Paradigma Lógico

Paradigmas de Programación

Facultad Regional Buenos Aires
Universidad Tecnológica Nacional

Ing. Lucas Spigariol

Buenos Aires – 2008

Presentación

¿Por qué Lógico?

Desde los orígenes de las ciencias de la computación, la imitación del razonamiento humano ha sido uno de sus grandes desafíos. La posibilidad de que un artefacto artificial, creado por el hombre, pueda comportarse, y sobre todo, "pensar" como una persona humana, no sólo alimentó el imaginario del hombre común mediante la literatura y el cine de ciencia ficción durante años, sino que tuvo su correlato científico en el ámbito de la informática, con serias investigaciones que han arrojado resultados sorprendentes.

En la década del 70, el origen del paradigma de la programación lógica estuvo íntimamente relacionado con este objetivo. Un gran entusiasmo, acompañado de una expectativa creciente respecto de sus alcances y posibilidades, le dio un significativo envión inicial. La evolución posterior del paradigma fue demostrando la validez del camino emprendido, pero el proceso fue y es en la actualidad, más complejo, tal vez, de lo que esperaban sus creadores y principales impulsores. En este sentido, los objetivos han ido variando: Por un lado, los lenguajes lógicos han incorporado características que los transforman en lenguajes de propósito general, con posibilidades de aplicación en los más variados campos informáticos, y por otro, el desarrollo de la "inteligencia artificial" manteniendo la centralidad de las premisas lógicas se ha diversificado incorporando nuevas herramientas de software y hardware.

El objetivo de este trabajo es presentar con claridad y sencillez los principios fundamentales que constituyen el paradigma lógico, como modesta introducción para la comprensión de una forma particular de programación que tiene un potencial aún no plenamente desarrollado. A quienes estén muy acostumbrados a trabajar siguiendo la mecánica de los paradigmas imperativos, o inclusive de objetos, tal vez les resultarán extraños algunas de los conceptos y estrategias que se utilizan, pero una vez entendida su fundamentación teórica y su filosofía de funcionamiento, propia de los paradigmas declarativos, descubrirán lo sencillo y consistente del modelo y se sorprenderán de cómo es posible que con un conjunto relativamente reducido de herramientas se puedan desarrollar un

sinnúmero de aplicaciones. Las características del paradigma no sólo permiten resolver problemas concretos, sino que, como podemos afirmar quienes desde hace años trabajamos con lenguajes lógicos, promueven el desarrollo de formas alternativas de razonamiento y planteo de soluciones, cuya utilidad y aplicación se extiende a numerosas problemáticas informáticas más allá de los límites del propio paradigma.

Para ilustrar y ejemplificar las explicaciones hemos elegido el Prolog, el lenguaje emblemático de la programación lógica que nació de manera indisoluble con el mismo paradigma. Un esquema claro de formulación de declaraciones y una implementación simple y consistente del backtracking como mecanismo de evaluación de reglas lógicas, hacen del Prolog una herramienta lo suficientemente potente como para desarrollar soluciones de los más variados y complejos problemas de aplicación.

En los primeros capítulos se desarrollan los principales conceptos del paradigma lógico, con ejemplos prácticos y luego, se presenta en detalle las características del lenguaje Prolog, con su sintaxis y semántica.

El presente material es una actualización de otros textos que anteriormente había elaborado, con nuevos ejemplos, ampliaciones conceptuales y ejemplos renovados, elaborado en base a mi experiencia de 15 años como docente en la carrera de Ingeniería en Sistemas de Información, en la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional, y en particular, en la cátedra de Paradigmas de Programación, desde su creación en el marco del plan de estudios de 1995.

El objetivo principal de este trabajo, es que en sus manos se convierta en una herramienta de utilidad para su formación permanente y su ejercicio profesional.

Ing. Lucas Spigariol

Capítulo 1

Conceptos generales

- **El paradigma lógico**
- **Principales características**
- **Campo de aplicación**
- **Historia y lenguajes**
- **PROLOG**

El paradigma lógico

Tiene como característica principal la aplicación de las **reglas de la lógica** para inferir conclusiones a partir de datos. Conociendo la información y las condiciones del problema, la ejecución de un programa consiste en la búsqueda de un objetivo dentro de las declaraciones realizadas. Esta forma de tratamiento de la información permite pensar la existencia de "**programas inteligentes**" que puedan responder, no por tener en la base de datos todos los conocimientos, sino por poder inferirlos a través de la **deducción**.

La importancia del concepto de declaratividad en este paradigma, permite encuadrarlo dentro de los **paradigmas declarativos**[1]**.** Al separar el control y la lógica, el programa se transforma en un conjunto de declaraciones formales de especificaciones que deben ser correctas por definición.

Un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado, sino que está formado por expresiones que describen la solución (o más precisamente, la "declaran"). De esta manera, la clave para hacer un programa lógico es poder explicitar una declaración que describa correctamente la solución del problema.

[1] El concepto de declaratividad y su vinculación con la organización de los paradigmas se puede ampliar en SPIGARIOL, Lucas. *Conceptos fundamentales de los Paradigmas de Programación.* Apunte CEIT

Principales características

El paradigma tiene sus fundamentos en las teorías de la **lógica proposicional**[2]. De ellas, se toman en particular las **Cláusulas de Horn**, que son una forma de lógica de predicados con una sola conclusión en cada cláusula y un conjunto de premisas de cuyo valor de verdad se deduce el valor de verdad de la conclusión: una conclusión es cierta si lo son simultáneamente todas sus premisas.

Por su **esencia declarativa**, un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado, sino que es el sistema internamente el que proporciona la secuencia de control.

No existe el concepto de asignación de variables, sino el de **unificación**. No hay un "estado" de las variables que se vaya modificando por sucesivas asignaciones, generalmente asociadas a posiciones de memoria**,** sino que las variables asumen valores al **unificarse** o "ligarse" con valores particulares temporalmente y se van sustituyendo durante la ejecución del programa[3].

Un programa lógico contiene una **base de conocimiento** sobre la que se hacen **consultas**. La base de conocimiento está formada por **hechos**, que representan la información del sistema expresada como **relaciones entre datos**, y por **reglas lógicas**[4] que permiten deducir consecuencias a partir de combinaciones entre los hechos y, en general, otras reglas. Se construye especificando la información del problema real en una base de conocimiento en un lenguaje formal y el problema se resuelve mediante un mecanismo de inferencia que actúa sobre ella. Así pues, una clave de la programación lógica es poder expresar apropiadamente todos los hechos y reglas necesarios que definen el dominio de un problema.

En otros paradigmas, las salidas son funcionalmente dependientes de las entradas, por lo que el programa puede verse abstractamente como la implementación de una transformación de entradas en salidas. En cambio, la programación lógica está basada en la noción de que el programa implementa una **relación**, en vez de una transformación. Los predicados son relaciones, que al no tener predefinido una "dirección" entre sus componentes, permiten que **sus argumentos actúen indistintamente como argumentos de entrada y salida**. Esta característica se denomina **inversibilidad**[5]. A su vez, a diferencia de las funciones donde está la restricción del a unicidad de la imagen para un elemento determinado del dominio, una relación permite vincular a cada elemento con muchos otros elementos, permitiendo **soluciones alternativas**. Dado que las relaciones son más generales que las transformaciones, la programación lógica es potencialmente de **más alto nivel** que la de otros paradigmas.

[2] Ver capítulo 2 "Fundamentos lógicos"
[3] Ver capítulo 3 "Unificación"
[4] Ver capítulo 4 "Reglas de inferencia"
[5] Ver capítulo 5 "Inversibilidad"

Internamente, existe un mecanismo, un "motor", que actúa como **control de secuencia**[6]. Durante la ejecución de un programa va evaluando y combinando las reglas lógicas de la base de conocimiento para lograr los resultados esperados. La implementación del mecanismo de evaluación puede ser diferente en cada lenguaje del paradigma, pero en todos los casos debe garantizar que se agoten todas las combinaciones lógicas posibles para ofrecer el conjunto completo de respuestas alternativas posibles a cada consulta efectuada. El más difundido se denomina **backtracking**, que utiliza una estrategia de búsqueda primero en profundidad.

La **recursividad**[7] como estrategia lógica para encontrar soluciones, junto con la utilización de **listas**[8] para representar conjuntos de valores, son dos características típicas de los programas lógicos.

Los lenguajes del paradigma lógico, en general incluyen herramientas para realizar **soluciones polimórficas** y manejar el concepto de **orden superior**[9]**,** entendido como la capacidad de un lenguaje para manejar su propio código como una estructura de datos más. Son un conjunto de funcionalidades que dotan de una **enorme expresividad y potencia a los programas**.

Existen también otras herramientas más complejas, como las que buscan incrementar la eficiencia o las que abren a la posibilidad de meta programación, que requieren de una cuidadosa utilización ya que se introducen en el interior del sistema mismo y permiten alterar la naturaleza declarativa del paradigma[10].

Campo de aplicación

El paradigma es ampliamente utilizado en las aplicaciones que tienen que ver con la **Inteligencia Artificial**, particularmente en el campo de sistemas expertos y procesamiento del lenguaje humano.

Un **sistema experto** es un programa que imita el comportamiento de un experto humano. Por lo tanto contiene información (es decir una base de conocimientos) y una herramienta para comprender las preguntas y encontrar la respuesta correcta examinando la base de datos (un motor de inferencia).

En el caso del **procesamiento del lenguaje humano** se trata de dividir el lenguaje en partes y relaciones y tratar de comprender su significado. Para plantear los problemas en términos del paradigma se definen reglas lógicas entre las diferentes partes.

También es útil en **problemas combinatorios** o que requieren gran cantidad o amplitud de soluciones alternativas, dada la naturaleza combinatoria del

[6] Ver capítulo 6 "Mecanismos de evaluación"
[7] Ver capítulo 7 "Estrategias de resolución"
[8] Ver capítulo 8 "Functores y listas"
[9] Ver capítulo 9 "Polimorfismo y orden superior"
[10] Ver capítulo 10 "Ejecución dinámica y control"

mecanismo de backtracking y como lenguaje de **prototipación y programación exploratoria**.

Otros casos de utilidad práctica del paradigma son:

- Paralelización automática de programas.
- Programación distribuida y multiagente.
- Validación automática de programas.
- Prototipado rápido de aplicaciones.
- Bases de datos deductivas.
- Acceso a bases de datos desde páginas Web.

Historia y lenguajes

La base conceptual de la lógica proposicional es desarrollada por Alfred Horn, en los años 50, en forma independiente al desarrollo computacional, al publicar "Sobre sentencias las cuales son verdaderas de la unión directa de las álgebras", en la cual presenta un modelo lógico para el tratamiento de oraciones del lenguaje natural, donde se explican las luego denominadas "cláusulas de Horn".

En la primera mitad de la década del 70, en base a las cláusulas de Horn, surgen las primeras versiones de lenguajes lógicos como una herramienta para resolver ciertos problemas en el área de la inteligencia artificial, originalmente vinculados al tratamiento computacional del lenguaje natural. Más concretamente, en 1972, en la **Universidad de Marsella**, **Alain Colmerauer**, **Philippe Roussel** y un grupo de investigadores presentan el lenguaje de programación lógica Prolog. Luego se va perfeccionando el lenguaje y se escribe el compilador.

Pocos años después se definen los principios que constituyen al Paradigma Lógico como un paradigma de programación, con la participación determinante de **Robert Kowalski**.

Inicialmente se trataba de un lenguaje totalmente interpretado hasta que, a mediados de los 70, David H.D. Warren desarrolló un compilador capaz de traducir **Prolog** en un conjunto de instrucciones de una máquina abstracta denominada Warren Abstract Machine, o abreviadamente, WAM. Desde entonces **Prolog** es un lenguaje semi-interpretado.

En 1979, en la **Universidad de Edimburgo** se escribe un nuevo compilador para el **Prolog**, adaptándolo al paradigma lógico planteando la ecuación Lógica + control + estructuras de datos = programas y se sigue perfeccionando el lenguaje.

Aunque con ciertas dificultades iniciales, debido principalmente a la novedad del paradigma y a la escasa eficiencia de las implementaciones

disponibles, el lenguaje se fue expandiendo rápidamente, sobre todo en Europa y en Japón. **Prolog** recibió un gran empuje en 1981, cuando el Instituto Japonés para la Nueva Generación de Tecnología Informática eligió a la programación lógica como su tecnología de software diferencial, y comenzó un proyecto para proveer la tecnología de hardware complementaria en la forma de máquinas de inferencia lógica rápidas (proyecto de ordenadores de quinta generación).

En 1995, el lenguaje se normaliza con el correspondiente estándar ISO. En la actualidad **Prolog** se ha convertido en una herramienta de desarrollo de software práctica y de gran aceptación para la que se dispone de múltiples compiladores, tanto comerciales como de libre distribución.

PROLOG

El **Prolog** es el lenguaje emblemático del paradigma lógico. Su evolución no puede separarse de la historia misma del paradigma.

El lenguaje **Prolog** no fue realmente diseñado como tal, sino que más bien fue evolucionando mayoritariamente en las Universidades de Marsella y Edimburgo, como una herramienta experimental de inteligencia artificial. Debido a la falta de una definición común, varios dialectos de **Prolog** evolucionaron. De ellos, el dialecto de Edimburgo es aceptado ampliamente como estándar.

Desde entonces, el lenguaje **Prolog** ha sido implementado con diferentes versiones y desarrollado por distintos fabricantes, manteniendo la misma base del lenguaje, pero con pequeñas variantes y herramientas adicionales para facilitar la programación.

En este trabajo, se utiliza como ejemplo el lenguaje **Prolog** en la versión desarrollada por la Universidad de Ámsterdam, denominada **SWI-Prolog**[11], que se basa en el dialecto de Edimburgo.

[11] Swi-Prolog (Multi-threaded, Version 5.4.7) Más información en **www.swi-prolog.org**

Capítulo 2

Fundamentos lógicos

- **Lógica proposicional**
  - Relaciones
  - Consultas
- **Cláusulas de Horn**

Lógica proposicional

El paradigma tiene sus fundamentos en las teorías de la **lógica proposicional**. De ella, se toma un tipo especial de lógica conocido como "lógica de predicados de primer orden".

A diferencia de otros paradigmas que se basan en la idea de "transformación" de datos, la programación lógica se fundamenta en la noción de "relación".

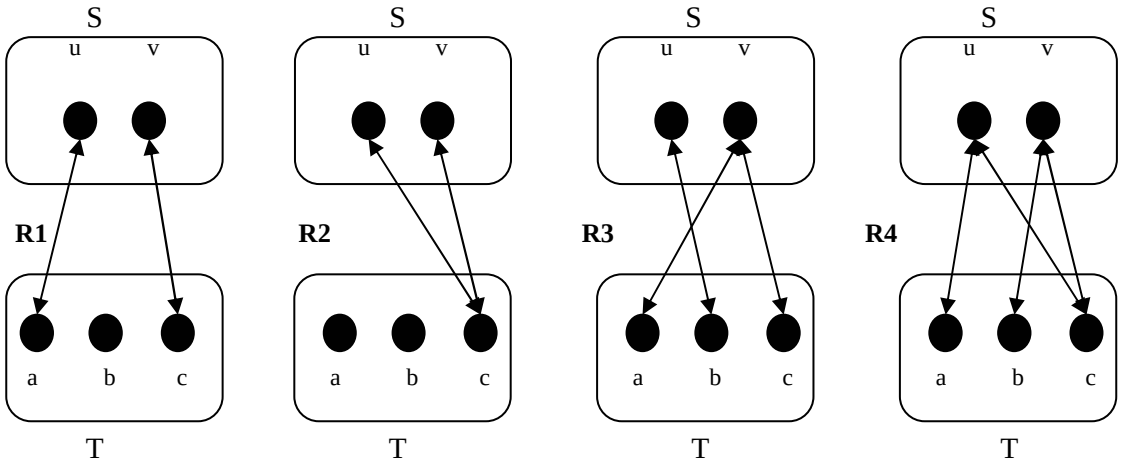
Relaciones

Si se consideran dos conjuntos de valores S y T, R es una relación entre S y T si, para cada x ∈ S e y ∈ T, R(x, y) es o verdadera o falsa. Si R(x, y) es verdadera, se dice que R se mantiene entre x e y.

Ejemplo:

Dados dos conjuntos S = {u, v} y T = {a, b, c}. Una flecha bidireccional conecta cada par de valores entre los cuales se mantiene una determinada relación.

La Relación R1 se mantiene sólo entre u y a y entre v y c. Las relaciones R1 y R1 tienen la propiedad de que cada x en S se corresponde exactamente con una y en T. Generalizando, sin embargo, una relación entre S y T puede mantenerse entre una x dada en S y varias y en T. Esas relaciones son R3 y R4.



Una diferencia importante entre el concepto de relación y el de transformación que utilizan las funciones, es que una relación no sólo es de uno a uno o de muchos a uno, sino que puede ser también de uno a muchos o de muchos a muchos. El concepto matemático de **relación** es **más amplio** que el de **función**.

Las relaciones no son sólo binarias, es decir, relaciones entre pares de elementos, sino también se puede hablar de relaciones unarias, escritas como R(x), relaciones ternarias, escritas como R(x, y, z), y así sucesivamente.

Consultas

Habiendo implementado una relación R, por ejemplo que sea binaria, podemos hacer consultas del tipo:

1. Dado un elemento en particular a y uno b, determinar si R(a, b) es verdadera.[12]
2. Dado un elemento en particular a, encontrar todos los valores posibles de X tal que R(a, X) es verdadera. [13]
3. Dado un elemento en particular b, encontrar todos los valores posibles X tal que R(X, b) es verdadera.
4. Encontrar todos valores posibles X e Y tal que R(X, Y) es verdadera.

Una consulta como la 1 tendrá una única respuesta: si o no, pero una consulta del tipo 2, 3 o 4 podrá tener varias respuestas (o ninguna). Más aún, las consultas 2 y 3 muestran que una relación no hace distinción entre entradas y salidas, lo que se denomina **inversibilidad**[14]. Estas consultas son características

[12] Ver capítulo 3 "Unificación" – Consultas de verificación
[13] Ver capítulo 3 "Unificación" – Consultas de búsqueda con variables
[14] Ver capítulo 5 "Inversibilidad".

de la programación lógica, y muestran porqué es potencialmente de **mayor nivel que otros paradigmas**.

A las consultas también se las denomina objetivos o metas.

Cláusulas de Horn

Los lenguajes del paradigma trabajan con lógica de primer orden, en particular, con las **Cláusulas de Horn**. Dichas cláusulas son una forma de lógica de predicados con **una sola conclusión en cada cláusula** y un conjunto de premisas de cuyo valor de verdad se deduce el valor de verdad de la conclusión: **una conclusión es cierta si lo son simultáneamente todas sus premisas.** Son el fundamento directo a la definición de **reglas de inferencia**[15] que utilizan los lenguajes de programación lógica.

