Fundamentos de la Programación Lógica

José de Jesús Lavalle Martínez
FCC, BUAP
Traducción de partes del libro
Logic Programming with Prolog
Max Bramer 2nd Edition (2013)
jlavalle@cs.buap.mx

Índice general

Introducción **5**

1. Iniciemos **13**
1.1. Iniciando Prolog 13
1.2. Programas Prolog 15
1.3. Objetos de datos en Prolog: términos de Prolog 21
Ejercicios prácticos 1 25

2. Cláusulas y Predicados **27**
2.1. Cláusulas 27
2.2. Predicados 29
2.3. Cargando cláusulas 33
2.4. Variables 37
Ejercicios prácticos 2 43

3. Satisfacción de Metas **45**
3.1. Introducción 45
3.2. Unificación 47
3.2.1. Unificando Términos de Llamada 48
3.3. Evaluación de Metas 53
3.4. Retroceso 58
3.5. Satisfacción de Reglas: Un Resumen 70
3.6. Eliminación de variables comunes 73
3.7. Una Nota sobre la Programación Declarativa 74
3.8. Nota Importante sobre el Retroceso Controlado por el Usuario 76
Ejercicios prácticos 3 77

4. Operadores y Aritmética **79**
4.1. Operadores 79
4.2. Aritmética 83
4.3. Operadores de Igualdad 87
4.4. Operadores Lógicos 91
4.5. Más sobre Precedencia de Operadores 92
Ejercicios prácticos 4 95

5. Entrada y Salida **97**
5.1. Introducción 97
5.2. Escritura de Términos 97
5.3. Lectura de Términos 99
5.4. Entrada y Salida Usando Caracteres 100
5.5. Escritura de Caracteres 101
5.6. Lectura de Caracteres 102
5.7. Uso de Caracteres: Ejemplos 103

Introducción

La Programación Lógica es el nombre que se le da a un estilo distintivo de programación, muy diferente al de los lenguajes de programación convencionales como C++ y Java. ¡Los fanáticos de la programación lógica dirían que 'diferente' significa más claro, más simple y, en general, mejor!

Aunque existen otros lenguajes de programación lógica, el más utilizado es **Prolog**. El nombre significa Programación en Lógica. Este libro enseña las técnicas de Programación Lógica a través del lenguaje Prolog. Prolog se basa en investigaciones realizadas por científicos de la computación en Europa en las décadas de 1960 y 1970, en particular en las universidades de Marsella, Londres y Edimburgo.

La primera implementación fue en la Universidad de Marsella a principios de la década de 1970. El desarrollo posterior en la Universidad de Edimburgo condujo a una versión estándar de facto, ahora conocida como Prolog de Edinburgh. Prolog ha sido ampliamente utilizado para desarrollar aplicaciones complejas, especialmente en el campo de la Inteligencia Artificial. Aunque es un lenguaje de propósito general, sus principales puntos fuertes son para el cálculo simbólico más que para el numérico.

Los desarrolladores del lenguaje eran investigadores que trabajaban en la automatización de la demostración de teoremas matemáticos. Este campo se conoce a menudo como lógica computacional. Pero si no eres un científico de la computación, un lógico o un matemático, ¡no dejes que esto te desanime! Este libro está dirigido al 99,9 % de la población que no es ninguno de estos. Quienes lo son, ya tienen una serie de excelentes libros de texto entre los que elegir.

La idea de que los métodos desarrollados por los lógicos computacionales podrían usarse como base para un poderoso lenguaje de programación de propósito general fue revolucionaria hace 30 años. Desafortunadamente, la mayoría de los otros lenguajes de programación aún no se han puesto al día.

La característica más llamativa de Prolog para el recién llegado es que los programas se ven mucho más simples que en otros lenguajes. Muchos diseñadores de lenguajes comenzaron con buenas intenciones pero no pudieron resistir, o tal vez no pudieron evitar, hacer sus creaciones demasiado elaboradas.

En algunos lenguajes, incluso escribir el programa de prueba habitual para imprimir las palabras ¡Hola mundo! a la pantalla del usuario es un trabajo duro. Todo lo que el usuario tiene que hacer en Prolog es escribir **write('¡Hola mundo!')**.

Los lenguajes de programación tradicionales tienen una característica en común. Todos ellos contienen una serie de instrucciones que deben ejecutarse una tras otra. Este estilo de programación se llama *procedimental*. Se corresponde estrechamente con la forma en que generalmente se construyen las computadoras. Este es un enfoque tentador que se ha utilizado desde la década de 1950 pero que, en última instancia, es defectuoso.

