vxy**
Con instanciación: "Lizy viajó a París" => **Vlp**
x = Lizy
y = París

"x nació es China" => **Nxc**
Con instanciación: "Mao nació en China" => **Nmc**
x = Mao

"x toca el piano y z canta" => **Tx ∧ Cz**
Con instanciación: "Jorge toca el piano y Robert canta" => **Tj ∧ Cr**
x = Jorge
z = Robert

Ejercicios:

1. Indique cuál es la diferencia entre la Lógica Sentencial y la Lógica Cuantificacional.

2. Indique cuál es la relación entre la matemática y el paradigma de programación lógica.

3. Defina con sus propias palabras los siguientes conceptos: *pensamiento, juicio, enunciado, y proposición.*

4. Describa cómo cada uno de los procesos que constituye el pensamiento es aplicado en la resolución de problemas matemáticos.

5. Describa cómo cada uno de los procesos que constituye el pensamiento es aplicado en la resolución de problemas informáticos.

6. Haga un resumen explicativo de las categorías en las que la Lógica Cuantificacional clasifica a los símbolos.

7. Para cada uno de los siguientes planteamientos identifique cuál es el argumento y cuál es el predicado:

    a) La casa de reuniones es la casa más vieja del pueblo.

    b) La isla de Makaokaha se encuentra al sur de la isla de Kauai.

    c) La bandera nacional nos une en hermandad.

    d) Arturo fue un gran jugador de la universidad. Logró batir cuatro records durante su carrera futbolística.

    e) La película tuvo una duración de 90 minutos. Fue bastante aburrida al inicio. De hecho, Juan se quedó dormido.

8. Simbolice cada uno de los siguientes planteamientos de acuerdo a la representación de proposiciones simples:

    a) El disco incluye el sencillo más popular.

    b) Richard quedó en penúltimo lugar durante la maratón.

    c) Mateo debutó como profesional haciendo lo que más le gusta: jugar póquer.

    d) John fue el oficial ejecutivo de la empresa. Su sucesor se llama Austero. Ambos comenzaron sus carreras como reporteros del New York Times.

e) Formica dirksi es una especia de hormiga. Pertenece a la familia Formicidae. Es una especia endémica en los Estados Unidos.

9. Simbolice cada uno de los siguientes planteamientos de acuerdo a la representación de proposiciones simples que utilizan conectores:

   a) El equipo Alfa es un batallón compuesto por cuatro grupos de soporte.

   b) 586 es un número que hace referencia a una cantidad, aunque también podría hacer referencia a una fecha. Si se tratara de una fecha, sería algo de los tiempos antiguos.

   c) El primer ministro es el ministro más viejo del gabinete. Al llegar al puesto es que pude seleccionar otros miembros, ya sea para promoverlos o para despedirlos.

   d) David es un neurocirujano, trabaja en la Universidad de Medicina. Es conocido por sus investigaciones en sinestesia. Podría deberse también a su fama de conquistador.

10. Adapte cada uno de los siguientes planteamientos a una función proposicional, indicando los argumentos a sustituir por variables (haga al menos dos versiones por planteamiento):

    a) Texas en un estado norteamericano.

    b) La diversidad de transmisión de una comunicación radial se origina desde una o dos fuentes independientes.

    c) El programa fue concebido por Carlos Parse y se produjo prácticamente durante dos temporadas.

    d) Nicole fue un cantautor y director de restaurantes durante la década de los noventa. Se le recuerda por su actitud cómica y despreocupada.

Conceptos introductorios

Prolog es un lenguaje *declarativo* e *interpretado*:

- Declarativo porque su sintaxis se encuentra definida en base a los elementos de la Lógica Sentencial y el Cálculo de Predicados de Primer Orden.

- Interpretado porque cada una de los enunciados lógicos es traducido a una secuencias de instrucciones entendibles por la computadora.

Por lo tanto, con ayuda de este lenguaje se puede representar el conocimiento sobre un determinado tema o área, y luego, a partir de ese conocimiento, pueden formularse preguntas e interrogar a Prolog para obtener las respuestas a dichos cuestionamientos, por medio de la inferencia que realiza con su motor interno de búsqueda.

A partir de lo anterior se pueden definir los siguientes conceptos:

Objeto:

En este lenguaje, un objeto representa cualquier entidad presente en la realidad o en la ficción, un objeto puede ser: una persona, un animal, una planta, un ser inanimado, un personaje de un cuento, etc. En resumen, cualquier cosa a la que se quiera hacer referencia.

Conocimiento:

En Prolog se pretende representar lo que se sabe acerca de los objetos: sus características y los vínculos que tienen unos objetos con otros. En un programa escrito en Prolog, a todo lo que constituye las características y vínculos de los objetos definidos se le conoce como el conocimiento del programa.

Base de conocimiento:

Es así como se les llama a los programas escritos en este lenguaje, puesto que se pretende utilizar el conocimiento que se tiene acerca de determinados objetos. Al igual que los programas de computadora escritos en otros lenguajes de programación, las bases de conocimiento se escriben en archivos de texto, y tienen su propio formato para escribir las diferentes partes que la componen.

En la versión de SWI-Prolog que se utilizará, las bases de conocimiento están escritas en archivos que terminan con la extensión *pl*.

Dominio:

Es el conjunto de objetos para los cuales es competente la base de conocimiento.

Hecho:

Es una característica o propiedad de un objeto, establecida en la base de conocimiento. Cada objeto puede tener uno o más hechos relacionados con él, dado que podría ser necesario establecer una o más de sus características o propiedades en la base de conocimiento.

Más adelante se presentarán ejemplos concretos acerca de hechos y de cómo acceder a ese conocimiento. Por ahora, sólo se mencionará el formato con que se escriben en la base de conocimiento, que es el siguiente:

<*hecho*> (<*objeto*>).

Los símbolos '<' y '>' indican que en esa posición se colocan los identificadores de los elementos: el identificador del hecho por fuera y a la izquierda de los paréntesis; y el identificador del objeto dentro de los paréntesis.

Por ejemplo, para establecer el hecho de que "Fido es un perro", se puede escribir:

perro(fido).

En donde estamos estableciendo que una característica del objeto *fido* es que es un *perro*, según lo especifica el formato arriba establecido.

Observe lo siguiente:

- En Prolog los nombres de los hechos y de los objetos deben comenzar con letra minúscula.
- Al establecer un hecho, la instrucción termina con un punto.
- La expresión se establece en su forma prefija, esto es, primero el identificador del hecho y luego el identificador del objeto, de acuerdo a lo establecido por la lógica cuantificacional.

Relación:

Cuando dos o más objetos distintos tienen un vínculo entre sí, éste puede ser establecido en la base de conocimiento y recibe el nombre de *relación*. Las relaciones también son hechos, pero hechos que involucran a dos o más objetos. Por otra parte, un objeto puede tener una o más relaciones con otros objetos.

Más adelante se presentarán ejemplos concretos acerca de relaciones y de cómo acceder a ese conocimiento. Por ahora solo se mencionará el formato con que se escriben en la base de conocimiento, que es el siguiente:

<*relación*> (<*objeto1*> , <*objeto2*> , …).

Por ejemplo, para establecer el hecho de que "Fido come carne", se puede escribir la relación:

come(fido, carne).

En donde se está estableciendo la relación *come* y los objetos *fido* y *carne*. En otras palabras, se le está diciendo a Prolog que los objetos, *fido* y *carne*, están vinculados por la relación *come*, en donde el objeto de la izquierda *se come* al objeto de la derecha.

Observe lo siguiente:

- Los nombres de las relaciones deben comenzar con letra minúscula, al igual que los hechos y los objetos.
- Al establecer una relación, la instrucción termina con un punto.
- La expresión se establece en su expresión prefija, esto es, primero el identificador de la relación y luego los identificadores de los objetos.
- El orden de los objetos dentro de la relación es arbitrario, pero debemos ser coherentes a lo largo de la base de conocimientos.

Regla:

Ésta se compone de dos partes: el antecedente y el consecuente. Para comprender mejor este concepto se debe recordar, de los cursos de matemáticas discretas, las expresiones condicionales $p \rightarrow q$, que se leen "p implica q", o "si p es cierto, entonces q también es cierto".

En Prolog, las reglas se establecen por medio de cláusulas de Horn:

Cláusulas de Horn

La regla es una cláusula de Horn de tipo "modus ponens", es decir, "Si es verdad el *antecedente*, entonces es verdad el *consecuente*".

Modus ponens es una regla de inferencia muy utilizada en los cursos de la matemática discreta:

Si p entonces q

o de manera simbólica:

$$p \rightarrow q$$

Ambas, p y q, son proposiciones, y p → q es una proposición condicional en donde p recibe el nombre de *antecedente* o *hipótesis* y q recibe el nombre de *consecuente* o *conclusión*.

En Prolog las cláusulas de Horn se escriben al revés de lo acostumbrado en los cursos de matemáticas discretas. Primero se escribe el consecuente y luego el antecedente.

$$q \leftarrow p$$

En donde ← se sustituye por :- en la computadora (dos puntos y un guión sin espacios intermedios). De esta forma, la cláusula de Horn se escribe así:

$$q :- p.$$

Notar que siempre debe de colocarse un punto al final.

Ejemplos:

1. Si se quiere establecer la siguiente proposición condicional:

   "Si Nerón ladra entonces Nerón es bravo"

   En donde, el que Nerón sea bravo se deduce a partir del hecho de que ladre, se escribe:

   La proposición condicional es: Si Nerón ladra entonces Nerón es bravo.

   Las proposiciones p y q son:

   p: Nerón ladra
   q: Nerón es bravo

   Ambas se escriben en Prolog así:

   p: ladra(nerón).

   q: bravo(nerón).

   Y la proposición condicional se escribe:

   ladra(nerón) → bravo(nerón)

   Escribiendo la cláusula como se haría en Prolog, se tiene:

bravo(nerón) :- ladra(nerón).

2. Si se quiere establecer la siguiente proposición condicional:

"Si Kiwi tiene plumas y Kiwi come masa y Kiwi habla entonces Kiwi es una lora"

Se escribe:

La proposición condicional es: Si Kiwi tiene plumas y Kiwi come masa y Kiwi habla entonces Kiwi es una lora.

Hay tres proposiciones p, q y r, que son:

p: Kiwi tiene plumas
q: Kiwi come masa
r: Kiwi habla

Notar que, en este caso, el antecedente lo constituyen varios hechos y relaciones, los cuales deben ser todos ciertos para que el consecuente sea cierto.

plumas(kiwi) y come(kiwi, carne) y habla(kiwi) → lora(kiwi).

Escribiendo la cláusula como se haría en Prolog, se tiene:

lora(kiwi) :- plumas(kiwi), come(kiwi, carne), habla(kiwi).

Observar que el "Y" lógico se sustituye por una coma.

A partir de los dos ejemplos anteriores se puede establecer que cada regla de la base de conocimiento es una cláusula de la forma:

$$r :- p_1, p_2, p_3, \ldots, p_n.$$

En donde:

- El consecuente recibe el nombre de *cabeza de la cláusula*.
- El antecedente recibe el nombre de *cuerpo de la cláusula*.
- Cada hecho o relación que se lista en el cuerpo de la cláusula se llama *objetivo*.
- El conjunto de hechos y relaciones que forman parte del cuerpo de la cláusula se conoce con el nombre de *secuencia de objetivos.*

Al evaluar un objetivo, éste puede ser cierto, en cuyo caso se dice que se ha obtenido un *éxito*, o puede ser falso, en este caso se dice que se ha obtenido un *fallo*.

Todos los objetivos de la secuencia de objetivos deben ser ciertos para que la evaluación de la cláusula retorne un éxito y la regla se cumpla. Si uno de los objetivos falla, la clausula falla y la regla no se cumple.

Como se mencionaba anteriormente, la coma que separa cada objetivo del otro tiene la función del "Y" lógico. A esto se le llama *conjunción de objetivos*.

Los hechos y relaciones pueden considerarse como si fueran *claúsulas sin cuerpo.*

Cuando se escriben las reglas en una base de conocimiento, se acostumbra colocar variables dentro de los paréntesis, no los identificadores de los objetos. Por ejemplo:

come(X, Y).

Nota: *Las variables siempre inician con letra mayúscula o guión bajo.*

En la terminología de Prolog, al identificador que está fuera de los paréntesis, ya sea hecho o relación, se le conoce como *predicado* o *functor*. A todo lo que está contenido dentro de los paréntesis se le llama *argumentos*, y a la cantidad de estos se le conoce como *aridad*, del francés *arité*. Así, el predicado *come* es de aridad 2; esto se simboliza: come/2.

Estructura general de un programa escrito en Prolog

Un programa escrito en Prolog está formado por una secuencia de enunciados lógicos, que pueden ser hechos, reglas y comentarios. Prolog es capaz de realizar procesos de inferencia sobre estas secuencias de enunciados lógicos para la realización de procesos de alto nivel (como demostraciones automáticas por ejemplo), por lo que a un sistema de aplicaciones Prolog que realicen este nivel de explotación sobre sus bases de conocimiento se le conoce usualmente con el nombre de *base de datos deductiva*.

En general, un programa escrito en Prolog tiene una estructura como la que se ilustra en el siguiente ejemplo:

```
% Bloque de hechos

mujer(maria).
hombre(pedro).
hombre(manuel).
hombre(arturo).
```

```
% Bloque de relaciones

padre(pedro,manuel).
padre(pedro,arturo).
padre(pedro,maria).


% Bloque de reglas

nino(X,Y):- padre(Y,X)
hijo(X,Y):-nino(X,Y),hombre(X).
hija(X,Y):-nino(X,Y),mujer(X).
hermano_o_hermana(X,Y):-padre(Z,X),padre(Z,Y).
hermano(X,Y):-hermano_o_hermana(X,Y),hombre(X).
hermana(X,Y):-hermano_o_hermana(X,Y),mujer(X).
```

Como puede observarse, en SWI-Prolog se utiliza el caracter '%' para la colocación de comentarios dentro de una base de conocimiento.

Ejercicios:

1. Defina con sus propias palabras los siguientes conceptos: *objeto, conocimiento, base de conocimiento, dominio, hecho, relación, cláusula de Horn, aridad*.

2. Describa la estructura de una cláusula de Horn, indicando cada uno de sus componentes y su significado.

3. Describa la manera en que Prolog da una respuesta a la resolución de una cláusula de Horn.

4. Describa la forma en que Prolog es capaz de utilizar las operaciones lógicas de conjunción y disyunción.

5. Escriba cada una de las siguientes proposiciones como un hecho en el lenguaje Prolog:

    a) El barco es grande.

    b) María es mujer.

    c) María es estudiosa.

    d) Luis es hombre.

e) El gato araña.

f) Kiwi es una lora.

g) Nerón es un perro.

h) El ladrón es malo.

i) La serpiente es un reptil.

j) El clima está lluvioso.

k) El oso es gris.

l) Kiwi habla.

6. Escriba cada una de las siguientes proposiciones como una relación en el lenguaje Prolog:

   a) El pez tiene escamas.

   b) Estela es abuela de Manuel.

   c) Anacleto tiene 40 años de edad.

   d) Marina es amiga de Claudia.

   e) El pájaro anida en el árbol.

7. Escriba un predicado de aridad 3 para cada una de las siguientes proposiciones:

   a) Rodrigo es jefe de Mario y Carolina.

   b) El 10 es un número entero y par.

   c) Escriba una relación para indicar la fecha "11 de diciembre de 1962".

8. Camilo es un camello, y para introducir conocimiento sobre él se le pide a usted que:

   a) Elabore un hecho y una relación que describan propiedades de Camilo.

   b) Elabore la cabera de la cláusula de Horn.

   c) Escriba la cláusula de Horn completa.

9. Traduzca a lenguaje natural el ejemplo mostrado al final de la sección (el ejemplo de la estructura general de un programa escrito en Prolog).

10. Transforme en hechos y relaciones el siguiente fragmento del poema "Bella" de Pablo Neruda:

Bella,
como en la piedra fresca
del manantial, el agua
abre un ancho relámpago de espuma,
así es la sonrisa en tu rostro,
bella.

Bella,
de finas manos y delgados pies
como un caballito de plata,
andando, flor del mundo,
así te veo,
bella.

Bella,
con un nido de cobre enmarañado
en tu cabeza, un nido
color de miel sombría
donde mi corazón arde y reposa,
bella.

Las variables en Prolog

En este lenguaje las variables no son vistas como casillas de memoria en las cuales se pueden almacenar valores para ser operados y transformados en el transcurso de la ejecución del programa.

El concepto de variable como depósito de datos es cambiado por el concepto de *variable lógica*.

En cuanto al manejo sintáctico y semántico de las variables puede mencionarse lo siguiente:

- En Prolog no se utiliza la declaración de variables.

- El nombre de las variables inicia con mayúscula o guión bajo. Cualquier otra etiqueta que comience con letra minúscula, es un término diferente a una variable (predicado, objeto, etc.).

- Mientras existe una variable, y mientras no esté un valor asociado a ella, se dice que es una *variable libre*.

- En el momento de asociar una variable con algún término, se dice que la variable se ha *instanciado*, y a partir de ese momento se dice que la variable está *ligada*.

- Cuando una variable se hace corresponder con otra, en sentido similar a asignar el contenido de una variable a otra, que es la manera en cómo se realiza en los lenguajes imperativos, se dice que ambas variables están *unificadas*.

El mecanismo de unificación de variables

La unificación es el mecanismo mediante el cual las variables lógicas toman valor en Prolog. El valor que puede tomar una variable consiste en cualquier tipo de elemento, un hecho, un número o una cadena de caracteres. Se debe tener el máximo cuidado para no confundir la unificación con la asignación de los lenguajes imperativos, puesto que la unificación se utiliza para representar la igualdad lógica.

Se dice que dos términos unifican cuando existe una posible ligadura (asignación de valor) de sus variables, de tal forma que ambos términos son idénticos si se sustituyen dichas variables por los valores asignados. Por ejemplo: a(X, 3) y a(4, Z) unifican dando valores a las variables: X vale 4, Z vale 3 (*Obsérvese que las variables de ambos términos entran en juego*).

Sin embargo, no todas las variables están obligadas a quedar ligadas. Por ejemplo: h(X) y h(Y) unifican aunque las variables X e Y no queden ligadas. No obstante, ambas variables permanecen unificadas entre sí. Si posteriormente se liga a la variable X al valor j(3), entonces automáticamente la variable Y tomará ese mismo valor.

En otras palabras, al unificar los términos dados, se impone la restricción de que X e Y deben tomar el mismo valor, aunque en ese preciso instante no se conozca dicho valor.

Es importante recalcar que la unificación no debe ser confundida con la asignación, y es relevante repetir la advertencia, ya que en Prolog muchas veces se unifican variables con términos en forma directa y de manera explícita, cosa que provoca la sensación de que se está realizando una asignación de valores a las variables al estilo imperativo.

En general, para saber si dos términos unifican podemos aplicar las siguientes normas:

- Una variable siempre unifica con un término, quedando ésta ligada a dicho término.

- Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.

- Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.

- Si dos términos no unifican, ninguna variable queda ligada.

Una vez que una variable se ha ligado, ya no se puede desligar de ese valor mientras la búsqueda continúa sin haber obtenido un fallo durante un proceso de resolución.

Ejercicios.

1. Defina con sus propias palabras los conceptos: *variable lógica, variable libre, variable ligada, variable instanciada y variable unificada.*

2. ¿Son los términos k(Z, Z) y k(4, H) capaces de ser unificados?

3. Si la variable H se cambia por el valor 3 en el ejercicio anterior, ¿podría realizarse una unificación?

4. Identifique los términos unificables en este ejemplo:
   a(b(j, K), c(X)) y a(b(W, c(X)), c(W)). ?

5. ¿Qué valores se ligan a las variables del ejercicio anterior?

6. Reflexione: ¿Es posible unificar cláusulas de Horn?

7. En el proceso natural del razonamiento humano, ¿cuál es la actividad equivalente a la utilización de variables lógicas?

8. En el proceso natural del razonamiento humano, ¿cuál es la actividad equivalente al proceso de unificación de variables?

9. Reflexione: ¿Cómo se utilizarían conceptualmente las variables lógicas en una demostración matemática por inducción?

10. Reflexione: ¿Cómo se utilizarían conceptualmente las variables lógicas en una demostración matemática por reducción al absurdo?

Los tipos de datos en Prolog

Como se mencionó en secciones anteriores, en Prolog no hay declaración de variables ni especificación de sus tipos. Sin embargo, esto no quiere decir que no existan tipos de datos en Prolog, ya que si son necesarios para categorizar los valores con los que una variable lógica puede ser unificada.

Los tipos de datos más importantes que Prolog soporta son:

a) String

Cualquier grupo de caracteres consecutivos (letras y números) que comience y termine con dobles comillas (“).

Ejemplo:“grande”, “árbol grande”

b) Integer

Cualquier número entero comprendido entre -32.768 y 32.767.

Ejemplo:4,-300,3004

c) Real

Cualquier número real en el rango +/- 1E-307 a +/-1E+308. El formato incluye estas opciones: signo, numero, punto decimal, fracción, E (exponente), signo para el exponente, exponente.

Ejemplo: 3,3.1415, 10E15, 3.1515E-12.30

d) Char

Cualquier carácter de la lista ASCII estándar, posicionado entre dos comillas sencillas (‘).

Ejemplos:‘t’,‘X’,‘f’

Predicados para realizar test de tipos

Prolog cuenta con predicados predefinidos para determinar, dada una variable lógica, el tipo de dato que contiene. Algunos predicados para el testeo de tipos son los siguientes:

Predicado	Significado
number/1	Retorna éxito si su argumento es un número entero o real. En caso contrario devuelve fallo.
integer/1	Retorna éxito si su argumento es un número entero. En caso contrario devuelve fallo.
float/1	Retorna éxito si su argumento es un número real. En caso contrario devuelve fallo.
var / 1	Devuelve éxito si su parámetro es una variable libre.
nonvar / 1	Devuelve éxito si su parámetro no es una variable libre.