Las cláusulas de Horn tienen la forma:

$$A_1 \text{ y } A_2 \text{ y } \ldots \text{ y } A_n \Rightarrow A_{n+1}$$

Cada A_i es una simple proposición de la forma $R_i(...)$ donde R_i es el nombre de la relación. Informalmente, está cláusula significa que, si A_1, A_2, ... y A_n son todas verdaderas, entonces podemos inferir que A_{n+1} es también verdadera. Pero no podemos inferir que A_{n+1} es falsa sólo porque algún A_i es falsa. Una cláusula de Horn se escribe más en base a "si" que a "si y sólo si", ya que puede haber otra cláusula en el programa que nos permite inferir también que A_{n+1} es verdadera.

Un programa lógico es una secuencia de cláusulas de Horn expresadas mediante la sintaxis de un lenguaje en particular. Ante un determinado requerimiento, cuyo resultado depende de probar la validez de una conclusión, se evalúa el valor de verdad de cada una de las premisas. La forma de escribir las cláusulas de Horn es al contrario de lo habitual: se escribe **primero el consecuente y luego el antecedente**. El antecedente puede ser una conjunción de condiciones que se denomina secuencia de objetivos, cada uno de los cuales puede ser a su vez un consecuente de una nueva cláusula.

Si bien es su fundamento matemático, los lenguajes de programación lógica no pueden explotar toda la potencialidad de la lógica matemática, porque hay formalismos que no son implementables. Se la puede usar limitadamente para especificar problemas que sean computables. Limitarse a cláusulas de Horn asegura la implementabilidad, y **le permite a la implementación ser aceptablemente eficiente**.

[15] Ver capítulo 4 "Reglas de inferencia"

Capítulo 3

Unificación

- **Valores y variables**
  - Variables anónimas
- **Términos**
- **Unificación**
- **Predicados**
- **Hechos**
- **Consultas**
  - Consultas de validación
  - Consultas de búsqueda con variables
- **Hechos universales**

Valores y variables

Los **átomos** son valores primitivos que no tienen otras propiedades que la habilidad de distinguirse unos de otros. Son usados para representar objetos del mundo real que son primitivos en lo concerniente a la aplicación.

Ejemplo:

rojo, verde, azul	pueden representar colores
juan, pedro, maria	pueden representar nombres de personas
1, 234, 4.6	pueden representar cantidades

En los ejemplos, se los representa comenzando con letras minúsculas, o representados por los dígitos correspondientes en el caso de valores numéricos.

Una **variable** lógica significa la especificación de un dato indeterminado. Las variables tienen un significado distinto al del que tienen en los lenguajes de

otros paradigmas donde se las asocia con posiciones de memoria donde se asignan valores. Las variables son **incógnitas**, **indeterminaciones** de valores, que al ejecutarse un programa asumen un conjunto posible de valores como resultados alternativos. Al no necesitar declaración de tipos de datos, una variable se puede unificar con cualquier tipo de valores. Una variable se **declara implícitamente** por su ocurrencia en la cláusula, y su **alcance** es sólo la cláusula en la cual aparece.

Ejemplo:

X, Y, UnaVariable pueden representar cualquier valor

En los ejemplos, se las representa comenzando con letras mayúsculas.

Variables anónimas

Existen variables sin nombre, que se representan mediante el símbolo de subrayado (_). Pero aunque todas las variables anónimas se escriben igual, son todas distintas. Es decir, mientras que dos apariciones de la secuencia de caracteres **Hola** se refieren a la misma variable, dos apariciones de la secuencia _ se refieren a variables distintas.

Se la utiliza cuando se quiere hacer referencia a una variable que después no se la vuelve a utilizar en ninguna otra proposición, por lo que significa que no interesa su valor para el propósito de la regla.

Términos

Los términos son el único elemento de un lenguaje lógico, es decir, los datos son términos, el código son términos, incluso el propio programa es un término. No obstante, es habitual, llamar término solamente a los datos que maneja un programa.

Todos los valores y variables mencionados son términos, pero el concepto de término es más amplio. También existen valores compuestos, que contienen varios átomos a la vez.[16]

En general, un término se compone de un **identificador** seguido de cero a N **argumentos** entre paréntesis y separados por comas. Los **argumentos** de un término pueden ser otro término o una variable lógica. La cantidad de argumentos se denomina **aridad**.

Ejemplo:

t(X,3)

Un término de aridad 2, denominado **t**

[16] Ver capítulo 8 "Functores y listas"

p(1,f(j),X)

Un termino de aridad 3, denominado **p**, cuyo segundo argumento es un término de aridad 1, llamado **f**.

rojo, 1, UnaVariable

También son términos.

Unificación

La unificación es el mecanismo mediante el cual **las variables lógicas toman valor**. El valor que puede tomar una variable consiste en cualquier término que representa un dato o conjunto de datos. Se le llama también **"Pattern Matching"** (Encaje de patrones), en analogía a otros paradigmas.

No existe el concepto de asignación a celdas de memoria y la noción de estado de una variable, sujeto a modificaciones reiteradas, sino que las variables son "unificadas" con valores particulares. La unificación no debe confundirse con la asignación de los lenguajes imperativos puesto que representa la igualdad lógica.

Cuando una variable no tiene valor se dice que está **libre**. Pero una vez que asume un valor, éste ya no cambia y se dice que la variable está **ligada, unificada** o **instanciada**.

Se dice que dos términos unifican cuando existen valores que hacen posible una ligadura (valor) de las variables tal que ambos términos son idénticos sustituyendo las variables por dichos valores.

La unificación se utiliza constantemente en la consultas de hechos y reglas, para ligar las variables, constantes y términos más complejos que se envían y reciben como argumentos.

Dentro de la definición de los predicados, para explicitar una unificación se utiliza la **igualdad** (=) entre valores, variables u otras expresiones.

Ejemplo:

?- X = 4 .

Unifica directamente asumiendo la variable **X** el valor **4**. Esto provoca la sensación de que se está asignando el valor a la variable al estilo imperativo, pero es una comparación.

?- 4 = X.

También unifica directamente asumiendo la variable **X** el valor **4**.

Un par de términos más complejos como los siguientes.

?- a(X, 3) = a(4, Z).

Unifican dando valores a las variables: **X** vale **4**, **Z** vale **3**.

Por otra parte, no todas las variables están obligadas a quedar ligadas.

Ejemplo:

?- h(X) = h(Y).

Unifican aunque las variables **X** e **Y** no quedan ligadas. No obstante, ambas variables permanecen unificadas entre sí. Si posteriormente se liga **X** al valor 3, entonces automáticamente la variable **Y** tomará ese mismo valor. Lo que esta ocurriendo es que, 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.

Para saber si dos términos unifican se pueden 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 identificador 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.

Ejemplo:

?- k(Z, Z) = k(4, H).

Una misma variable puede aparecer varias veces en los términos a unificar. Por el primer argumento, **Z** se liga al valor **4**. Por el segundo argumento, **Z** y **H** unifican, pero como **Z** se liga a un valor, entonces **H** se liga a ese mismo valor, que es **4**.

?- k(Z, Z) = k(4, 3) .

No unifican. Una variable no puede ligarse a dos valores distintos.

?- a(b(j, K), c(X)) = a(b(W, c(X)), c(W)).

Unifican, para **j = W**, **K = c(X)**, y **X = W**

?- k(_, _) = k(3, 4).

Las variables anónimas, al ser todas distintas, unifican perfectamente.

Como la igualdad no es una asignación, sino una comparación, no tiene sentido hablar de variables que sean contadores o acumuladores como en otros paradigmas.

Ejemplo:

?- X = X + 1.

no

?- X = X + K.

no

Representan comparaciones que son falsa por definición ya que no existe valor alguno de **X** que permita que sean verdadera, por lo tanto no se produce la unificación. No tiene sentido hablar del "estado anterior" de **X** y el "nuevo estado" de **X** como si fuera el resultado de una asignación.[17]

Predicados

Toda relación que se quiera utilizar en un programa lógico se la representa mediante predicados. Un predicado es un término que tiene un identificador y cada uno de los datos que intervienen constituyen sus argumentos. La cantidad de argumentos puede de dos, tres o más argumentos. También puede darse el caso de hechos con un sólo átomo, donde la relación es unaria, o incluso, sin argumentos.

Los predicados son **los elementos ejecutables en el paradigma**. Están determinados en la base de conocimiento mediante hechos y **a partir de ellos se efectúan las consultas**.

Hechos

Los hechos son una abstracción de la realidad que el sistema pretende modelar. Es la información del sistema que es considerada correcta y se expresan como **relaciones entre datos**.

Los hechos son cláusulas que forman parte de la **base de conocimiento** de un programa.

Ejemplo:

padre(luis, santiago).

Indica que hay una relación de filiación entre luis y santiago, por la cual luis "es el padre de" santiago y santiago "es el hijo de" luis.

persona(juan).

Es un hecho de un solo argumento que indica que juan es una persona.

regalo(daniel, andrea, auto, rojo).

[17] Para realizar operaciones aritméticas, se utiliza el predicado **is/2.** Ver anexo "Prolog" – Cálculos aritméticos

Es un hecho de cuatro argumentos, en el que se indica que **daniel** le regaló a **andrea** un **auto** de color **rojo**.

En la base de conocimiento, para cada predicado puede haber una cantidad de hechos tan numerosa como se requiera.

Ejemplo:

vive(raul, cordoba, 1985).

vive(cristina, cordoba, 1985).

vive(cristina, rosario, 2005).

Indica que **raul** se fue a vivir a **cordoba** en el año **1985** y que **cristina** también los hizo en el mismo año. Luego, **cristina** fue a vivir a **rosario** en el **2005**.

Consultas

El procesamiento consiste en comprobar el valor de verdad de una proposición dada, mediante lo que se denomina una **consulta** u **objetivo (goal)**. Si podemos inferir de los hechos del programa que la proposición es verdadera, entonces decimos que la consulta tuvo éxito. Si no podemos inferir que es verdadera, entonces decimos que falló. Esto no significa que la proposición es definitivamente falsa, sólo significa que no se puede inferir que es verdadera de la información que cuenta el programa.

Las consultas son la expresión de las **preguntas, requerimientos o pedidos** que se hacen sobre la base de conocimiento para encontrar todas las soluciones correctas posibles a un problema.

Una consulta es una llamada concreta a un predicado. Todas las consultas tienen un resultado de éxito o fallo tras su ejecución indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso. Cuando tiene éxito, las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables libres.

Ante una consulta cualquiera se hace una búsqueda secuencial de toda la base de conocimiento hasta encontrar las soluciones válidas. Para efectuar una consulta se utilizan los mismos nombres de predicados que existen en la base de conocimiento.

Una consulta puede ser una **validación** o una **búsqueda**.

Consultas de validación

La consulta más sencilla consiste en preguntar acerca de **si se cumple cierta relación entre datos**. Satisfacer una consulta de este tipo significa **validar la veracidad de una relación** confrontándola con la información de la base de

conocimiento, por lo que hay sólo dos respuestas posibles, de naturaleza booleana.

Ejemplo:

?- padre(luis, santiago).

yes

?- vive(raul, cordoba, 1985).

yes

?- persona(juan).

yes

Como en la base de conocimiento hay algún hecho en el que todos sus datos coinciden con los argumentos de la consulta, se responde afirmativamente.

?- padre(luis, estela).

no

?- vive(raul, cordoba, 1988).

no

?- persona(jorge)

no

Como en la base de conocimiento no hay hechos que coincidan totalmente con los solicitados, se responde negativamente

Consultas de búsqueda con variables

Una consulta que contiene variables se debe comprender como la búsqueda en la base de conocimiento de **los valores que puedan asumir la variable para que la consulta sea cierta**, es decir, una consecuencia lógica de la información que se dispone.

Ante una consulta variable, se buscarán todos los hechos de la base de conocimiento que permitan **unificar** los valores con las variables y así satisfacer la consulta.

Una consulta con una variable puede tener **varias soluciones**.

Ejemplo:

?- padre(luis, X).

X = santiago

La consulta se lee como ¿existe un **X** para el que **luis** sea su padre? La búsqueda en la base de conocimiento encuentra el hecho del ejemplo anterior, en el cual coincide el

primer argumento con el valor **luis** y el segundo argumento provoca la unificación de la variable **X** con el valor **santiago**.

?- padre(X, santiago).

X = luis

En forma similar, ante la consulta con una variable en el primer argumento, se encuentra un valor posible para **X** que satisface la consulta.

?- vive(X, cordoba, 1985).

X = raul

X = cristina

Esta consulta encuentra dos hechos diferentes para los cuales la consulta se satisface, por lo tanto cada valor constituye una solución alternativa para la variable **X**.

También se pueden realizar **consultas con varias variables**, en cuyo caso cada solución estará formada por un conjunto de valores: uno para cada incógnita.

Ejemplo:

?- vive(raul, X, Y).

X = cordoba Y = 1985

Hay una única solución con un valor para cada variable

?- vive(cristina, X, Y).

X = cordoba Y = 1985

X = rosario Y = 2005

Hay dos soluciones alternativas, cada una formada con un valor para cada variable

Cuando habiendo recorrido toda la base de conocimiento no se encuentran datos que satisfagan la consulta, la respuesta es negativa y debe interpretarse como que **no hay soluciones**, es decir, que la consulta **falla**.

Ejemplo:

?- padre(X, luis).

No

No hay ningún hecho del predicado pedro que tenga el valor **luis** como segundo argumento, En otras palabras, el sistema no cuenta con la información sobre quién es el padre de **luis**.

Los predicados no son funciones, por lo que las consultas que se hacen sobre ellos no retornan un valor de respuesta en la expresión misma. Son proposiciones que al evaluarlas permiten obtener un valor de verdad. Cuando

se invoca un predicado con todos los argumentos con valores constantes excepto uno, generalmente el último, que es variable, lo que podría interpretarse una función con varios parámetros de "entrada" y uno de "salida", hay que tener presente, no sólo que la solución no es necesariamente única, sino también que es la variable utilizada como argumento la que asume los valores y no la expresión del predicado como tal.

Hechos universales

Los hechos, además de contener valores constantes, pueden tener variables para **permitir generalizar una relación para todos los elementos,** y se denominan hechos universales.

En estos casos, aunque la unificación se produce en forma similar a las consultas con variables.

Ejemplo:

En la base de conocimiento todas las personas son de nacionalidad argentina:

nacionalidad(X, argentina).

Se pueden hacer consultas como:

?- nacionalidad(juan, argentina).

yes

Cualquiera sea el valor del primer argumento, se unificará con la variable del hecho universal.

Por otra parte, no se pueden hacer consultas como:

?- nacionalidad(X, argentina).

Ya que no hay valores de los que se puedan deducir.

Capítulo 4

Reglas de inferencia

- **Consultas simultáneas**
- **Reglas lógicas**
  - Variables que no "varían"
- **Predicados con varias cláusulas**

Consultas simultáneas

Una consulta formada por varias **proposiciones unidas por comas**, que son a su vez otras consultas, representa una conjunción, un "**y" lógico** de todas las proposiciones, es decir, requiere la **verificación simultánea** de todas las consultas. Se debe tener en cuenta:

- Las consultas se ejecutan secuencialmente de izquierda a derecha.
- Si una consulta falla, las siguientes consultas ya no se ejecutan y la conjunción, en total, falla.
- Si una consulta tiene éxito, algunas o todas sus variables quedan ligadas, y por tanto, dejan de ser variables libres para el resto de las consultas.
- Si todas las consultas tienen éxito, la conjunción tiene éxito y mantiene las ligaduras de los objetivos que la componen.

Ejemplo:

En la base de conocimiento están los hechos:

```
edad( luis, 32).
edad( juan, 25).
```

Se realiza la siguiente consulta

```
?- edad( luis, Y), edad( juan, Z), Y > Z.
Y = 32
Z = 25
```

La ejecución de la primera consulta tiene éxito y liga la variable **Y**, que antes estaba libre, al valor **32**. Al ejecutar la segunda consulta, su variable **Z** también está libre, pero el objetivo tiene éxito y liga dicha variable al valor **25**. Cuando se ejecuta la tercera consulta sus variables ya no están libres porque fueron ligadas en las consultas anteriores. Como el valor de **Y** es mayor que el de **Z** la comparación tiene éxito.

Como todos los objetivos han tenido éxito, la conjunción tiene éxito, y deja las variables **Y** y **Z** ligadas a los valores **32** y **25** respectivamente.

Reglas lógicas

Las reglas son **cláusulas** que permiten **definir nuevas relaciones en función de otras ya existentes**. Las reglas se basan en las **cláusulas de Horn**, de la siguiente manera:

B:- A_1, A_2, ... , A_n.

Esta notación indica que si se cumplen simultáneamente A_1, A_2 ... y A_n ... entonces se cumple B. Tanto B como los A_i son predicados. B es el consecuente y los A_i son los antecedentes. [18]

La lógica consiste en que **cuando todos los antecedentes son ciertos, el consecuente también lo es**.

Las soluciones de una consulta sobre una regla se deducen si todos los antecedentes de la regla se verifican con la información existente; para lo cual, estos antecedentes se convierte en nuevas consultas que deben equipararse con hechos o resolverse por otras reglas. El proceso termina cuando todas las consultas han sido probadas. La solución final viene determinada por los valores unificados de las variables de la meta inicial.

La evaluación así descrita es **puramente declarativa**. Y asume que si se selecciona una regla: o bien sólo existe esa posibilidad o la regla que se necesita para alcanzar la solución es la que de algún modo ha sido seleccionada. Se alcanza la solución si existe un conjunto apropiado de reglas y sustituciones, tales que aplicando las sustituciones a las reglas se puede deducir la meta desde los hechos conocidos.

Ejemplo:

mortal(X) :- humano(X).

Todos los humanos son mortales.

hombre(X) :- varon(X), adulto(X).

Un hombre es un varón adulto

pareja(X,Y) :- varon(X), mujer(Y).

Una pareja está formada por un varón y una mujer.

[18] Generalizando una definición común a todas las cláusulas, un hecho es una regla donde la cantidad de antecedentes es 0. Su valor es siempre verdadero ya que no depende de otras proposiciones.

En otras palabras, permite definir un **nuevo predicado**, a partir de **consultas simultáneas** en el sentido de "y" lógico, tal como se explica anteriormente.