La forma en que los usuarios escriben programas debe depender lo menos posible de cómo se construyó la máquina subyacente y lo más posible de lo que el usuario está tratando de hacer. De la misma manera, las instalaciones que utilizo cuando conduzco mi automóvil deben depender lo menos posible de cómo se diseñó el motor o funciona el carburador. Quiero que se me oculte todo ese nivel de detalle, aunque en la práctica entiendo que puede no ser completamente posible ignorarlo.

Los programas Prolog a menudo se describen como *declarativos*, aunque inevitablemente también tienen un elemento procedimental. Los programas se basan en las técnicas desarrolladas por los lógicos para formar conclusiones válidas a partir de la evidencia disponible. Solo hay dos componentes en cualquier programa: hechos y reglas.

El sistema Prolog lee el programa y simplemente lo almacena. Luego, el usuario ingresa una serie de preguntas, conocidas como *consultas*, que el sistema responde utilizando los hechos y las reglas disponibles. Este es un ejemplo sencillo, una serie de consultas y respuestas sobre animales. El programa consta de solo siete líneas (las líneas en blanco se ignoran).

```
dog(fido).
dog(rover).
dog(henry).
cat(felix).
cat(michael).
cat(jane).
animal(X):-dog(X).
```

Este programa no es demasiado difícil de descifrar. Las primeras tres líneas son *hechos*, con la interpretación obvia de que fido, rover y henry son todos perros. Los siguientes tres hechos dicen que felix, michael y jane son todos gatos.

La línea final es una *regla* que dice que cualquier cosa (llamémosla X) es un animal si es un perro. Los amantes de los gatos pueden sentir que los gatos también pueden afirmar que se les llama animales, pero el programa no dice nada al respecto.

Una vez cargado el programa, el usuario se encuentra con el símbolo de dos caracteres **?-**, que se denomina *indicador del sistema*. Para verificar si fido es un perro, todo lo que se necesita es escribir la consulta **dog(fido)** seguida de un punto y presionar la tecla 'return', que indica al sistema que se necesita una respuesta. Esto da el diálogo completo:

?-dog(fido).
true.

El usuario puede ingresar una serie de consultas en el indicador, solicitando más información.

?-dog(jane).	[**¿jane es un perro?**]
false.	[**No - es un gato**]
?-animal(fido).	[**¿fido es un animal?**]
true.	[**sí - porque es un perro y cualquier perro es un animal**]
?-dog(X).	[**¿Es posible encontrar algo, llamémosle X, que sea un perro?**]
X = fido;	[**Proporciona las tres respuestas posibles**]
X = rover;	
X = henry	

?-animal(felix). **[felix es un gato y por lo tanto no califica como un animal, en lo que respecta al programa]**

false.

Aunque sencillo, este ejemplo muestra los dos componentes de cualquier programa Prolog, *reglas* y *hechos*, y también el uso de *consultas* que hacen que Prolog busque en sus hechos y reglas para encontrar la respuesta. Determinar que fido es un animal implica una forma muy simple de razonamiento lógico:

DADO QUE
cualquier X es un animal si X es un perro

AND
fido es un perro

SE DEDUCE QUE
fido debe ser un animal

Este tipo de razonamiento es fundamental para la demostración de teoremas en Matemáticas y para la escritura de programas en Prolog.

Incluso consultas muy simples como:

?-dog(fido).

pueden verse como pedirle al sistema Prolog que demuestre algo, en este caso que fido es un perro. En los casos más simples, puede hacerlo simplemente encontrando un hecho como **dog(fido)** que se le ha dado. El sistema responde '**true**' para indicar que este simple 'teorema' ha sido probado.

Ahora ha visto los tres elementos necesarios para la programación lógica en Prolog: hechos, reglas y consultas. No hay otros. Todo lo demás se construye a partir de ellos.

Una palabra de advertencia es necesaria en esta etapa. Es fácil para el recién llegado comenzar con Prolog, pero no se deje engañar por estos ejemplos y piense que Prolog solo es capaz de manejar problemas simples (¿Mickey Mouse?).

Al juntar estos bloques de construcción muy básicos, Prolog proporciona una herramienta muy poderosa para crear programas que resuelvan problemas complejos, especialmente los que involucran razonamiento, pero todos

los programas de Prolog tienen una forma simple y están sólidamente basados en la idea matemática de probar resultados a partir de los hechos y reglas disponibles.