Otros predicados similares, que pueden ser bastante útiles, son:

Predicado	Significado
round/1	Redondea al entero más cercano.
sign/1	Retorna 1 si el argumento es mayor que cero, retorna -1 si el argumento es menor que cero, y cero si el argumento es cero.
truncate/1	Retorna la parte entera de su argumento.
atomic/1	Retorna éxito si su argumento es un átomo, es decir, un entero, real o cadena.

Ejemplos:

Para los siguientes ejemplos se ilustrará la manera en que se realizan consultas a una base de conocimientos en la interfaz de Swi-Prolog:

- '?-' representa el cursor de la consola e indica que Prolog se encuentra listo para contestar preguntas hechas por el usuario.
- Las preguntas que se escriban (en forma de hechos o relaciones) siempre deben de ser finalizadas con un punto.
- Prolog siempre dará una de dos respuesta posibles: 'true' si la resolución fue exitosa, y 'false' si la resolución fue fallida.
- Si parte de la respuesta incluye el valor actual de una o más variables, Prolog también las mostrará.

a) Predicado *number/1*

```
?- number(34).
true.
```

```
?- number(gato).
false.
```

b) Predicado *integer/1*

```
?- integer(12).
true.
```

```
?- integer(12.1).
false.
```

c) Predicado *float/1*

```
?- float(12).
false.
```

```
?- float(12.1).
true.
```

d) Predicado *var/1*

```
?- var(X).
true.
```

```
?- X is 4, var(X).
false.
```

e) Predicado *nonvar/1*

```
?- nonvar(X).
false.
```

```
?- X is 4, nonvar(X).
true.
```

f) Predicado sign/1

```
?- X is sign(11).
X = 1.
```

```
?- X is sign(-11).
X = -1.
```

g) Predicados *round/1*y *truncate/1*

```
?- X is round(3.5).
X = 4.
```

```
?- X is truncate(3.5).
X = 3.
```

h) Predicado *atomic/1*

```
?- atomic(gato).
true.
```

```
?- atomic(tiene(plumas, polly)).
false.
```

El ámbito de las variables

Cuando en una regla aparece una variable, el ámbito de esa variable es únicamente esa regla. Supónganse las siguientes reglas:

```
abuelo_de(X, Y) :- hijo(Z, X), hijo(Y, Z).
puede_prestar(X, P) :- necesita(A, P), conoce(X, A), confia(X, A).
```

Aunque en ambas reglas aparece la variable X, la X de la primera regla no tiene nada que ver con la X de la segunda regla, y, en consecuencia, la instanciación de la X en la primera regla no implica la instanciación en la segunda. Sin embargo, todas las ocurrencias de X dentro de una misma regla se consideran la misma variable y se instanciarían con el mismo valor.

Ejecución de cláusulas de Horn

La ejecución de una cláusula de Horn se invoca desde la consola, en la línea de comando de Prolog, escribiendo el nombre del consecuente (cabeza de la cláusula) y terminando con un punto, así:

```
?- predicado(<lista de argumentos>).
```

Donde, en la lista de argumentos se realiza lo siguiente:

- En la posición del argumento del que se tiene alguna duda, que se quiere conocer o consultar, se coloca una variable.
- En la posición del argumento conocido se coloca el término correspondiente.

Ejemplos:

Dada una base de conocimientos sumamente sencilla como la siguiente:

```
pez(dori).
come(kiwi, galletas).
```

Se pueden realizar consultas como estas:

```
?- pez(X).

X = dori
```

Donde Prolog contesta que Dori es un pez, y en caso conociera más peces también los informaría.

```
?- come(kiwi, X).

X = galletas
```

En este segundo caso Prolog informa que lo que a Kiwi le gusta comer son las galletas.

Se puede incluso consultar de esta manera:

```
?- come(X, galletas).

X = kiwi
```

O de esta:

```
?- come(X, Y).

X = kiwi
Y = galletas
```

Como puede verse, un predicado de consulta puede tener varios argumentos. A Prolog le es indistinto cuál de ellos es el que se desea consultar desde la consola, y cuál le es proporcionado como valor conocido.

En todo caso, Prolog contesta de acuerdo a la base de conocimientos que se le ha introducido.

Ejercicios:

1. Defina con sus propias palabras los siguientes conceptos: *tipo de dato, ámbito y ejecución de base de conocimiento.*

2. Elabore una base de conocimiento que permita hacer interrogantes acerca de la fecha y hora de entrada y salida de los empleados de una empresa.

3. Amplíe el ejercicio anterior para que también se puedan preguntar el nombre de cada uno de los empleados.

4. Amplíe el ejercicio anterior para permitir además preguntar la sucursal de la empresa en la que laboran los empleados.

5. Elabore una base de conocimiento que permita hacer interrogantes sobre el significado de cada uno de los predicados de verificación de tipos estudiados en esta sección.

6. Amplíe el ejercicio anterior para que también se pueda preguntar la sintaxis de cada uno de los predicados.

7. Elabore una base de conocimiento que permita hacer interrogantes sobre el padrón electoral durante el período de elecciones. Interrogantes como el número de dui y el lugar de votación.

8. Amplíe el ejercicio anterior para permitir además preguntar sobre la última fecha de actualización del dui y los datos que fueron actualizados.

9. Elabore una base de conocimiento que permita hacer interrogante sobre el recorrido de algunas rutas del transporte público: lugar de salida, paradas principales, lugar de llegada.

10. Amplíe el ejercicio anterior para que también se pueda preguntar la hora de llegada y de salida de cada una de las paradas del recorrido, además de preguntar sobre la cantidad de pasajeros que abordaron y se bajaron del autobús.

Construcción de cláusulas en Prolog

En secciones anteriores se ha analizado cómo construir hechos y relaciones a partir de proposiciones simples. También se comentó acerca del uso de variables y algunos operadores comunes. En esta sección se procederá a explicar el proceso de construcción de cláusulas a partir de una serie de hechos fundamentales.

En primer lugar, se presenta el clásico ejemplo del árbol genealógico:

Supongamos que se tiene el árbol genealógico de una familia.

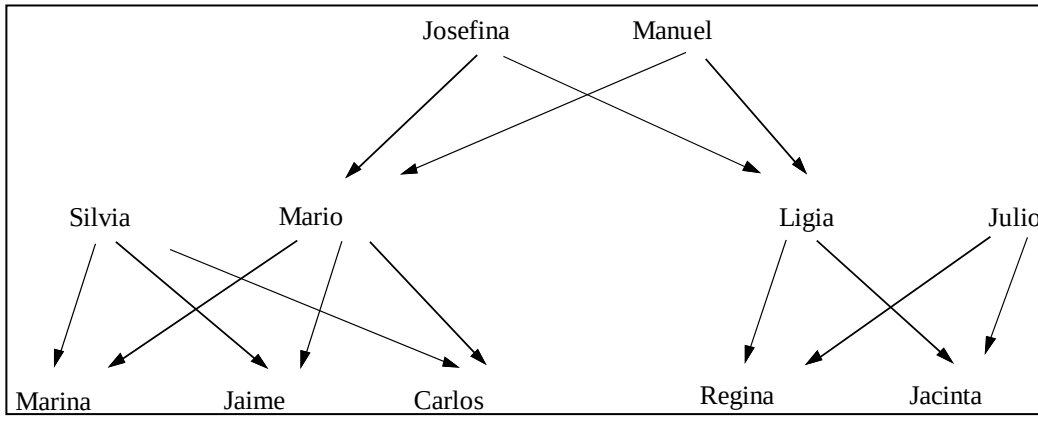


Figura 1. Árbol genealógico de una familia.

Utilizando el diagrama de la figura 1, se pueden establecer en Prolog las relaciones *madre/2* y *padre/2*. La base de conocimiento se vería así:

```
madre(josefina, mario).
madre(josefina, ligia).
madre(silvia, marina).
madre(silvia, jaime).
madre(silvia, carlos).
madre(ligia, regina).
madre(ligia, jacinta).
padre(manuel, mario).
padre(manuel, ligia).
padre(mario, marina).
padre(mario, jaime).
padre(mario, carlos).
padre(julio, regina).
padre(julio, jacinta).
```

A partir de este conjunto de relaciones puede entonces preguntársele a Prolog:

1. ¿Es Silvia madre de Marina?
   Consulta: madre(silvia, marina).
   A lo que Prolog responderá afirmativamente.

2. ¿Es Ligia madre de Jacinta?
   Consulta: madre(ligia, jacinta).
   A lo que Prolog responderá afirmativamente.

3. ¿Es Manuel padre de Ligia?
   Consulta: padre(manuel, ligia).
   A lo que Prolog responderá afirmativamente.

Utilizando variables pueden hacerse consultas más interesantes:

1. ¿De quién es madre Josefina?
   Consulta: madre(josefina, X).
   A lo que Prolog responderá desplegando todos los posibles valores de X que satisfagan esta proposición.

2. ¿De quién es padre Manuel?
   Consulta: padre(manuel, X).
   A lo que Prolog responderá desplegando todos los posibles valores de X que satisfagan esta proposición.

Las variables también se pueden poner en el sentido inverso de la relación:

3. ¿Quién es madre de Jaime?
   Consulta: madre(X, jaime).
   A lo que Prolog responderá desplegando el nombre de la madre de Jaime: Silvia.

4. ¿Quién es padre de Ligia?
   Consulta: padre(X, ligia).
   A lo que Prolog responderá desplegando el nombre del padre de Ligia: Manuel.

5. ¿Quién es madre de julio?
   Consulta: madre(X, julio).
   A lo que Prolog responderá que no tiene información al respecto.

Aparentemente, Prolog no podría deducir más allá de lo que tenemos planteado hasta ahora en la base de conocimiento. Pero, al explorar más detenidamente, hay relaciones familiares que se pueden deducir del árbol genealógico, aún cuando estas no han sido señaladas. Por ejemplo: los abuelos, primos, tíos y hermanos.

Este tipo de relaciones se pueden establecer por medio de cláusulas de Horn.

Ejemplo:

La relación abuela se puede establecer dado que una persona puede ser madre de otra y esta segunda ser madre, o padre, de una tercera.

Se coloca la cláusula:

```
abuela(X, Y):-  madre(X, Z), madre(Z, Y).
```

Personas que cumplen esta relación en el árbol son:

Josefina con Regina.
Josefina con Jacinta.

Notar que Josefina es abuela de Marina, Jaime y Carlos, pero la cláusula de Horn no considera la relación padre(Z, Y).

Para ello debe de establecerse una segunda cláusula, sin excluir la anterior, así:

```
abuela(X, Y):-  madre(X, Z), padre(Z, Y).
```

Algo que no debe de causar extrañeza, es que hay dos cláusulas con el mismo nombre. Siempre que se consulte abuela/2, Prolog va a agotar todas las posibilidades ejecutando todas las cláusulas que aparezcan con ese nombre.

Así, podríamos decir que, para que se cumpla que X es abuela de Y, se debe cumplir:

madre(X, Z), madre(Z, Y). ó **madre(X, Z), padre(Z, Y).**

Muchos autores suelen decir que, dado que la coma es la conjunción, dos cláusulas con el mismo functor y aridad son la disyunción.

Sin embargo, Prolog permite otra forma de escribir el "O lógico", y es a través de la utilización del punto y coma dentro de la misma cláusula, como se puede ver a continuación:

```
abuela(X, Y):-
        madre(X, Z) , madre(Z, Y) ; madre(X, Z) , padre(Z, Y).
```

Que, aplicando la ley distributiva, se reescribe como:

```
abuela(X, Y):-
        madre(X, Z) , (madre(Z, Y) ; padre(Z, Y)).
```

Ejercicios:

1. Suponga que, en el árbol genealógico visto anteriormente, alguno de los descendientes de la tercera generación tienen hijos. Escriba las cláusulas que determinan la existencia de bisnietos.

2. Elabore las bases de conocimiento y cláusulas necesarias para resolver los siguientes problemas:

    a) Siempre se dice que Ana habla más suave que Rebeca, quien es bastante gritona. Sin embargo, se dice Claudia habla tan fuerte que siempre le gana a Rebeca. ¿Habla Ana más suave o más fuerte que Claudia?

    b) Durante una maratón en las olimpiadas, cuatro corredores llegaron a la meta, se sabe que Carlos llegó justo después de Bruno, y Darío llegó después de Alberto pero antes de Carlos. ¿En qué orden llegaron los cuatro corredores?

    c) Seis amigos deciden hacer un viaje de negocios, y para hacer más divertida la travesía deciden ir de dos en dos en diferentes medios de transporte. Alejandro no irá en carro porque acompañará a Beatriz, quien por cierto, no irá en avión. El que si viajará en avión será Andrés. Si Camilo no quiere ir con David, y si tampoco le gustan los aviones, ¿En qué medio de transporte se irá Tatiana?

    d) Anastasia tiene cuatro perros: Pedro, Pablo, Paco y Wilber. Éste último come más que Pedro; Paco come más que Pedro y menos que Pablo, pero éste come más que Wilber. Wilber come más que Paco. ¿Con cuál perro Anastasia gasta menos en comida?

    e) En un partido del prestigioso torneo de tenis de Roland Garros se enfrentaron Nadal y Federer. El triunfo correspondió al primero por 6-3 y 7-5. Comenzó sacando Nadal y no perdió nunca su saque. Federer perdió su servicio dos veces. Nadal rompió el servicio de su rival en el segundo juego del primer set. ¿En qué juego del segundo set rompió el servicio Nadal?

    f) Jack Sparrow fue rodeado por un grupo de serpientes marinas, muchas de ellas ciegas. Tres no veían con los ojos a estribor, 3 no veían nada a babor, 3 podían ver a estribor, 3 a babor, 3 podían ver tanto a estribor como a babor, en tanto que otras 3 tenían ambos ojos arruinados. ¿Cómo mínimo, cuántas serpientes marinas había?

    g) El caballo de Mario es más oscuro que el de Silvio, pero más rápido y más viejo que el de Juan, que es aún más lento que el de Walter, que es más joven que el de Mario, que es más viejo que el de Silvio, que es más claro que el de Walter, aunque el de Juan es más lento y más oscuro que

el de Silvio. ¿Cuál caballo es el más viejo, cuál el más lento y cuál el más claro?

h) Ángel, Boris y César salieron a tomarse un trago y se sentaron en la barra de un bar. El que bebió champaña estaba en el extremo derecho. El que bebió cerveza se sentó a la izquierda del que bebió vino. A César no le gusta la cerveza. Boris estaba en un puesto a la izquierda de César. Ni César ni Boris bebieron vino. Angel estaba sentado al centro. ¿Qué puesto ocupó y qué bebida tomó cada uno de ellos?

i) Don Chilo, Doña Canuta, Don Mario, la Tenchis y Doña Marina viven en un edificio de cinco plantas. Todos ellos viven en distintos pisos. Don Chilo no vive en el quinto piso y Doña Canuta no vive en el primero. Don Mario no vive en el piso superior ni en el inferior, y no está en un piso adyacente al de Doña Marina ni al de Doña Canuta. La Tenchis vive en algún piso por encima de Doña Canuta. ¿Quién vive en cada piso?

Operadores y expresiones

Las expresiones aritméticas se escriben de manera similar a las relaciones, es decir, se establecen en su forma prefija. Por ejemplo:

3 + 5 debe de escribirse **+(3, 5).**

Es bastante similar a la expresión polaca prefija propuesta por el matemático Jan Lukasiewicz en 1920, con la excepción que incluye el uso de paréntesis y comas. A pesar que la forma prefija de los operadores es la forma predominante dentro de Prolog, la forma infija también puede ser utilizada.

Algunos de los operadores en Prolog, son:

Operador aritmético	Significado	Ejemplo
+/2	Suma	+(2, 3)
-/1	Cambio de signo	- 3
-/2	Resta	-(5, 7)
*/2	Producto	*(4, 10)
//2	División real	/(5, 2)
///2	División entera	//(13, 3)
mod/2	Residuo de la división entera	mod(13, 6)
abs/1	Valor absoluto	abs(-7)
^/2 ó **/2	Potencia	^(2,3) ó **(2,3)

Prolog también cuenta con un conjunto predicados predefinidos para diversas funciones matemáticas. Algunos de ellos son:

Algunas funciones aritméticas	Significado	Ejemplo
max/2	Valor máximo de dos números	max(32,40)
min/2	Valor mínimo de dos números	min(17,28)
pi/0	Valor de la constante PI	X is pi
log/1	Logaritmo natural	A is 2 + log(5.5).
sqrt/1	Raíz cuadrada	B is sqrt(4).
cos/1	Coseno, el argumento se toma en radianes	C is cos(0)
tan/1	Tangente, el argumento se toma en radianes	D is tan(0)
sin/1	Seno, el argumento se toma en radianes	E is sin(0)
acos/1	Arcocoseno, el argumento se toma en radianes	F is acos(1)
atan/1	Arcotangente, el argumento se toma en radianes	G is atan(0)

asin/1	Arcoseno, el argumento se toma en radianes	H is asin(0)
floor/1	Obtiene el número entero inmediato inferior al argumento	X is floor(3.2) X = 3
ceil/1	Obtiene el número entero inmediato superior al argumento	X is ceil(3.2) X = 4

Entre otros operadores existentes dentro del lenguaje Prolog se encuentran los de carácter relacional:

Operador de comparación aritmética	Significado	Ejemplo
</2	Menor que	<(3,5)
=</2	Igual o menor que	=<(5,5)
>/2	Mayor que	>(7,4)
>=/2	Mayor o igual que	>=(10,5)
=\=/2	Verifica la desigualdad numérica de dos expresiones argumento	=\=(5,7)
=:=/2	Comprueba la igualdad numérica de las expresiones argumento	=:=(5,5) =:=(+(2 , 3),5)

Para establecer la forma prefija de expresiones aritméticas más complejas se recomienda reescribirla paso a paso, iniciando por los operadores de mayor precedencia y terminando con los operadores de menor precedencia. Si en una parte de la expresión hay varios operadores con igual precedencia, estos se traducen en el orden en el que son encontrados.

Operadores de unificación

Prolog cuenta con dos operadores para la realización de unificaciones de variables con otros elementos:

=/2	Comprueba si dos términos son unificables. Si lo son, los unifica.	A = 'hola', B = A. unifica A con el término 'hola' y luego unifica B con A.
is/2	Evalúa la expresión de la derecha e instancia la variable de la izquierda con el valor numérico resultante.	is(A, 25). is (B, 10 * 4).

El uso de estos dos operadores puede resultar confuso. Es importante reconocer que la diferencia radica en si se efectúa o no una evaluación de expresiones: 'is' evalúa la

expresión a la derecha antes de instanciar (unificar con un valor), mientras que '=' no realiza ningún tipo de evaluación y simplemente unifica.

Ejemplos:

1. ?- =(A, 10).
   A = 10.

   En este ejemplo, la variable A se ha unificado con la expresión 10. Es sumamente importante hacer notar que esto no se trata de una asignación, sino de una unificación de expresiones lógicas. El siguiente ejemplo deja un poco más claro este concepto:

2. ?- =(A, 2+3).
   A = 2+3.

   Nótese que Prolog responde que A es 2+3, no que A es 5. Esto quiere decir que A se ha ligado a la expresión 2 + 3, sin ningún tipo de evaluación de la misma, al contrario de lo que se podría haber pensado, de que en la variable A almacena el valor 5.

3. ?- =(A, 5).
   A = 5.

   Lo anterior es lo que debe de realizarse si se desea ligar a la variable A con el valor 5 utilizando el operador '='.

4. ?- is(A, 2+3).
   A = 5.

   Como se observa en el ejemplo anterior, es por medio del operador 'is' que puede ligarse una variable al resultado de una expresión. No obstante, esto sigue sin ser considerado como una asignación.

5. ?- =(B, 10), is(A, B + 3).
   B = 10,
   A = 13.

   En este caso se realizó primero una unificación de la variable B con el valor 10, para luego ligar el resultado de la expresión 10 + 3 a la variable A.

6. ?- =(B, 10), =(A, B + 3).
   B = 10,
   A = 10 + 3.

En este caso se han realizado dos unificaciones: primero se unifica el valor 10 con la variable B, y segundo, se unifica la expresión B + 3 con la variable A. Nótese que Prolog informa que el valor ligado a A es 10 + 3, en lugar de B + 3.

7. ?- =(C, 5), =(A, C).
   C = 5,
   A = 5.

   C se liga a 5, y luego A se unifica con C, quedando ligado también con 5.

Uso del operador '=='

Este operador se utiliza para indicar si dos términos no numéricos son iguales, por ejemplo:

```
?- A = 'hola', B = 'hola', A == B.

A = hola,
B = hola .
```

Prolog despliega los valores de cada una de las variables en señal de éxito, en caso de fallo responde con un *false*.

El operador '\=='

Comprueba si dos términos son distintos, así mismo, si dos variables tienen distintos valores instanciados.

Ejemplos:

1. ?- is(A, 5), is(B, 10), \==(A, B).

   En esta consulta se liga a las variables A y B con valores distintos, luego, en el tercer objetivo se pretende comprobar si ambas variables tienen distintos valores instanciados, lo que resulta en un éxito.

2. ?- =(Cad1, hola), =(Cad1, Cad2), \==(Cad1, Cad2).

   En esta consulta se liga a las variables Cad1 y Cad2 con el mismo término. La comprobación del último de los objetivos dará un fallo puesto que las variables no tendrán distinto valor instanciado.