Ejemplo:

```
edad( luis, 32).
edad( juan, 25).
```

Con hechos como los del ejemplo anterior, la consulta de varias consultas simultáneas se puede implementar con el predicado **mayor/2** que consta de la siguiente regla:

```
mayor( Y, Z ) :-
        edad( luis, Y ),
        edad( juan, Z),
        Y > Z.
```

Ahora, para realizar la misma invocación anterior, se hace:

```
?- mayor( Y, Z ).
Y = 32
Z = 25
```

Una definición más genérica de la regla, permite relacionar mediante un nuevo predicado **mayor/2** a cualquier persona con otra.

```
mayor( Persona1, Persona2 ) :-
        edad( Persona1, Y ),
        edad( Persona2, Z),
        Y > Z.
```

En este caso, la invocación resulta:

```
?- mayor( P1, P2 ).
P1 = luis
P2 = juan
```

Las variables incógnitas se unifican con los nombres de las personas, mientras que las variables **X** e **Y**, si bien se unifican con los correspondientes valores y verifican la desigualdad, al no ser argumentos de la regla, no son devueltos. Si se desea obtener los cuatro valores, es decir, los nombres y las correspondientes edades, se puede definir un predicado **mayor/4**:

```
mayor( Persona1, Y, Persona2, Z ) :-
        edad(Persona1, Y ),
        edad(Persona2, Z),
        Y > Z.
```

En este caso, la invocación resulta:

```
?- mayor( P1, E1, P2, E2 ).
P1 = luis
E1 = 32
P2 = juan
E2 = 25
```

Responder una consulta consiste en determinar si es una consecuencia lógica de la información de la base de conocimiento, aplicando las reglas de deducción o inferencia.

Ante una consulta por un predicado que está implementado por una regla, **se deben evalúan cada una de las proposiciones que la componen como nuevas consultas** y así devolver las soluciones.

En las reglas se utilizan variables para que la regla sea genérica, ya que son **variables que están cuantificadas universalmente**. Su alcance es toda la regla. Una vez que una variable del consecuente se unificó con el valor invocado como argumento en la consulta, **no lo cambia** durante la evaluación de las demás proposiciones de la regla.

También pueden utilizarse **nuevas variables** en las proposiciones que actúan como antecedentes, que se comportan como las variables de cualquier consulta, es decir, se van a unificar con los valores que se obtengan como resultados alternativos.

Si la consulta es de **validación**, se evaluará que para los valores dados se cumplan todas las proposiciones. Si se trata de una consulta de **búsqueda con variables**, se devolverán los valores alternativos de dichas variables que satisfacen simultáneamente todas las proposiciones de la regla.

Ejemplo:

Los siguientes hechos definen una relación **estrella/1** sobre cuerpos celestiales.

```
estrella( sol ).
estrella( sirio ).
estrella( alfa ).
```

Los siguientes hechos definen una relación **orbita/2** entre cuerpos celestiales.

```
orbita( mercurio, sol ).
orbita( venus, sol ).
orbita( tierra, sol ).
orbita( marte, sol ).
orbita( luna, tierra ).
orbita( fobos, marte ).
orbita( deimos, marte ).
```

La siguiente cláusula define una relación **planeta/1**:

```
planeta( B ) :- orbita( B, sol ).
```

Esto significa que el cuerpo B es un planeta si realiza su órbita alrededor del sol.

Dadas las siguientes consultas se obtiene como respuesta:

```
?- planeta( marte ).
yes
```

?- planeta(P).

P = mercurio ;

P = venus ;

P = tierra ;

P = marte

La siguiente cláusula define la relación satélite:

satelite(B) :- orbita(B, P), planeta(P).

Esto significa que el cuerpo **B** es una satélite si realiza su órbita alrededor de un cuerpo **P** y ese **P** es un planeta.

Dadas las siguientes consultas se obtiene como respuesta:

?- satelite(fobos).

yes

?- satelite(S).

S = luna ;

S = fobos ;

S = deimos

La consulta **satelite(S)** significa: "¿Existe algún S tal que satelite(S)?"

En general, cada variable dentro de una consulta es existencialmente cuantificada. La cláusula que define **satelite/1** tiene el siguiente significado en un predicado lógico: "Para todo B: satelite(B) si existe P tal que orbita(B, P) y planeta(P)."

En general, cada variable que está en el lado izquierdo de la cláusula es universalmente cuantificada ("para todo ...: ..."). La cuantificación siempre es implícita. Esto es a causa de que el alcance de cada variable es la cláusula en la cual está.

Variables que no "varían"

Dentro de una regla, las variables unificadas no cambian su valor como resultado de la evaluación de las nuevas consultas que forman parte de ella. Todas las variables unificadas en alguna de las consultas, permanecen unificadas con los mismos valores hasta que se terminan de valuar las otras consultas de la regla. Como se considera que un conjunto de unificaciones de variables es solución a una consulta cuando el mismo conjunto **verifica todas las consultas** que forman parte de la regla, no tiene sentido que para algunas de las consultas sean unos los valores de las variables que las satisfacen y que para las demás consultas sean otros los valores para las mimas variables.

Cuando se trata de varias respuestas alternativas, las variables cambian de valor y asumen un abanico de posibles valores, pero dentro de cada solución válida, las variables que intervienen se unificaron cada una con un valor para todas las consultas de la regla.

Ejemplo:

```
regla ( X, Y ) :-
        proposicion1( X, Y ),
        proposicion2( Y ),
        proposicion3( X ),
        proposicionN( X, Y).
```

Si se invoca a la regla con variables, y al evaluar la **proposicion1** se encontró alguna solución por la cual se unificaron las variables **X** e **Y**, la evaluación de **proposicion2** no puede hacer variar el valor de **Y**, ni la **proposicion3** el valor de **X**, ni las restantes en las que intervengan estas variables ya que se convierten en consultas de validación. Las variables **X** e **Y** se mantendrán con sus valores unificados por el resto de las proposiciones de la regla. Si hubiera diferentes soluciones para **X** e **Y**, en cada par de soluciones, los valores de **X** e **Y** serían únicos para verificar simultáneamente las proposiciones 1 a N.

Predicados con varias cláusulas

Un **predicado** se puede implementar mediante **una o más cláusulas**, que pueden ser **tanto hechos como reglas**. Para resolver una consulta, se evalúan secuencialmente todas las cláusulas.

Un predicado con varias cláusulas, representa la existencia de alternativas diversas para llegar a los resultados, para verificar una consulta. En otras palabras, representa el **"o" lógico**, de manera que una consulta por dicho predicado se verifica si se cumple una cláusula o si se cumple otra, o así sucesivamente, si se cumple alguna otra de las cláusulas definidas.

En una consulta de validación, ante la primera de ellas que permite satisfacer positivamente la consulta, se interrumpe la búsqueda y se devuelve el valor de verdad. Retornará falso cuando habiendo evaluado todas las cláusulas ninguna permitió satisfacer la consulta.

En una consulta de búsqueda con variables se devuelve el conjunto total de las soluciones alternativas encontradas por cada una de las cláusulas. Independientemente de que si al evaluar la primer cláusula hubo soluciones o no, se evalúa la segunda cláusula en búsqueda de nuevas soluciones y así sucesivamente hasta agotar todas las posibles alternativas.

También se puede expresar una alternativa con el símbolo punto y coma (;) en vez de la coma (,) para unir las consultas en una regla. Para indicar precedencia y agrupación se pueden utilizar paréntesis.

Ejemplo:

Siguiendo con el ejemplo anterior, las siguientes cláusulas define la relación **solar/1**:

```
solar( sol ).
solar( X ) :- planeta( X ).
```

solar(X) :- satélite(X).

La primera cláusula es un hecho que afirma que el sol es un miembro del sistema solar. La segunda cláusula indica que otra forma por la cual un cuerpo **X** es un miembro del sistema solar consiste en un regla que deduce de que el cuerpo **X** sea un planeta. Por último otra regla dice que también un cuerpo **X** es miembro del sistema solar si **X** es un satélite.

Dadas las siguientes consultas se obtiene como respuesta:

?- solar(sol).

yes

Al buscar en la base de conocimiento el predicado **solar/1** se encuentra con la primera cláusula que es un hecho que satisface la consulta. Se detiene la búsqueda y se da la respuesta afirmativa.

?- solar(luna).

yes

Al buscar el predicado **solar/1** no se unifica con la primera cláusula, que es un hecho en que no coinciden los valores. Se pasa a la siguiente cláusula, donde se unifica la variable **X** con el valor **luna** y se evalúa una nueva consulta mediante el predicado planeta cuyo resultado final va a ser negativo. Entonces se pasa a la última cláusula donde también se unifica la variable y el argumento. En este caso, cuando se evalúa la nueva consulta con el predicado **satelite/1**, la respuesta es afirmativa, por lo que la respuesta final de la consulta original también lo es.

?- solar(sirio).

no

En este caso, habiendo evaluado todas las cláusulas en forma análoga al ejemplo anterior y no habiendo encontrado en la base de conocimiento ningún hecho o regla que permita dar una respuesta afirmativa, se deduce que la respuesta es negativa.

?- solar(B).

B = sol ;

B = mercurio ;

B = venus ;

B = tierra ;

B = marte ;

B = luna ;

B = fobos ;

B = deimos

En esta consulta con variable, se van a evaluar todas las cláusulas. Con la primera cláusula se unifica la variable **B** con el valor **sol** y se obtiene así la primera respuesta. Se pasa a la siguiente cláusula, donde sustituyendo las variable **B** por **X** se evalúa una nueva consulta mediante el predicado **planeta/1** que obtiene como soluciones alternativas los valores **venus**, **tierra** y **marte**. Estos valores son devueltos como nuevas soluciones a la consulta original. Luego se pasa a la última cláusula donde también se evalúa una nueva

consulta con el predicado **satelite/1**, cuya resolución aporta nuevas soluciones para la incógnita **B** que se añaden a las ya encontradas anteriormente. No habiendo más cláusulas que evaluar para el predicado **solar/1**, se concluye la búsqueda.

Ejemplo:

En la base de conocimientos existen hechos que indican los números de teléfono de cada persona y la vinculación que tienen con la persona que trabaja en una oficina.

```
datos( juan, 44444444, laboral ).
datos( ana, 43333333, familiar ).
datos( cecilia, 45555555, personal ).
```

Las siguientes reglas controlan la central telefónica de una oficina discriminando cuáles llamados deriva o no según la hora y el origen de la llamada.

```
derivar( Te,   _  ):- datos( _, Te, familiar ).
derivar( Te, Hora ):- datos( _, Te, laboral ), Hora > 10, Hora < 18.
derivar( Te, Hora ):- datos( _, Te, personal ), Hora < 17.
derivar( Te, Hora ):- datos( _, Te, personal ), Hora > 22.
```

La primera regla indica que los llamados familiares se derivan durante todo el día. Se usa una variable anónima para la hora, ya que no hace falta volver a utilizarla. **Te** es la variable que representa el número telefónico y **familiar** es una constante. También se usa otra variable anónima para el nombre de la persona. Se consulta por el predicado **datos/3** para saber si el número de teléfono recibido corresponde a una persona que sea familiar.

La segunda regla indica que los llamados laborales se derivan entre las 10 y las 18 hs. Se realizan las comparaciones de la variable **Hora** que simultáneamente debe ser mayor a la constante **10** y menor que la constante **18**.

Las dos últimas reglas indican que los llamados personales se derivan cuando se producen antes de las 17 o después de las 22 hs. Las dos comparaciones entre la variable **Hora** y las constantes **17** y **22** se expresan en reglas diferentes ya que no se trata de condiciones que deben cumplirse simultáneamente ("y"), sino en forma alternativa ("o").

Algunas consultas válidas son:

```
?- derivar( 42222222, 16 ).
Yes

?- derivar( 45555555, 17 ).
No

?- derivar( X, 14 ).
X = 44444444
X = 43333333
X = 45555555
```

Capítulo 5

Inversibilidad

- **Inversibilidad**
- **Indeterminación**

Inversibilidad

En otros paradigmas, las salidas son funcionalmente dependientes de las entradas, por lo que el programa puede verse abstractamente como la implementación de una transformación de entradas en salidas. La programación lógica está basada en la noción de que **el programa implementa una relación, en vez de una transformación**. Los predicados son relaciones, que al no tener predefinido una "dirección" entre sus componentes, permiten que **sus argumentos actúen indistintamente como argumentos de entrada y salida**. Dado que las relaciones son más generales que las transformaciones, la programación lógica es potencialmente de más **alto nivel** que la de otros paradigmas.

La **reversibilidad o inversibilidad** es la propiedad de los predicados de ser definidos mediante cláusulas que permiten realizar consultar con variables en los diferentes argumentos, de manera de poder obtener un conjunto de resultados con soluciones alternativas. A su vez, en los predicados con varios argumentos, cambiando cuál es el argumento en el que se indica una variable, se hace que **un mismo predicado tenga varias funcionalidades lógicas**.

En otras palabras, inversibilidad es el nombre conceptual que recibe la posibilidad de realizar consultas con variables.

Ejemplo:

Dadas las siguientes cláusulas y consultas:

```
padre( carlos, ernesto ).
padre( ernesto, alfredo ).
```

padre(carlos, jose).

padre(jose, ariel).

abuelo (X, Y) :- padre(X, Z), padre(Z, Y).

Permite obtener tanto abuelos como nietos según cuál sea la incógnita en la consulta. El predicado es, entonces, inversible.

?- abuelo (A, alfredo).

A = carlos

Con una variable en el primer argumento, obtiene el abuelo de **alfredo**, en general, el o los abuelos de quien sea el segundo argumento.

?- abuelo (carlos, N).

N = Alfredo ;

N = ariel

Se obtienen los nietos de **carlos**, en general, los nietos de quien sea el primer argumento.

?- abuelo (A , N).

A = carlos
N = alfredo ;

A = carlos
N = ariel

En la consulta más amplia posible, se obtienen todos los pares de abuelos y nietos que se deduzcan de la base de conocimiento.

Ejemplo:

Se conoce la información de los resultados de los partidos de básquet de las diferentes ruedas realizadas luego de la primera rueda, que se definen mediante eliminación directa:

partido(cuartos, argentina, 98, grecia, 76).

partido(cuartos, holanda, 99, estadosunidos, 100).

partido(cuartos, italia, 90, turquia, 80).

partido(cuartos, uruguay, 104, rusia, 100).

partido(semi, argentina, 98, estadosunidos, 91).

partido(semi, italia, 95, uruguay, 93).

partido(final, argentina, 89, italia, 88).

partido(tercerpuesto, estadosunidos, 87, uruguay, 88).

El siguiente programa permite obtener los países ganadores y las medallas que obtiene cada uno, como también los que clasificaron, es decir los que no quedaron eliminados en la primera rueda.

```
resultado(Rueda, Ganador, Perdedor) :-
        partido( Rueda, Ganador, X, Perdedor, Z ),
        X > Z.
resultado(Rueda, Ganador, Perdedor) :-
        partido( Rueda, Perdedor, X, Ganador, Z ),
        X < Z.
```

El predicado **resultado/3** relaciona una determinada rueda con el país ganador y el perdedor, a partir de consultar por el partido y de comparar los puntos obtenidos. En a primera regla, si los puntos (**X**) del primer país (**Ganador**) superan a los puntos (**Z**) del segundo país (**Perdedor**), el ganador es el primero. La otra regla deduce el caso opuesto, donde de acuerdo a lo que indica el predicado **partido/5**, el país del cuarto argumento supera en puntos al país del segundo argumento. Se supone que no hay posibilidad de empate.

```
gano( Rueda, Ganador ):-    resultado(Rueda, Ganador, _).
perdio( Rueda, Perdedor):-  resultado(Rueda, _ , Perdedor).
jugo( Rueda, Pais):-        resultado(Rueda, Pais, _).
jugo( Rueda, Pais):-        resultado(Rueda, _, Pais).
```

Estos predicados se basan en el predicado anterior para indicar que un país ganó, perdió o jugó en una determinada rueda del campeonato. En particular, el predicado **jugo/2**, requiere de dos reglas para contemplar ambos resultados posibles.

```
medalla( Pais, oro ):-      gano( final, Pais).
medalla( Pais, plata ):-    perdio( final, Pais).
medalla( Pais, bronce ):-   gano( tercerpuesto, Pais).
```

Sabiendo qué país ganó o perdió cada rueda del campeonato, con el predicado **medalla/2** se deducen las medallas correspondientes. Tanto los valores de las medallas como de las ruedas son expresados con constantes.

```
clasifico( Pais ):-         jugo( cuartos, Pais).
```

El predicado **clasifico/1** indica qué países superar la clasificación e ingresaron a las ruedas de eliminación directa. Con sólo comprobar que el país haya jugado un partido de cuartos de final es suficiente para afirmar que clasificó.

Con los hechos mencionados de la base de conocimiento, las consultas arrojan los siguientes resultados:

```
?- medalla( argentina, oro).
yes
```

Es una consulta de validación, con constantes, donde la respuesta es de carácter booleana. Se lee: ¿Es cierto que a argentina le corresponde una medalla de oro?

```
?- medalla( Pais, oro).
Pais = argentina
```

En esta consulta, se comprueba cómo el primer argumento es inversible, al poder representar una incógnita mediante una variable. Se lee ¿A qué país(es) les corresponde la medalla de oro?

?- medalla(argentina, Medalla).

Pais = argentina

En forma similar, se ve que el segundo argumento es inversible. Se lee: ¿Qué medalla(s) le corresponde a argentina?

?- medalla(Pais, Medalla).

Pais = argentina
Medalla = oro ;

Pais = italia,
Medalla = plata ;

Pais = estadosunidos
Medalla = bronce

Esta consulta permite ver la inversibilidad de ambos argumentos, el que representa el país y la medalla, ya que ambos pueden ser incógnitas.

Con el otro predicado, sucede en forma similar, aún siendo un solo argumento.

?- clasifico(grecia).

yes

?- clasifico(brasil).

no

?- clasifico(Pais).

Pais = argentina ;

Pais = grecia ;

Pais = holanda ;

Pais = estadosunidos ;

Pais = italia ;

Pais = turquia ;

Pais = uruguay ;

Pais = rusia

Más precisamente, cada argumento puede ser inversible en la medida que pueden realizarse consultas con variable en él. Cuando todos los argumentos son inversibles se dice que el predicado también lo es.

No todos los predicados son inversibles, y en muchos de ellos hay argumentos que son inversibles y otros que no. Esto determina que haya consultas que se puedan realizar y otras que no.

Indeterminación

El mecanismo de evaluación requiere que las variables se unifiquen con valores para poder retornar los resultados posibles de una consulta. Cuando se realizan consultas variables, hay situaciones donde **no es posible establecer el valor de verdad de una proposición**. Esto se produce cuando **las variables no se logran unificar satisfactoriamente con algún valor**, sin que ello signifique que la respuesta sea negativa, sino que hay una **indeterminación** acerca de cuales serían esos valores.