Prolog se ha utilizado para una amplia variedad de aplicaciones. Muchas de éstas son de Matemáticas y Lógica, pero muchas no. Algunos ejemplos del segundo tipo de aplicaciones son

- programas para procesar un texto en 'lenguaje natural', para responder preguntas sobre su significado, traducirlo a otro idioma, etc.
- sistemas de asesoramiento para aplicaciones legales
- solicitudes de formación
- mantenimiento de bases de datos para el proyecto Genoma Humano
- un sistema de personal para una empresa informática multinacional
- generación automática de historias
- analizar y medir las 'redes sociales'
- una plataforma de ingeniería de software para apoyar el desarrollo de sistemas de software complejos
- generar automáticamente licencias legalmente correctas y otros documentos en varios idiomas
- un sistema electrónico de apoyo a los médicos.

Prolog también se está utilizando como base para un "lenguaje de representación del conocimiento" estándar para la Web Semántica, la próxima generación de tecnología de Internet.

Prolog es uno de los principales lenguajes utilizados por los investigadores en Inteligencia Artificial, con muchas aplicaciones desarrolladas en ese campo, especialmente en forma de Sistemas Expertos, programas que 'razonan' la solución a problemas complejos utilizando reglas.

Muchos libros de texto sobre Prolog asumen que usted es un programador experimentado con una sólida formación en Matemáticas, Lógica o Inteligencia Artificial (preferiblemente las tres). Este libro no hace tales suposiciones. Comienza desde cero y tiene como objetivo llevarlo a un punto en el que pueda escribir programas bastante potentes en el lenguaje, a menudo con una

cantidad de líneas de 'código' de programa considerablemente menor que la que se necesitaría en otros lenguajes.

No es necesario ser un programador experimentado para aprender Prolog. Cierta familiaridad inicial con los conceptos básicos de computación, como programa, variable, constante y función, lo harían más fácil de lograr, pero, paradójicamente, demasiada experiencia en la escritura de programas en otros lenguajes puede dificultar la tarea; puede ser necesario desaprender los malos hábitos de pensamiento aprendidos en otro lugar.

Algunos detalles técnicos

Los programadores experimentados buscarán en vano en este libro características del lenguaje estándar como declaraciones de variables, subrutinas, métodos, bucles for, bucles while o sentencias de asignación. (Si no sabe lo que significan estos términos, no se preocupe, no los necesitará).

Por otro lado, a los lectores experimentados les puede gustar saber que Prolog tiene una sintaxis sencilla y uniforme, programas que son equivalentes a una base de datos de hechos y reglas, un probador de teoremas incorporado con retroceso automático, procesamiento de listas, recursividad y facilidades para modificar programas (o bases de datos) en tiempo de ejecución. (Probablemente tampoco sepa lo que significan la mayoría de estos, pero los usará todos al final de este libro).

Prolog se presta a un estilo de programación que hace un uso particular de dos poderosas técnicas: recursividad y procesamiento de listas. En muchos casos, los algoritmos que requerirían cantidades sustanciales de codificación en otros lenguajes se pueden implementar en unas pocas líneas en Prolog.

Hay muchas versiones de Prolog disponibles para PC, Macintosh y sistemas Unix, incluidas versiones para Microsoft Windows, para vincular Prolog a una base de datos relacional de Oracle y para usar con el diseño de programas 'orientados a objetos'. Estos van desde sistemas comerciales con muchas funciones hasta versiones de dominio público y 'freeware'.

Todos los programas de este libro están escritos utilizando la 'sintaxis de Edimburgo' estándar y deberían ejecutarse sin cambios en prácticamente cualquier versión de Prolog que encuentre (desafortunadamente, encontrar diferencias sutiles ocasionales entre las implementaciones es uno de los riesgos laborales de aprender cualquier lenguaje de programación).

No se han incluido deliberadamente características tales como interfaces gráficas, enlaces a bases de datos externas, etc., ya que generalmente varían de

una implementación a otra. Todos los ejemplos proporcionados se probaron con la versión 6.2.6 de **SWI-Prolog**, una versión popular de dominio público del lenguaje que está disponible en una variedad de plataformas.

Esta segunda edición se ha ampliado con la adición de dos capítulos más que ilustran el uso de reglas gramaticales para analizar oraciones en inglés y el uso de Prolog para aplicaciones de inteligencia artificial.

Cada capítulo tiene ejercicios de autoevaluación que le permiten verificar su progreso.