3. Un papá tiene varios hijos, y uno de ellos tiene el mismo nombre que su padre. Supóngase entonces las relaciones:

```
padre(mario, marina).
padre(mario, jaime).
padre(mario, carlos).
padre(mario, mario).
```

Al hacer la consulta *padre(mario, X)* se obtiene:

```
X = marina ;
X = jaime ;
X = carlos ;
X = mario .
```

Prolog muestra a todos los hijos, incluso al hijo llamado Mario, como es lógico suponer.

Pero si se construye la cláusula:

```
es_papa(X, Y) :- padre(X, Y), X \== Y.
```

Al realizar la consulta *?- es_papa(mario, X)*, se obtiene:

```
X = marina ;
X = jaime ;
X = carlos .
```

El uso del operador '\==' permitió omitir la opción *X = mario*.

Ejercicios:

Escribir en su forma prefija las siguientes expresiones, y posteriormente tradúzcalas a Prolog:

a) 2 + 3 * 7.5 / 4

b) (4.3 – 3) / 5 * 4 + 2

c) 3^2 - sen(0.5)

d) max(80, 73) // 4^3

e) sen(3^2 + 5.5 / 4 * 8) / (6 + 2)

f) (sqrt(7.5))3 + 6 * sen(2 * 4)

g) $\frac{20+30}{2*40}-\sqrt{\frac{9^3-5}{10}}$

h) $\frac{3*(sen(37)-\log(4^5))}{10+\max(100,200)}$

i) $\frac{4*\tan 25-3}{\sqrt{\frac{9}{2}-sen15}}-10*\log 43$

j) $\frac{\cos 47}{3}+\frac{\log(sen28)-\min(50,25)}{5^3-\frac{3}{5}}$

Especifique cual sería el resultado de los siguientes intentos de unificación:

k) =(C,A),=(C,5).

l) =(Y, X), =(X, 'hola').

m) =(Y, X), =(X, 5), =(W, 3), is(W, Y).

Desarrolle lo que se le pide:

n) Elabore una base de conocimientos de su universidad, en la que considere las relaciones para identificar: catedráticas(os), alumnas(os) y áreas de especialidad (ejem: química, computación, etc.). Luego establezca la cláusula tutoría(...):-, en la que determina que una persona puede tutoriar a otra, en base a la conjunción (disjunción) de objetivos determinados por las relaciones mencionadas.

o) Agregue a la base de conocimientos anterior hechos que determinen el sexo de las personas que allí se identifican. Luego construya una cláusula que considere que tanto el tutoriado como su tutor deben ser del mismo sexo.

p) Reescriba los hechos que determinan el sexo, como relaciones. Luego, elabore una cláusula en la que se pueda solicitar si el asesor es hombre o mujer.

q) Elabore una base de conocimientos de un grupo familiar, y utilice los hechos y relaciones necesarios para construir la cláusula hermana_o(X, Y):-:

i) Sin el objetivo X \== Y

ii) Con el objetivo X \== Y.

r) Utilizando la base de conocimiento del ejemplo anterior, elabore la cláusula tia_o_polit(X, Y):- (Sugerencia: elabore primero la cláusula tia_o(X, Y).).

Comandos para entrada y salida de datos en consola

Como en todos los lenguajes, en Prolog existe la posibilidad de manejar entrada/salida de datos. Esto es, archivos, pantalla, impresoras, etc. Todo ello se hace a través de flujos (streams), concepto idéntico al de otros lenguajes, como C++ y Java.

Los flujos son espacios de memoria (buffers) para escribir y/o leer de dispositivos como el teclado, la pantalla, el disco, etc. De modo que consideramos tres tipos de flujos:

flujos de entrada (lectura).
flujos de salida (escritura).
flujos de entrada y salida.

Existen dos grupos de predicados para manejar flujos:

- Aquellos donde el flujo es implícito. Manipulan la entrada y salida estándar: teclado y la pantalla (consola).

- Aquellos donde el flujo se indica explícitamente. Se suelen utilizar para manipular archivos y otros puertos del sistema (sockets).

Salida de datos en consola

Prolog cuenta con los siguientes predicados para la salida de datos en la consola:

- **write/1**, recibe un término como argumento.

- **writeln/1**, recibe un término como argumento.

- **put/1,** escribe caracteres simples, recibiendo como argumento el código ASCII correspondiente.

- **display/1,** cuyo efecto es el mismo que write/1, excepto al desplegar listas (las cuales serán estudiadas en secciones posteriores).

- **nl/0,** que escribe un salto de línea en la salida estándar.

Ejemplos:

1. ?- write(miTermino).
   Resultado: miTermino
   true.

2. ?- write('primer mensaje').
   Resultado: primer mensaje

true.

3. ?- write(‘primer mensaje’), write(‘segundo mensaje’).
   Resultado: primer mensajesegundo mensaje
   true.

4. ?- write(‘primer mensaje’), nl, write(‘segundo mensaje’).
   Resultado: primer mensaje
   segundo mensaje
   true.

5. ?- writeln(‘primer mensaje’), write(‘segundo mensaje’).
   Resultado: primer mensaje
   segundo mensaje
   true.

6. ?- put(90).
   Resultado: Z
   true.

7. ?- display(‘mensaje con display’).
   Resultado: mensaje con display
   true.

8. ?- A is *(5, 3), write('Resultado: '), write(A).
   Resultado: Resultado: 15
   A = 15 .

9. ?- X is 2, Y is sqrt(X), write('La raiz de '), write(X), write(' es: '), write(Y).
   Resultado: La raiz de 2 es: 1.41421
   X = 2
   Y = 1.41421
   true.

10. ?- X is 2, Y is sqrt(X), write('La raiz de '), write(X), nl, write('es: '), write(Y).
    Resultado: La raiz de 2
    es: 1.41421
    X = 2
    Y = 1.41421
    true.

Entrada de datos desde consola

Prolog cuenta con los siguientes predicados para la entrada de datos desde consola:

- **read/1**, lee un argumento por el teclado. La entrada se debe terminar con un punto.

- **get_char/1**, lee un carácter desde teclado. No requiere de punto al final.

- **get/1**, lee un carácter desde teclado. No requiere de punto al final. A diferencia de get_char/1, la variable dentro de los paréntesis es ligada con el valor ASCII del carácter digitado.

Ejemplos:

1. ?- write('Digite un numero: '), read(A), B is *(A, 10), write('Resultado: '), write(B).
   Resultado: Digite un numero: 30.
   Resultado: 300
   A = 30
   B = 300 ;

2. ?- write('Digite dos enteros, finalice cada uno con punto: '), read(A), read(B), C is +(A, B), write('Resultado: '), write(C).
   Resultado: Digite dos enteros, finalice cada uno con punto: 50.
   | 100.
   Resultado: 150
   A = 50
   B = 100
   C = 150 ;

3. ?- write('Introduzca un término: '), read(X).
   Resultado: Introduzca un término: mitermino.
   X = mitermino
   true.

4. ?- write('introduzca una frase: '), read(X).
   Resultado: introduzca una frase: 'Esta es la frase'.
   X = 'Esta es la frase' ;

5. ?- write('Introduzca un caracter: '), get_char(Car).
   Resultado: Introduzca un caracter: n (Notar que no se coloca punto)
   Car = n

6. ?- write('Introduzca un caracter: '), get(Car).
   Resultado: Introduzca un caracter: n (Notar que no se coloca punto)
   Car = 110

Ejercicios:

1. Elabore un programa en Prolog que explique paso a paso la estructura general de un programa escrito en Prolog.

2. Elabore un programa en Prolog que solicite el nombre del usuario y lo encripte en base a la siguiente regla:

   "Cada letra se debe sustituir por la letra que le sucede en el alfabeto. Ej: *juan* se encriptaría como *kvbo*."

   La *z* se sustituiría por una *a* para cerrar el círculo. El programa debe informar el nombre encriptado.

3. Elabore un programa en Prolog que solicite un término, y que informe a qué tipo de dato corresponde.

4. Elabore un programa en Prolog que le pregunte al usuario el tipo de dato con el que desea trabajar, para posteriormente solicitarle un dato del tipo seleccionado. Se deben incluir validaciones.

5. En secciones anteriores se realizó un ejercicio en base a un árbol genealógico, construyendo cláusulas para evaluar quién es abuelo de quién. Modifique el ejercicio para que al responder las interrogantes se muestre el camino de descendencia. Por ejemplo, si María es abuela de Martín, y el padre de Martín es Alberto, se mostraría algo como:

   María -> Alberto -> Martín

6. Elabore un programa en Prolog que dado un número entero, lo desglose en cada una de sus cifras y muestre cada una de ellas. Por ejemplo, dado el número 103, el programa desplegaría

   1 es la cifra de las centenas

   0 es la cifra de las decenas

   3 es la cifra de las unidades

   Considere un límite para su programa.

7. Elabore un programa en Prolog que cuente con una base de conocimiento que describa el organigrama de una empresa. La base se utilizará para contestar interrogantes acerca de quién es el jefe directo de un empleado, y quién es el jefe absoluto, desplegando la cadena de jefaturas.

8. Elabore un programa en Prolog que cuente con una base de conocimiento que describa los géneros musicales, principales discos y sencillos de varios artistas. El usuario puede seleccionar un artista para que el programa le informe los géneros musicales que utiliza el artista. Luego, puede seleccionar un género para conocer los discos que responden al género, y puede seleccionar después un disco para conocer las canciones que contiene, y saber cuáles fueron sencillos.

9. Elabore un programa en Prolog que dados los valores a, b y c de una ecuación cuadrática, sea capaz de resolver la ecuación.

10. Elabore un programa en Prolog que dado un nombre encriptado utilizando el método del ejercicio 2, informe cual es el nombre original.

Operadores de control

El predicado fail

fail (fallo) es un predicado que siempre produce fallo. Es útil cuando se quieren detectar casos explícitos en los que falla la evaluación de una cláusula.

Ejemplo:

Elabore un programa en Prolog para resolver la función:

$$f(x) = \begin{cases} 1, & x < 0 \\ 2, & x > 0 \end{cases}$$

Una primera versión del programa es la siguiente:

```
f(X, 1):- X < 0.

f(X, 2):- X > 0.
```

Este programa fallará cuando se quiera evaluar la función con cero. Esto podría ser muy probablemente lo que se busca que haga Prolog, pero en este caso el lenguaje no alertará por medio de algún mensaje que eso ha sucedido.

Una segunda versión podría ser entonces así:

```
f(0, Y):- write('Valor indefinido para f(x)').

f(X, 1):- X < 0.

f(X, 2):- X > 0.
```

En esta segunda versión observe que la primera cláusula informa que el valor de la función para el X solicitado es indefinido, cosa que es correcta, pero en este caso el programa devolverá un éxito. Para evitar esto se puede utilizar el predicado *fail*, colocándolo al final de la secuencia de objetivos.

Así que una tercera versión sería de esta manera:

```
f(0, Y):- write(' Valor indefinido para f(x)'), fail.

f(X, 1):- X < 0.
```

```
f(X, 2):- X > 0.
```

En esta última versión del programa se cumplen los requisitos antes planteados: que el programa mismo indique por medio de un mensaje el resultado para el caso en que X se ha ligado con cero, y que esa situación se convierte en un fallo.

Uso de variables anónimas

En ocasiones suele darse que al compilar un programa se obtiene una advertencia (*warning*) que señala que alguna variable nunca es usada. Esto puede darse en las clausulas que contemplan los casos triviales, o en cláusulas como la del ejemplo de la función escalonada.

$$\mathbf{f(x)} = \begin{cases} \mathbf{1,\ x < 0} \\ \mathbf{2,\ x > 0} \end{cases}$$

Como se mencionó, el programa que resuelve esta función en su versión final es:

```
f(0, Y):- write(' Valor indefinido para f(x)'), fail.

f(X, 1):- X < 0.

f(X, 2):- X > 0.
```

Observe que en la primera de las cláusulas la variable Y nunca es utilizada durante la secuencia de objetivos:

```
f(0, Y):- write(' Valor indefinido para f(x)'), fail.
```

En estos casos se puede utilizar lo que se conoce como *variable anónima,* para indicarle a Prolog que no es de interés lo que se evalúe en esa posición. Es también otra forma de evitar las advertencias en tiempo de compilación.

Para utilizar una variable anónima en lugar de la variable Y se utiliza el guión bajo ('_') de la siguiente manera:

```
f(0, _):- write(' Valor indefinido para f(x)'), fail.
```

El guión bajo es el símbolo que Prolog utiliza para indicar el lugar donde se coloca una variable anónima. Este tipo de variable no se opera ni se utiliza en ningún momento durante la resolución de la secuencia de objetivos.

El predicado de corte

En Prolog la resolución de la secuencia de objetivos de una cláusula se realiza mediante un proceso llamado *árbol de resolución SLD* (en la siguiente sección se discutirá con mayor detalle este proceso).

El corte es un predicado predefinido que no recibe argumentos y se representa mediante un signo de admiración (!). El predicado de corte se utiliza para "podar" ramas del árbol de resolución, con lo que se consigue que la finalización del proceso de búsqueda de una respuesta ocurra más rápido.

Es decir, cuando se ejecuta el corte el resultado del la cláusula en resolución queda en dependencia total de los objetivos que aparecen a continuación y que aún no han sido evaluados. Es como si a Prolog "se le olvidase" que dicha cláusula puede tener varias soluciones. Debe tenerse mucho cuidado al utilizarlo, ya que puede podar ramas que contengan soluciones, impidiendo que el sistema las encuentre.

El uso del predicado de corte es apropiado en los casos en que, habiendo varias respuestas, solo interesa obtener la primera que se encuentre al evaluar una cláusula.

Efectos del predicado corte:

- Siempre se cumple.

- Si se intenta re-evaluar la cláusula para encontrar alguna otra solución, el predicado elimina las alternativas restantes de los objetivos que hay desde su posición hasta la cabeza de la regla en donde aparece.

Ejemplo:

1. Considérense los siguientes predicados, los cuales se diferencia en un corte:

```
% Sin corte.
p(X, Y) :- X > 25, Y > 60.
p(X, Y) :- X > Y.

% Con corte.
q(X, Y) :- X > 25, !, Y > 60.
```

```
q(X, Y) :- X > Y.
```

Supóngase se realiza la consulta p(35, 22):

a. Prolog encuentra la primera opción de la cláusula *p(X, Y)* y liga las variables X e Y con los valores 35 y 22 respectivamente. Es en este punto que se observa que existe una segunda opción de resolución, lo cual se deja "apuntado" en alguna parte.

b. Prolog procede a evaluar el primer objetivo: X > 25, lo cual resulta en éxito.

c. Prolog evalúa el segundo objetivo: Y > 60, lo cual resulta en fallo.

d. La primera opción de la cláusula resultó en fallo, pero Prolog recuerda que existe una segunda opción (la que dejó "apuntada").

e. Prolog comienza la evaluación de la segunda opción con el primer objetivo: X > Y, lo cual resulta en éxito.

f. Al no haber más objetivos, la segunda opción de la cláusula resulta en un éxito.

Al final, la consulta p(35, 22) es exitosa.

Ahora supóngase se realiza la consulta q(35, 22):

a. Prolog encuentra la primera opción de la cláusula *q(X, Y)* y liga las variables X e Y con los valores 35 y 22 respectivamente. Es en este punto que se observa que existe una segunda opción de resolución, lo cual se deja "apuntado" en alguna parte.

b. Prolog procede a evaluar el primer objetivo: X > 25, lo cual resulta en éxito.

c. Se ejecuta el corte, provocando que las opciones alternas de resolución sean "olvidadas".

d. Prolog evalúa el segundo objetivo: Y > 60, lo cual resulta en fallo.

e. La primera opción de la cláusula resulta en fallo, y dado que Prolog "olvidó" que existían opciones alternas, ya no puede realizar nada más.

Al final, la consulta q(35, 22) es fallida.

Como puede verse sin ninguna dificultad, los resultados son sustancialmente diferentes. La segunda cláusula del predicado q/2 ni siquiera pudo ser evaluada debido a que el corte ha comprometido el resultado de la resolución al resultado de Y>15 en la primera cláusula.

2. Teniendo una base de conocimiento como la siguiente:

```
tiene(aguila, plumas).
tiene(aguila, pico).
tiene(ornitorrinco, pico).
tiene(pinguino, plumas).
tiene(pinguino, pico).
tiene(halcon, plumas).
tiene(halcon, pico).
tiene(avestruz, plumas).
tiene(equidna, pico).
tiene(avestruz, pico).
tiene(kiwi, plumas).
tiene(kiwi, pico).

vuela(aguila).
vuela(murcielago).
vuela(halcon).

no_vuela(ornitorrinco).
no_vuela(pinguino).
no_vuela(avestruz).
no_vuela(equidna).
no_vuela(kiwi).

come(aguila, carne).
come(pinguino, pescado).
come(halcon, carne).
come(avestruz, omnivoro).

oviparo(aguila).
oviparo(ornitorrinco).
oviparo(pinguino).
oviparo(halcon).
oviparo(avestruz).
oviparo(equidna).

es_ave(X):- tiene(X,plumas), tiene(X,pico).

ave_que_no_vuela1(X):- es_ave(X), no_vuela(X).
```

```
ave_que_no_vuela2(X):- es_ave(X), no_vuela(X), !.
```

Supónganse que se hacen las siguientes consultas:

```
?- ave_que_no_vuela1(X).
Resultado: X = pinguino ;
           X = avestruz ;
           X = kiwi.

?- ave_que_no_vuela2(X).
Resultado: X = pinguino ;
```

La segunda consulta es exactamente igual a la anterior, a excepción del uso del corte, el cual es el responsable de que las respuestas alternas a *X = pinguino* hayan sido "podadas".

3. Observar la siguiente consulta:

```
?- A is 5, write('Antes del corte '), !, B is 10, write(' Después del corte ').
Resultado: Antes del corte  Después del corte
           A = 5
           B = 10
```

El corte no evita, en este caso, que los objetivos que se encuentran a continuación, y hasta el final de la cláusula, se ejecuten. Recuerde que el corte siempre devuelve un éxito, así que Prolog seguirá evaluando objetivos hasta encontrar, ya sea un objetivo que falle, o bien, el final de la cláusula.

Usos del corte

El corte se utiliza muy frecuentemente, de hecho, suele suceder que mientras más experiencia tenga un programador, mayor es la frecuencia en que usará el corte.

Las razones principales por las que se usa el corte son las siguientes:

- Para optimizar la ejecución. El corte sirve para evitar que se exploren puntos de elección de opciones que, con toda seguridad, no llevan a otra solución (fallan).

- Para facilitar la legibilidad y comprensión del algoritmo que está siendo programado. A veces se sitúan cortes en puntos donde, con toda seguridad, no van a existir puntos de elección para eliminar, en estos casos el corte ayuda a entender que la ejecución sólo depende de la cláusula en resolución.

- Para implementar algoritmos diferentes según la combinación de argumentos de entrada. Algo similar al comportamiento de las sentencias case en los lenguajes imperativos.

- Para conseguir que un predicado solamente tenga una solución. Lo cual puede resultar necesario en algún momento.

Corte y fallo

Es común que en un programa escrito en Prolog se encuentra la secuencia de objetivos corte-fallo: **!,fail**. Esta secuencia de objetivos es útil en los casos donde, a la vez que es necesario eliminar soluciones alternas de la resolución, también es necesario que la resolución termine en un fallo.

Se utiliza para detectar prematuramente combinaciones de argumentos que no llevan a una solución, evitando la ejecución de un montón de código que se sabe con seguridad terminaría en fallo.

La negación por fallo

La negación en Prolog consiste en el predicado predefinido **\+/1**. La negación por fallo recibe como argumento un objetivo. Si dicho objetivo tiene éxito la negación falla y viceversa. Por ejemplo: \+ (X < 10) es equivalente a X >= 10.

Por simple que esta operación pueda parecer, la negación en Prolog presenta un detalle capaz de generar confusión: dicha negación **no** es la negación lógica, sino la **negación por fallo**.

La negación lógica se dedica a invertir el sentido de una proposición sin preocuparse por su valor de verdad, la negación por fallo invierte el valor de verdad de una proposición después de habérsele sometido a un intento de demostración.

Es importante tener clara la forma en que se lleva a cabo la resolución de cláusulas en Prolog: se asume que aquellos objetivos que no tienen solución, es decir, los objetivos que fallan, son falsos. Esto se denomina *asunción de mundo cerrado* porque se da por hecho que todo aquello que no se puede deducir (porque no ha sido escrito en el programa) es falso.

La negación por fallo solamente coincide con la negación lógica cuando el objetivo negado es un término que no contiene variables libres, lo que se llama un *término cerrado*. El programador es el responsable de asegurarse del cumplimiento de esta condición.

Ejemplo:

Considérese el siguiente programa:

```
estudiante(luis).
estudiante(juan).

informatico(luis).

hobby(X, musica) :-
informatico(X),
estudiante(X).
```

Se hace ahora la siguiente consulta: \+ hobby(X, musica), es decir, se le pregunta a Prolog ¿a quién no le gusta la música como hobby?

La negación lógica (y el sentido común) indica que la respuesta sería que el hobby de Juan no es la música. Sin embargo, Prolog contestaría que no hay nadie a quien no le guste la música.

Recuerde siempre lo siguiente: **en Prolog todos los predicados son falsos hasta que se demuestre lo contrario**. El problema es que a veces no se puede demostrar.