En general, son casos donde hay **infinitas soluciones posibles** o consultas que siendo válidas en su formulación en términos del lenguaje, representan **consultas sin sentido** en la realidad.

Ejemplo:

Dada una regla que permite afirmar que un valor es mayor que otro

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

Cualquier consulta de validación funciona correctamente:

```
?- mayor( 6, 5 ).
yes

?- mayor( 6, 7 ).
no
```

Se produce la unificación de la variables **X** e **Y** con los valores de los argumentos, con ellos se evalúa la desigualdad y su resultado es devuelto a la consulta original.

En cambio, ante una consulta de búsqueda con variables:

```
?- mayor( 6, N ).
```

Hay un error, porque si bien se produce la unificación de la variable **X** con el valor **6** del argumento, cuando se evalúa la desigualdad se está comparando:

```
6 > X
```

En esta proposición habría infinitos valores posibles para **X** que la satisfacen, pero no hay forma de obtenerlos. La variable **X** queda sin unificar y la consulta, imposibilitada de responder una solución válida, ya sea positiva o negativa, produce un error.

La indeterminación de las variables marca un límite a la inversibilidad, ya que no todos los predicados son inversibles. Sus restricciones están dadas por las variables que ante determinadas consultas no se pueden unificar. Un predicado puede ser no inversible por su misma definición o solamente ser inversible para ciertos argumentos, dependiendo de la forma en que se efectúan las consultas.

Una definición correcta de predicados, y por lo tanto la documentación correspondiente, debe ser clara respecto a la indeterminación e inversibilidad,

indicando que argumentos deben o no estar instanciados (sin variables libres) para que una consulta tenga sentido y se pueda resolver.

Las formas posible de uso de un predicado en cuanto a la inversibilidad de sus argumentos, se puede especificar mediante una convención simbólica como la siguiente, que es usada por numerosos intérpretes:

- De entrada y/o salida indistintamente. Estos argumentos se denotan con un símbolo de interrogación (?).
- De solamente entrada. Estos se indican con un símbolo de suma (+).
- De solamente salida. Con un símbolo de resta (-).

Ejemplo:

predi1(+A, +B, -C).

Los dos primeros argumentos deben estar instanciados, son de "entrada", el tercero es de "salida".

predi2(?A, ?B).

Ambos argumentos son inversibles.

Capítulo 6

Mecanismos de evaluación

- **Control de secuencia**
- **Backtracking**
- **Evaluación de una regla**

Control de secuencia

Internamente, existe un mecanismo, un "**motor**", que actúa como **control de secuencia**. Durante la ejecución de un programa va evaluando y combinando las reglas lógicas de la base de conocimiento para lograr los resultados esperados. La implementación del mecanismo de evaluación puede ser diferente en cada lenguaje del paradigma, pero en todos los casos debe garantizar que se agoten todas las combinaciones lógicas posibles para ofrecer el conjunto completo de respuestas alternativas posibles a cada consulta efectuada. El más difundido se denomina **backtracking**, que utiliza una estrategia de búsqueda primero en profundidad.

Backtracking

El **backtracking** consiste en una búsqueda en la base de conocimiento que se efectúa **de arriba hacia abajo**, realizando **una búsqueda primero en profundidad**. Las reglas son resueltas **de izquierda a derecha**. Cuando se precisa seleccionar una regla, este algoritmo de control selecciona la primera que encuentra, si conduce a un punto muerto, selecciona la segunda, y así hasta que todas las posibilidades han sido probadas.[19]

El mecanismo de backtracking puede expresarse como "**volver atrás y probar de nuevo**". Permite **evaluar todas las combinaciones posibles** y así **encontrar todas las soluciones alternativas** a un problema dado.

[19] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Paraninfo.

La evaluación empieza definiendo como objetivo una determinada consulta e intentando probar que dicho objetivo se ajusta a un hecho o se deduce de alguna regla.

Cada objetivo determina un subconjunto de cláusulas susceptibles de ser ejecutadas. Cada una de ellas se denomina punto de elección. Se selecciona el primer punto de elección y sigue ejecutando el programa hasta determinar si el objetivo es verdadero o falso. En caso de ser falso entra en juego el backtracking propiamente dicho, que consiste en deshacer todo lo ejecutado situando el programa en el mismo estado en el que estaba justo antes de llegar al punto de elección. Entonces se toma el siguiente punto de elección que estaba pendiente y se repite de nuevo el proceso. Todos los objetivos terminan su ejecución bien en éxito ("verdadero"), bien en fallo ("falso"). Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado. Durante la ejecución, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades se vuelve a la elección anterior y se toma la siguiente opción. Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento recursivo. Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.

El backtracking le da a la programación lógica un estilo distintivo, y es responsable de gran parte de su potencialidad y expresividad. Por otra parte, para resolver problemas complejos requiere uso de muchos recursos, tanto en tiempo como en espacio de almacenamiento.

Esquemáticamente, el backtracking funciona de la siguiente manera:

Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuantas soluciones alternativas puede tener. Cada una de las alternativas se denomina punto de elección. Dichos puntos de elección se anotan internamente y de forma ordenada.

- Se escoge el primer **punto de elección** y se ejecuta el objetivo eliminando el punto de elección en el proceso.
- Si el objetivo tiene **éxito** se continúa con el siguiente objetivo aplicándole estas mismas normas.
- Si el objetivo **falla**, da **marcha atrás** recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y **deshaciendo las ligaduras de sus variables**. Es decir, comienza el backtracking.
- 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.

- El proceso se repite mientras haya objetivos y puntos de elección anotados. De hecho, se puede decir que un programa ha terminado su ejecución cuando no le quedan puntos de elección anotados ni objetivos por ejecutar en la secuencia.
- Además, los puntos de elección se mantienen aunque al final la conjunción tenga éxito. Esto permite posteriormente conocer todas las soluciones posibles.

Evaluación de una regla

En una regla, para evaluar que se cumplan todas las proposiciones, se las va evaluando de izquierda a derecha, en el orden en que están dispuestas en la definición de la regla. Cuando cualquiera de las proposiciones implica una nueva consulta en la que alguna o algunas de sus variables se unifica con más de un valor posible, o sea que devuelve varias soluciones alternativas, evalúa las demás proposiciones asumiendo los valores unificados en la primer solución (y se coloca internamente una "marca" que permite saber dónde fue encontrada esa solución). Ya sea que con dicha unificación se logró encontrar o no respuestas satisfactorias a la consulta original, al terminar de evaluar todas las posibilidades de las proposiciones restantes, retorna a continuar con la evaluación de la consulta que había quedado detenida en su primer solución, (gracias a la marca dejada oportunamente) hasta encontrar la siguiente solución con una nueva unificación de variables. Así, con estos nuevos valores unificados, y en general, con cada una de las soluciones, se vuelven a evaluar todas las restantes proposiciones en búsqueda de otras soluciones a la consulta original, y como en el caso anterior, encontrando o no soluciones, al terminar "vuelve hacia atrás y prueba de nuevo" hasta agotar todas las posibles respuestas.

Este mecanismo se repite para cada una de las proposiciones de una regla, teniendo en cuenta cada una de sus soluciones, por lo que en definitiva provoca una evaluación de todas las combinaciones posibles entre las variables que intervienen.

Cuando en una regla ya se evaluó alguna de las proposiciones mediante una consulta variable, logrando así una unificación de variables, y al seguir con el proceso de evaluación de las proposiciones se encuentra frente a una proposición que implica una consulta y su resultado es negativo, ya no tiene sentido que siga evaluando las restantes proposiciones con dicha unificación. Entonces, como en el caso anterior, se vuelve atrás hasta la anterior consulta variables de la cual se obtuvo la solución parcial, se la descarta y se retoma su evaluación a partir de donde fue detenida en búsqueda de una nueva solución.

Si se encuentra otra solución alternativa, con los nuevos valores unificados se volverá a evaluar la consulta, a ver en este caso cuál es su respuesta.

Si el predicado tiene varias cláusulas, se repite el procedimiento para cada una y son devueltas juntas las soluciones provistas por todas las cláusulas.

Ejemplo:

```
varon( juan ).
varon( pedro ).
varon( raul ).
adulto( juan ).
adulto( raul ).
joven( pedro ).
mujer( ana ).
hombre( X ) :- varon( X ), adulto( X ).
```

La regla afirma que un varón que a la vez es adulto es considerado hombre.

```
?- hombre( juan ).
yes
```

Ante la invocación de la consulta, se unifica el valor del argumento **juan** con la variable **X** del consecuente de la regla del predicado **hombre/1**. Para poder responder si es cierto o no, se evalúan las proposiciones sustituyendo el valor **juan** en cada aparición de la variable **X**.

```
varon( juan ), adulto( juan ).
```

Se evalúan la primera proposición, que actúa como una nueva consulta. Al encontrar un hecho coincidente, devuelve verdadero.

Entonces evalúa también la otra proposición, que en forma similar, también devuelve verdadero. Habiéndose satisfecho todos los antecedentes, se deduce indefectiblemente que también se cumple el consecuente.

```
?- hombre( pedro ).
no
```

Ante la invocación de la consulta, se unifica el valor del argumento pedro con la variable **X** del consecuente de la regla hombre y se sustituye en las demás proposiciones:

```
varon( pedro ), adulto( pedro ).
```

La primera consulta responde satisfactoriamente, pero cuando se evalúa la segunda consulta, no se encuentra ningún hecho en la base de conocimiento que coincida, por lo que devuelve un valor falso. Como una de las proposiciones no se cumple, el valor de verdad de toda la regla también es falso.

```
?- hombre( ana ).
no
```

Este caso se comporta en forma similar, con la diferencia que como la primera de las proposiciones a evaluar ya retorna falso, se devuelve directamente que todo el consecuente es falso sin necesitar evaluar la segunda proposición.

?- hombre(H).

Ante la consulta con variable, se asocia la variable **H** del argumento con la variable **X** de la regla, pero lo que sigue siendo una expresión variable sin un valor definido. La evaluación de las proposiciones se resuelve como nuevas consultas:

varon(X), adulto(X).

La primera consulta es con variables y retornará tres valores alternativos para la variable **X**, que serán:

X = juan ;

X = pedro ;

X = raul

Como la forma en que trabaja el backtracking es mediante una búsqueda "primero en profundidad", estas soluciones alternativas se van encontrando de a una y para cada una de ellas se evalúan todas las demás consultas antes de pasar a buscar la siguiente. Por lo tanto, la primera solución que se encuentra es

X = juan

Con ese valor unificado se evalúa el predicado **adulto/1** como una nueva consulta, ésta de verificación.

adulto(juan)

yes

Habiendo encontrado una unificación que hace cierta todas las proposiciones de la regla, se devuelve a la consulta original el valor correspondiente como una primera solución.

H = juan

En este momento, el mecanismo de backtracking retoma la última consulta variable pendiente, que es la consulta por **varon(X)** interrumpida en su primera solución e intenta encontrar nuevas soluciones alternativas. De esta manera, continuando su búsqueda en la base de conocimiento, se obtiene la unificación:

X = pedro

Con ese nuevo valor unificado para **X** se evalúa el predicado **adulto/1** como una nueva consulta, también de validación.

adulto(pedro)

no

Como la respuesta es negativa el valor encontrado no puede ser solución de la consulta original. En este caso se detiene la evaluación con dicha unificación para la variable **X** y se retoma nuevamente la consulta anterior del predicado **varon/1** buscando otras soluciones alternativas. Se encuentra entonces una nueva respuesta:

X = raul

Otra vez, con el nuevo valor, se evalúa el predicado **adulto/1**.

adulto(raul)

yes

En este caso, como la respuesta vuelve a ser afirmativa se está ante un nuevo valor que satisfizo todas las proposiciones y es otra solución alternativa de la consulta original. Por lo tanto se retorna.

H = raul

Al volver hacia atrás e intentar buscar nuevas soluciones para la consulta del predicado **varon/1** no se van a encontrar otras más, por lo que la consulta original tampoco va a tener más soluciones alternativas y así concluye el mecanismo de su evaluación.

Capítulo 7

Estrategias de resolución

- **Recursividad**
  - Inducción
  - Construcción de predicados recursivas
- **La suposición de un "mundo acotado"**
- **Generación**

Recursividad

Una característica que lejos de ser exclusiva del paradigma pero que tiene importancia en la formulación de soluciones, es la recursividad, que se ve expresada en el uso de **predicados recursivas**. La recursividad, entendida como iteración con asignación no destructiva, está relacionada con el principio matemático de **inducción**.

Un predicado recursivo tiene al menos una cláusula que sea una regla recursiva, definida en función de sí misma, y una cláusula no recursiva para detener la recursividad.

Una regla recursiva es una regla que entre las proposiciones que actúan como antecedentes, se encuentre el mismo nombre de predicado que el consecuente de la regla, invocado con un argumento diferente.

Inducción

El **principio de inducción** sostiene que para demostrar que una determinada propiedad se cumple para todos los números naturales, es suficiente con demostrar que se cumplen dos condiciones.

- La propiedad **P** se cumple para **1**.
- **Si** una propiedad **P** se cumple para el número natural **n entonces** se cumple para **n+1**.

Construcción de predicados recursivos

Aplicándolo a las reglas del paradigma lógico, una expresión recursiva permite resolver numerosos problemas. La solución de un problema se plantea con dos casos:

- Un "**caso base**" de nivel "1", no recursivo, generalmente de simple resolución, a veces, inclusive, de naturaleza trivial.
- Un "**caso genérico**" de nivel "n + 1"donde la recursividad consiste en invocar a la mismo predicado para un caso de nivel "n". Asumiendo como cierto que el predicado para el caso "n" devuelve soluciones correctas, se lo utiliza dentro de una expresión que represente la resolución de caso de nivel "n + 1".

Al invocar el predicado mediante una consulta con un argumento cualquiera, ésta se irá invocando a sí misma tantas veces como sea necesario, evaluándose cada vez la expresión del caso genérico, y se evaluará una vez la expresión del caso base, que evitará una recursividad infinita.

Ejemplo:

Si se conocen las relaciones de paternidad entre personas de diferentes generaciones se pueden deducir nuevas relaciones que forman parte de un árbol genealógico más complejo.

Sean los siguientes hechos, donde se interpreta que la persona representada por el primer argumento es el padre y la del segundo es el hijo:

```
padre( justino, daniel ).
padre( daniel, carlos ).
padre( carlos, ernesto ).
padre( carlos, jose ).
padre( ernesto, alfredo ).
padre( jose, ariel ).
```

Si se desea establecer la relación abuelo-nieto, se lo puede hacer con el predicado:

```
abuelo( X, Y ) :- padre( X, Z ), padre( Z, Y ).
```

Las siguientes consultas evalúan las proposiciones y obtienen los correspondientes resultados:

```
?- abuelo( P, alfredo ).
P = carlos
```

Se obtiene el abuelo de **alfredo**

```
?- abuelo( carlos, P).
P = Alfredo ;
P = ariel
```

Se obtienen los nietos de **carlos**

Si se quisiera obtener los bisabuelos, y por lo tanto también los bisnietos se puede definir:

bisabuelo(X, Y) :- padre(X, Z), abuelo(Z, Y).

Se interpreta como "el padre del abuelo de una persona es su bisabuelo" o, desde el punto de vista del bisabuelo, que "los bisnietos son los nietos de sus hijos".

?- bisabuelo(P, alfredo).

P = daniel

Por último, si se desea obtener todos los antepasados de una persona, es decir, tanto padre, abuelo, bisabuelo como todos los anteriores, es evidente que no se pueden especificar todos los casos uno por uno.

antepasado(X, Y) :- padre(X, Y).

antepasado(X, Y) :- abuelo(X, Y).

antepasado(X, Y) :- bisabuelo(X, Y).

antepasado(X, Y) :- ...

Entonces se lo define en forma recursiva:

```
antepasado( X, Y ):-
                padre( X, Y ).
antepasado( X, Y ):-
                padre( X, Z ),
                antepasado( Z, Y ).
```

Se interpreta como que el antepasado de una persona es su padre o bien es el padre de un antepasado. La primera cláusula actúa como caso base y la segunda cláusula como caso genérico que invoca la recursividad. En otras palabras, el caso genérico plantea que suponiendo que se cumple que una persona es un antepasado, entonces su padre también lo es.

?- antepasado(P, alfredo).

P = ernesto ;

P = carlos ;

P = daniel ;

P = justino

Ejemplo:

En la base de conocimiento se encuentran las relaciones directas de correlatividad de una materia con otra, dentro del plan de estudios de una carrera.

correlativa(analisisMatematicoII, algebra).

correlativa(analisisMatematicoII, analisisMatematicoI).

```
correlativa( modelosNumericos, analisisMatematicoII).
correlativa( disenioSistemas, analisisSistemas).
correlativa( disenioSistemas, paradigmas).
correlativa( analisisSistemas, ingenieriaSociedad).
correlativa( paradigmas, sintaxis ).
correlativa( sintaxis, algoritmos ).
correlativa( analisisSistemas, algoritmos).
correlativa( algoritmos, matematicaDiscreta).
```

El siguiente programa permite saber si una materia es correlativa de otra, directa o indirectamente y calcular la cantidad de materias intermedias en la secuencia de correlatividades sucesivas.

```
cantCorrelativasIntermedias( Mat1, Mat2,  0  ):-  correlativa( Mat1, Mat2 ).
cantCorrelativasIntermedias( Mat1, Mat2, Cant ):-
                        correlativa( Mat1, MatAux ),
                        cantCorrelativasIntermedias( MatAux, Mat2, CantAux ),
                        Cant is CantAux + 1.
```

En la primera regla, se afirma que si dos materias son correlativas directas, evidentemente son correlativas y la cantidad de materias intermedias es **0**. En la segunda, se plantea que si existe una materia intermedia (**MatAux**), de la cual la primera (**Mat1**) es correlativa directa, y a su vez, dicha materia intermedia es correlativa (directa o indirectamente) de la segunda materia (**Mat2**), entonces la primera es correlativa de la segunda. Además, la cantidad de materias intermedias entre las materias originales (**Cant**), será 1 más que las que haya entre la materia auxiliar y la segunda (**CantAux**).[20]

Las consultas pueden ser:

```
?- cantCorrelativasIntermedias( paradigmas, algoritmos, X ).
X = 1

?- cantCorrelativasIntermedias( disenioSistemas, matematicaDiscreta, X ).
X = 2;
X = 3

?- cantCorrelativasIntermedias( X, Y, 2 ).
X = disenioSistemas
Y = matematicaDiscreta;
X = disenioSistemas
Y = algoritmos;
X = paradigmas
Y = matematicaDiscreta
```

[20] Ver predicado **is/2** en anexo "Prolog" – Cálculos aritméticos

La suposición de un "mundo acotado"

Una proposición puede fallar, pero no significa que es definitivamente falsa; simplemente significa que no se puede inferir de las cláusulas del programa que sea verdadera. En realidad, pocas proposiciones son definitivamente falsas (estas excepciones son proposiciones triviales como "perro = gato" y "2 > 3"). Cuando una proposición es evaluada, el éxito significa verdadero y la falla significa tanto desconocido como falso.