Capítulo 1

Iniciemos

Objetivos del Capítulo

Después de leer este capítulo, debería ser capaz de:

- Escribir y cargar un programa Prolog simple
- Introducir metas en el indicador del sistema Prolog
- Comprender la terminología básica del lenguaje Prolog
- Distinguir entre diferentes tipos de términos (objetos de datos).

1.1. Iniciando Prolog

Iniciar el sistema Prolog suele ser sencillo, pero los detalles precisos variarán de una versión a otra. Consulte la documentación si es necesario. Iniciar Prolog generalmente producirá una cantidad de líneas de encabezados seguidos de una línea que contiene solo

```
?-
```

Este es el indicador del sistema (en algunas versiones de Prolog, se puede usar una combinación de caracteres ligeramente diferente).

El indicador indica que el sistema Prolog está listo para que el usuario ingrese una secuencia de uno o más objetivos, que debe terminar con un punto, por ejemplo:

?- write('Hello World'),nl,write('Welcome to Prolog'),nl.

nl significa 'comenzar una nueva línea', como se explicará más adelante. Como todas las demás entradas del usuario, la línea anterior no tiene ningún efecto hasta que se presiona la tecla 'return'.

Hacer eso produce la salida

Hello World
Welcome to Prolog
true.
seguido de otro indicador del sistema **?-**.

En este libro, una secuencia de metas introducidas por el usuario generalmente se mostrará precedida por el indicador **?-**. El indicador no debe ser escrito por el usuario. Es generado automáticamente por el sistema Prolog para mostrar que está listo para recibir una secuencia de metas.

En el ejemplo anterior, el usuario ingresó una secuencia de cuatro metas: escribir ('Hello World'), nl (dos veces) y escribir ('Welcome to Prolog'). Las comas que separan las metas significan 'y'.

Para que la secuencia de metas

write('Hello World'),nl,write('Welcome to Prolog'),nl.
tenga éxito, cada una de las siguientes metas tienen que tener éxito en orden.

write('Hello World')
Hello World tiene que mostrarse en la pantalla del usuario

nl
se debe enviar una nueva línea a la pantalla del usuario

write('Welcome to Prolog')
Welcome to Prolog tiene que mostrarse en la pantalla del usuario

nl
se debe enviar una nueva línea a la pantalla del usuario.

El sistema Prolog puede lograr todos estas metas simplemente enviando líneas de texto a la pantalla del usuario. Lo hace y luego escribe **true** para indicar que la secuencia de metas ha tenido éxito.

Desde el punto de vista del sistema, lo importante es si la secuencia de metas introducidas por el usuario tiene éxito o no. La generación de salida a la pantalla se considera mucho menos importante y se describe (simplemente)

como un efecto secundario de evaluar las metas de **write('Hello World')**, etc.

Los significados de **write** y **nl** están predefinidos por el sistema Prolog. Se conocen como predicados interconstruidos, a veces abreviados como BIP.

Otros dos predicados interconstruidos que se proporcionan como estándar en casi todas las versiones de Prolog son **halt** y **statistics**.

?-halt.
hace que el sistema Prolog termine.

?-statistics.
hace que se generen estadísticas del sistema (de valor principalmente para los usuarios más experimentados) como las siguientes.

0.250 seconds cpu time for 61,957 inferences
4,179 atoms, 2,858 functors, 1,936 predicates, 36 modules, 62,926 VM-codes
1 garbage collections gained 14,448 bytes in 0.000 seconds.
Stack shifts: 2 local, 1 global, 1 trail in -0.000 seconds.
true.

Tenga en cuenta que esta salida termina con la palabra **true**, lo que significa que la meta ha tenido éxito, como **statistics**, **halt** y muchos otros predicados interconstruidos siempre tienen. Su valor radica en los efectos secundarios (generación de estadísticas, etc.) que se producen cuando se evalúan.

Una secuencia de una o más metas ingresadas por el usuario en el indicador a menudo se denomina una *consulta*. En general, utilizaremos el término "secuencia de metas" en este libro.

1.2. Programas Prolog

Ingresar una meta o una secuencia de metas en el indicador del sistema usando solo predicados interconstruidos sería de poco valor en sí mismo. La forma normal de trabajar es que el usuario cargue un programa escrito en el lenguaje Prolog y luego ingrese una secuencia de una o más metas en el indicador, o posiblemente varias secuencias en sucesión, para hacer uso de la información que se ha cargado en la base de datos.