Algunos ejemplos del uso de la negación por fallo utilizando términos cerrados podrían ser:

```
?- X = 3, >(X, 5).
false.

?- X = 10, >(X, 5).
X = 10 .

?- X = 10, \+ (>(X, 5)).
false.

?- X = 3, \+ (>(X, 5)).
X = 3.
```

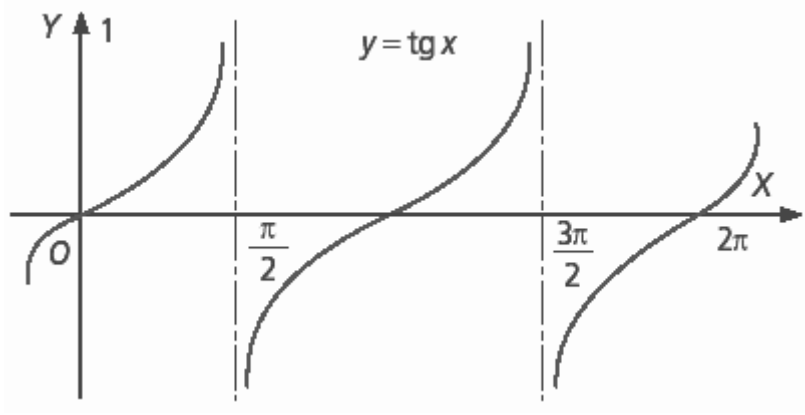
Ejercicios:

1. Reflexione sobre el motivo de esta condición: cuando un objetivo falla sus variables no se ligan. No obstante, su negación tiene éxito, entonces ¿a qué valor se ligan las variables de dicha negación?

2. Hacer un programa en Prolog que emule una sentencia *case* tal y como ésta funciona en los lenguajes de programación imperativos.

3. Modificar el programa sobre aves presentado en la sección del predicado corte para que no haga uso de la relación *no_vuela*.

4. Elabore un programa en Prolog para resolver evaluaciones de la función f(x) = tan (x), en el intervalo de cero a 2 * Pi.



5. Dada una base de conocimiento con relaciones de la forma: conoce_a(<persona>, <persona>), elabore cláusulas para resolver la siguiente consulta: una persona X desea realizar una modesta reunión con personas que conoce directamente, para lo cual debe enviar invitaciones.
   Elabore la cláusula de Horn con encabezado invita_a(X, Y), para resolver la consulta de a cuales Y debe invitar el anfitrión X.

6. Modifique la cláusula del ejercicio anterior para que resuelva a lo siguiente: una persona X desea conocer a personas conocidas por sus amistades, que ella aun no conoce.

7. Utilizando la base de conocimiento de los ejercicios anteriores construya una cláusula para que una persona X que desea realizar una modesta reunión, invite a

   - Personas que conoce directamente.
   - Personas conocidas de sus conocidos, que X aun no conoce.

   Su respuesta debe ser:

   i. Elaborando dos cláusulas de Horn, con mismo nombre y misma aridad.

   ii. Reutilizando las cláusulas de los ejercicios anteriores.

   iii. Utilizando el O lógico de Prolog (';').

8. Ahora la situación es la siguiente: una persona X quiere conocer nuevos amigos y decide enviar tarjetitas con saludos a gente que no conoce. Además sus destinatarios deben ser de sexo contrario. ¿Con quienes entablará contacto?

Agregue a la base de conocimiento relaciones de la forma sexo(<persona>, <sexo>).

La resolución de objetivos

Hasta el momento se ha discutido como crear un programa en Prolog, y como a partir de éste se pueden formular interrogantes para preguntarle a Prolog aspectos de la base de conocimiento, proceso que ha sido denominado *consulta*.

Se ha podido observar también que típicamente una consulta tiene el mismo formato que un hecho, y que a partir de una pregunta como:

```
?- quiere(rebeca, pedro).
```

Prolog buscara por toda la base de conocimiento hechos que coincidan con el que ha sido descrito en dicha pregunta.

Se sabe que dos hechos coinciden si sus predicados son iguales, y sus correspondientes argumentos son unificables uno a uno, y que si Prolog encuentra un hecho que coincida con la pregunta, responderá *true*, respondiendo *false* en caso contrario.

Además, se describió como una pregunta puede contener variables. Caso en el cual, Prolog buscará por toda la base de conocimientos aquellos objetos que pueden ser representados por la variable, siendo un ejemplo sencillo de esto:

```
?- quiere(rebeca, Alguien).
```

Con lo que se obtendría un resultado como:

```
Alguien = armando
```

Donde muy probablemente un hecho similar a *quiere(rebeca, armando)*. coincide con la pregunta, provocando una instanciación de la variable Alguien con el objeto *armando*.

Sin embargo, aún no se ha discutido la manera en que Prolog lleva a cabo este proceso de resolución de consultas. Conviene, por lo tanto, conocer cuál es el mecanismo de control de Prolog, con el fin de comprender el porqué y el cómo de sus respuestas.

El mecanismo de control de Prolog

El mecanismo empleado por Prolog para poder dar un respuesta a las preguntas hechas por el usuario hace uso de tres técnicas: el razonamiento hacia atrás (*backward*), la búsqueda en profundidad (*depth first*) y la vuelta atrás o reevaluación (*backtracking*).

- **El razonamiento hacia atrás**: partiendo de un objetivo a probar, Prolog busca los hechos que pueden probar el objetivo. Si en un punto caben varios opciones o caminos de resolución, éstas se recorren en el orden que aparecieron durante la resolución del programa, en general, de arriba a abajo y de izquierda a derecha.

- **La búsqueda en profundidad**: al momento de ejecutar cada una de las opciones encontradas durante el *backward*, no se abandona ninguna de ellas hasta agotar todos los recursos posibles en la base de conocimiento que pudieran provocar un éxito en la resolución, es decir, por cada opción Prolog intenta hasta lo último para saber si se trata de un éxito o un fallo.

- **La reevaluación**: si en un momento dado una variable se instancia con un determinado valor con el fin de alcanzar una solución, y se llega a un camino que provoca un fallo, Prolog retrocede al punto en el cual se instanció la variable, la des-instancia y busca otra instanciación que pudiera provocar un éxito, comenzando entonces un nuevo camino de búsqueda.

La estrategia anterior se puede ilustrar con el siguiente ejemplo:

Supóngase la consulta:

```
?- se_pueden_casar(rebeca, X).
```

Prolog recorre la base de conocimiento en busca de un hecho que coincida con la cuestión planteada y se encuentra con la siguiente regla:

```
se_pueden_casar(X, Y) :- quiere(X, Y), hombre(X), mujer(Y).
```

Prolog reconoce que la cabeza de la regla y la consulta coinciden, por lo que realiza una instanciación de la variable X de la regla con el objeto *rebeca*. Prolog ve ahora a la regla de la siguiente forma:

```
se_pueden_casar(rebeca,Y) :-
                quiere(rebeca,Y), hombre(rebeca), mujer(Y).
```

Con esto comienza el razonamiento hacia atrás, se sabe que lo que se quiere demostrar es el hecho *se_pueden_casar(rebeca, Y)*, por lo que se buscarán objetos, hechos y relaciones que lo demuestren. Se comienza por buscar una instanciación de la variable Y que verifique los hechos del cuerpo de la regla. El primer paso consiste en probar el primer objetivo de la regla:

```
quiere(rebeca, Y).
```

Una vez más Prolog recorre la base de conocimiento. En este caso encuentra un hecho que coincide con el objetivo:

```
quiere(rebeca, armando).
```

Prolog instancia la variable Y con el objeto *armando*. Siguiendo el orden dado por la regla, aún quedan dos objetivos por probar después de instancias a Y:

hombre(rebeca), mujer(armando).

Se recorre de nuevo la base de conocimiento, y como se podrá inferir no se encuentra ninguna coincidencia con el hecho *hombre(rebeca)* (es aquí donde se hace notar la búsqueda en profundidad).

A causa del fallo anterior, Prolog recurre a la vuelta atrás, des-instanciando el valor de la variable Y, y retrocediendo con el fin de encontrar una nueva instancia de la misma que haga cierto el hecho *quiere(rebeca, Y)*. Un nuevo recorrido de los hechos da como resultado una coincidencia con:

quiere(rebeca, natalia).

Se repite el proceso anterior. La variable Y se instancia con el objeto *natalia* y se intenta otra vez probar los hechos:

hombre(rebeca), mujer(natalia).

Esto provoca un nuevo fallo que provoca la des-instanciación de la variable Y, así como una vuelta atrás en busca de nuevos hechos que informen a quién más quiere Rebeca.

Una nueva reevaluación da como resultado la instanciación de Y con el objeto *ana*, el cual constituye la última opción dentro de la base de conocimiento, opción que una vez más terminaría en fallo a causa del objetivo:

hombre(rebeca).

Ante esta triste situación, donde el *backward* y el *depth first* jugaron un papel importante, Prolog da por imposible la regla. Esto provoca una vuelta hacia atrás en busca de alguna otra regla que coincida con la consulta, encontrando la siguiente:

```
se_pueden_casar(rebeca, Y) :-
                quiere(rebeca, Y), mujer(rebeca), hombre(Y).
```

Se repite todo el proceso anterior, buscando nuevas instanciaciones de la variable Y, y encontrando de nuevo como primera opción al hecho:

quiere(rebeca, armando).

Lo que provoca la instanciación de la variable Y con el objeto *armando*. Prolog tratará de probar ahora el resto del cuerpo de la regla con las instanciaciones actuales:

mujer(rebeca), hombre(armando).

Un recorrido de la base de conocimiento da un resultado positivo en ambos hechos, quedando probado en su totalidad el cuerpo de la regla, y por lo tanto su cabeza, entendida ahora como una de las soluciones al objetivo inicial.

X = armando
"Rebeca y Armando se pueden casar"

De todo lo anterior, puede reconocerse que Prolog utiliza un mecanismo de búsqueda independiente de la base de conocimiento. Aunque esto pudiera parecer ilógico, es en realidad una muy buena estrategia, debido a que garantiza el procesamiento de todas las posibilidades. Conocer esta estrategia es útil para que un programador sea capaz de realizar depuraciones y optimizaciones de sus programas.

Otros aspectos sobre el *backtracking*

Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuantas soluciones alternativas puede tener. Cada una de estas alternativas se denomina *punto de elección*. Dichos puntos de elección se anotan internamente y de forma ordenada y se trabaja de acuerdo al siguiente algoritmo:

1. Se escoge el primer punto de elección y se ejecuta el objetivo eliminando el punto de elección en el proceso.

2. Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicándole estas mismas normas.

3. Si el objetivo falla, Prolog da marcha atrás recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y deshaciendo las ligaduras de sus variables. En otras palabras, se inicia el backtracking.

4. Cuando uno de esos objetivos tiene un punto de elección anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución continúa de nuevo hacia adelante. El punto de elección se elimina en el proceso.

5. El proceso se repite mientras haya objetivos y puntos de elección anotados.

De hecho, se puede decir que un programa escrito en Prolog ha terminado su ejecución únicamente cuando no le quedan puntos de elección anotados y no le quedan objetivos por ejecutar durante un proceso de resolución.

Los puntos de elección no son eliminados al encontrar un éxito en la resolución. Esto es lo que permite que posteriormente se puedan conocer todas las soluciones posibles para una consulta.

Ejemplo:

Supóngase que se dispone de dos predicados p/1 y q/1 que tienen varias soluciones con un orden significativo:

p(1) tiene éxito.
p(2) tiene éxito.
q(2) tiene éxito.

Donde no existirían más soluciones que éstas.

A continuación considere la secuencia de objetivos *p(X), q(X).*

Prolog tiene que ejecutar p(X) y sabe que existen dos soluciones. En consecuencia, anota dos puntos de elección.

Se ejecuta p(X) usando el primer punto de elección, que se elimina de la lista en el proceso. Dicho objetivo tiene éxito y la variable X queda ligada al valor 1.

Ahora Prolog debe ejecutar q(X), que solamente tiene un punto de elección, el cual queda anotado.

Se ejecuta q(X) eliminando su único punto de elección. Sin embargo, X ya no es una variable libre, por lo que se está ejecutando realmente q(1). El predicado falla porque no es una de sus soluciones.

La conjunción falla.

Comienza el backtracking, recorriendo los objetivos en orden inverso hasta encontrar un punto de elección anotado.

El primer punto se elección anotado que encuentra Prolog es el segundo punto de elección para el objetivo p(X). Se deshace la ligadura de la variable X con el valor 1, es decir, X vuelve a ser variable libre.

Con lo anterior la ejecución puede continuar hacia adelante.
Se ejecuta de nuevo p(X), pero esta vez se usa el punto de elección encontrado. Se liga entonces a la variable X con el valor 2. El punto de elección se elimina en el proceso.

Ahora Prolog debe ejecutar q(X), que solamente tiene un punto de elección, el cual queda anotado.

Se ejecuta q(X) eliminando su único punto de elección. La variable X ya no está libre, por lo que se está ejecutando realmente q(2). El objetivo tiene éxito esta vez.

La conjunción tiene éxito manteniendo la ligadura de la variable X al valor 2.

La respuesta es X = 2.

Ejercicios:

Describa el proceso de resolución de los siguientes problemas, indicando los momentos en que se realiza el backtracking:

1. Siempre se dice que Ana habla más suave que Rebeca, quien es bastante gritona. Sn embargo, se dice que Claudia habla tan fuerte que siempre le gana a Rebeca. ¿Habla Ana más suave o más fuerte que Claudia?

2. Seis amigos deciden hacer un viaje de negocios, y para hacer más divertida la travesía deciden ir de dos en dos en diferentes medios de transporte. Alejandro no irá en carro porque acompañará a Beatriz, quien por cierto, no irá en avión. El que si viajará en avión será Andrés. Si Camilo no quiere ir con David, y si tampoco le gustan los aviones, ¿En qué medio de transporte se irá Tatiana?

3. Anastasia tiene cuatro perros: Pedro, Pablo, Paco y Wilber. Éste último come más que Pedro; Paco come más que Pedro y menos que Pablo, pero éste come más que Wilber. Wilber come más que Paco. ¿Con cuál perro Anastasia gasta menos en comida?

Árboles SLD

Un árbol SLD es una representación gráfica del proceso de resolución de una consulta al utilizar la base de conocimiento de un programa escrito en Prolog.

Las siglas SLD se deben a que un árbol de este tipo utiliza:

- Una regla de **S**elección de objetivos y reglas para llevar a cabo intentos de resolución (los puntos de elección del backtracking).
- Una resolución **L**ineal, es decir, una vez escogido un punto de elección no deben de realizarse bifurcaciones.
- Cláusulas **D**efinidas, las escritas por el programador en la base de conocimiento.

La regla de selección en Prolog sigue estas convenciones:

- La selección de objetivos se realiza de izquierda a derecha.
- La selección de reglas se realiza en el orden en que fueron escritas en el programa.

Ejemplo:

Dada la siguiente base de conocimiento en Prolog:

```
p(X,Y) :- r(Y), q(X,Y).
q(X,Y) :- r(X), r(Y).
q(X,b) :- r(X), t(X).
r(a).
r(b).
t(b).
```

Y una consulta a dicha base: *p(Z, b).*

El árbol de resolución SLD se muestra en la figura 2.

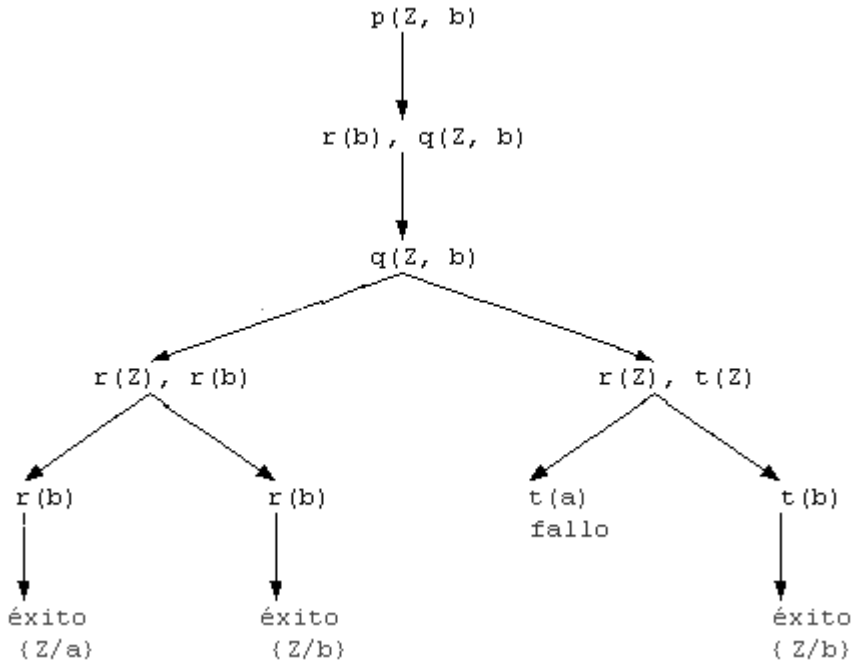


Figura 2. Ejemplo de árbol SLD.

Dado que un árbol SLD es una estructura estrechamente ligada al proceso de backtracking, los puntos clave para su construcción son los puntos de elección y las variables libres.

Por lo tanto, el árbol del ejemplo anterior puede analizarse de la siguiente manera:

a. La regla p(X, Y) unifica con la consulta, por lo que X se unifica con Z y Y se liga con b.

b. Sólo existe un punto de elección para p(X, Y), el cual es anotado.

c. Se ejecuta p(X, Y) eliminando su único punto de elección, dando paso al siguiente nivel del árbol, el cual consta de la secuencia de objetivos para el predicado p(X, Y) tomando en cuenta la unificación de X con Z, y el ligamiento de Y con b.

d. El primer objetivo, r(b), consta de un hecho, el cual produce un éxito, dejando para la evaluación únicamente a q(Z, b).

e. La regla q(X, Y) unifica con el objetivo, por lo que X (la X del predicado q/2) se unifica con Z, y la variable Y (la Y del predicado q/2) se liga a b.

f. Existen dos puntos de elección para q(X, Y), ya que q(X, b) es el mismo predicado con la excepción que Y se instancia en la cabeza de la regla. Se anotan estos dos puntos que pasarán a constituir ramas en el árbol.

g. Se ejecuta la primera opción de q(X, Y) eliminando su primer punto de elección. La secuencia de objetivos es ahora r(X), r(Y), que con los valores unificados e instanciados es r(Z), r(b).

h. El primer objetivo, r(Z), tiene dos puntos de elección. Estos son apuntados y serán más ramas para el árbol.

i. Se ejecuta la primera opción para r(Z), que es r(a), eliminando su primer punto de elección, ligando Z con a, y provocando un éxito.

j. El siguiente objetivo en la secuencia es un hecho ya analizado como cierto, por lo que el primer camino de resolución termina en éxito y se tiene como primera solución Z = a.

k. Comienza el backtracking, encontrando que aún existe un punto de elección para r(Z): r(b). Se ejecuta esta opción eliminando el último punto de elección de r(Z), desligando Z con a, ligando Z con b, y provocando un éxito, tanto en la opción como en el camino de resolución. Z = b se convierte en la segunda solución.

l. Se resume el backtracking hasta descubrir que queda un punto de elección para q(X, Y): la secuencia de objetivos r(X), t(X). Se ejecuta esta secuencia eliminando el último punto de elección del proceso.

m. El primer objetivo, r(Z), tiene dos puntos de elección. Se ejecuta el primero: r(a). Se elimina el primer punto de elección y se genera un éxito ligando Z con a.

n. El siguiente objetivo, t(a), provoca un fallo para la opción y para el camino de resolución.

o. Se inicia un backtracking hasta encontrar el segundo punto de resolución para r(Z), que es r(b). Se desliga Z de a, se liga con b y se genera un éxito.

p. El siguiente objetivo es ahora t(b), lo cual es un hecho y provoca un éxito en la resolución. Sin embargo, Z = b ya ha sido registrado como respuesta.

q. Al no haber más puntos de elección registrados, termina la resolución y la construcción del árbol SLD.

Como puede verse, al construir un árbol SLD ocurren generalmente los siguientes eventos:

1. Unificaciones generan niveles en el árbol.

2. Puntos de elección generan ramas en el árbol.

3. Los éxitos y fallos constituyen los extremos de las ramas del árbol, estos extremos son llamados *hojas del árbol*.

Utilizando estas consideraciones se le pide al lector analizar el siguiente ejemplo:

Dada la base de conocimiento del ejemplo anterior, y realizando la consulta *p(b, Z).*, el árbol SLD correspondiente se muestra en la figura 3.

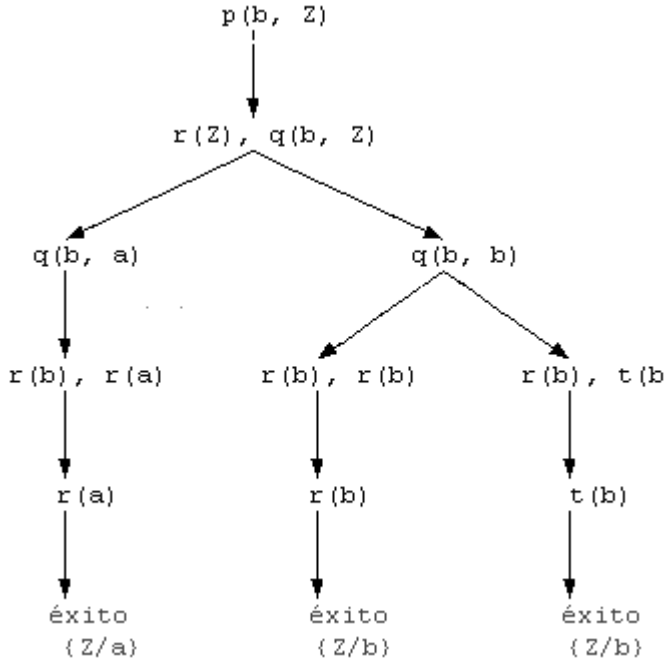


Figura 3. Otro ejemplo de árbol SLD.