En forma práctica se ignora la distinción entre desconocido y falso. En otras palabras, **una proposición es asumida como falsa si no puede ser inferida como verdadera**. Esto es llamado como la suposición de un mundo acotado ("closed-world assumption"). Se asume que **la base de conocimiento incluye toda la información relevante** sobre el "mundo", es decir, el dominio de aplicación.[21]

Ejemplo:

Dada la base de conocimiento:

```
limite( argentina, chile ).
limite( argentina, bolivia ).
limite( argentina, paraguay ).
limite( argentina, brasil ).
```

Ante la consulta:

```
?- limite( argentina, uruguay).
no
```

La respuesta es negativa porque simplemente se omitió hacer proposición alguna que relacione a **argentina** con **uruguay**. Por lo tanto, la suposición de un mundo acotado obliga a incluir toda la información relevante.

Generación

Una estrategia frecuente para implementar predicados, es que en las reglas definidas por varias consultas, **las primeras generen soluciones posibles** y **las últimas filtren las válidas**. En general, sobre las primeras se hacen consultas de búsqueda con variables y en las últimas consultas de validación.

Ejemplo:

```
regla( X ):- generador( X ),  filtro( X ).
```

Ante una consulta con variables, de todas las respuestas posibles de **generador/1** son respuestas válidas de **regla/1** las que se verifiquen mediante **filtro/1**.

[21] **WATT, David**. *Programming Languajes Concepts and Paradigms*. Capítulo 14.

Ejemplo:

```
animal( leon ).
animal( elefante ).
animal( perro ).
animal( gato ).
paraLaCasa( helecho ).
paraLaCasa( perro ).
paraLaCasa( parrilla ).
paraLaCasa( gato ).
mascota( P ):- animal( A ), paraLaCasa( A )
```

De todos los animales posibles, que se obtienen con la consulta a **animal/1**, se considera mascota aquellos que son aptos para llevar a la casa, con el predicado **paraLaCasa/1**. Que haya otros elementos que también se puedan llevar a la casa no afecta en la solución.

```
?- mascota( X).
X = perro
X = gato.
```

La utilización de **predicados generadores** es una estrategia que facilita la construcción **de reglas inversibles**.

Ejemplo:

El predicado **clave/3** permite obtener todas las claves posibles formadas por tres dígitos diferentes.

```
digito( 0 ).
digito( 1 ).
digito( 2 ).
digito( 3 ).
digito( 4 ).
digito( 5 ).
digito( 6 ).
digito( 7 ).
digito( 8 ).
digito( 9 ).
clave( X, Y, Z ):- digito( X ), digito( Y ), digito( Z ), X \= Y, Y \= Z, X \= Z.
```

La invocaciones al predicado **digito/1** generan todos los posibles valores y las últimas condiciones filtran a algunos de ellos para dar cómo solución sólo las combinaciones con dígitos diferentes.

```
?- clave( X, Y, Z ).
```

```
X = 0,
Y = 1,
Z = 2;

X = 0,
Y = 1,
Z = 3;

…

X = 9,
Y = 8,
Z = 7
```

Se obtendrán algo menos de 1000 soluciones alternativas, formadas por todas las combinaciones posibles de tres dígitos diferentes. De esta manera, los tres argumentos son inversibles.

De no haber estado las consultas por **digito/1** en la definición de la regla **clave/3**, se podrían realizar solamente consultas de verificación:

```
clave( X, Y, Z ):- X \= Y, Y \= Z, X \= Z.


?- clave( 0, 1, 2 ).

yes

?- clave( 0, 1, 0 ).

no
```

Capítulo 8

Functores y listas

- **Tipos de datos compuestos**
- **Functores**
  - Descomposición
- **Listas**
  - Descomposición
  - Utilización

Tipos de datos compuestos

Los tipos de datos compuestos representan **estructuras de datos** que contienen simultáneamente un conjunto de valores y tienen la particularidad de poder ser tratados en conjunto **como una unidad** o **distinguiendo por separado sus componentes** y utilizados independientemente.

Los tipos de datos compuestos más utilizados son las **listas** y los **functores**.

Las estructuras de datos pueden contener valores de **diferentes tipos de datos.**

Functores

Los functores son **estructuras de datos** que pueden **contener varios elementos juntos** y tratarlos a todos como una unidad o descomponerlos en sus partes.

Se representa encerrando entre paréntesis **( )**, separando por comas a cada uno de los elementos que contiene y **anteponiendo una etiqueta** o nombre que permite identificarlos y así diferenciarlos de otras estructuras con que tienen la misma cantidad de componentes pero representan distintos elementos. Esta

forma de nomenclatura es compartida con los términos en general, y con los predicados, ya que internamente son manejados con la misma lógica.

Ejemplo:

fecha(2000, enero, 1)

fecha(1989, marzo, 26)

fecha es el nombre del functor con tres componentes que pueden ser usadas para representar fechas con año mes y día.

punto(2, 3)

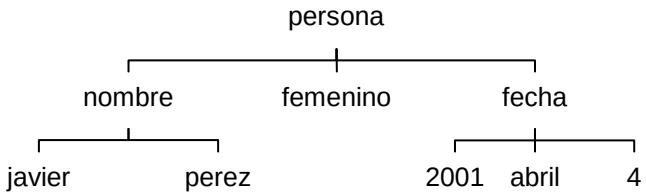
complejo(2, 3)

Ambos functores tienen los mismos componentes, pero en el caso de punto representan las coordenadas que indican su ubicación en el plano y en el caso de complejo representan a la componente real y la imaginaria.

Los componentes de una estructura pueden ser de cualquier valor, incluyendo **otras estructuras**, por lo que las estructuras son jerárquicas.

Ejemplo:

persona(nombre(javier, perez), femenino, fecha(2001, abril, 4))



Descomposición

En la declaración de una cláusula cualquiera, se puede tratar a **todo el functor como un conjunto**, unificándolo a una variable o **descomponerlo en sus componentes**, utilizando tantas variables o constantes como elementos tenga, entre paréntesis, separados con comas y con la etiqueta correspondiente.

Ejemplo:

ejeX(punto(0, _)).

ejeY(punto(_ , 0)).

Las reglas afirman que un punto está sobre el eje dado si el valor de dicha coordenada es 0, sin importar el valor de la otra coordenada.

adentro(punto(X, Y), R) :- X*X + Y*Y < R*R.

La regla devuelve si el punto de primer argumento está adentro del círculo cuyo radio es el segundo argumento y el centro es el origen de coordenadas.

origen(punto(0, 0)).

La cláusula afirma que un punto es el origen si ambas coordenadas tienen un valor **0**.

origen(P):- ejeX(P), ejeY(P).

Es otra forma de escribir el mismo predicado utilizando los predicados anteriores, sin descomponer el functor, unificándolo con la variable **P**. Afirma que un punto cualquiera es el origen si está sobre los dos ejes a la vez.

Se pueden realizar consultas como las siguientes:

?- origen(punto(0, 2)).

no

?- origen(P).

P = punto(0, 0).

?- ejeX(punto(0, 2)).

yes

?- ejeY(punto(5, 2)).

no

?- adentro(punto(1, 2), 3).

yes

?- adentro(punto(3, 3), 4).

no

Recordando las propiedades de inversibilidad, estas consultas no son válidas.

?- adentro(punto(1, 2), R).

La consulta intenta preguntar por el radio de todos aquellos círculos que incluyan al **punto(1, 2)**. Esto significa encontrar todo valor de **R** tal que **5 < R*R**, pero la relación **"<"** sólo funciona cuando ambos argumentos son expresiones ligadas, sin variables indeterminadas.

?- adentro(P, 10).

Se intenta preguntar por todos los puntos que estén dentro de un círculo de radio conocido. Esto significa encontrar todo valor de **X** e **Y** tal que **X*X + Y*Y < 100**, que tampoco es posible.

?- adentro(P, R).

Por ambos motivos anteriores, tampoco es posible hallar soluciones. Por lo tanto, la relación adentro funciona sólo cuando ambos argumentos están ligados.

Listas

Una lista representa una estructura de datos que contiene una **cantidad variable** de elementos, que pueden ser de **diferente tipo**, que están dispuestos secuencialmente dentro de la estructura. El tipo de dato de las listas está definido en forma **recursiva**, con dos posibilidades:

- Una lista es una estructura **vacía**
- Una lista es **un elemento** concatenado con **otra lista**

Una lista vacía se representa con [ ].

Una lista con elementos se representa enunciado el elemento ubicado en la primera posición (llamado comúnmente cabeza de la lista), unida por una barra vertical (**|**) a la otra lista (llamada comúnmente cola de la lista), todo encerrado entre corchetes (**[ ]**).

Ejemplo:

[1|[ ]]	Una lista con el entero 1 como primer elemento y la lista vacía como cola, por lo tanto una lista con un único elemento, el 1.
**[juan	[pedro	[ ]]**	Una lista con el valor juan como cabeza y una cola que es una lista formada a su vez por una cabeza pedro y una cola vacía. En definitiva, una lista con dos elementos: juan y pedro.

Las listas no vacías pueden ser construidas enunciando explícitamente sus elementos, separándolos con comas (**,**) y encerrando la expresión completa entre corchetes (**[ ]**). Esta notación es equivalente a la anterior para construir una lista con elementos constantes.

Ejemplo:

[1|[ ]]	es equivalente a	**[1]**
**[juan	[pedro	[ ]]**	es equivalente a	**[juan, pedro]**
**[juan, pedro	[raul]]**	es equivalente a	**[juan, pedro, raul]**

Descomposición

En la declaración de una cláusula cualquiera, se puede tratar a **toda la lista como un conjunto** o **descomponerla en su cabeza y su cola** para poder unificarlas por separado y luego ser utilizados sus valores en forma independiente, utilizando el carácter de la barra vertical (**|**). Se puede combinar el uso de variables y constantes.

Ejemplo:

Dado el hecho:

vocales([a, e, i, o, u]).

Se pueden realizar las siguientes consultas:

?- vocales(X).

X = [a, e, i, o, u]

?- vocales([H | T]).

H = a

T = [e, i, o, u]

?- vocales([X | _]).

X = a

Ejemplo:

Dado un predicado que dice si una lista empieza con el valor 0:

empiezaConCero([0 | XS]).

Ante una invocación:

?- empiezaConCero([0, 4, 6]).

yes

Se evalúa la consulta descomponiendo la lista en cabeza y cola. Se unifica la cabeza **0** del argumento que coincide con el valor **0** de la definición de la cláusula y la variable **Xs** se unifica con **[4, 6]**, que es la cola de la lista. Como se trata de un hecho universal y por lo tanto no tiene expresiones a evaluar, la respuesta es afirmativa.

Ante una invocación:

?- empiezaConCero([3, 4, 6]).

no

Al no coincidir los valores constantes del argumento con la definición de la cláusula no se produce la unificación, y no habiendo otras cláusulas a evaluar, se responde negativamente.

Para definir el predicado pudo haberse utilizado una variable anónima para la variable, ya que luego no es utilizada.

empiezaConCero([0 | _]).

Ejemplo:

Se define un predicado que permite saber si un elemento pertenece a una lista[22]:

pertenece([X | _], Y) :- X =Y.

pertenece([_ | Xs], Y) :- pertenece(Xs, Y).

La primera regla sólo es cierta cuando la cabeza de la lista que se recibe como argumento coincide con el segundo argumento. La segunda regla descompone la lista en

[22] Existe un predicado predefinido, llamado **member/2** que tiene la misma funcionalidad que **pertenece/2**

cabeza y cola, y vuelve a utiliza la cola como argumento en la invocación de una nueva consulta.

?- pertenece([3, 4, 6], 3).

yes

En esta consulta se descompone la lista, unificándose el **3** con la variable **X** y la lista **[4,6]** con la variable **Xs** y también se unifica el **3** del segundo argumento con la variable **Y**. Al evaluar luego la comparación entre **X** e **Y**, como ambos valen **3**, retorna verdadero, por lo tanto se satisface la consulta original.

?- pertenece([3, 4, 6], 4).

yes

Se unifica en la primera cláusula, pero al evaluar la igualdad se obtiene un valor falso. Se avanza entonces a la segunda cláusula, donde se unifica nuevamente el valor **3** con la variable **X** y la lista formada por los valores **4** y **6** con la variable **Xs**. Para encontrar la solución se invoca una nueva consulta del predicado **pertenece/2** con un argumento diferente.

pertenece([4, 6], 4).

La consulta retornará responderá que es verdadero (porque se asume que la regla funciona correctamente para un caso genérico menor que el que se está evaluando) y por lo tanto, no habiendo otras consultas por evaluar, también se retorna una respuesta afirmativa a la consulta original.

?- pertenece([ ], 4).

no

No hay ninguna cláusula que se unifique con la lista vacía del primer argumento, por que lo que la consulta retorna un valor negativo. Esta consulta puede haberse hecho originalmente con un lista vacía o ser consecuencia de invocaciones recursivas desde la misma regla pertenece, con una consulta original con un lista con varios elementos y un elemento que no se encuentra en la lista, en cuyo caso es correcto que la respuesta sea negativa.

?- pertenece([3, 4, 6], X).

X = 3;

X = 4;

X = 6

Se unifica en la primera cláusula y se obtiene la primera solución. Luego, al continuar evaluando la segunda cláusula, al ser variable el argumento se unifica nuevamente y mediante la invocación recursiva con la cola de la lista se obtienen las otras dos soluciones. En última instancia, ante la invocación con la lista vacía no se obtienen nuevas soluciones ya que como se muestra en el caso anterior no unifica con ninguna cláusula.

En predicado pudo haberse escrito sin antecedentes, es decir mediante un hecho universal utilizando dos veces la misma variable, de la siguiente manera.

pertenece([X | _], X).

pertenece([_ | Xs], Y) :- pertenece(Xs, Y).

Hay numerosas relaciones sobre listas que recursivas, dado que las listas son **estructuras recursivas**.

Ejemplo:

Las siguientes cláusulas definen el predicado un predicado que relaciona a un lista y un elemento con una lista que equivale a la primera lista con el elemento al final.

```
alFinal( [ ], X, [ X ] ).

alFinal( [ Y|Ys ], X, [ Y|Zs ] ) :- alFinal( Ys, X,  Zs ).
```

Significa que una lista vacía (**[ ]**) con un elemento cualquiera (**X**) se relacionan (mediante el predicado **alFinal/3**) con una lista únicamente con dicho elemento (**[X]**). En la otra regla, se afirma que si por medio de la relación **alFinal/3** una primer lista (**Ys**) con un elemento (**X**) al final es una segunda lista (**Zs**), entonces, una nueva primer lista (**[Y|Ys]**), formada por una cabeza cualquiera (**Y**) y una cola que es la primer lista anterior (**Ys**), con el mismo elemento (**X**) al final resulta en una nueva segunda lista (**[Y|Zs]**) formada por una cabeza igual a la primera (**Y**) y una cola que es la anterior segunda lista (**Zs**).

```
?- alFinal( [ ], 7, L ).

L = [ 7 ]
```

Es el caso básico, que unifica en la primera regla y no en la segunda.

```
?- alFinal( [ 2, 3, 5 ], 7, L ).

L = [ 2, 3, 5, 7 ]
```

Así invocada, la relación permite agregar un elemento al final de una lista. Lo hace recursivamente suponiendo que se lo puede agregar al final de la cola de esa lista.

```
?- alFinal( L, V, [ 2, 3, 5 ] ).

L = [ 2, 3 ] ,
V = 5
```

Con esta consulta, la regla separa un lista en dos, por un lado el último elemento y por otro la lista con todos los restantes.

```
?- alFinal( L, V, [ ] ).

no
```

La lista vacía, por definición no tiene ningún elemento y no se puede descomponer, por lo tanto, no unifica con ninguna de las dos reglas definidas y la consulta falla.

Ejemplo:

Las siguientes cláusulas definen un predicado que relaciona a dos listas con una tercera que es la concatenación de las dos primeras.

```
concatenacion( [ ], Ys, Ys ).

concatenacion( [ X|Xs ], Ys, [ X|Zs ] ) :- concatenacion( Xs, Ys, Zs ).
```

Concatenando la lista vacía y la lista **Ys** se obtiene la lista **Ys**, y concatenando las listas **[X|Xs]** e **Ys** se obtiene la lista **[X|Zs]**, sendo **Zs** la concatenación de las listas **Xs** e **Ys**.

Las consultas:

?- concatenacion([2, 3], [5, 7], L).

L = [2, 3, 5, 7]

?- concatenacion([2, 3], L, [2, 3, 5, 7]).

L = [5, 7]

?- concatenacion(L1, L2, [5, 7]).

L1 = [ ]
L2 = [5, 7] ;

L1 = [5]
L2 = [7] ;

L1 = [5, 7]
L2 = [ ]

En el último caso, las respuestas alternativas son todas las formas posibles de dividir la lista en sublistas.

La relación **concatenacion/3** puede ser usada para concatenar dos listas dadas, o para remover una lista dada del frente (o el final) de otra, o para encontrar todas las formas de dividir una lista dada. En lenguajes de otros paradigmas, habría que desarrollar diferentes funciones o subprogramas para realizar cada uno de estos cálculos por separado.

?- concatenacion([2, 3], X, L).

?- concatenacion(X, [2, 3], L).

?- concatenacion(X, Y, L).

Estas consultas no arrojan soluciones correctas ya que se produce una indeterminación que limita la inversibilidad del predicado, dado que las listas que podrían darse como respuesta son incontables, podrían tener cualquier longitud y contenido.

Ejemplo:

Se quiere averiguar si un elemento pertenece a la lista resultante de la concatenación de dos listas, utilizando el predicados **concatenacion/3** y **pertenece/2** de ejemplos anteriores, no es válido utilizar expresiones como:

?- L = concatenacion([2, 4, 5], [3, 6], X), pertenece(L, 3). --ERROR

?- pertenece (concatenacion([2, 4, 5], [3, 6], X), 3). --ERROR

Recordando el concepto de predicado, la invocación a **concatenacion/3** no es una expresión del estilo de una "función" que devuelve en sí misma la lista resultante para ser pasada como argumento de otro predicado o unificada con una variable.

En el caso anterior, la solución a la incógnita planteada, o sea, la lista que es concatenación de las otras dos, está representada en **X**, por lo tanto, se debe hacer:

?- concatenacion([2, 4, 5], [3, 6], X), pertenece(X, 3).