La forma más sencilla (y más habitual) de crear un programa Prolog es escribirlo en un editor de texto y guardarlo como un archivo de texto, digamos *prog1.pl*.

Este es un ejemplo simple de un programa Prolog. Tiene tres componentes, conocidos como cláusulas, cada uno terminado por un punto. Tenga en cuenta que las líneas en blanco para mejorar la legibilidad se ignoran.

```
dog(fido).
cat(felix).
animal(X):-dog(X).
```

Luego, el programa se puede cargar para que lo use el sistema Prolog utilizando el predicado interconstruido **consult**.

?-consult('prog1.pl').

Siempre que exista el archivo *prog1.pl* y que el programa sea sintácticamente correcto, es decir, que contenga cláusulas válidas, la meta tendrá éxito y, como efecto secundario, producirá una o más líneas de salida para confirmar que el programa se ha leído correctamente, por ejemplo.

?-
% prog1.pl compiled 0.02 sec.
true.
?-

> Si el sistema Prolog tiene una interfaz gráfica de usuario, probablemente habrá una opción 'Load' o 'Consult' en un menú como alternativa al uso del predicado **consult**. Estas y otras opciones de menú como 'Exit' no son una parte estándar del lenguaje Prolog y no se describirán en este libro.

Cargar un programa simplemente hace que las cláusulas se coloquen en un área de almacenamiento llamada base de datos de Prolog. Introducir una secuencia de una o más metas en respuesta al indicador del sistema hace que Prolog busque y utilice las cláusulas necesarias para evaluar las metas. Una vez colocadas en la base de datos, las cláusulas generalmente permanecen allí hasta que el usuario sale del sistema Prolog y, por lo tanto, pueden usarse para evaluar metas adicionales ingresadas por el usuario.

Terminología

En el programa de arriba las tres líneas:

```
dog(fido).
cat(felix).
animal(X):-dog(X).
```

son todas cláusulas. Cada cláusula termina con un punto. Aparte de los comentarios y las líneas en blanco, los programas de Prolog consisten únicamente en una secuencia de cláusulas. Todas las cláusulas son hechos o reglas.

dog(fido) y **cat(felix)** son ejemplos de *hechos*. Se pueden interpretar de forma natural en el sentido de 'fido es un perro' y 'felix es un gato'.

a **dog** se le llama *predicado*. Tiene un argumento, la palabra **fido** encerrada entre ( ). a **fido** se le llama *átomo* (lo que significa que es una constante que no es un número). La línea final del programa.

```
animal(X):-dog(X).
```

es una regla. El carácter **:-** (dos puntos y guión) se puede leer como 'si'. A X se le llama variable. El significado de una variable utilizada en una regla o hecho se describe en el Capítulo 2. En este contexto, X representa cualquier valor, siempre que tenga el mismo valor en ambas ocasiones. La regla se puede leer de forma natural como *X es un animal si X es un perro (para cualquier X)*.

De las cláusulas anteriores es simple (para los humanos) deducir que **fido** es un animal. Prolog también puede hacer tales deducciones:

?-animal(fido).
true.

Sin embargo, no hay evidencia que sugiera que felix es un animal:

?-animal(felix).
false.

Más Terminología

Decimos que una meta tiene *éxito* o *fracasa*, o alternativamente que se *satisface* o que *no se puede satisfacer*. El término *evaluar una meta* se usa para determinar si se satisface o no. De manera equivalente, podemos decir que una meta se evalúa como verdadera (es decir, tiene éxito) o falsa (es decir,

falla). Todo esto encaja bien con la definición cotidiana de una meta como 'algo que debe lograrse'.

Tenga en cuenta que, a veces, una meta introducida por el usuario puede interpretarse como un comando, por ejemplo.

?-halt.

En otras ocasiones puede considerarse como una pregunta, por ejemplo.

?-animal(fido).
true.

Aquí hay otro programa sobre animales. Éste consta de ocho cláusulas. Todo el texto entre /* y */ se considera un comentario y se ignora.

```
/* Animals Program 1 */
dog(fido).
cat(mary). dog(rover).
dog(tom). cat(harry).
dog(henry).
cat(bill). cat(steve).
/*Aparte de los comentarios y las líneas en blanco, que son ignorados, los
programas Prolog constan de una serie de cláusulas. Una cláusula siempre
termina con un punto. Puede ocupar más de una línea, o pueden haber
varias en la misma línea, separadas por al menos un espacio. Hay dos tipos
de cláusula: hechos y reglas. dog(tom) es un ejemplo de un hecho */
```

Hay cuatro cláusulas para el predicado **dog** y cuatro para el predicado **cat**. Decimos que el programa consta de cuatro cláusulas que definen el predicado **dog** y cuatro que definen el predicado **cat**.