Utilización de cortes en SLD

El predicado corte (!) es utilizado para evitar que Prolog dé soluciones alternas durante un proceso de resolución, tal y como se describió en secciones anteriores. En términos del backtracking esto es equivalente a decir que el corte elimina los puntos de elección que han sido anotados durante el proceso de resolución.

Con el conocimiento de los árboles SLD, puede mencionarse una versión más de esta afirmación: el corte elimina ramas en un árbol de resolución, es decir, es un predicado que efectivamente realiza un *corte*.

Ejemplo:

Dada la siguiente base de conocimiento en Prolog:

```
p(X,Y) :- r(Y), q(X,Y).
q(X,Y) :- !,r(X), r(Y).
q(X,b) :- r(X), t(X).
r(a).
r(b).
t(b).
```

Y la consulta p(Z, b).

El árbol SLD puede observarse en la figura 4.

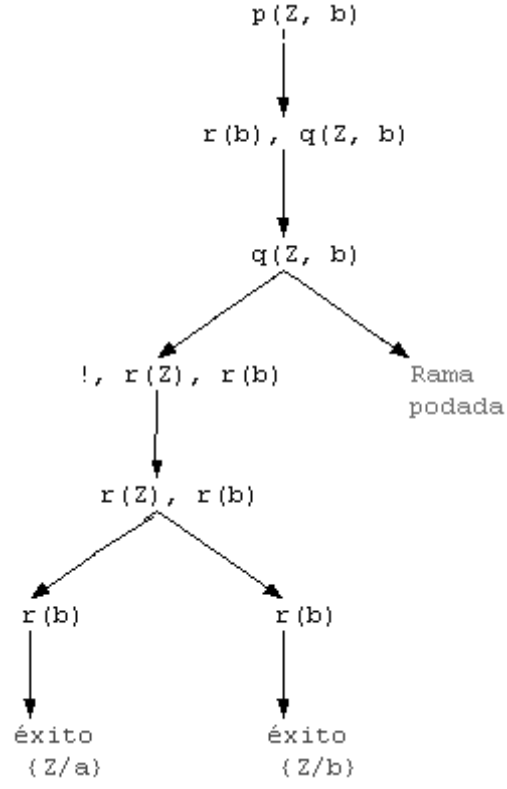


Figura 4. Ejemplo de árbol SLD con corte.

A continuación se presenta el análisis de este árbol con corte:

a. La regla p(X, Y) unifica con la consulta, por lo que X se unifica con Z y Y se liga con b.

b. Sólo existe un punto de elección para p(X, Y), el cual es anotado.

c. Se ejecuta p(X, Y) eliminando su único punto de elección, dando paso al siguiente nivel del árbol, el cual consta de la secuencia de objetivos para el predicado p(X, Y) tomando en cuenta la unificación de X con Z, y el ligamiento de Y con b.

d. El primer objetivo, r(b), consta de un hecho, el cual produce un éxito, dejando para la evaluación únicamente a q(Z, b).

e. La regla q(X, Y) unifica con el objetivo, por lo que X (la X del predicado q/2) se unifica con Z, y la variable Y (la Y del predicado q/2) se liga a b.

f. Existen dos puntos de elección para q(X,Y), ya que q(X,b) es el mismo predicado con la excepción que Y se instancia en la cabeza de la regla. Se anotan estos dos puntos que pasarán a constituir ramas en el árbol.

g. Se ejecuta la primera opción de q(X, Y) eliminando su primer punto de elección. La secuencia de objetivos es ahora !, r(X), r(Y)., que con los valores unificados e instanciados es !, r(Z), r(b).

h. Se ejecuta el corte, eliminando los puntos de elección no evaluados hasta el momento (se cortan ramas del árbol).

i. El segundo objetivo, r(Z), tiene dos puntos de elección. Estos son apuntados y serán más ramas para el árbol.

j. Se ejecuta la primera opción para r(Z), que es r(a), eliminando su primer punto de elección, ligando Z con a, y provocando un éxito.

k. El siguiente objetivo en la secuencia es un hecho ya analizado como cierto, por lo que el primer camino de resolución termina en éxito y se tiene como primera solución Z = a.

l. Comienza el backtracking, encontrando que aún existe un punto de elección para r(Z): r(b). Se ejecuta esta opción eliminando el último punto de elección de r(Z), desligando Z con a, ligando Z con b, y provocando un éxito, tanto en la opción como en el camino de resolución. Z = b se convierte en la segunda solución.

m. Se resume el backtracking, y se descubre que ya no existen más puntos de elección, ya que fueron eliminados por el corte. Por lo que el proceso de resolución termina.

Ejercicios:

1. Dada la siguiente base de conocimiento:

```
digito(0).
digito(1).
digito(2).
digito(3).
digito(4).
digito(5).

numero(A, B):- digito(A),digito(B), A is (2 * B ).
```

   Construir el árbol de resolución SLD que responda a las consultas:

   - *numero(1,B).*
   - *numero(A,B).*

2. Dada la base de conocimiento anterior, agregarle la regla:

```
numero(C, D):-digito(C), digito(D), C is 3*D.
```

   Y repetir las mismas consultas.

3. Dada la base de conocimiento anterior, agregarle las reglas:

```
numero(C, D):- numeroPar(C), C =:=D.

numeroPar(C):-  C1 is ( C  mod 2),  C1 =:=0.
```

   Y repetir las mismas consultas.

4. Construir un árbol SLD para la siguiente base de conocimiento al efectuar la consulta *nieto(c, X).*:

```
nieto(X, Z) :- hijo(X, Y), hijo(Y, Z).
hijo(X, Y) :- madre(Y, X).
hijo(X, Y) :- padre(Y, X).
padre(a, b).
madre(b, c).
```

5. Construir un árbol SLD para la siguiente base de conocimiento al efectuar la consulta *alumno(A, jose_a).*:

```
alumno(A, P) :- estudia(A, C), enseña(P, C).
estudia(ana, ia).
```

```
estudia(ana, pl).
estudia(eva, ra).
enseña(jose_a, ia).
enseña(jose_a, ra).
enseña(rafael, pl).
```

6. ¿Qué sucedería al construir un árbol SLD para la siguiente base de conocimiento al efectuar la consulta *hermano(a,X).*?

```
hermano(X,Y) :- hermano(Y,X).
hermano(b,a).
```

7. ¿Qué sucedería en el ejercicio anterior al cambiar la base de conocimiento por la siguiente?

```
hermano(a, b).
hermano(b, c).
hermano(X, Y) :- hermano(X, Z), hermano(Z, Y).
```

8. En un comedor cercano a la entrada peatonal de la UCA se sirven modestos almuerzos que incluyen una entrada, un plato principal, vegetales, bocadillo adicional, bebida y algo para empujar. Elabore la base de conocimientos que contiene los hechos:

   a) empujar/1 para pan y tortilla.

   b) entrada/1 para frijoles, papa, y macarrones.

   c) pl_princ/1 para carne, pollo y pescado.

   d) vegetal/1 para ensalada fresca y verduras cocidas.

   e) bebida/1 para leche, gaseosa, y jugo de naranja.

   Luego, elabore la cláusula almuerzo/5 que resuelve todas las posibilidades de combinación de los platillos antes mencionados, para el almuerzo de los clientes. Elabore El árbol SLD resolución para la consulta almuerzo(pan, Ent, carne, Veg, jugo_naranja).

9. Diseñe una estrategia para construir árboles SLD para un programa escrito en Prolog que utilice la negación por fallo (*Sugerencia*: investigar sobre los árboles SLDNF).

10. Modifique el programa elaborado en el ejercicio 1 para que muestre el árbol SLD durante el proceso de resolución.

Implementación de recursión en la solución de problemas

Al llegar a este punto ya son conocidas varias de las características de Prolog, como que no cuenta con instrucciones de control de flujo, que no cuenta con declaración de variables, que la coma separando dos objetivos implementa el conector lógico Y, que el O lógico se puede implementar mediante dos o más cláusulas con el mismo predicado y misma aridad, entre otras cosas. También se conoce la manera de resolver problemas que encadenan objetivos hasta encontrar una solución aceptable, pero surge la siguiente interrogante: ¿cómo se pueden resolver los problemas que involucren la realización repetida de determinadas operaciones?

En los lenguajes imperativos y procedimentales, como C, Java, Pascal o PHP, se pueden realizar repetidamente un conjunto de operaciones por medio de instrucciones de control de flujo. Estas instrucciones ayudan a llevar el control de las repeticiones por medio de conteos o mediante una evaluación de condiciones. Las instrucciones iterativas de este tipo más conocidas son el *para* (*for*), el *mientras* (*while*) y el *hacer ... mientrasque* (*do ... while*).

La buena noticia es que en Prolog sí se pueden resolver problemas que involucran la realización repetida de operaciones a pesar que no se cuenta con instrucciones iterativas. Esto se logra implementando lo que se conoce con el nombre de *recursión*. La recursión es un concepto antiguo e indispensable en la solución de problemas que requieren del uso de memoria dinámica (problemas como el control de estructuras de datos complejas en la memoria del sistema). En esta área hay ciertos problemas que sería muy difícil o imposible resolver si no se implementaran mecanismos recursivos.

En los lenguajes que permiten la implementación de la recursión, esto básicamente consiste en iniciar una nueva ejecución de un procedimiento o función antes de haber terminado una ejecución previa de dicho procedimiento o función. En este sentido puede decirse que hay, al menos, dos ejecuciones de ese mismo bloque de instrucciones en la memoria de la computadora, pendientes de finalizar.

El número de llamadas recursivas a un procedimiento o función depende de la cantidad de memoria del equipo que se esté utilizando; y la cantidad de llamadas recursivas que se harán dentro del programa no puede determinarse de antemano, sino que a medida que se avance en la solución del problema, los mecanismos recursivos implementados irán dando la pauta para ir terminando las ejecuciones en curso o realizar otras más. Muchos lenguajes de programación --algunos de los más tradicionales y otros más recientes-- que incorporan instrucciones para control de flujo, también permiten la implementación de soluciones recursivas. Este es el caso de los lenguajes mencionados anteriormente y de muchos otros. Para el caso de Prolog, la implementación de la recursión es el único medio para la solución de problemas que requieren la ejecución de un mismo bloque de instrucciones varias veces, es decir, en el caso de este lenguaje, una forma de realizar la ejecución repetida de una cláusula.

Como sucede en cualquier lenguaje, al resolver un problema mediante el uso de una cláusula, es muy común el hacer uso de otras cláusulas que resuelve una sub-tarea, como en el siguiente ejemplo:

```
tia_o(X, Y):- hermana_o(X, Z), (padre(Z, Y) ; madre(Z, Y)).
```

Observe que la cláusula *tia_o( )* hace uso de la cláusula *hermana_o( )*. En la figura 1 se muestra el proceso normal de una llamada desde un proceso a otro: cuando el proceso llamado termina de ejecutarse, el control regresa al proceso que hizo la llamada y continúa su ejecución normal. La figura 5 muestra cómo funcionan las llamadas no recursivas.

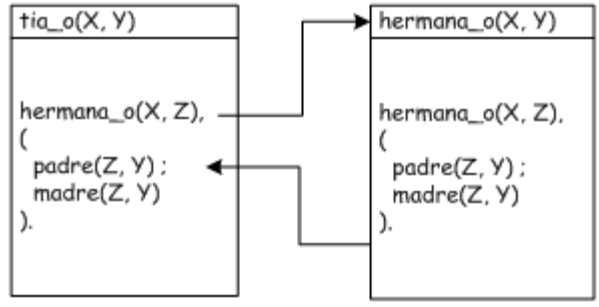


Figura 5. Proceso de ejecución de la llamada de una cláusula a otra cláusula.

Las llamadas recursivas a los procesos se pueden dar de dos formas: una se conoce como recursión directa y la otra como recursión indirecta. Ambas se explican a continuación.

- *Recursión directa*: consiste en que la llamada recursiva a una cláusula se encuentra en el cuerpo de la misma cláusula (ver figura 6).

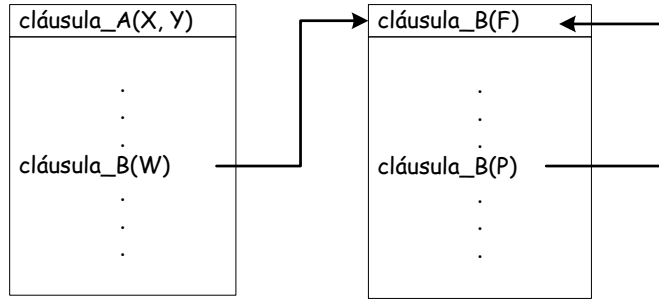


Figura 6. Proceso de ejecución de una llamada recursiva directa.

En la figura 6 se puede observar que hay dos llamadas a ejecución de *cláusula_B(F)*. La primera se realiza desde *cláusula_A(X, Y)*, que es una llamada no recursiva. La segunda llamada se realiza desde *cláusula_B(F)*, ésta es la llamada recursiva. Como puede notarse, una vez esta cláusula entra en ejecución la primera vez, habrá una serie de llamadas a ejecución de la misma debido a la invocación que se hace desde su mismo cuerpo. Naturalmente, debe haber un mecanismo de control que evite que se realice una cantidad de llamadas innecesarias y que facilite los retornos de todas las ejecuciones en curso. De lo contrario se corre el riesgo de provocar el agotamiento de la memoria del equipo y el fallo de la ejecución del programa. Estos mecanismos de control se verán más adelante.

- *Recursión indirecta*: consiste en la existencia de una cláusula recursiva que contiene una llamada a una segunda cláusula. Esta segunda cláusula contiene una llamada a la cláusula anterior. En este sentido hay, al menos, dos cláusulas recursivas que están invocándose una a otra (ver figura 7).

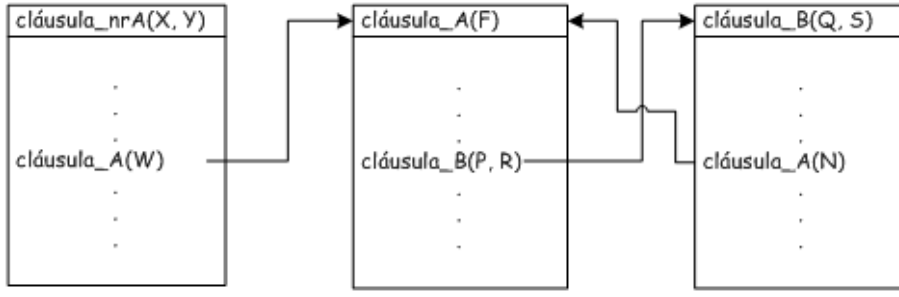


Figura 7. Proceso de ejecución de una llamada recursiva indirecta.

En la figura 7 pueden notarse las llamadas recursivas indirectas entre *cláusula_A(F)* y *cláusula_B(Q, S)*. Esta forma de realizar llamadas recursivas puede involucrar más de dos cláusulas. En general, dependiendo del problema a resolver, así será el mecanismo a implementar.

Como se mencionó anteriormente, el uso de la recursión es de especial importancia en Prolog debido a que no cuenta con estructuras de control repetitivas. Cada vez que se realiza la ejecución recursiva de una cláusula --o de una función, procedimiento o módulo en cualquier lenguaje que permita la recursión-- se crea espacio para contener nuevas copias de las variables locales. En cada nueva ejecución de la cláusula recursiva, la referencia a una variable local no interfiere con la variable local del mismo nombre en las demás ejecuciones de la cláusula que se encuentran en memoria. Las variables que se pasan como parámetro en la llamada unifican o ligan de la misma manera como se ha visto hasta ahora, esto no cambia.

Uso de la recursión

La forma más natural de explicar muchos problemas es la forma recursiva. El ejemplo más típicamente usado es el del factorial de un número, que se define así:

El factorial de un número entero positivo, simbolizado por n!, es otro entero que se obtiene de la siguiente manera:

$$n! = n * (n-1) * ... * 1$$

o, de otra forma:

$$n! = \begin{cases} \mathbf{1, si\ n = 0} \\ \mathbf{n * (n-1)!, si\ n >= 1} \end{cases}$$

(Forma recursiva de definir el factorial de un número)

Nótese que para obtener el factorial de un número que es mayor o igual a 1, se especifica que debe multiplicarse dicho número por el factorial del entero que le precede. De esta forma, se tiene una definición recursiva implícita en la definición. Así que su implementación recursiva por medio de un programa se considera hasta cierto punto natural.

Ejemplo:

Escriba un programa que realice un conteo iniciando en 1 y terminando en un valor X, con ayuda de un proceso recursivo:

```
contarHasta(X) :-
        hacerCiclo(1, X).

hacerCiclo(N, X):-
        =<( N, X),
        write('Ciclo '), write(N), nl,
        is(X1, +(N, 1)),
        hacerCiclo(X1, X).
```

Nótese que hay un objetivo que funciona como una validación, =<(N, X), que es el que controla que se realice, o no, el resto del cuerpo de la cláusula y la siguiente llamada recursiva.

Cuando la llamada recursiva es la última instrucción, como en este caso, se le llama *recursión de cola*. Es el caso más parecido a los procesos iterativos con instrucciones para el control de flujo de los lenguajes imperativos, y se puede reescribir fácilmente utilizando estas instrucciones.

En las cláusulas que se resuelven por medios recursivos es muy importante determinar correctamente:

a) Los objetivos que habrá que repetir.

b) Un objetivo que funcionará como condición o chequeo para la continuación o detención del proceso recursivo, que sea suficientemente consistente y que se sabe de antemano que siempre se va a alcanzar.

c) Los casos especiales --también conocidos como particulares, únicos o triviales-- que deben ser especificados en procedimientos o cláusulas aparte. Estos casos generalmente serán utilizados como mecanismos de parada de la recursión. Por ejemplo: en el caso del factorial de n, se sabe que siempre se alcanza el caso particular de 1!. Otro caso particular es el de 0!, que no se resuelve mediante productos, sino por definición.

Ejemplo 1:

Ejemplo de conteo ascendente con ayuda de procesos recursivos:

```
contarHasta(X) :-
        hacerCiclo(1, X).

hacerCiclo(X1, LimSup):-
        ==( X1, LimSup),
        write(‘Ultima ejecución: ‘),
        write(X1),
        write('. Fin de llamadas recursivas'), nl.

hacerCiclo(N, X):-
        <(N, X),
        write('Ciclo '), write(N), nl,
        is(X1, +(N, 1)),
        hacerCiclo(X1, X).
```

Observe dos cosas:

- La colocación del objetivo <(N, X), que controla el ingreso a la cláusula y la realización de una nueva llamada recursiva.

- La colocación del caso especial en una cláusula diferente de la cláusula principal; este es el último valor del conteo, cuando se ha alcanzado el límite superior. Este caso especial sirve también como mecanismo de parada de la recursión, no contiene una llamada recursiva y siempre se alcanza, obteniéndose un éxito. Es una buena práctica considerar los casos especiales en cláusulas independientes.

Debido a la forma en cómo funciona el mecanismo de la recursividad en cualquier lenguaje, se espera que al finalizar la última ejecución recursiva se regrese a las invocaciones precedentes, concluyéndolas una tras otra en el orden inverso en que fueron invocadas. Ver el siguiente ejemplo:

Ejemplo 2:

```
contarHasta(X) :-
        hacerCiclo(1, X).

hacerCiclo(X, X):-
        write('Ultima llamada: '),
        write(X),
        nl.

hacerCiclo(N, X):-
        write('Ciclo '), write(N), nl,
        N < X,
        X1 is N + 1,
        hacerCiclo(X1, X),
        write('Retornando a concluir llamada '),
        write(N),
        nl.
```

Observe lo que sucede al terminar la última invocación recursiva: se continúa ejecutando los objetivos pendientes de las llamadas previas al ir retornando. Cuando la llamada recursiva no es la última instrucción de un bloque de programa, sino que hay más instrucciones a continuación, a esto se le llama *recursión por posposición*, ya que existen instrucciones que quedan en espera de ser ejecutadas cuando se regrese de la llamada recursiva.

Notar que en el programa del *Ejemplo 1*, no se retorna a concluir las ejecuciones recursivas de la cláusula debido a que la última invocación recursiva no unifica, obteniéndose un fallo: el *objetivo =<(N, X)* no se cumple en la última llamada recursiva.

Ejemplo 3:

```
contarHasta(X) :-
        hacerCiclo(1, X).

hacerCiclo(N, X):-
        N =< X,
        write('Ciclo '), write(N), nl,
        X1 is N + 1,
        hacerCiclo(X1, X),
```

```
    write('Retornando a concluir llamada '),
    write(N),
    nl.
```

Notar la salida en la consola:

```
Ciclo 1
Ciclo 2
Ciclo 3
Ciclo 4
Ciclo 5
Ciclo 6
Ciclo 7
Ciclo 8
Ciclo 9
Ciclo 10

false.
```

Notar que los objetivos que le siguen a la llamada recursiva no se ejecutan, dado que no se realizó ningún despliegue en la consola.

¿Recursión por cola o recursión por posposición?

Muchas veces puede surgir una duda acerca de cuál de los dos tipos de recursión es mejor. En la historia, esta interrogante ha acompañado a los mecanismos de recursión como una sombra. Sólo existe una respuesta posible: *depende*.

¿Depende de qué? De diversos factores:

- *De las características del problema a resolver.*

Algunas situaciones requieren la repetición estricta de una secuencia de operaciones sin esperar ningún tipo de respuesta al finalizar la última ejecución de la misma: un programa que solicite una lista de alumnos para escribirlos en un archivo con algún formato determinado, un programa que lea desde dicho archivo la lista de alumnos ingresada, un programa que despliegue una frase un número determinado de veces, etc. Para estos casos es necesario implementar la recursión por cola.