Ejemplo:

Las siguientes cláusulas definen un predicado que relaciona a una lista con otra, en la que están sólo los números positivos de la primera lista.

```
positivos( [ ], [ ] ).
positivos( [ X|Xs ], [ X|Ys ] ):- X  >  0, positivos( Xs, Ys ).
positivos( [ X|Xs ],     Ys     ):- X =< 0, positivos( Xs, Ys ).
```

La invocación:

```
positivos( [ 1, -5, -2, 4 ], L ).
L = [ 1, 4 ]
```

Se puede ver cómo el predicado plantea dos cláusulas recursivas, parecidas, con las primeras proposición a evaluar que son mutuamente excluyentes. En cada invocación, cuando se satisface completamente una de las cláusulas, la otra necesariamente falla. Se interpreta que cuando el elemento **X**, cabeza de la primer lista, es positivo, queda como cabeza de la otra lista. Cuando no es positivo, no queda en la otra lista.

Ejemplo:

Para relacionar una lista con su longitud, una posible definición es[23]:

```
longitud( [ ], 0 ).
longitud ( [_|Cola], Cont ):-
                    longitud ( Cola, Cont1 ),
                    Cont is Cont1 + 1.
```

Se interpreta como que la cantidad de elementos de una lista vacía es 0 (el caso base), y en otro caso, si la lista tiene elementos, la cantidad de elementos de la lista completa coincide con la cantidad de elementos de la cola más 1. Para resolver la cuenta, se utiliza el predicado predefinido **is/2**[24].

```
?- longitud( [ 5, 4 ], X ).
X = 2

?- longitud( [ ], X ).
X = 0
```

Ejemplo:

Para calcular la sumatoria de los elementos de una lista, se puede definir:

```
sumatoria( [ ], 0 ).
sumatoria( [ Cab|Cola ], S ):-
                sumatoria( Cola, SCola),
                S is SCola + Cab.
```

La sumatoria de la lista equivale a la sumatoria de la cola más la cabeza.

[23] Existe un predicado predefinido, llamado **length/2** que tiene la misma funcionalidad que **longitud/2**

[24] Ver anexo "Prolog" – Cálculos aritméticos

Utilización

Las listas son estructuras muy utilizadas dentro del paradigma, en combinación con los otros elementos. En los ejemplos anteriores, se les utilizaba como argumentos de predicados de uso genéricos para procesamiento de listas. En general, en la base de conocimiento muchos de los hechos pueden incluir listas, por lo que predicados que en sus argumentos no incluyen listas requieren de su uso auxiliar.

La mayoría de las responsabilidades sobre conjuntos de elementos se realiza con listas. Incluso las soluciones alternativas a una consulta, que son devueltas como soluciones independientes, pueden tratarse como listas. [25]

A continuación, ejemplos más complejos de resolución de problemas mediante la utilización de listas.

Ejemplo:

El siguiente programa permite relacionar a dos clubes con sus jugadores en común. En la base de conocimiento, para cada jugador se conoce la historia de todos los clubes por los que pasó.

```
historia( diego, [boca, velez, central, estudiantes] ).
historia( pepe, [boca, central] ).
historia( roberto, [colon, chicago] ).
historia( jose, [estudiantes, central, velez] ).
enComun( X, Club1, Club2 ):-
        historia( X, Lista ),
        member(Club1, Lista),
        member(Club2, Lista),
        Club1 \= Club2.
```

Dos clubes (**Club1** y **Club2**) tienen a un jugador (**X**) en común, si ambos clubes son miembro de la lista con la historia del jugador. Además, se valida que sean diferentes clubes.

Algunas consultas posibles son:

```
?- enComun( X, boca, central ).
X = diego
X = pepe

?- enComun( pepe, Y, Z ).
Y = boca
Z = central
Y = central
Z = boca
```

[25] Ver capítulo 9 "Polimorfismo y orden superior"

```
?- enComun( X, boca, Z ).

X = diego
Z = velez

X = diego
Z = central

X = diego
Z = estudiantes

X = pepe
Z = central
```

Ejemplo:

Dada la siguiente base de conocimiento.

```
empresa( [ administracion, personal, produccion ] ).
```

Están registradas todas las secciones de una empresa.

```
empleados( administracion, 3 ).

empleados( personal, 1 ).

empleados( produccion, 10 ).
```

Se sabe cuántos empleados trabajan en cada sección.

```
sueldoPromedio( administracion, 1000 ).

sueldoPromedio( personal, 2000 ).

sueldoPromedio( produccion, 500 ).
```

Se conoce también el sueldo promedio de los empleados de cada sección.

```
ventas( 9000 ).
```

Es la información del monto que se recauda en concepto de ventas.

El predicado **recorte/1** calcula el porcentaje de ajuste que habría que aplicar sobre los sueldos de los empleados para que los costos no superen el importe de ventas.

```
recorte( Porc ):-
        empresa( Sectores ),
        totalSueldos( Sectores, Tot ),
        ventas( Venta ),
        porcentaje( Venta, Tot, Porc ).
```

Dentro de la regla, con la primer consulta **empresa/1** se unifica **Sectores** con la lista de sectores de la empresa. La consulta por **totalSueldos/2**, a partir de que la variable **Sectores** está unificada, permite obtener en la variable **Tot** el total de gastos de sueldos. La consulta por el predicado **ventas/1** remite al hecho de la base de conocimiento por el que la variable **Venta** asume un valor constante. Por último, el predicado **porcentaje/3**, unifica un valor para **Porc**, a partir de los valores que tengan **Tot** y **Venta**.

```
totalSueldos( [ ],  0 ).
totalSueldos( [S| Secs ], Tot ):-
        sueldoPromedio( S, Sueldo ),
        empleados( S, Cant ),
```

```
totalSueldos( Secs, TotAux ),
Tot is TotAux + Sueldo * Cant.
```

Es una estructura lógica similar al ejemplo anterior de sumatoria, pero en vez de sumarle a la sumatoria de la cola (**TotAux**) el valor de la cabeza de la lista (**S**), se suma una expresión mas compleja, con los valores **Sueldo** y **Cant** que surgen de las consultas a **sueldoPromedio/2** y **empleados/2**, respectivamente.

```
porcentaje( Ingreso, Gasto,  0 ):-      Ingreso >= Gasto.
porcentaje( Ingreso, Gasto, Porc ):-
        Ingreso < Gasto,
        X is Tot - Ingreso,
        Porc is X * 100 / Gasto.
```

La primera regla afirma que el recorte es nulo cuando los ingresos superan el total de gastos. La segunda plantea que si las ventas son menores a los gastos, y se resuelven las operaciones aritméticas del cálculo porcentual, entonces el valor **Porc** representa el porcentaje de ajuste.

Con la base de conocimiento dada, la consulta clave es

```
?- recorte( X ).
X = 10
```

Si se modifican el hecho

```
ventas( 12000 ).
```

El resultado de la consulta varía:

```
?- recorte( X ).
X = 0.
```

Capítulo 9

Polimorfismo y orden superior

- **Polimorfismo**
- **Predicados de orden superior**
- **Negación**
- **Listas con respuestas alternativas**
- **Predicados predefinidos**
- **Llamadas de orden superior**

Polimorfismo

El **polimorfismo permite obtener soluciones más genéricas**, que sean válidas para **diferentes tipos de datos contemplando las particularidades** de cada uno de ellos. Se basa en el carácter débilmente tipado del lenguaje, que permite que no haya que predeterminar el tipo de dato de cada argumento de un predicado, variable, functor o cualquier término.

Si bien todos los predicados pueden recibir cualquier tipo de argumento, muchas veces el uso que se hace de ellos en el interior de las cláusulas que lo definen, delimita un rango de tipos de valores que tiene sentido recibir. No sucede un error si se envía un dato de diferente a los esperados, sino que en general directamente el predicado falla por no poder unificar.

Ejemplo:

Un predicado **ultimo/2** que relaciona una lista con el último elemento, puede ser definido así:

```
ultimo( [ X ], X )
ultimo( [ _|Xs ], X ):- ultimo( Xs, X ).
```

Si se invoca con una lista que contenga cualquier tipo de elementos, el predicado funciona correctamente

```
?- ultimo( [ juan, pedro ], X ).
X = pedro

?- ultimo( [ 1, 2, 3, 4 ], X ).
X = 4
```

Pero si se invoca en el primer argumento con otro tipo de dato que no sea una lista

```
?- ultimo( 6, X ).
no
```

La consulta falla unifica debido a que el valor **6** no unifica en ninguna de las dos cláusulas del predicado, que por su misma definición, sólo unifican con listas.

En forma similar sucede con los functores. El predicado **sueldo/2** relaciona un functor que representa a una persona que trabaja de jefe, donde su primer componente es el nombre, la segunda es el sueldo básico y la tercera es un adicional, con el sueldo total que le corresponde, como suma de los dos valores.

```
sueldo( jefe( _, Basico, Adicional ), Sueldo) :- Sueldo is Basico + Adicional.
```

Las invocaciones que respetan el functor del primer argumento, funcionan correctamente:

```
?- sueldo( jefe( laura, 3000, 800 ), X ).
X = 3800
```

Las invocaciones que en el primer argumento usen valores o expresiones de cualquier otro tipo, fallan siempre.

```
?- sueldo( laura, X ).
No
```

Una forma de aprovechar al máximo la propiedad polimórfica de que un predicado pueda recibir cualquier tipo de dato y tratarlos indistintamente, cuando luego en las cláusulas se utilizan expresiones que limitan los tipos de datos que dan respuestas positivas, es **agregar más cláusulas** para contemplar los **otros posibles tipos de datos**, y tratarlos de la forma adecuada.

Ejemplo:

Continuando con el ejemplo anterior, se pueden ampliar la base de conocimiento y agregar nuevas cláusulas del predicado sueldo, para que calcule el sueldo para el personal de una empresa representados con diferentes functores, cada uno a su manera.

```
basico( categoria1, 1000 ).
basico( categoria2, 1500 ).
valorHora( 15 ).
```

Son nuevos hechos

sueldo(jefe(_, Basico, Adicional), Sueldo) :- Sueldo is Basico + Adicional.

Se mantiene igual al ejemplo anterior.

sueldo(empleado(_, Categoría), Sueldo) :- basico(Categoria, Sueldo).

Esta nueva regla indica que los empleados de la empresa cobran un sueldo correspondiente a su categoría. Para ello también se agregan hechos del predicado **basico/2**.

sueldo(contratado(_, Cant), Sueldo) :- valorHora(Valor), Sueldo is Valor * Cant.

En esta otra, se indica que el personal contratado cobra un sueldo según la cantidad de horas que trabajó cada uno y el valor de la hora que está registrado con el predicado valorHora.

Pueden hacerse nuevas consultas:

?- sueldo(empleado(juan, categoria1), X).

X = 1000

?- sueldo(empleado(raul, categoria2), X).

X = 1500

?- sueldo(contratado(esteban, 20)], X).

X = 300

Completando el ejemplo, se incorpora un hecho con el predicado **personal/1** con una lista donde están todos los empleados, jefes y contratados de la empresa.

personal([empleado(juan, categoria1), empleado(ana, categoria1), empleado(raul, categoria2), jefe(jorge, 2000, 500), jefe(laura, 3000, 800), contratado(esteban, 20)]).

También se agrega un nuevo predicado **nombre/2**, con varias cláusulas en las que se relaciona al functor que representa a cada trabajador con su nombre.

nombre(jefe(Nombre, _, _), Nombre).

nombre(empleado(Nombre, _), Nombre).

nombre(contratado(Nombre, _), Nombre).

A partir de lo anterior, se define un predicado **cuantoGana/2**, que relaciona el nombre de una persona que trabaja en la empresa con su correspondiente sueldo.

```
cuantoGana( Nombre, Sueldo ):-
        personal( Lista ),
        member( P, Lista ),
        nombre( P, Nombre ),
        sueldo( P, Sueldo ).
```

Se pueden realizar diferentes consultas, siendo la más amplia:

?- cuantoGana(Nombre, Sueldo).

Nombre = juan
Sueldo = 1000 ;

```
Nombre = ana
Sueldo = 1000 ;

Nombre = raul
Sueldo = 1500 ;

Nombre = jorge
Sueldo = 2500 ;

Nombre = laura
Sueldo = 3800 ;

Nombre = esteban
Sueldo = 300
```

El polimorfismo permite poder construir **predicados genéricos** que contemplen un abanico de casos posibles que se definan y a la vez permitir que ante modificaciones futuras o nuevos problemas a resolver, el impacto del cambio sea mínimo, de manera que muchos predicados puedan quedar expresados de la misma manera y seguir funcionando correctamente.

Ejemplo:

El predicado **cuantoGana/2** es lo suficientemente polimórfico como para soportar cualquier tipo de trabajador con que se lo invoque y seguirá siendo válido, sin modificaciones, para cualquier otro tipo de dato como primer argumento, siempre y cuando los predicados **nombre/2** y **sueldo/2** lo contemplen adecuadamente.

Se agrega otro tipo de trabajador, con un cálculo de sueldo en particular, que para el ejemplo se resuelve con otro predicado auxiliar **calculoCualquiera/4** a partir de los datos del nuevo functor.

```
nombre( unTrabajador( Nombre, _, _, _ ), Nombre ).

sueldo( unTrabajador( _, X, Y, Z ), Sueldo) :- calculoCualquiera( X, Y, Z, Sueldo ).
```

Se pueden implementar soluciones polimorfitas con **cualquier tipo de dato**. Para los que tienen una nomenclatura específica, como los functores o las listas, es más simple y se expresa directamente en la forma de los argumentos para que el mecanismo de unificación ("pattern matching") lo resuelva. Para los otros, se pueden incorporar en cada regla, consultas de **comprobación de tipos de datos**[26].

Predicados de orden superior

El concepto de **orden superior** es la capacidad de un lenguaje para manejar su propio código como una estructura de datos más.

Las capacidades de orden superior del lenguaje, son un conjunto de funcionalidades generalmente desconocidas en lenguajes de diferentes paradigmas que dotan de una **enorme expresividad y potencia a los**

[26] Ver anexo "Prolog" - Comprobación de tipos de datos

programas. También son una forma de implementar **polimorfismo** en el paradigma lógico.

En particular, un predicado de orden superior es aquél que tiene la capacidad de **recibir un predicado como argumento y poder utilizarlo**. Se escribe un predicado y se lo pasa como argumento a otro y esta último lo ejecuta sin necesidad de saber exactamente qué está evaluando.

Existen algunos **predicados predefinidos** de orden superior que resultan muy útiles y son ampliamente utilizados y también es posible **definir predicados propios** de orden superior.

En cualquier predicado de orden superior, en el argumento que se recibe un predicado, puede enviarse una consulta simple o una expresión que incluye a varias consultas entre paréntesis, ya sea unidas por comas (,) o puntos y comas (;) para representar consultas simultáneas o alternativas respectivamente.

Negación

El predicado de orden superior más simple, y a la vez muy utilizado, es el **not/1**, que representa una negación.

Si **P** es una proposición, entonces **not(P)** es una proposición que niega el valor de verdad asumido para **P**. Así, la negación de algo falso se toma por verdadera.

Ejemplo:

Dada la base de conocimiento:

```
varon( pedro ).
varon( juan ).
varon( raul ).
mujer( X ) :- not( varon( X ) ).
```

Sean las consultas:

```
?- varon(pedro).
yes

?- varon( ana ).
no

?- mujer( pedro ).
no
```

?- mujer(ana).

yes

Las consultas por **varon/1** se responden directamente en función de los hechos. Las consultas por **mujer/1**, evalúan la regla **varon/1** y responden el valor opuesto.

El uso de not encierra tiene también algunas **limitaciones** y se pueden escribir cláusulas que fácilmente engañosas.

Ejemplo:

Continuando con el ejemplo anterior

?- mujer(carlos).

yes

?- mujer(sultan).

yes

?- mujer(cualquiercosa).

yes

Todo elemento que en la base de conocimiento no esté registrado como varon va a ser deducido como que es mujer.

Otra característica es que las variables del predicado que se niega deben estar ligadas. Cuando se evalúa una consulta donde sus variables no han sido instanciadas, los valores que busca son los que hacen cierta la expresión y luego al ser negados dan un valor falso. No hay forma que se busquen los valores que dan resultado falso para que al negarlos se obtenga un valor verdadero. Sólo si no hay ninguna solución para la consulta variable que es argumento del **not/1** y por lo tanto falla, la consulta con la negación dará verdadero, pero sin ligar a la variable libre con un valor.

Ejemplo:

Con el mismo ejemplo, la consulta:

?- mujer(M).

No puede encontrar soluciones, ya que se evalúa la consulta **varon(X)** no permite obtener valores de **X** que hagan falsa la expresión.

Una forma de obtener a todas las mujeres sería incluyendo un predicado **persona/1** con hechos que representen a toda las personas ya sean varones o mujeres, dejando los hechos de **varon/1** y modificando el predicado **mujer/1**.

persona(pedro).

persona(juan).

persona(raul).

persona(ana).

persona(julia).

mujer(X) :- persona(X), not(varon(X)).

De esta manera, se afirma que una mujer es una persona que no es varón. Al consultar primero por **persona(X)**, la variable **X** se unifica y cuando se evalúa la consulta **varon(X)**, se trata de una consulta de validación, sin variables indeterminadas, por lo que el **not/1** funciona correctamente. En otras palabras, **persona/1** actúa como predicado generador[27].

Ejemplo:

El siguiente predicado relaciona tres listas, cuando la tercera contiene la unión de los elementos de las dos primeras listas, sin repetidos.

```
unionSinRepetidos(    [ ]    , YS,    YS   ).
unionSinRepetidos( [X|XS], YS,    ZS   ):-
                        member( X, YS ),
                        unionSinRepetidos( XS, YS, ZS ).
unionSinRepetidos( [X|XS], YS, [X|ZS] ):-
                        not( member( X, YS ) ),
                        unionSinRepetidos( XS, YS, ZS ).
```

El caso base indica que si una de las listas es vacía, la otra, cualquiera sea, coincide necesariamente con la tercera.

En la primera regla recursiva, se afirma que la cabeza de la primera lista también es cabeza de la lista unión del tercer argumento (**[X|ZS]**) si cumple con la condición de ser miembro de la segunda lista, mediante **member(X, YS)**, siendo **ZS** la unión de los demás elementos de la primer lista con toda la segunda, lo que se garantiza con la invocación recursiva. En la otra regla, si se verifica que no cumple con dicha condición mediante **not(member(X, YS))** , el elemento no integra la lista unión (**ZS**), por lo que el tercer argumento de la regla es sólo la lista unión (**ZS**) que surge del llamado recursivo.

Ejemplo:

De cada profesión se conoce el conjunto de nombres de los profesionales

```
profesion( programador, [juan] ).
profesion( diseniador, [raul, jorge, juan] ).
profesion( analista, [diego, ana] ).
```

El predicado **equipo/3** permite relacionar a cualquier lista de profesiones, con cualquier listas de profesionales, en el que cada uno de los profesionales es de la profesión de la posición correspondiente de la otra lista. En otras palabras, relaciona a un conjunto de profesiones con un equipo interdisciplinario de profesionales. Es importante que en el equipo no se repita el profesional dos veces, ya que es posible consultar por un equipo con profesiones que se repiten o un profesional puede tener más de una profesión.

```
equipo( [ ], [ ] ).
equipo( [P|Ps], [X|Xs] ):-
```

[27] Ver capítulo 7 "Estrategias de resolución"

```
equipo( Ps, Xs ),
profesion( P, Lista ),
member( Lista, X ),
not( member( Xs, X )).
```

La consulta por **not(member(Xs,X))** implica que conociendo la cola de la lista del equipo de profesionales (**Xs**), ya unificada mediante el llamado recursivo, y conociendo al posible profesional que completaría al equipo (**X**), unificado por la consulta por si es miembro de la lista de profesionales (**Lista**) de la primera profesión (**P**) de la lista de profesiones, se verifica que no sea uno de los que ya forma parte del equipo.

```
?- equipo( [ programador, diseniador ], [ juan, raul ] ).
Yes
```

Esta consulta verifica que **juan** y **raul** forman un equipo consistente con las profesiones programador y diseñador.

```
?- equipo( [ programador, diseniador ], Y ).
Y = [ juan, raul ]
Y = [ juan, jorge ]
```

Se buscan todos los posibles equipos formados por un programador y un diseñador. Como el tercer posible diseñador, **juan**, ya está en el equipo por ser también programador, no se devuelve la solución formada por dos veces **juan**.

```
?- equipo( [ analista, analista ], Y ).
Y = [ ana, diego ]
Y = [ diego, ana ]
```

Se buscan todos los posibles equipos formados por dos analistas. Como se verifica que no se pueden repetir los profesionales, son sólo estas dos las soluciones. Por más que sean los mismos integrantes en las dos soluciones, se trata de listas diferentes. Ambas verifican todas las condiciones planteadas en el predicado.

```
?- equipo( [ analista, programador, diseniador ], Y ).
Y = [ diego, juan, raul ]
Y = [ ana, juan, raul ]
Y = [ diego, juan, jorge ]
Y = [ ana, juan, jorge ]
```

Son todas las combinaciones posibles.

```
?- equipo( Y, [ diego, juan, raul ] ).
Y = [ analista, programador, diseniador ],
```

También se puede hacer la consulta con el otro argumento como incógnita.

Listas con respuestas alternativas

Hay un conjunto de predicados cuya finalidad es almacenar en una lista todas las soluciones de un predicado dado, entendiendo como tales, las ligaduras que se producen en una o varias variables libres que se indican explícitamente. Algunos de ellos son los siguientes:

- **findall/3**

Genera una lista con todas las soluciones del predicado dado según el orden en que se van sucediendo.

Tiene tres argumentos: una primera expresión, la consulta que se quiere evaluar y la lista con un elemento para cada solución alternativa a la consulta.

El primer argumento es una variable o un término que contiene la o las variables libres que interesan de la solución. Es posible que no interesen todas, por lo que se indican sólo las variables necesarias, que han de ser un subconjunto de las que aparecen en el segundo argumento.

El segundo argumento es el predicado del cual se quiere obtener soluciones, para ello, debe contener una o más variables sin ligar. Por otra parte, dicha consulta debe tener un número finito de soluciones, si no, se entraría en un bucle infinito al ejecutar alguno de los predicados.

El tercer argumento es la lista con las soluciones. Cada elemento de la lista va a tener la forma de la expresión del primer argumento. Si no hay soluciones, se obtiene la lista vacía. La lista de soluciones no se ordena y puede contener elementos repetidos. Solamente se obtiene una solución.

Ejemplo:

Ante una invocación de la forma

?- findall(T, C, L)

L es la lista de todas las instancias del término **T** tales que la consulta **C** se cumple. Tras la consulta, las variables de **T** y **C** permanecen sin unificar.

Ejemplo:

Usos correctos de **findall/3**:

findall(X, predicado(X), Lista).

findall(X, predicado(X, Y), Lista).

findall(elemento(X, Y), predicado(X, Y), Lista).

Estas son algunas formas de uso incorrecto del predicado, en las que hay variables del primer argumento que no aparecen en el segundo.

findall(X, predicado(Y), Lista).

findall(X, predicado(Y, Z), Lista).

findall(elemento(X, Y), predicado(Y, Z), Lista).

Ejemplo:

Dada la siguiente base de conocimiento, donde se indica cada materia que cursa cada alumno:

```
cursa( juan, paradigmas ).
cursa( jose, paradigmas ).
cursa( juan, operativos ).
cursa ( ana, sintaxis ).
```

Mediante la siguiente consulta:

```
?- findall( Y, cursa( X, Y ), Lista ).
Lista = [ paradigmas, paradigmas, operativos, sintaxis ]
```

Se obtiene una lista con las soluciones del segundo argumento de **cursa/2**. Las soluciones del primer argumento, con la variable **X**, se ignoran. En la lista aparecen todas las soluciones, incluidas las repetidas. En este caso, son todas las materias que alguien cursa.

```
?- findall( par( X, Y ), cursa( X, Y ), Lista ).
Lista = [ par( juan, paradigmas ), par( jose, paradigmas ), par( juan, operativos ), par( ana, sintaxis ) ]
```

Se obtiene una lista con las soluciones de los dos argumentos de **cursa/2**. Se las ubica en la lista con la estructura del functor indicado como primer argumento (**par/2**). Son todos los pares de alumno y materia que cursa que hay en la base de conocimiento.

Ejemplo:

Un predicado común de manejo de listas como la concatenación se puede resolver con **findall/3**.

```
concatenacion( XS, YS, ZS ):-
        findall( X, ( member( X, XS ); member( X, YS) ), ZS ).
```

El argumento que se recibe como un predicado está formado por dos consultas, unidas lógicamente por una relación de "o" (**;**). Se puede leer: la concatenación de dos listas es la lista formada por todos los elementos que cumplen la condición de ser miembros de la primera lista o ser miembros de la segunda.

Ejemplo:

```
intersección( XS, YS, ZS ):-
        findall( X, ( member( X, XS ), member( X,YS) ), ZS ).
```

Un predicado **intersección/3** relaciona dos listas con una tercera que contiene los elementos comunes a ambas, se expresa de manera muy parecida a la anterior, pero con el segundo argumento con una consulta doble unida por una relación de "y" (**,**), validando que los elementos sean miembros de las dos listas simultáneamente.

- **bagof/3**

Similar a **findall/3**, ya que genera una lista con todas las soluciones del predicado dado. La diferencia es que tiene una solución por cada variable libre que no haya sido indicada como parte de la solución y además falla cuando no hay soluciones.

Ejemplo:

Para la misma base de conocimiento del ejemplo anterior.

```
?-- bagof( Y, cursa ( X, Y ), Lista ).

X = juan
Lista = [ paradigmas, operativos ] ;

X = jose
Lista = [ paradigmas ] ;

X = ana
Lista = [ sintaxis ]
```

Para cada solución de las variables libre, en este caso **X**, la lista contiene todas las soluciones de las otras variables, en este caso **Y**. O sea, la lista de materias que cursa cada alumno.

```
?-- bagof( X, cursa (X, Y), Lista).

Y = paradigmas
Lista = [ juan, jose ] ;

Y = operativos
Lista = [ juan ] ;

Y = sintaxis
Lista = [ ana ]
```

Para cada solución de las variables libre, en este caso **Y**, la lista contiene todas las soluciones de las otras variables, en este caso **X**. O sea, la lista de alumnos que cursa cada materia.

- **setof/3**

Similar a **bagof/3**. La diferencia es que no incluye las soluciones duplicadas que pudieran existir y que ordena la lista del tercer argumento.

Ejemplo:

Como una alternativa para uno de los predicados auxiliares de los ejemplos anteriores, el siguiente predicado relaciona dos listas con una tercera que es la unión sin repetidos de todos los elementos de ambas listas.

```
unionSinRepetidos( XS, YS, ZS ):-
                    setof( X, ( member( X, XS ); ( member( X, YS ) ), ZS ).
```

Predicados predefinidos

Existe una cantidad enorme de otros predicados de orden superior que realizan las más diversas tareas. A continuación se enumeran sólo algunos.

- **forall/2**

Para todas las soluciones del predicado del primer argumento, evalúa el predicado del segundo argumento. Si alguno de los predicados de los argumentos falla, falla el predicado principal, sin que haya forma de indicar cuál de los predicados falló.

Ejemplo:

Para definir un predicado que indique si un conjunto, representado por una lista, está incluido en otro, sin importar el orden de los elementos, se podría desarrollar una formulación recursiva. Otra forma es invocando al presente predicado.

```
incluido( L1, L2 ):- forall( member( X, L1 ), member( X, L2 ) ).
```

Se afirma que una lista **L1** está incluida en otra lista **L2**, si para todo elemento **X** que es miembro de la lista **L1** se verifica que también es miembro de la lista **L2**.

- **maplist/3**

Permite aplicar un predicado de "mapeo" entre dos listas de elementos, por la cual cada elemento de una lista se relaciona mediante el mapeo con el elemento de la misma posición de la otra lista.

Ejemplo:

El predicado **capital/2** relaciona cada país con su capital.

```
capital( argentina, buenosaires ).
capital( peru, lima ).
capital( urugual, montevideo ).
```

Para obtener una lista con las capitales, a partir de una lista de países, se puede hacer:

```
?- maplist( capital, [ argentina, peru ], Lista ).
Lista = [ buenosaires, lima ].
```

Si se agregan a la base de conocimiento nuevos hechos:

```
capital( bolivia, lapaz ).
capital( bolivia, sucre ).
?- maplist( capital, [ argentina, peru, bolivia ], Lista ).
Lista = [ buenosaires, lima, lapaz ] ;
Lista = [ buenosaires, lima, sucre ]
```

Se obtienen dos respuestas alternativas, dado que el predicado que se evalúa también tiene dos soluciones para uno de los valores. Si hubiera más alternativas para cada consulta individual, se devolverían todas las posibles listas con las combinaciones de valores.

Llamadas de orden superior

Para desarrollar predicados de orden superior es necesario poder evaluar los predicados que se reciben como argumentos. Para ello, y en general, para convertir cualquier término o expresión en una consulta, existe el predicado **call**, con diferentes aridades, de acuerdo a la aridad del predicado que se quiere evaluar.

El primer argumento es el nombre del predicado y los siguientes son los argumentos con los que se desea evaluar al predicado.

Ejemplo:

Un predicado de orden superior llamado **selecciona/3**[28], que permite obtener una sublista con los elementos de una lista que cumplen cierta condición, recibiendo como primer argumento dicha condición que es un predicado de un solo argumento.

```
selecciona( _, [ ], [ ] ).
selecciona( P, [ X|Xs ], [ X|Ys ] ) :-
                call( P, X ),
                selecciona( P, Xs, Ys ).
selecciona( P, [ X|Xs ], Ys ) :-
                not( call( P, X ) ),
                selecciona( P, Xs, Ys ).
```

La variable **P** se unifica con el predicado que se envíe como argumento. Al evaluar **call**, se envía como primer argumento el predicado **P** y como segundo una variable para que se evalúe como el único argumento de **P**. Si se verifica, el elemento **X** es seleccionado y forma parte de la lista del tercer argumento, y si no, se lo excluye.

Se esperan predicados **P** con un solo argumento, ya que **call** está siendo invocado con dos argumentos: **P** y el argumento de **P**.

Una forma de uso del predicado **selecciona/3** puede ser:

```
varon( juan ).
varon( pedro ).
?- selecciona( varon, [ olga, juan, estela, pedro ], L).
L = [ juan, pedro ]
```

Ejemplo:

Con los mismos conceptos se puede construir un predicado **mapea/3** que tenga la misma funcionalidad que el predicado predefinido **maplist/3** mencionado anteriormente.

```
mapea( _, [ ], [ ] ).
```

[28] Existe un predicado predefinido con esta misma funcionalidad, llamado **sublist/3**

```
mapea( P, [ X|Xs ], [ Y|Ys ] ) :-
                call( P, X, Y ),
                mapea( P, Xs, Ys ).
```

La variable **P** se unifica con el predicado que se envíe como argumento. Como se esperan predicados **P** con dos argumentos, que puedan relacionar los elementos de una lista con los de la otra, se invoca a **call** con tres argumentos: **P** y sus dos argumentos.

Ejemplo:

Combinando varias de los predicados de los ejemplos anteriores, el siguiente programa es más complejo y permite relacionar a dos novios con la lista de invitados para la fiesta de su casamiento. Para ello hay información en la base de conocimiento:

```
amigos( maria, [ lau, charly, caro, ceci, ana, fede, marce, juan ] ).
amigos( leo, [ charly, lucas, santi, pipa, chiqui ] ).
```

Representa la lista de personas que son amigas de cada uno de los novios.

```
muyAmigo( maria, caro ).
muyAmigo( maria, charly ).
muyAmigo( leo, charly ).
```

Representa las personas que son muy amigas de cada uno de los novios.

```
familia( perez, [ isa, carlos, maria ] ).
familia( sanchez, [ leo, gaston ] ).
```

Son los familiares de cada uno. El primer argumento es el apellido de la familia y cada novio está dentro de la lista de su correspondiente familia.

```
invitoASuFiesta( maria, [ santi, juan ] ).
invitoASuFiesta( leo, [ santi ] ).
```

Estas listas incluyen a los amigos que invitaron previamente a su casamiento a alguno de los novios.

El predicado **invitados/3** tiene como primeros argumentos a los dos novios y como tercer argumento a la lista de invitados. El criterio de armado de la lista consiste en que cada uno va a invitar, de entre todos sus amigos, a los que sean muy amigos o quienes los hayan invitado previamente a su casamiento. También van a invitar a los miembros de la familia. A aquellas personas que tengan motivos para ser invitadas por ambos novios, no aparecerán dos veces en la lista de casamiento. Los mismos novios también integran la lista.

```
invitados( Novio, Novia, Lista ):-
                invitadosUno( Novio, L1 ),
                invitadosUno( Novia, L2 ),
                unionSinRepetidos( L1, L2, Lista ).
```

La lista de invitados (**Lista**) es la unión sin repetidos de la lista que arma cada novio con el predicado **invitadosUno/2**.

```
invitadosUno( Novio, Lista ):-
                invitaFamilia( Novio, L1 ),
```

```
                invitaAmigos( Novio, L2 ),
                concatenacion( L1, L2, Lista ).
```

La lista de invitados de cualquiera de los novios (**Lista**) es la concatenación de la lista de familiares y de amigos invitados.

```
invitaFamilia( Novio, Lista ):-
                familia( _, Lista ),
                member( Novio, Lista ).
```

La lista de familiares de cualquiera de los novios es la lista de la familia, que sin importar el apellido, de la cual el novio es miembro.

```
invitaAmigos( X, LS ):-
                amigos( X, XS ),
                sublist( invitado( X ), XS, LS ).
```

La lista de amigos a invitar de cualquiera de los novios es un subconjunto de la lista de todos sus amigos.

```
invitado( Novio, A ):-
                muyAmigo( Novio, A ).
invitado( Novio, A ):-
                invitoASuFiesta( Novio, L ),
                member( A, L ).
```

En la primera regla, si el amigo es muy amigo, entonces lo invita. En la segunda, si el amigo es miembro de la lista de aquellos que lo invitaron a su fiesta, también lo invita.

Finalmente, la consulta esperada es la siguiente:

```
?- invitados( maria, leo, L ).
L = [ isa, carlos, maria, caro, leo, gaston, charly, santi, juan ]
```

Otra solución más simple al mismo problema se consigue utilizando otros predicados de orden superior, sobre la misma base de conocimiento.

```
invitados( Novio, Novia, Lista ):-
                setof( P, ( invitado( Novio, P) ; invitado( Novia, P ) ), Lista ).
```

La lista de invitados (**Lista**) está formada por todas las personas (**P**) que hayan sido invitadas (mediante el predicado **invitado/2**) por el novio (**Novio**) o (**;**) la novia (**Novia**). El predicado **setof/3** garantiza que no haya repetidos. El predicado **invitado/2** es el mismo de la solución anterior, que contempla a los amigos, pero agregándole una regla más para también tenga en cuenta a cada familiar.

```
invitado( Novio, F ):-
                familia( _, Lista ),
                member( Novio, Lista ),
                member( F, Lista ).
```

Capítulo 10

Ejecución dinámica y control

- **Predicados dinámicos**
  - Declaración de predicados dinámicos
  - Inserción de cláusulas
  - Eliminación de cláusulas
- **Control de ejecución**
  - Cortes (cut)
  - Fallos (fail)

Predicados dinámicos

Una de las características más útiles es la posibilidad de **añadir y eliminar cláusulas** de los predicados presentes en el programa, y hacerlo en **tiempo de ejecución**. Los predicados con esta posibilidad se denominan **predicados dinámicos**. Uno de los usos apropiados es para implementar la noción de estado, de un estado que va cambiando a lo largo del problema.

Una característica que ayuda a realizar programas **coherentes**, es que teniendo en cuenta los puntos de elección en el programa en los que se basa el backtracking para su funcionamiento, no se permite que un programa se modifique a sí mismo hasta que no termine de ejecutarse la parte afectada.

En general, es sencillo usar predicados dinámicos para almacenar **hechos**. Al manipular dinámicamente predicados con **reglas**, a la vez que aumenta la **capacidad de aprendizaje y evolución de un sistema**, es más delicado tu tratamiento.

Haciendo un mal uso de los predicados dinámicos se puede provocar ilegibilidad del código, realizar inconsistencias y dificultar la resolución de tareas.

Declaración de predicados dinámicos

Estos predicados deben ser previamente marcados mediante la directiva **dynamic/1**, indicando el nombre del predicado dinámico. Un predicado dinámico es como otro cualquiera excepto en que puede no tener cláusulas. En ese caso, una llamada al predicado simplemente falla, pero no provoca un error de predicado indefinido.

Ejemplo:

Se declara el predicado **prueba/1** como dinámico.

```
:- dynamic prueba/1.
prueba( abc ).
prueba( 123 ).
```

Inserción de cláusulas

Para insertar cláusulas de un predicado dinámico existe la familia de predicados **assert**, consistente en los siguientes predicados:

asserta/1	Inserta una nueva cláusula al principio del programa.
assertz/1	Inserta una nueva cláusula al final del programa.
assert/1	Idéntico a **assertz/1**.