Otras situaciones requieren que después de cada ejecución de una secuencia de operaciones se realicen operaciones adicionales: un programa que codifica una cadena de caracteres en un número entero cuya cifra de las decenas depende de la cifra de las unidades y así sucesivamente, un programa que calcule el factorial de un número, y en

general la mayoría de propuestas matemáticas recursivas. Para estos casos es necesario implementar la recursión por posposición.

- *De los datos disponibles.*

Independientemente de las características del problema a resolver, un informático debe analizar los datos que se utilizarán en la resolución de un problema, tomando en cuenta tanto los datos ya existentes, como los datos que se obtendrán durante el proceso de resolución del problema.

Los datos conocidos de antemano no son determinantes en la selección de la estrategia de recursión a implementar (al menos no la mayoría de las veces), caso contrario a los datos que se originarán durante la ejecución. Cuando los datos son independientes de la secuencia de operaciones, en el sentido que surgen de datos calculados durante su ejecución y no tanto de su resolución final, lo más conveniente es utilizar la recursión por cola. Un ejemplo de esto podría ser un programa que cuente los dígitos de un número mientras se analiza si el número cumple con alguna propiedad determinada.

Cuando los datos son dependientes de la secuencia de operaciones, en el sentido que dependen totalmente de la resolución de la ejecución, lo más conveniente es utilizar la recursión por posposición. Como ejemplo se pueden mencionar nuevamente el factorial de un número: para calcular el factorial de un número N, es obligatorio conocer primero el factorial de N-1, y así sucesivamente.

- *De la asimilación y el gusto del programador.*

Para algunos programadores puede resultar más sencillo entender el mecanismo de recursión por cola, por considerarlo más natural o lógico, dado que utiliza una elaboración lineal del resultado en un programa.

Para otros podría resultar más sencillo entender el mecanismo de la recursión por posposición, por considerarlo más formal o estructural, dado que se apega a las definiciones formales de los problemas recursivos matemáticos.

Podría pensarse que lo anterior genera un problema para quienes están familiarizados principalmente con la recursión por cola y se encuentran con una situación cuyas características y disponibilidad de datos sugieren el uso de la recursión por posposición, y viceversa. Sin embargo, es importante mencionar que, aunque para una situación particular la teoría indique que lo más apropiado es cierto tipo de recursión, la mayoría de las veces es posible adaptar una solución al otro tipo de recursión.

Es decir, al final es el programador (en base a su experiencia, gustos personales, y el análisis que realice sobre los problemas que debe resolver) quien decidirá cuál estrategia de resolución recursiva utilizar.

Ejemplo:

"Construir un programa que calcule el factorial de un número."

Como se mencionó anteriormente, esta es una situación que sugiere el uso de la recursión por posposición. Por lo tanto, una primera versión del programa podría ser:

```
factorial(0,1).
factorial(1,1).
factorial(N,R):- N1 is N-1,factorial(N1,R1),R is N*R1.
```

Sin embargo, un programador puede si desea adaptar la solución anterior para utilizar la recursión por cola. Por lo que una segunda versión del programa sería:

```
factorial(N):- factorial_aux(1,1,N).
factorial_aux(_,R,0):- write('El resultado es: '),write(R).
factorial_aux(_,R,1):- write('El resultado es: '),write(R).
factorial_aux(C,R,N):- C > N, write('El resultado es: '),write(R).
factorial_aux(C,R,N):- C <= N, C1 is C + 1, R1 is R * C, factorial_aux(C1,R1,N).
```

En cuanto a cuál de las dos versiones es la mejor, una vez más dependerá de lo que quiera decidir el programador en su momento.

Parámetros de acumulación

La técnica de parámetros de acumulación se suele utilizar en combinación con la recursividad. Consiste en un argumento auxiliar (o varios de ellos) que almacena la solución parcial en cada paso recursivo. Cuando se llega al caso base, la solución parcial es la solución total.

Ejemplo:

```
conteo_aux(0, Contador, Contador).

conteo_aux(N, Contador, Cuenta) :-
        NContador is Contador+1,
        NN is N -1,
        conteo_aux(NN, NContador, Cuenta).

conteo(N, Cuenta) :-
        conteo_aux(N, 0, Cuenta).
```

En este ejemplo, el valor inicial del parámetro de acumulación es cero. Este valor inicial es importante para que la solución sea correcta. Por eso se ha creado el predicado conteo/2, que se asegura del correcto uso del parámetro de acumulación. El predicado conteo_aux/3 no debería ser ejecutado directamente.

La ventaja del parámetro de acumulación es que genera recursividad de cola, esto es, la llamada recursiva es siempre la última en ejecutarse. Esto permite a los compiladores optimizar considerablemente el uso de recursos ocasionado por la recursividad.

Recomendaciones

Al utilizar cláusulas recursivas hay que tener cuidado de no cometer el error de escribirlas de tal forma que se generen bucles infinitos, o de que Prolog no pueda llegar a la respuesta aún cuando ésta se pueda obtener a partir de los hechos establecidos en la base de conocimiento. Para ello se recomienda escribir las cláusulas en el siguiente orden:

I. Colocar en primer lugar las versiones más sencillas de una cláusula: hechos, casos triviales y las versiones no recursivas de la misma.

II. Debajo coloque las versiones recursivas de la cláusula.

III. Dentro de la cláusula recursiva coloque los objetivos más sencillos primero, es decir, de los que hay menos ocurrencias en la base de hechos.

IV. Luego coloque los objetivos más complicados, es decir, de los que hay más ocurrencias en la base de hechos y los que se resuelven por medio de otras cláusulas de Horn.

V. Por último coloque las llamadas recursivas de la cláusula.

Ejercicios:

1. Defina con sus propias palabras los siguientes conceptos: *recursión, recursión directa y recursión indirecta*.

2. Escriba un programa que realice un conteo ascendente iniciando en el valor N y terminando en M, con ayuda de un proceso recursivo. Hágalo de dos maneras diferentes.

3. Dado un valor entero, contar la cantidad de dígitos que lo componen:

    a) Con validaciones.

    b) Sin validaciones.

4. Generar y desplegar N términos de la serie de Fibonacci a partir de los primeros dos términos, los cuales se le proporcionan al programa:

5. Elabore un programa para realizar un conteo en reversa. El programa debe iniciar en un valor N y finaliza al llegar a cero. Hágalo de dos maneras distintas.

6. Elabore un programa en Prolog que despliegue el número binario equivalente a un número entero positivo.

7. En una base de conocimiento se establece información sobre la cadena alimenticia de ciertas especies de animales, con el predicado come/2, en donde el animal que se especifica en el primer parámetro se come al animal que se especifica en el segundo. Escriba la cláusula *cadena_alimenticia(X, Y)*, que resuelve la interrogante ¿están X y Y en la misma cadena alimenticia?, en donde X está arriba de la cadena y Y abajo.

```
come(alf, gato).
come(raton, escarabajo).
come(escarabajo, heces).
come(mono, guineo).
come(leon, humano).
come(humano, pez).
come(pez, renacuajo).
come(renacuajo, algas).
come(aguila, mono).
come(gato, raton).
```

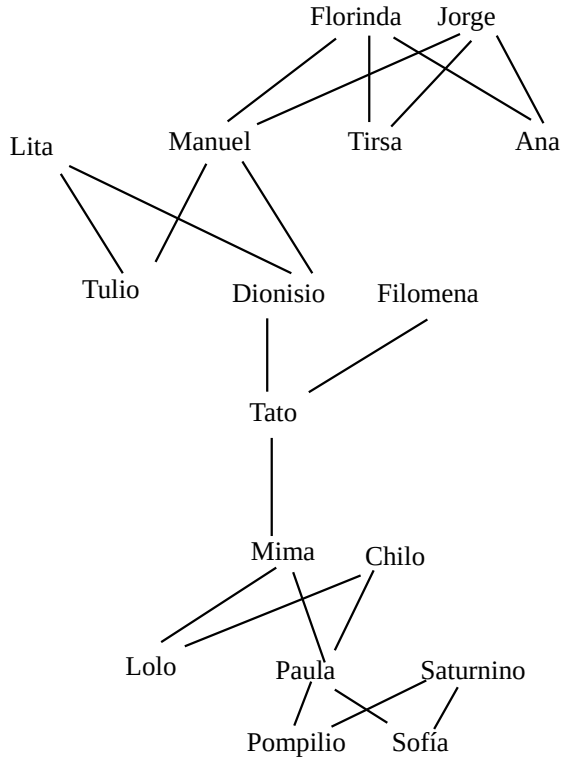
8. Elabore un programa que realice la descomposición factorial de un número entero dado, en sus factores primos. Ejemplo: *descomp_fact(15)*, desplegaría los factores: 3 y 5. El programa debe funcionar para los números N <=50. Establezca hechos *primo/1*, para los números primos inferiores a 25 y la cláusula recursiva *descomp_fact/1*.

9. Dado un número N, generar una nueva cantidad entera que se componga de los mismos dígitos colocados en forma inversa.

10. Dados una cantidad entera y un dígito, contar cuantas veces aparece dicho dígito en dicha cantidad.

11. Dado un valor entero y un dígito, eliminar todas las ocurrencias de dicho dígito en el número entero.

12. De acuerdo al árbol genealógico que se muestra a continuación elabore la cláusula antepasadoDirecto/2, en la que el objeto proporcionado en el primer parámetro es antepasado del objeto proporcionado en el segundo. En la misma base de conocimientos establezca relaciones padre/2 y madre/2, que serán utilizadas en la cláusula solicitada.



13. Elabore una cláusula de aridad dos que recibe como parámetros una cantidad entera y un dígito. Luego elimine de dicha cantidad todos los dígitos que sean menores o iguales al dígito introducido. Establezca procedimientos de validación para ambos parámetros; el primer parámetro debe ser un número entero positivo y el segundo no puede ser mayor que nueve, ni negativo.

Aspectos adicionales sobre la evaluación de funciones

Aun cuando hay lenguajes especializados para la programación de fórmulas, Prolog es una buena alternativa para expresar ideas de lógica matemática. Sin embargo, Prolog es un lenguaje que suele utilizarse más comúnmente para resolver problemas relacionados con la gramática del lenguaje natural.

Por ejemplo, se podrían tener funciones como la siguiente:

La Función Escalonada

$$F(x) = \begin{cases} -1, & x < 0 \\ 1, & x >= 0 \end{cases}$$

El programa Prolog que resuelve esta función es el siguiente:

```
f(X, Y):- X < 0, Y is -1.

f(X, Y):- X >= 0, Y is 1.
```

Existe una manera optimizar la ejecución de los programas, adelantando la ligadura de algunas variables. Esto se hace colocando el término o cantidad con el que se ligará la variable, dentro de los paréntesis en el encabezado de la cláusula. En este ejemplo esto se pude realizar con la variable Y, así:

```
f(X, -1):- X < 0.

f(X, 1):- X >= 0.
```

En los casos anteriores se evalúan ambas cláusulas para cualquier valor de X. Podemos utilizar el corte para evitar esto y optimizar aun más, así:

```
f(X, -1):- X < 0, !.

f(X, 1):- X >= 0.
```

En general, lo que se pretende es permitir que las funciones matemáticas sean ejecutadas con la mayor eficiencia y eficacia posibles.

En base al ejemplo anterior se pueden resumir las siguientes recomendaciones sobre la evaluación de funciones:

1. Se deben separar las secciones de la función matemática a programar en cláusulas. Una cláusula por sección.

2. Se debe optimizar el uso de cada cláusula adelantando el valor de variables en la cabeza de la cláusula (cuando sea posible).

3. Se debe optimizar el proceso total de la resolución mediante el uso de predicados de corte.

Ejercicios:

1. Construya una cláusula de aridad uno, que, dado un valor entero, positivo, calcula la sumatoria $\sum \frac{1}{k}$; 1 <= k <= n.

2. Elabore un programa en Prolog que resuelva la función unitaria de Heaviside:

$$H(x) = u(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

3. Elabore un programa en Prolog que resuelva la función signo:

$$\operatorname{sgn}(x) = \begin{cases} 1, & \text{si } x > 0 \\ 0, & \text{si } x = 0 \\ -1, & \text{si } x < 0 \end{cases}$$

4. Elabore un programa en Prolog para resolver la siguiente función:

$$f(x) = \begin{cases} 2, \text{ si } x \in [-3, 2) \\ -1, \text{ si } x \in (2, 4] \end{cases}$$

5. Elabore un programa en Prolog para resolver esta función escalonada:

$$P(X \leq a) = F(a) = \begin{cases} 0 & \text{para } a < 30 \\ 0,4 & \text{para } 30 \leq a < 50 \\ 0,7 & \text{para } 50 \leq a < 60 \\ 0,9 & \text{para } 60 \leq a < 100 \\ 1 & \text{para } a \geq 100 \end{cases}$$

6. Elabore un programa en Prolog para resolver las siguientes funciones escalonadas:

$$F(x) = \begin{cases} 0, & x < 3 \\ 2, & 3 <= x <= 6 \\ 4, & x > 6 \end{cases}$$

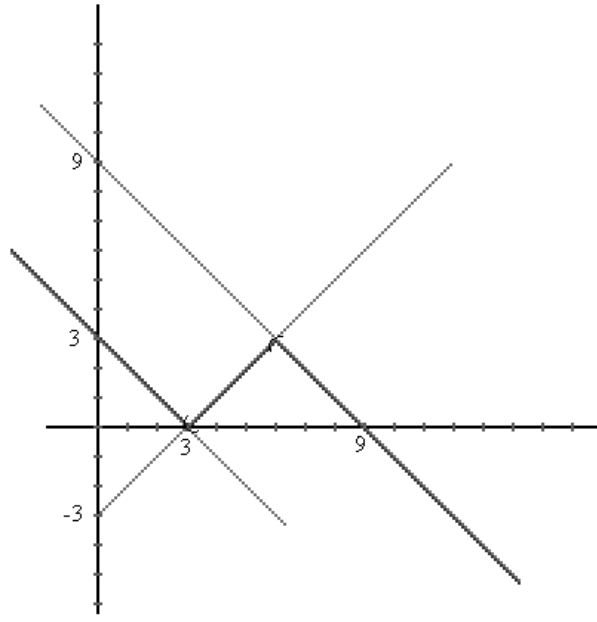
$$F(x) = \begin{cases} -1, & x < -2 \\ 0, & -2 <= x < 2 \\ 1, & x >= 2 \end{cases}$$

7. Elabore un programa en Prolog para resolver la función:

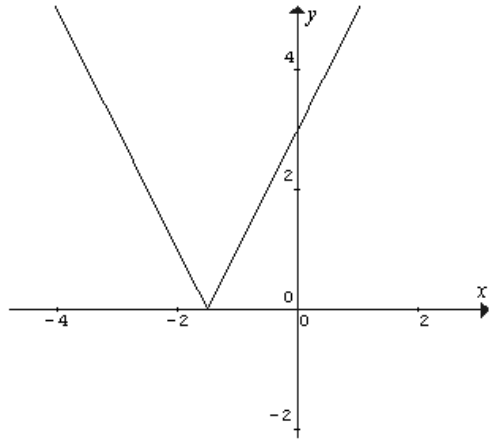
$$f(x) = \begin{cases} 3-x, & x < 3 \\ x-3, & 3 <= x < 6 \\ 9-x, & x >= 6 \end{cases}$$

   Haga una versión sin optimización, y una versión con optimización.

8. Elabore un programa Prolog que resuelva la función que se traza entre las intersecciones de las rectas, tal y como lo muestra la siguiente figura:



9. Elabore un programa en Prolog que resuelva la función que puede observarse en el siguiente gráfico:


y
4
2
0
x
-4
-2
0
2
-2

Manejo de listas en Prolog

Las listas pueden verse como estructuras que son utilizadas por Prolog para la representación de secuencias ordenadas de elementos. Pueden ser utilizadas para agrupar elementos a los que se les va a dar el mismo tratamiento dentro de la ejecución de un programa escrito en Prolog.

La forma de representar listas es por medio de corchetes, [ ], especificando los elementos de la lista en su interior, separándolos por coma:

[<lista de elementos>]

Ejemplos:

[cafe, chocolate, leche]

[a, b, c, d, e]

[perro(fido), gato(misho), raton(bolita)]

[ ]

- La primera lista consta de tres átomos o términos.
- La segunda lista consta de cinco átomos o términos.
- La tercera lista consta de tres hechos.
- El cuarto ejemplo es la forma de representar una lista vacía.

El orden en el que están colocados los elementos sí es importante, así que la lista:

[uno, dos, tres]

no es lo mismo que la lista:

[uno, tres, dos]

Por supuesto que, para poder realizar manipulaciones sobre listas, siempre se requerirá del uso de variables.
Así, por ejemplo para poder manipular la lista

[hugo, jacinta, petrona]

se haría lo siguiente en la consola de Prolog:

```
?- L = [hugo, jacinta, petrona].
```

Se puede incluso, instanciar en una variable una lista vacía:

```
?- X = [ ].
```

De esta forma, una vez contenida la lista en una variable, se puede, por ejemplo, realizar las operaciones básicas con listas, que son:

a) Agregar elementos a la lista.

b) Quitar elementos de la lista.

c) Desplegar los elementos de una lista

Agregando elementos a una lista

Al agregar un elemento a una lista, este elemento debe de agregarse por el frente o inicio de la lista: la *cabeza* de la lista. La cabeza de la lista es el primer elemento de la misma. El resto de la lista se llama *cola* de la lista.

Ejemplo:

1. Agregando a *pepito* en la lista L:

```
?- L = [hugo, jacinta, petrona], Z = [pepito|L].

L = [hugo, jacinta, petrona]
Z = [pepito, hugo, jacinta, petrona]
```

2. Se puede agregar más de un elemento. Ejemplo:

```
?- L = [hugo, jacinta, petrona], Z = [pepito, maria|L].
L = [hugo, jacinta, petrona]
Z = [pepito, maria, hugo, jacinta, petrona]
```

3. Puede permitirse la existencia de listas formadas por distintos elementos:

```
?- Y = [hugo, paco], W is 5, Z = [W|Y].

Y = [hugo, paco]
W = 5
Z = [5, hugo, paco]
```

4. Notar el siguiente ejemplo:

```
?- L1 = [francisco, chico], L2 = [paco, pancho], L3 = [L1|L2].

L1 = [francisco, chico]
L2 = [paco, pancho]
L3 = [[francisco, chico], paco, pancho]
```

   - L1 es una lista formada por dos elementos.
   - L2 es una lista formada por dos elementos.
   - L3 es una lista formada por tres elementos. Observar la forma en que se ha establecido la concatenación: L1 a la cabeza, la cola son los elementos de L2. La cabeza es considerada como un elemento y el primero de la lista.

Es importante mencionar que los elementos de las listas pueden ser heterogéneos, como lo sucedido en el ejemplo anterior. Una lista puede contener cualquier combinación de términos, hechos, relaciones, e incluso, otras listas. Por ejemplo: [1, p(a), [a, b], f(g(h))].

Eliminando elementos de una lista

El primer elemento a eliminar de una lista es el elemento que se encuentra en la cabeza de la lista. Como resultado, después de haber eliminado la cabeza de la lista, se tiene únicamente la cola de la lista.

Ejemplo:

Observar la cláusula:

```
quitarPrimerElemento( [Cabeza|Cola], Cola ).
```

Al invocar desde la consola:

```
?- quitarPrimerElemento([a, b, c, d], Z).
```

Se obtiene:

```
Z = [b, c, d] ;
```

Observar: según el orden de los parámetros, el parámetro actual [a, b, c, d] se unifica con el parámetro formal [Cabeza|Cola]. Así que Cabeza unifica al valor *a* y Cola, que se instancia con la sublista [b, c, d], unifica con Z.

Esta es una forma de manipular las listas ampliamente utilizada en los ambientes de desarrollo de la programación lógica, donde en lugar de definir todos los elementos, se definen únicamente la lista distinguiendo entre la cabeza y el resto: [Cabeza|Cola]. Donde la variable Cabeza representa el primer elemento, y Cola el resto. Por ejemplo:

```
L = [1, 2, 3, 4, 5],
M = [0|L].
```

La lista M sería equivalente a [0, 1, 2, 3, 4, 5]. Es importante no confundir los términos [Cabeza|Cola] y [Cabeza, Cola]. La diferencia es muy sutil:

```
L = [1, 2, 3, 4, 5],
M = [0,L].
```

El resultado sería M = [0,[1, 2, 3, 4, 5]], que es una lista de dos elementos.

Despliegue del contenido de una lista

Utilizando la estrategia de unificación con las variable Cabeza y Cola del último ejemplo, es que pueden realizarse la mayoría de programas que efectúan algún nivel de manipulación de listas. Por ejemplo, un programa en Prolog que despliegue todos los elementos de una lista podría verse como este:

```
desplegarLista([]).

desplegarLista([Cab|Col]):-
        write(Cab),
        write(' '),
        desplegarLista(Col).
```

Cualquier lista que se despliegue utilizando las cláusulas anteriores unificará su primer elemento con la variable Cab y el resto de sus elementos con la variable Col para cada una de las llamadas recursivas del proceso.

Clasificación y búsqueda de elementos en listas

La búsqueda de elementos en una lista es un proceso que trata de determinar la existencia de elementos particulares en el conjunto de datos que conforman las listas.

Con la clasificación se pretende reagrupar los elementos de la lista de acuerdo a ciertas características determinadas.