La diferencia entre insertar las cláusulas por el principio o por el final es importante puesto que determina el orden de sucesión de las soluciones.

El argumento que toman estos predicados es la nueva cláusula a insertar, pero teniendo en cuenta:

- Las **variables ligadas** dentro de la cláusula se sustituyen por su valor en el momento de insertar dicha cláusula. Esto es lo lógico y deseable.
- Las **variables libres** dentro de la cláusula se sustituyen por variables nuevas en el momento de la inserción. Es decir, posteriores unificaciones no afectan a la cláusula ya insertada.
- Si existen **puntos de elección** para el predicado modificado (aquel para el que se inserta una nueva cláusula), estos se mantienen y no se generan nuevos puntos de elección. Es decir, la nueva cláusula no se tendrá en cuenta hasta que se ejecute un nuevo objetivo para el predicado en cuestión.

Ejemplo:

Teniendo en cuenta el ejemplo anterior, si se añade una cláusula ejecutando:

```
?- assert( prueba( 666 ) )
```

La base de conocimiento queda:

```
prueba( 666 ).
```

prueba(abc).

prueba(123).

Si se añade otra cláusula mediante:

?- J = 999, assert(prueba(J)).

Resulta:

prueba(666).

prueba(abc).

prueba(123).

prueba(999).

Por último, se genera una cláusula más compleja:

assert((prueba(X) :- X > 1024))

El programa queda:

prueba(666).

prueba(abc).

prueba(123).

prueba(999).

prueba(H) :- H > 1024.

Eliminación de cláusulas

Es posible eliminar cláusulas de predicados dinámicos mediante la familia de predicados **retract** consistente en:

retract/1	Elimina únicamente la primera cláusula que unifique con el argumento. Siempre se elimina por el principio del programa.
rectractall/1	Elimina todas las cláusulas que unifiquen con el argumento.

Hay que considerar que:

- Los **puntos de elección** que ya existieren a causa de las cláusulas eliminadas permanecen mientras sea necesario.
- **retract/1** es constructivo. Las variables libres se ligan a los elementos de la cláusula eliminada.
- **retract/1** tiene tantas soluciones como cláusulas existan que unifiquen con el argumento. Si se hace backtracking sobre él, se eliminan todas las cláusulas que unifiquen. Falla cuando no hay más cláusulas a unificar.
- **retractall/1** solamente tiene una solución, y no es constructivo. No liga las variables libres.

Control de ejecución

En un principio, asumiendo los criterios del **paradigma declarativo** en su forma más pura, el orden en que se realiza la resolución no debería afectar el conjunto de respuestas a una consulta. Sin embargo, en la programación lógica práctica el orden tiene importancia.

El comportamiento real de un programa lógico depende del orden en el cuál se realiza la resolución. Para definir correctamente el control sobre los cálculos, se asume un orden de resolución predefinido. Las proposiciones en el lado derecho de la cláusula son evaluadas en orden de izquierda a derecha. Si hay varias cláusulas definiendo la misma relación, entonces son probadas de la primera a la última.

Como consecuencia, **todo programa es determinístico**. Si una consulta tiene múltiples respuestas, se puede predecir el orden en que las respuestas serán encontradas.

Para definir el orden de las cláusulas y las proposiciones dentro de la cláusula, la principal consideración es la posibilidad de la **indeterminación**, de variables que no se unifiquen y el riesgo de ciclos infinitos. Mediante un cuidadoso orden de los hechos y reglas, se puede aumentar la eficiencia de la selección, así como su terminación.

Existen herramientas concretas que permiten interrumpir o alterar de alguna manera el **mecanismo de backtracking**. Entre ellas, los **cortes** y **fallos**.

Cortes (cut)

Si se sabe de antemano que una consulta tendrá sólo una respuesta, no tendrá sentido permitir que el mecanismo de backtracking siga buscando otra respuesta cuando la primera ya ha sido encontrada. Aún sabiendo que la consulta pueda tener muchas respuestas alternativas, si para una aplicación determinada sólo interesa la primera de ellas, tampoco es necesario encontrar todas las soluciones.

Existe un predicado, llamado **cut** o comúnmente "corte", que se representa mediante el signo de exclamación (**!**).

El corte tiene la propiedad de eliminar los puntos de elección del predicado que lo contiene. Es decir, cuando se ejecuta el corte, el resultado de la consulta (no sólo la cláusula en cuestión) queda comprometido al éxito o fallo de los objetivos que aparecen a continuación. Es como si "se le olvidase" que dicha consulta puede tener varias soluciones.

Su funcionalidad consiste en que **altera el backtracking,** impidiendo volver atrás a continuar buscando nuevas soluciones de consultas anteriores (borra todas las "marcas" dejadas hasta el momento). No detiene completamente el mecanismo de evaluación, sino que este continúa con las demás proposiciones

de la regla que se esta evaluando. Provoca que las variables ya unificadas en la cláusula no podrán volver a unificarse con otros valores mediante la búsqueda de otras soluciones alternativas, ni se podrán buscar otras cláusulas para el predicado que se esta evaluando.

Ejemplo:

Si se está probando un predicado **A0** cualquiera de la forma:

```
A0 :- A1, !, A2.

A0 :- A3.
```

Si **A1** falla, entonces se vuelve atrás y se intenta encontrar otra cláusula para **A0**, como de costumbre. Pero si **A1** tiene éxito, se acepta la primera respuesta dada por **A1**, pasa el corte, y sigue comprobando **A2**. Pasar el corte tiene el efecto de que más allá que **A2** falle o no, no se hace luego ningún intento por encontrar otras soluciones alternativas para **A1**, ni buscar las siguiente cláusulas para **A0**.

El corte tiene un valor **verdadero**, por lo que no hace fallar la regla que se está evaluando. Puede estar ubicado en cualquier posición entre las proposiciones de una regla, o inclusive como única proposición. Sin ser lo más frecuente, en la misma regla podría utilizarse más de una vez.

El corte se utiliza para los siguientes casos:

- Para **optimizar la ejecución**. El corte sirve para evitar que por culpa del backtracking se exploren puntos de elección que, con toda seguridad, no llevan a otra solución (fallan). En otras palabras, es podar el árbol de búsqueda de posibles soluciones.
- Para **facilitar la legibilidad y comprensión del código** que está siendo programado. A veces se sitúan cortes en puntos donde, con toda seguridad, no van a existir puntos de elección para eliminar, pero ayuda a entender que la ejecución sólo depende de la cláusula en cuestión.
- Para conseguir que un predicado solamente tenga **una solución**. Esto nos puede interesar en algún momento. Una vez que el programa encuentra una solución ejecutamos un corte. Así evitamos que Prolog busque otras soluciones aunque sabemos que éstas existen.

Por otra parte comparte las **desventajas** de los operadores de **bajo nivel**, de tener que conocer al detalle la forma de implementación interna del mecanismo de evaluación, en particular del backtracking, **perdiéndose la característica declarativa** de que desde el punto de vista del programa, se despreocupa de los detalles internos del control de la ejecución.

Fallos (fail)

Existe un predicado interno, **fail/0**, que siempre falla, ya que su valor de verdad es **falso**.

Se utiliza como mecanismo para introducir casos falsos en una base de conocimiento donde se asume que todas las proposiciones son verdaderas, que permiten establecer por ejemplo algún tipo de excepción a alguna regla o demostrar por el "absurdo".

Es muy habitual encontrar la secuencia de objetivos corte-fallo: **!, fail**. El predicado **fail/0** se utiliza para detectar prematuramente combinaciones de los argumentos que no llevan a solución, evitando la ejecución de un montón de código que al final va a fallar de todas formas.

Anexo

PROLOG

- **Sintaxis básica**
  - Comentarios
  - Valores comunes
  - Comparaciones
  - Operadores
- **Cálculos aritméticos**
- **Comprobaciones de tipos**
- **Entrada y Salida estándar**
  - Entrada
  - Salida

Sintaxis básica

La sintaxis utiliza expresiones que incluyen básicamente valores atómicos, estructuras de datos, variables, y predicados.

Los nombres de los **predicados**, como también las etiquetas de los functores, se representan con una secuencia de caracteres que empiezan con una letra **minúscula.**

Los nombres de las **variables** empiezan siempre con **mayúsculas**.

Comentarios

Los comentarios se escriben comenzando la línea con un símbolo de porcentaje.

Ejemplo:

```
% Hola, esto es un comentario.
```

```
% Y esto también.
```

Para comentarios de varias líneas, se usa una barra y un asterisco para comenzar y al revés para terminar.

Ejemplo:

```
/* todo esto es un comentario
que es extenso
y continua por varias lineas */
```

Valores comunes

PROLOG es un lenguaje **débilmente tipado**. Los números, los átomos y las estructuras pueden ser usados indistintamente siempre. Por lo tanto, no es necesario declarar el tipo de dato de cada expresión, ya sea un argumento, un tipo de dato compuesto como listas o factores u otro término. Así, entre otras cosas, es posible definir diferentes reglas del mismo predicado con expresiones que se unifiquen con diferentes tipos de datos y se pueden construir listas heterogéneas conteniendo diferentes tipos de valores.

Los **símbolos** son secuencias de caracteres, sin espacios, que se utilizan directamente para reflejar alguna información. No van encerrados entre comillas, no pueden contener signos especiales, ni pueden comenzar con mayúscula.

Los **valores numéricos** se representan en forma literal por sus correspondientes representaciones mediante los dígitos, y en el caso de valores con decimales, mediante el punto decimal.

Las **cadenas de caracteres** se pueden representar con caracteres encerrados entre comillas dobles, pero cuando son devueltos, se muestran mediante su representación interna que consiste en una lista con los valores ASCII de los caracteres.

Ejemplo:

```
?- X = hola.
X = hola

?- X = 145.67.
X = 145.67

?- X = “hola”.
X = [104, 111, 108, 97]
```

Comparaciones

La **igualdad** entre valores o variables se representa mediante el operador igual (=), mientras que la **diferencia**, es el operador (\=).

Ejemplo:

```
2 = 2
X = Y
3 \= 4
X \= Y
```

Las **desigualdades** son representadas mediante los siguientes operadores:

>	mayor
<	menor
>=	mayor o igual
=<	menor o igual

Ante expresiones aritméticas, el "=" unifica las expresiones pero sin evaluarlas, es decir, no se resuelven las operaciones aritméticas de la derecha, sino que se unifica la expresión tal cual como fue expresada.

Ejemplo:

```
?- 5 = 4 + 1.
no

?- Y = 4 + 1.
Y = 4 + 1

?- 4 + 1 is 4 + 1
yes

?- X = 4, Y = X +1.
X = 4
Y = 4 +1

?- Y = (4 * 5 + 6) / 2.
Y = (4 * 5 + 6) / 2
```

Operadores

Algunos identificadores pueden estar declarados como **operadores**, bien de manera predefinida, o bien por el programador. Los operadores sirven para

escribir términos unarios o binarios de una manera más cómoda. Los operadores binarios tienen en general notación infija, que permiten escribir el identificador entre los dos argumentos y eliminar los paréntesis

Ejemplo:

Un identificadores definido como operador infijo es la suma (**+**).

a + b

Es una expresión perfectamente válida, aunque en realidad no es más que el término:

+(a, b)

Cálculos aritméticos

Para realizar **cálculos aritméticos**, existe un predicado de unificación para los valores numéricos, **is/2**, que permite que las variables asuman el resultado de cálculos aritméticos.

El **is/2** es similar al **=/2** en cuanto que compara las expresiones a izquierda y derecha, pero además, a la expresión de la derecha la reduce hasta lograr un valor aritmético.

A la izquierda, como primer argumento, debe haber una variable o una constante, y a la derecha, como segundo argumento, puede utilizarse una variable, una constante o cualquier término válido, que además, para que sea posible la unificar, todas sus variables deben estar ligadas.

Su utilidad principal es cuando a la izquierda se utiliza una variable sin unificar, por lo que como resultado de la evaluación, **la variable asume el valor aritmético de la expresión ubicada a la derecha**, que como en las consultas variables, hace verdadera la expresión en su conjunto.

Si la expresión de la izquierda está ligada, se trata de una consulta de validación, devolviendo si es verdadero o falso que el valor de la expresión aritmética es el indicado.

En el predicado **is/2** las expresiones que se pueden utilizar en el segundo argumento pueden ser:

+/2	Suma	**X is A + B.**
***/2**	Producto	**X is 2 * 7.**
-/2	Resta	**X is 5 - 2.**
'/'/2	División	**X is 7 / 5.**
-/1	Cambio de signo	**X is -Z.**
'//'/2	División entera	**X is 7 // 2.**
mod/2	Resto de la división entera	**X is 7 mod 2.**
'^'/2	Potencia	**X is 2^3.**
abs/1	Valor absoluto	**X is abs(-3).**

pi/0	La constante PI	**X is 2*pi.**
sin/1	seno en radianes	**X is sin(0).**
cos/1	coseno en radianes	**X is cos(pi).**
tan/1	tangente en radianes	**X is tan(pi/2).**
asin/1	arcoseno en radianes	**X is asin(7.2).**
acos/1	arcocoseno en radianes	**X is acos(Z).**
atan/1	arcotangente en radianes	**X is atan(0).**
floor/1	redondeo por defecto	**X is floor(3.2).**
ceil/1	redondeo por exceso	**X is ceil(3.2).**

Ejemplo:

```
?- 5 is 4 + 1
yes

?- Y is 4 + 1.
Y = 5

?- 4 + 1 is 4 + 1
no

?- X = 4, Y is X +1.
X = 4
Y = 5

?- Y is (4 * 5 + 6) /2.
Y = 13
```

Comprobaciones de tipos

En **Prolog** no existen declaraciones de tipo. En el caso en que se desee **asegurarse de que un argumento es de un tipo determinado**, se puede realizar una **comprobación de tipo**. Son predicados que generalmente reciben un dato como argumento y fallan si el argumento no es del tipo esperado.

Existen predicados predefinidos para comprobar algunos tipos básicos. Estos son:

integer/1	Comprueba si su argumento es un número entero
float/1	Comprueba si el argumento es un número decimal
number/1	Comprueba si el argumento es un número (entero o decimal)

atom/1	Comprueba si el argumento es un término cero-ario excluyendo las constantes numéricas
var/1	Comprueba si el argumento es una variable libre
nonvar/1	Comprueba si el argumento está instanciado
ground/1	Comprueba si el argumento es un término que no contiene variables libres (está cerrado)

La desventaja de las comprobaciones de tipo es que resulta necesario ejecutarlos. Esto añade un tiempo extra de ejecución a las aplicaciones. Gracias a la **unificación**, la mayoría de los predicados no requieren comprobación de tipo.

El **polimorfismo** que aporta la ausencia de declaraciones de tipo es deseable en muchas ocasiones. En contrapartida, una adecuada implementación puntual de comprobación de tipos en algunos predicados permite que otros predicados, sin implementar las comprobaciones, sean polimórficos[29].

Las comprobaciones acerca de las variables instanciados o libres es de mucha utilidad para evitar indeterminaciones y ampliar la propiedad de inversibilidad de las reglas.

Entrada y Salida estándar

La posibilidad de manejar entrada/salida estándar, habitualmente el teclado y la pantalla, a través de **streams**, que son buffers para escribir y/o leer de los dispositivos.

Hay tres tipos de streams:

- Streams de entrada (lectura).
- Streams de salida (escritura).
- Streams de entrada y salida (híbridos).

Entrada

El predicado principal para leer de la entrada estándar es **read/1**, que es capaz únicamente de leer términos que estarán separados unos de otros por un punto (.) y un salto de línea. Cuando ya no hay más términos para leer en la entrada estándar, el predicado retorna el término **end_of_file/0**.

Para leer caracteres uno a uno, existe el predicado **get/1**, que retorna el código ASCII del carácter leído. Cuando ya no hay más caracteres retorna -1.

[29] Ver capitulo 9 "Polimorfismo y orden superior"

Cuando la entrada estándar es el teclado, se muestra un "prompter" representando por el símbolo "¦" para indicar que está preparado para que se teclee.

Cuando **read/1** intenta leer un término pero se encuentra con un error sintáctico (el término está mal construido), emite un mensaje de error y el predicado falla.

Nada impide que **read/1** lea un término que contenga **variables libres**. Hay que tener en cuenta que dichas variables son siempre distintas a las variables del programa en ejecución y de las variables leídas en llamadas anteriores, independientemente del nombre que se les ponga a dichas variables.

Salida

Para escribir en la salida estándar se utiliza el predicado **write/1** que recibe un término como argumento. Análogamente a la entrada estándar, el predicado **put/1** escribe caracteres simples, recibiendo como argumento el código ASCII correspondiente.

Adicionalmente existe el predicados **display/1** cuyo efecto es el mismo que **write/1**, y el predicado **nl/0** que escribe un salto de línea en la salida estándar.

Ejemplo:

Este es un programa que lee dos números de la entrada estándar (separados por punto), los suma y escribe el resultado por la salida estándar.

```
suma( Total ) :-
        leerNumeros( Num1, Num2 ),
        Total is Num1 + Num2,
        mostrarTotal( Total ).

leerNumeros( Num1, Num2 ) :-
        read( Num1 ),
        number( Num1 ),
        read( Num2 ),
        number( Num2 ).

mostrarTotal( Total ) :-
        nl,
        display( 'LA SUMA ES : ' ),
        display( Total ),
        nl.
```

Bibliografía

- **ALONSO AMO, F y SEGOVIA PEREZ, F**. *Entornos y Metodologías de Programación*. Paraninfo.
- **WATT, David**. *Programming Languajes Concepts and Paradigms*. Prentice Hall.
- **GHEZZI, Carlo y JAZAYERI, Mehdi**. *Conceptos de Lenguajes de Programación*. Díaz de los Santos.
- **RAVI y SETHI.** *Lenguajes de programación - Conceptos y constructores.* Addison Wesley.
- **LABRA, José E**. *Programación Práctica en Prolog*, Universidad de Oviedo, 1998.
- **WIELEMAKER, Jan.** *SWI-Prolog Reference Manual,* Universidad de Amsterdam, 2007.
- **PINEDA, Ángel Fernández.** *Tutorial del lenguaje Prolog*, Universidad Politécnica de Madrid.

Fuentes en Internet

- **Sitio oficial de Swi-Prolog:**

  www.swi-prolog.org
- **Swi-Prolog User (TWiki) web:**

  gollem.science.uva.nl/twiki/pl/bin/view/Main/WebHome
- **Tutoriales de Prolog**

  www.programacion.com/tutoriales/prolog/

Índice