Ejemplos:

1. Localizar un elemento en una lista e indicar la posición que ocupa de izquierda a derecha (desde la cabeza, siendo la cabeza la posición 1).

```
localizaElemento(L, Elto):-
        localizaElto(L, Elto, 1).

localizaElto([], _, _):-
        write('El elemento no está en la lista'), nl.

localizaElto([Cab|Col], X, Pos):-
         X \== Cab,
        Pos2 is +(Pos, 1),
         localizaElto(Col, X, Pos2).

localizaElto([Cab|Col], X, Pos):-
        X == Cab,
        write('Elemento encontrado en la posicion '),
        write(Pos).
```

   El programa anterior utiliza muchas de las estrategias utilizables en Prolog estudiadas hasta el momento:

   - Utiliza parámetros de acumulación para el conteo de la posición en la que se encuentra en elemento buscado.

   - Utiliza la unificación con variables Cabeza/Cola para un apropiado estudio de cada uno de los elementos de la lista.

   - El uso de variables anónimas en una cláusula para los datos que no interesa conocer.

   - Y el uso de la recursión para el proceso repetitivo del recorrido de la lista.

2. Dada una lista indicar cuantas veces se encuentra en la misma un elemento determinado.

```
cuentaOcurrencias(L, Elto):-
        cuentaOcur(L, Elto, 0).

cuentaOcur([], _, Cont):-
        write('El elemento se encuentra '),
        write(Cont),
        write(' veces').

cuentaOcur([Cab|Col], X, Cont):-
        X == Cab,
        Cont2 is +(Cont, 1),
        cuentaOcur(Col, X, Cont2).

cuentaOcur([Cab|Col], X, Cont):-
        X \== Cab,
        cuentaOcur(Col, X, Cont).
```

Este programa constituye, en general, una modificación del ejemplo anterior, donde los mecanismos de registro de posición en la lista han pasado a convertirse en un contador para los elementos recorridos que cumplen con alguna restricción, en este caso, ser el número X.

Una lista como un conjunto de sub-listas anidadas

Las listas en Prolog podrían definirse como un conjunto de sub-listas anidadas, donde cada sub-lista posee dos elementos:

- El primero es uno de los elementos de la lista.
- El segundo es una sub-lista con el resto de los elementos de la lista. Cada una de estas cumpliendo este mismo formato.
- La sub-lista que contiene el último elemento de la lista posee una lista vacía como sub-lista.

Este comportamiento es parecido a la manera en que funcionan los árboles (otro tipo de estructura que se tratará en secciones posteriores) puesto que los términos se pueden anidar todas las veces que sea necesario.

Ejemplo:

Por ejemplo, la lista de números:

[1, 2, 3, 4, 5]

Puede desglosarse en sub-listas anidadas de la siguiente manera:

[1, 2, 3, 4, 5]

lista(1, 2, 3, 4, 5)

lista(1, lista(2, 3, 4, 5))

lista(1, lista(2, lista(3, 4, 5)))

lista(1, lista(2, lista(3, lista(4, 5))))

lista(1, lista(2, lista(3, lista(4, lista(5, ())))))

De hecho, esta es la manera en que internamente Prolog construye, analiza y manipula las listas. Lo que sucede es que las listas también son términos, pero términos que se escriben de una manera más cómoda. Así, la lista [1,a,2] es en realidad el término '.'(1, '.'(a, '.'(2, []))).

Afortunadamente, Prolog permite un formato predefinido que permite la construcción de programas con una mayor comodidad.

Cadenas de caracteres

Las cadenas de caracteres en Prolog son entendidas como listas de número enteros, números que representan los códigos ASCII de cada uno de los caracteres que conforman la cadena. Afortunadamente, Prolog también permite escribir las cadenas de caracteres en forma cómoda poniendo los caracteres entre comillas dobles.

Por ejemplo, la expresión "ABC" es en realidad la lista [65, 66, 67]. Así, podemos tratar las cadenas de caracteres como cadenas o como listas según sea necesario en la resolución de un problema.

Naturalmente, todo el código construido para la manipulación de listas, puede ser utilizado también para la manipulación de cadenas. Por ejemplo, el predicado que concatena dos listas también sirve para concatenar dos cadenas de texto, el predicado que busca un elemento en la lista puede usarse para buscar caracteres en una cadena, etc.

Ejercicios:

1. Escriba un programa en Prolog que construya una lista incorporando a la misma n repeticiones de un número entero dado.

2. Escriba un programa que genere una lista de los valores enteros existentes en el intervalo [N, M].

3. Elaborar un programa en Prolog que cuente la cantidad de elementos de una lista.

4. Elaborar un programa que indique si una variable contiene o no una lista. Unos ejemplos de uso de este programa serían:

   ```
   ?- esLista(5).
   false.

   ?- esLista([5]).
   true.

   ?- esLista([]).
   true.
   ```

5. Escriba un programa que resuelva el problema de eliminar todas las ocurrencias de un elemento determinado de una lista.

6. Escriba un programa que resuelva el problema de eliminar todas las ocurrencias de un elemento determinado de una lista e indique cuantas veces se eliminó.

7. Escriba un programa que resuelva el problema de eliminar, de una lista, el elemento ubicado en una posición determinada.

8. Elaborar un programa que, a partir de una lista que contiene elementos de diverso tipo, cree una lista que contenga solamente los números enteros.

9. Elabore un programa que concatene dos listas, una a continuación de otra.

10. Elabore un programa que concatene dos listas, los elementos deben ser intercalados como se ve en el ejemplo:

    [a1, b1, c1] [a2, b2, c2] ==> [a1, a2, b1, b2, c1, c2]

    ?-concatenaListas([1, 2, 3], [6, 7, 8], L). ==> [1, 6, 2, 7, 3, 8]

11. Combinar dos listas de tal manera que los elementos se inserten en parejas, colocando el más pequeño antes que el más grande. ejemplo:

    [4, 25, 13] [7, 18, 20] ==> [4, 7, 18, 25, 13, 20]

12. Elabore un programa en Prolog que ordene los elementos de una lista en orden ascendente.

13. Elabore un programa en Prolog que ordene los elementos de una lista en orden descendente.

14. Elabore un programa en Prolog que ordene los elementos de una lista y que le pide al usuario el tipo de ordenamiento: ascendente o descendente (validar la entrada).

15. Elabore un programa en Prolog que invierta los elementos de una lista.

Listas de listas (*Multilistas*)

La existencia de listas de listas conlleva una conclusión importante: los elementos de una lista pueden no ser átomos, sino que también pueden ser otras listas. De hecho, una lista puede contener una mezcla de elementos atómicos y no atómicos, como por ejemplo una lista que contenga número enteros y listas de caracteres.

Ejemplos:

1. [[3, 5, 7], [2, 4, 6, 8]]

   En este ejemplo se tiene una lista de dos elementos, cada uno de los cuales es una lista a su vez: la primer lista es de tres elementos y la segunda lista es de cuatro. El ejemplo puede comprenderse mejor mediante la figura 8:

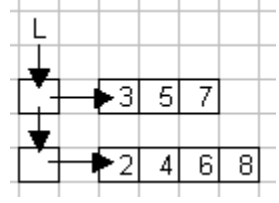


Figura 8. Ilustración de una multilista.

   En la figura se observa que L, la lista del ejemplo, hace referencia a dos elementos, uno consecutivo al otro. Mientras que cada uno de estos elementos hace referencia a una lista de números enteros.

2. [[7, 2, 12], dos, [9, 5], [6, 15, 25], cinco]

   En este ejemplo se tiene una lista de cinco elementos, tres de los cuales son listas, los otros dos son átomos.

3. [[[14, 9], 2, 12], dos, [9, 5, 0], [4, ocho]]

   Es importante hacer notar que los elementos de esta lista son cuatro. El primero de ellos es una lista de tres elementos, seguido de un átomo, y por último dos listas más. La primera de las listas contiene a su vez como elemento a una lista de dos elementos atómicos.

   Es decir, las listas en Prolog son elementos anidables, pudiendo anidar listas en listas tantas veces como sea necesario para la resolución de un problema particular.

Ejemplo:

Elaborar un programa que, dada una lista que puede estar formada por elementos atómicos y otras listas, despliegue todos sus elementos en el orden de aparición.

```
mostrarLista([]).

mostrarLista([Cab|Col]):-
        atomic(Cab),
        write(Cab), write(' '),
        mostrarLista(Col).

mostrarLista([Cab|Col]):-
      \+ atomic(Cab),
      mostrarLista(Cab),
      mostrarLista(Col).
```

El programa anterior analiza elemento por elemento el contenido de la lista de entrada, y por medio de la negación por fallo decide si simplemente mostrar el elemento utilizando un predicado write, o si es necesario realizar una llamada recursiva.

Ejercicios:

1. Insertar, en una lista, elementos que se introducen desde teclado. La inserción termina con 0. Los elementos pueden ser listas.

2. Elabore un programa para manejar una lista que clasifica los elementos que contiene en sublistas de: enteros y reales.

3. Elabore la cláusula listaEnteros(L) que, dada una lista L, afirme o niegue que está formada por números enteros. Retornará una lista R donde cada elemento será una sublista de dos elementos: el elemento de L evaluado y si es o no un número entero.

4. Construya la cláusula promedioLista(L, P) que, dada una lista de números L, instancie P con el promedio de la lista. Si no es posible obtener el promedio, P se instanciará con 0 y se escribirá un mensaje de error. Luego se construirá una lista R donde cada elemento será una sublista de dos elementos: un elemento de L y su diferencia en cuanto al promedio calculado.

5. Elabore una cláusula que, dada una serie de números enteros y reales, que se introducen desde teclado, construya, dentro de la lista LFinal, dos sublistas, de enteros y reales respectivamente. La entrada finaliza cuando se introduce –1. Luego, muestre las dos sublistas.

6. Construya un programa en Prolog que considere una multilista que simule un tablero de ajedrez, es decir, una cuadrícula de 8x8. Usted deberá colocar una reina (‘R’) en cualquier celda del tablero y una ficha adicional (‘F’), en otra. El programa le deberá indicar si la reina puede comerse a la ficha en un solo movimiento. Recordar: la reina puede moverse en línea recta o diagonal, en cualquier dirección.

7. En un tablero de ajedrez coloque la pieza del caballo en la celda formada por la intersección de la fila I y la columna J. Deberá indicar, luego de cada movimiento, las nuevas coordenadas o si el caballo se sale del tablero. El programa termina al introducirle el movimiento 0.

Árboles

Como se estudia en la matemática discreta, los árboles son un caso particular de los grafos, y en Prolog pueden ser manejados implementando listas de una forma especial. Dentro del tema de árboles, un caso particular son los árboles binarios, en donde, a partir de cada nodo del árbol se puede llegar a otros dos nodos como máximo. Este es el tipo de árbol que se estudiará principalmente.

Un árbol binario tiene el siguiente aspecto:

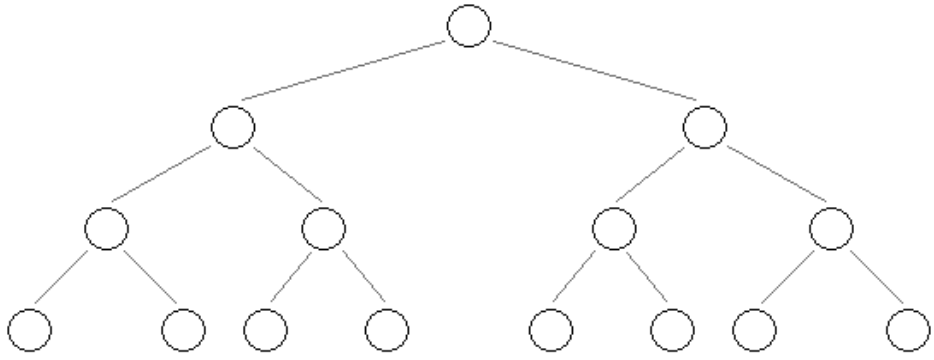


Figura 9. Ejemplo de árbol binario.

Los nodos de un árbol se puede clasificar en niveles, iniciando desde el nivel cero, donde se encuentra el nodo raíz, los hijos primarios se encontrarían en el nivel uno, los hijos de estos en el nivel dos, y así sucesivamente. En la representación gráfico de un árbol, el sentido del arco que une dos nodos siempre es de arriba hacia abajo.

Los árboles son ampliamente utilizados para la realización de búsquedas de elementos y para ejecutar algoritmos de ordenamiento. Tienen gran importancia en la indexación de las tablas de una bases de datos. También tienen aplicación en los procesos que verifican el formato de una expresión, llamados *procesos de evaluación de expresiones*.

En la figura 10, puede apreciarse que cada nodo del árbol posee, en su interior, el dato que almacena, y que a través de ese nodo se puede avanzar hacia los *subárboles* izquierdo y derecho.

Dicha estructura se representaría en Prolog utilizando listas de la siguiente manera:

[dato, subárbol izq, subárbol der]

Una lista de tres elementos que constituyen el elemento raíz, el subárbol izquierdo y el subárbol derecho.

El ejemplo anterior utiliza el formato de representación prefija (o pre-orden). También puede utilizarse el formato de representación infija:

[subárbol izq, dato, subárbol der]

Para ambos casos, *subárbol izq* y *subárbol der* son sublistas que cumplen con esa misma estructura.

Ante lo anterior surge una interrogante: ¿cómo saber si se ha llegado a un nodo que ya no conduce a subárboles izquierdo y derecho, es decir, un nodo que no cuenta con uno o ambos subárboles? Esto se puede solucionar colocando, en lugar de *subárbol izq* y *subárbol der*, el átomo *nulo*.

Ejemplo:

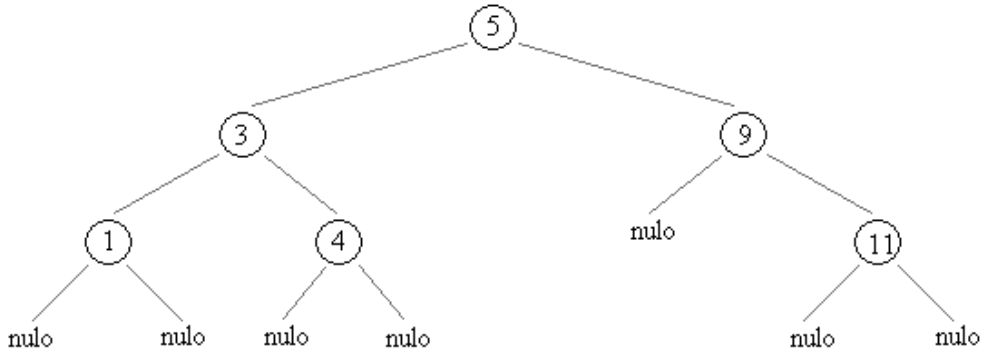


Figura 10. Ejemplo de árbol binario con hojas nulas.

- El subárbol cuya raíz es 11 se representaría: [11, nulo, nulo]
- El subárbol cuya raíz es 9 se representaría: [9, nulo, [11, nulo, nulo]]
- El árbol completo, expresado en una estructura de lista se representaría:

  [[5, [3, [1, nulo, nulo], [4, nulo, nulo]], [9, nulo, [11, nulo, nulo]]]]

Puede inferirse entonces que si un árbol, no importando su extensión, se representa mediante una lista, un árbol vacío se representaría mediante una lista vacía.

Operaciones con árboles

Una vez conocida la estructura de un árbol, tal y como es entendida por Prolog, pueden entonces construirse cláusulas para su manipulación.

Ejemplo:

1. Elabore un programa en Prolog que despliegue el contenido de todos los nodos de un árbol binario. El despliegue debe hacerse en el siguiente orden:

   a) nodo padre – subárbol izquierdo – subárbol derecho.

```
mostrarArbolCID(nulo).

mostrarArbolCID([ArbolIzq, Nodo,ArbolDer]):-
        \==(Nodo, nulo),
        write(Nodo), write(' '),
        mostrarArbolCID(ArbolIzq),
        mostrarArbolCID(ArbolDer).
```

   El ejemplo anterior muestra los principales comportamientos de la manipulación de árboles en Prolog:

   - El tratamiento es estrictamente recursivo: se realiza alguna operación con el nodo raíz del subárbol en evaluación, mientras que para sus hijos izquierdo y derecho se realizan llamadas recursivas.

   - La condición de paro hace uso del elemento *nulo*, ya que es el que indica que durante el recorrido de una rama del árbol se ha alcanzado una hoja, por lo que no es posible realizar ninguna otra operación.

   b) subárbol izquierdo – nodo padre – subárbol derecho.

```
mostrarArbolICD(nulo).

mostrarArbolICD([ArbolIzq, Nodo,ArbolDer]):-
        \==(Nodo, nulo),
        mostrarArbolICD(ArbolIzq),
        write(Nodo), write(' '),
        mostrarArbolICD(ArbolDer).
```

   Nótese como el código es esencialmente el mismo, y sólo se ha cambiado de posición una de las líneas de código.

   c) subárbol derecho – nodo padre - subárbol izquierdo.

```
mostrarArbolDCI(nulo).

mostrarArbolDCI([ArbolIzq,Nodo,ArbolDer]):-
        \==(Nodo, nulo),
        mostrarArbolDCI(ArbolDer),
        write(Nodo), write(' '),
```

```
mostrarArbolDCI(ArbolIzq).
```

Para este caso también sólo se tuvieron que hacer pequeñas modificaciones.

Árbol binario de búsqueda

Se llama árbol binario de búsqueda a un árbol binario en el que los nodos se insertan siguiendo un orden determinado.

Por ejemplo, un árbol binario de búsqueda muy utilizado es el que, para insertar un nodo, se compara el elemento a insertar con cada nodo ya existente en el árbol. Si es menor, se avanza por el subárbol izquierdo, y si es mayor, se avanza por el árbol derecho. Todo nuevo nodo se inserta al final de una ruta.

Para ilustrar mejor el comportamiento de este tipo de árbol se presentan ejemplos para buscar un elemento, para insertar un elemento y para eliminar un elemento.

Ejemplos:

1. Elabore un programa que, dado un número y un árbol binario de búsqueda, realice la búsqueda de dicho número en el árbol.

```
buscarNodo(Dato, nulo):-
        write('El dato: '),
        write(Dato),
        write(', NO se encuentra en el arbol'),
        fail. % Es necesario para indicar que no se logró encontrar el
dato.

buscarNodo(Dato,[_,Dato, _]):-
          write('Se ha ENCONTRADO el dato: '), write(Dato).

buscarNodo(Dato,[ArbolIzq, Nodo, _]):-
          <(Dato, Nodo),
          buscarNodo(Dato, ArbolIzq).

buscarNodo(Dato,[_, Nodo, ArbolDer]):-
          >(Dato, Nodo),
          buscarNodo(Dato, ArbolDer).
```

Nótese cómo se compara el elemento buscado con cada nodo del árbol para saber si se debe continuar la búsqueda por la derecha o por la izquierda. El proceso continua hasta llegar al nodo que contiene el dato buscado, o al encontrar una hoja nula.

2. Elabore un programa que inserte un nuevo nodo en un árbol binario de búsqueda.

```
insertar(Dato, LL):-
        insNodo(Dato, LL, LLFinal).

insNodo(Dato, [], [nulo, Dato, nulo]).

insNodo(Dato,[ArbolIzq, Dato, ArbolDer], [ArbolIzq, Dato, ArbolDer]):-
        write('El Dato YA EXISTE: '), writeln(Dato), nl.

insNodo(Dato, nulo, [nulo,Dato, nulo]).

insNodo(Dato,[ArbolIzq,Nodo, ArbolDer],
                        [ArbolIzq2,Nodo, ArbolDer]):-
        <(Dato, Nodo),
        insNodo(Dato, ArbolIzq, ArbolIzq2).

insNodo(Dato,[ArbolIzq, Nodo, ArbolDer],
                        [ArbolIzq, Nodo, ArbolDer2]):-
        >(Dato, Nodo),
        insNodo(Dato, ArbolDer, ArbolDer2).
```

El programa anterior compara el elemento a ingresar al árbol con cada nodo para decidir si continuar el proceso por la izquierda o por la derecha. Cada vez que toma una decisión, la opción descartada se unifica al árbol resultante, mientras que la opción seleccionada se modifica para albergar al nuevo elemento del árbol.

3. Elabore un programa que elimine un nodo de un árbol binario de búsqueda.

```
eliminar(Dato, L):-
        suprNodo(Dato, L, LFinal),
        mostrarArbolICD(LFinal).

suprNodo(Dato, nulo, _):-
        write('NO existe un nodo con el dato: '), write(Dato), nl.

suprNodo(Dato,[Nodo,ArbolIzq,ArbolDer|_], nulo):-
        Nodo == Dato,
        ArbolIzq == nulo, ArbolDer == nulo.

suprNodo(Dato,[Nodo,ArbolIzq,ArbolDer|_],
                        [Nodo,ArbolIzq2,ArbolDer|_]):-
        Dato < Nodo,
        suprNodo(Dato, ArbolIzq, ArbolIzq2).

suprNodo(Dato,[Nodo,ArbolIzq,ArbolDer|_],
```

```
                                    [Nodo,ArbolIzq,ArbolDer2|_]):-
        Dato > Nodo,
        suprNodo(Dato, ArbolDer, ArbolDer2).

mostrarArbolICD(nulo).

mostrarArbolICD([Nodo,ArbolIzq,ArbolDer|_]):-
        Nodo \== nulo,
        mostrarArbolICD(ArbolIzq),
        write(Nodo), write(' '),
        mostrarArbolICD(ArbolDer).
```

Este último ejemplo utiliza las mismas estrategias de los anteriores, además que agrega cláusulas para mostrar el árbol luego de haber eliminado el elemento solicitado.

Ejercicios:

1. Dado un dato y un árbol binario de búsqueda, localizar el nodo en el árbol que contiene el dato y extraer el subárbol que tiene a este nodo como raíz.

2. La altura de un árbol se calcula en base a la cantidad de nodos recorridos en el camino más largo de la raíz a una hoja en un árbol. Haga un programa en Prolog que calcule la altura de un árbol.

3. Un árbol binario se dice que es equilibrado el número de niveles de sus subárboles izquierdo y derecho no difieren en más de una unidad. Haga un programa en Prolog que evalúe si un árbol es equilibrado.

4. Dada una lista de números enteros, haga un programa que coloque todos los elementos de la lista en un árbol binario de búsqueda.

5. Dado un árbol binario de búsqueda que contiene sólo números enteros, haga un programa que coloque todos sus elementos en una lista.

6. Elabore un programa en Prolog que cuente todos los nodos de un árbol.

7. Elabore un programa en Prolog que invierta un árbol binario, es decir, todos los hijos izquierdos pasan a ser hijos derechos, y viceversa.

8. Dados dos árboles binarios, haga un programa en Prolog que evalúe si los dos árboles son iguales.

9. Elabore un programa en Prolog que traslade el contenido de un árbol binario de números enteros a otro árbol que sea binario de búsqueda.

10. Elabore un programa en Prolog que elimine de un árbol binario de búsqueda todos los elementos que se encuentran en otro árbol binario.

Manejo de archivos

Importancia de los archivos

Los archivos digitales, en cualquier aplicación, adquieren especial relevancia en las organizaciones, ya que son los utilizados para almacenar todos los datos que resultan de las transacciones diarias. Las diversas herramientas de software actuales permiten la creación y administración de archivos que registran cientos de miles o millones de transacciones.

Algunas herramientas para la administración de bases de datos que los alumnos conocerán en el futuro son: MS SQL Server, Oracle, MySQL, PostgreSQL; y algunos de los lenguajes que permiten el manejo de archivos son: Pascal, C, C++, Java, etc.

Prolog permite al programador el poder manejar archivos en disco por medio de una serie de predicados predefinidos.

Apertura de archivos y sus modalidades

Al igual que en C++ y en Java, en Prolog se maneja el concepto de Flujo (stream). Esto es, un movimiento o traslación de bytes que se dan desde o hacia un programa en ejecución. Toda entrada/salida de información que existe se realiza a través de un flujo.

El primer paso para conectarse a un archivo mediante un flujo (stream) es ejecutar una apertura del archivo. La apertura de un flujo se realiza mediante el predicado *open*/3.

El formato de esta predicado es el siguiente:

open(<Archivo físico>, <Modo de apertura>, <Flujo>)

Donde:

- <Archivo físico> es el nombre del archivo en disco que se conectará al flujo.
- <Modo de apertura> puede ser uno de los siguientes:
    - *read*: prepara el flujo para leer de él (entrada de datos).
    - *write*: prepara el flujo para escribir en él (salida de datos). Si el archivo físico conectado al flujo ya tenía información, ésta es borrada.
    - *append*: prepara el flujo para salida de datos, pero coloca el puntero del archivo al final, para añadir información.

- *update*: prepara el flujo para salida de datos, colocando el puntero del archivo de salida al inicio del archivo.

- <Flujo> es una variable que servirá para identificar el archivo dentro del programa.

El paso final, después de haber realizado todas las operaciones de apertura y E/S, consiste en el cierre del archivo. El cierre de un flujo se realiza mediante el predicado *close*(<Flujo>). Si el flujo no ha sido abierto, o si fue cerrado anteriormente, Prolog emitirá un mensaje de error.

Flujos de entrada y salida

Existe un flujo que está conectado a la entrada estándar de datos (la entrada desde teclado), éste se utiliza de forma implícita cuando se le piden valores al usuario desde teclado.

Los flujos de entrada estándar se encuentran definidos dentro de los métodos de entrada/salida de cada lenguaje en particular. Por ejemplo, para Prolog se utiliza el predicado *read*/1, mientras que para C se utiliza la función *scanf*.

Por otro lado, también se maneja otro flujo de manera implícita sin ni siquiera saberlo. Este es la salida de datos estándar, mejor conocido como el monitor de la computadora, aunque a nivel de programación no se tiene una referencia explícita de este otro flujo a pesar de que siempre se utiliza cuando se despliegan mensajes en pantalla.

Los flujos de salida estándar, al igual que en el caso anterior, se encuentran definidos dentro de los métodos de entrada/salida de cada lenguaje en particular. Por ejemplo, para Prolog se utiliza el predicado *write*/1 y para C se utiliza la función *printf*.

Para el manejo de archivos, los flujos de entrada y salida de datos se emplean de manera explícita.

Comandos para entrada y salida de flujos de archivos

Entrada de datos

Se utiliza para proporcionarle al programa datos que provienen de un archivo. El formato de esta instrucción es:

read(<Flujo>, <término>)

Donde:

- <Flujo> es el nombre lógico que le fue asignado al archivo en el momento de la apertura.

- <término> es una variable en la que se depositará el dato proveniente del archivo.

Ejemplo:

1. El siguiente programa obtiene el nombre de una persona desde un archivo y lo muestra en pantalla..

```
leerDeArchivo:-
        write('El nombre almacenado es: '),
        open('nombre.txt', read, DatosPers),
        read(DatosPers, Var),
        close(DatosPers)
        write(Var).
```

   Ejemplo de corrida de este programa:

```
?- leerDeArchivo.
   El nombre almacenado es: 'Guillermo Cortes'.

   true.
```

Salida de datos

Se utiliza para incluir en un archivo datos que provienen de un programa. El formato de esta instrucción es:

write(<Flujo>, <término>)

Donde:

- <Flujo> es el nombre lógico que le fue asignado al archivo en el momento de la apertura.

- <término> es un dato que se depositará en el archivo y que puede estar contenido en una variable.

Ejemplos:

1. El siguiente programa solicita el nombre de una persona y la graba en un archivo en disco.

```
grabarEnArchivo:-
        write('Introduzca su nombre: '),
        read(Var),
        open('nombre.txt', write, DatosPers),
        write(DatosPers, Var),
        close(DatosPers).
```

   Ejemplo de corrida de este programa:

```
?- grabarEnArchivo.
   Introduzca su nombre: Enmanuel Amaya.

   true.
```

   En el disco se podrá encontrar el archivo *nombre.txt*. Al abrirlo se podrá encontrar el valor que se le introdujo a la variable Var.

2. El siguiente programa solicita al usuario el nombre del archivo en disco, el nombre lógico del flujo, y el dato a ingresar al archivo.

```
grabarEnArchivo2:-
        write('Introduzca el nombre del archivo en disco: '),
        read(NomArch),
        write('Introduzca el nombre el Flujo: '),
        read(NomFlujo),
        write('Introduzca su nombre: '),
        read(Var),
        open(NomArch, write, NomFlujo),
        write(NomFlujo, Var),
        close(NomFlujo).
```

   Ejemplos de corridas de este programa:

```
?- grabarEnArchivo1.
   Introduzca el nombre del archivo en disco: 'miNombre.txt'.
   Introduzca el nombre el Flujo: MisDatos.
   Introduzca su nombre: 'Guillermo Cortés'.

   true.
?- grabarEnArchivo1.
   Introduzca el nombre del archivo en disco: 'mis_datos.dat'.
```

```
Introduzca el nombre el Flujo: DatosPers.
Introduzca su nombre: 'Enmanuel Amaya'.

true.
```

Convención de formato para el uso de archivos de texto en Prolog

Algunas cosas de importancia respecto a la entrada desde flujos provenientes de archivos son:

- Cada dato que se encuentre en el archivo de texto debe terminar con un punto.
- Cuando sean leídos por Prolog con la instrucción *read*, éste leerá el flujo hasta encontrar el punto.
- La variable en la instrucción de lectura será instanciada con esta entrada, sin incluir el punto final.
- Para separar una entrada de otra, se ha de utilizar un carácter espacio en blanco, es decir, un símbolo de retorno de carro, un símbolo de tabulación o un espacio en blanco dado con la barra espaciadora.
- Si se desea leer un número real, debe estar escrito tal cual es en el archivo y se le coloca un punto a continuación de la última cifra decimal también.

El contenido de un archivo de texto entendible por Prolog sería el siguiente:

```
10.
20.
25.33.
10.11.
casa.
perro.
gato.
come(carne, león).
```

Para leer un archivo de texto en el que cada línea contenga, por ejemplo, una frase o un nombre, éste debe estar entre comillas simples y terminar en punto. Por ejemplo, un listado de nombre de personas, direcciones y teléfonos, se vería así:

```
‘Perico Palotes’.
‘Col. Las Arboledas, Pje. Ramilletes, #25’.
‘2255-5555’.
‘Alam Brito’.
```

'Col. El Progreso, Pje. La alhambra, #10'.
'2257-7777'.
'Mari Posas'.
'Col. Jardines, Av. Fragancias, # 1'.
'2233-4455'.

Realización repetida de instrucciones de lectura en archivos

En un proceso repetido de lectura, en el que es posible agotar la entrada, se llegaría al final del archivo sin tener más datos que leer. El lenguaje proporciona predicados que se pueden utilizar previendo estas situaciones al escribir programas que leen repetidamente desde un archivo. El átomo predefinido que proporciona el lenguaje para verificar este tipo de situaciones es: *end_of_file*. Su funcionamiento es el siguiente:

Cuando se lee de un flujo de entrada, la variable en la instrucción de lectura se instancia con el valor proveniente del archivo. A su vez, el puntero del archivo avanza hacia el siguiente caracter que se va la leer. Si se ha llegado al final del archivo y se ejecuta de nuevo una lectura, la variable no puede adquirir un valor proveniente del archivo, así que Prolog la instancia con el átomo *end_of_file*. De esta forma, se puede verificar el contenido de la variable repetidamente, luego de cada lectura, para determinar la acción a tomar.

Ejemplos:

1. Ejemplo del uso del átomo *end_of_file*:

```
leerDeArchivoVacio:-
        open('listaVacia.txt', read, MiArchivo),
        read(MiArchivo, Dato),
        probar(Dato),
        close(MiArchivo).

probar(Dato):-
        ==( Dato, end_of_file),
        writeln('Se ha llegado al final del archivo').

probar(Dato):-
        \==( Dato, end_of_file),
        writeln('Todavía hay datos que leer en el archivo').
```

2. Leyendo las filas de un archivo y desplegándolas en pantalla:

```
leerDeArchivo:-
```

```
        open('lista_numeros.txt', read, ListaNumeros),
        lectura(ListaNumeros),
        close(ListaNumeros).

lectura(Flujo):-
        read(Flujo, Valor),
        probar(Valor).

probar(Valor):-
        \==( Valor, end_of_file),
        write(Valor), nl,
        lectura(Flujo).

probar(Valor):-
        ==( Valor, end_of_file),
        write('Se agotaron los datos'), nl.
```

Para escribir cada dato en una línea distinta de un archivo de texto se utiliza el símbolo '*n'*. Ejemplo:

```
write(Flujo, Valor),
write(Flujo, '.'),
writeFlujo, '\n'),
```

'*n'* es el símbolo de nueva línea. Este símbolo también se utiliza en otros lenguajes de programación como C, C++ y Java.

Ejemplo:

Dada una lista de números enteros, incorporarla a un archivo de texto en el que cada dato se encuentre en una línea distinta. Proporcione la lista por medio de un parámetro y solicite el nombre del archivo desde teclado.

```
grabarLista(L):-
        write('Digite el nombre del archivo destino: '),
        read(NomArch),
        open(NomArch, write, ListaNumeros),
        meteNumeros(ListaNumeros, L),
        close(ListaNumeros).

meteNumeros(_, []).

meteNumeros(Flujo, [Cab|Col]):-
        write(Flujo, Cab),
```

```
write(Flujo, '.'),
write(Flujo, '\n'),
meteNumeros(Flujo, Col).
```

Si no se utilizaran cambios de línea, el aspecto del archivo sería el siguiente:

```
10.20.30.40.
```

Pero al intentar realizar alguna lectura Prolog emitiría un mensaje de error.

Lectura y escritura caracter a caracter

Para leer un archivo caracter por caracter se utiliza el predicado *get/2*. Su formato de utilización es:

get(<Flujo>, <término>)

Donde:

- <Flujo> determina el flujo de donde proviene la entrada de datos.
- <término> es la variable que se instancia con el valor proveniente de la entrada. A diferencia de read, get proporciona a la variable el valor ASCII del caracter leído. Si se llega al final del archivo y se realiza una lectura, la variable se instancia con –1, que no es un valor ASCII.

Cuando se realiza la lectura de un caracter, el puntero del archivo avanza hacia el siguiente caracter válido, saltándose los caracteres espacio en blanco que encuentre. Los caracteres espacio en blanco son: retorno, caracteres de tabulación y el espacio en blanco.

Para escribir caracter a caracter en un archivo se utiliza el predicado *put/2*. Cuyi formato de utilización es:

put(<Flujo>, <término>)

Donde:

- <Flujo> determina el flujo hacia donde se dirige la salida.
- <término> es valor ASCII del caracter que se grabará en el archivo de salida.

Ejemplo:

Elabore un programa que realice una copia del contenido de un archivo hacia otro.

```
copiarArchivo:-
        open('archivoIn.txt', write, In),
        open('archivoOut.txt', read , Out),
        get(Out, C),
        copiar(C, In, Out),
        close(In),
        close(Out).

copiar(-1,_,_).

copiar(C, In, Out):-
        put(In, C),
        get(Out, C1),
        copiar(C1, In, Out).
```

El programa anterior colocaría el contenido de *archivoOut.txt* en *archivoIn.txt*. Sin embargo, dado que el predicado *get* omite los espacios en blanco, el contenido no sería copiado correctamente.

Lectura de caracteres con get_char/2

A diferencia de get/2, el comando get_char/2 lee todos los caracteres del archivo sin importar que sean caracteres espacio en blanco, *get_char* los interpreta como el caracter en sí, y no como el valor ASCII. Además, *get_char* instancia con *end_of_file* al encontrar al final del archivo, y no con –1 como lo hace *get*. Su formato de utilización es:

get_char(<Flujo>, <término>)

Donde:

- <Flujo> determina el flujo de donde proviene la entrada de datos.
- <término> es la variable que se instancia con el valor proveniente de la entrada.

Escritura de carácteres con put_char/2

Al igual que get_char, el predicado put_char/2 trabaja con los caracteres tal y como son, sin utilizar en ningún momento su código ASCII. Su formato de utilización es:

put_char(<Flujo>, <término>)

Donde:

- <Flujo> determina el flujo hacia donde se dirige la salida.

- <término> es carácter que se grabará en el archivo de salida.

Utilizando estos predicados, el ejemplo anterior puede ser modificado de la siguiente manera:

```
copiarArchivo2:-
        open('archivoIn.txt', write, In),
        open('archivoOut.txt', read , Out),
        get_char(Out, C),
        copiar(C, In, Out),
        close(In),
        close(Out).

copiar(end_of_file, _, _).

copiar(C, In, Out):-
        put_char(In, C),
        get_char(Out, C1),
        copiar(C1, In, Out).
```

Donde esta vez, el contenido de *archivoOut.txt* quedaría fielmente copiado en *archivoIn.txt*.

Otros comandos útiles para el manejo de archivos

Prolog proporciona otra buena cantidad de predicados predefinidos para la manipulación de archivos. Entre los más importantes se pueden mencionar:

Predicado	**Significado**
delete_file(<Archivo>)	Borra el archivo cuyo nombre recibe como parámetro.
exists_file(<Archivo>)	Verifica la existencia del archivo cuyo nombre recibe como parámetro.
rename_file(<Archivo>, <Nombre>)	Cambia el nombre del archivo que se especifica en el primer parámetro, por el nombre del archivo que se especifica en el segundo parámetro.
size_file(<Archivo>, <Tamaño>)	Unifica <Tamaño> con la cantidad de bytes que contiene <Archivo>.
absolute_file_name(<Archivo>, <Nombre absoluto>)	Unifica <Nombre absoluto> con el nombre absoluto, es decir, incluye el nombre y ruta absoluta de <Archivo>.

file_directory_name(<Archivo>, <Directorio>)	Unifica el segundo parámetro con la parte del nombre de <Archivo> correspondiente a la ruta de directorios.
file_base_name(<Archivo>, <Nombre>)	Unifica el segundo parámetro con la parte del nombre de <Archivo> separada de la ruta de directorios.
exists_directory(<Directorio>)	Retorna éxito si la ruta de directorios especificada existe, o fallo en caso contrario.

Ejemplos:

1. borrarArchivo(Archivo):-
   delete_file(Archivo).

   ?- borrarArchivo('notas_finales.xls').

2. cambiarNombre(Archivo, NvoNombre):-
   rename_file(Archivo, NvoNombre).

   ?- cambiarNombre('archivo_prueba.txt', 'pruebas.txt').

3. longArchivo(Archivo, Longitud):-
   size_file(Archivo, Longitud).

   ?- longArchivo('pruebas.txt', X).

   X = 21

4. ?- nombreAbsoluto('pruebas.txt', N).

   N = 'c:/documents and settings/all users/escritorio/prolog/ejercicios/pruebas.txt'

5. directorioDelArchivo(Archivo, Directorio):-
   file_directory_name(Archivo, Directorio).

   ?- directorioDelArchivo('c://mi_directorio//programas//pruebas.txt', D).

   D = 'c://mi_directorio//programas/'

6. nombreDelArchivo(Archivo, Nombre):-
   file_base_name(Archivo, Nombre).

   ?- nombreDelArchivo('c://mi_directorio//programas//pruebas.txt', N).

```
N = 'pruebas.txt' ;
```

7.
```
verificarDirectorio(Directorio):-
                        exists_directory(Directorio).
```

```
?- verificarDirectorio('c://mi_directorio//programas').

true.

?- verificarDirectorio('c://mi_directorio//prog').

false.
```

Ejercicios:

1. Dado un archivo que contiene una lista de números, realizar lecturas de cada elemento para irlos anexando a una lista. El contenido de la lista debe de verse así:

```
10.
20.
30.
40.
50.
```

2. Elaborar un programa que cuente la cantidad de líneas de un archivo de texto.

3. Se requiere de un programa que despliegue una frase aleatoria de bienvenida cuando se ingresa a una red en un recinto universitario. Las frases "célebres" se coleccionan en un archivo de texto de 100 líneas. Elabore un programa en Prolog que genere un número entero aleatorio, en el rango de 1 y 100, lea la frase de la línea correspondiente a ese número y la despliegue en pantalla. (*Investigar sobre el predicado random/1*)

4. El administrador de una red colecciona diferentes tipos de frases, las cuales clasifica en: dichos, consejos, insultos y frases célebres. Las frases las guarda en cuatro archivos de texto dependiendo de su clasificación. Hay un quinto archivo, llamado coleccionFrases.txt cuyo contenido es:

```
'dichos.txt'
##
'consejos.txt'
##
'insultos.txt'
```

##
'frases_celebres.txt'
##

En donde el ## debajo de cada nombre de archivo es la cantidad de líneas que cada uno contiene. Elabore un programa en Prolog, que genere un número de 1 a 4 para seleccionar el tipo de frase a desplegar. Luego genere un número entre 1 y la cantidad de frases contenidas en el archivo elegido. Finalmente, despliegue la frase correspondiente.

5. Dado un archivo de texto con información acerca de individuos: nombre completo, carnet y edad, elabore un programa que lea la información de dicho archivo y la coloque en una multilista en la que, cada sublista contiene nombre, carnet y edad de cada persona.

6. Dado un árbol binario de búsqueda realizar el recorrido en pre-orden (hijo izq – padre – hijo der) y almacenar todos los nodos del árbol en un archivo, cada nodo en una línea del archivo, terminando con punto.

7. Dado un árbol binario de búsqueda realizar la búsqueda de un elemento y guardar el recorrido que se realiza desde la raíz, en un archivo, ya sea que se encuentre o no el nodo buscado.

8. Las temperaturas diarias de una zona desértica se han almacenado en archivos, uno por cada día de la semana, nombrados temp_lunes.txt, temp_martes.txt, ..., temp_domingo.txt. En cada archivo hay cinco valores de temperatura tomados a las horas: 07:00 am, 10:00 am , 12:00 m, 3:00 pm, 5:00 pm y 7:00 pm. Los nombres de archivos y las horas de recolección de datos están almacenados en días.txt y horas.txt respectivamente. Elabore un programa que calcule:

    a) La temperatura promedio diaria para cada día de la semana.

    b) La temperatura promedio para cada hora establecida.

    Luego muestre los resultados en pantalla.

9. Elabore la cláusula *muestraLineaN(Arch, Nlinea, Linea)* que, dado el nombre de un archivo y un número de línea, ligue la variable *Línea* con el contenido de la línea *Nlinea* de ese archivo. El archivo almacena frasea románticas. ¿De qué manera esperaría usted que las frases estuvieran almacenadas?, escriba un ejemplo con cuatro frases.

10. Elabore un programa que abra un archivo que contiene diversos tipos de datos, lea su contenido y lo almacene procurando clasificarlo así: a) un archivo donde solo se almacenen números enteros, b) un archivo donde solo se almacenen números reales, c) un archivo donde se almacene otro tipo de contenido. Cada dato en uno de estos tres nuevos archivos sepárelo por líneas.

Bibliografía

- Iranzo, P. J., *Lógica Simbólica para Informáticos*, Alfaomega Grupo Editor, Primera edición, México, 2005.

- Iranzo, P. J.; Frasnedo, M., *Programación Lógica, Teoría y Práctica,* Pearson Prentice Hall, Primera edición, España, 2007.

- Clocksin, W., *Programación en Prolog*, Editorial Gustavo Gili, Segunda edición, España, 1993.

- Ferrater, J.; LeBlanc, H., *Lógica Matemática*, Fondo de Cultura Económica, Primera Edición, España, 1975.

- Canedo, J,, *Lógica formal y simbólica*, Producciones Cima, Vigésima octava edición, Bolivia, 1996.

- http://www.swi-prolog.org