Suponiendo que el programa se haya guardado en un archivo de texto 'animals1.pl', el resultado generado al cargar el programa e ingresar una secuencia de metas en el indicador del sistema se muestra a continuación.

?-consult('animals1.pl'). Indicador del sistema
% animals1.pl compiled 0.00 sec. Se cargó animals1.pl usando consult
true.

?-dog(fido).
true.

?-dog(daisy).
false.

?-dog(X).
X = fido se pausa - el usuario oprime la tecla return

?-dog(Y).
Y = fido ; se pausa - el usuario oprime ;
Y = rover ; se pausa - el usuario oprime ;
Y = tom ; se pausa - el usuario oprime ;
Y = henry No se pausa - pasa a la siguiente línea

?-cat(X).
X = mary ; se pausa - el usuario oprime ;
X = harry se pausa - el usuario oprime return

?-listing(dog). Lista todas las cláusulas que definen el predicado dog

dog(fido).
dog(rover).
dog(tom).
dog(henry).
true.
?-

Hay varias características nuevas de Prolog introducidas en este ejemplo. La consulta

?-dog(X).
(una sola meta) significa 'encuentra un valor de X para el cual se satisfaga la meta **dog(X)**', o efectivamente 'encuentra un valor de X que sea el nombre de un perro'. Prolog responde

X = fido

Sin embargo, hay otras respuestas posibles (**rover**, **tom** y **henry**). Debido a esto, Prolog hace una pausa y espera a que el usuario presione la tecla 'return' antes de mostrar el indicador del sistema **?-**.

La siguiente consulta ingresada es

?-dog(Y).

Esta es esencialmente la misma consulta que antes. No es importante qué variable (X o Y) se utiliza. La consulta significa 'encontrar un valor de Y que sea el nombre de un perro'. Prolog responde

Y = fido
y vuelve a hacer una pausa. Esta vez el usuario presiona la tecla ; (punto y coma). Prolog ahora busca una solución alternativa o, más precisamente, un valor alternativo de Y que satisfaga la meta **dog(Y)**. Prolog responde

Y = rover
Se vuelve a hacer una pausa y el usuario vuelve a pulsar la tecla ;. Se da otra solución

Y = tom
Prolog se detiene de nuevo. El usuario presiona nuevamente la tecla ; produciendo una solución adicional

Y = henry
Esta vez no hay más soluciones disponibles y Prolog reconoce esto al no hacer una pausa, sino que pasa inmediatamente a generar el indicador del sistema **?-**.

El proceso de encontrar formas alternativas de satisfacer una meta introduciendo un punto y coma en el indicador del sistema se conoce como *retroceso* o, más precisamente, "forzar al sistema Prolog a retroceder". El retroceso se discutirá con más detalle en el Capítulo 3.

El ejemplo también introduce un nuevo predicado interconstruido. Ingresando la meta

?-listing(dog).
hace que Prolog liste las cuatro cláusulas que definen el predicado **dog**, en el orden en que se cargaron en la base de datos (que es el mismo orden en que aparecieron en el archivo animals1.pl).

El siguiente ejemplo muestra más sobre el uso de variables en las consultas. La secuencia de metas

?-cat(X),dog(Y).
da todas las combinaciones posibles de un perro y un gato.

?-cat(X),dog(Y).
X = mary,
Y = fido;
X = mary,
Y = rover;
X = mary,

Y = tom;
X = mary,
Y = henry;
etc.

Por el contrario, la secuencia de metas

?-cat(X),dog(X).
da todos los animales que son *a la vez* un gato y un perro (no hay tales animales en la base de datos). Aunque X significa 'cualquier valor' tanto en **cat(X)** como en **dog(X)**, ambos deben tener el *mismo* valor.

?-cat(X),dog(X).
false.

1.3. Objetos de datos en Prolog: términos de Prolog

Los objetos de datos en Prolog se denominan términos. Ejemplos de términos que se han utilizado en los programas Prolog hasta ahora vistos en este libro son **fido**, **dog(henry)**, **X** y **cat(X)**.

Hay varios tipos diferentes de términos, que se enumeran a continuación.

1. **Números**

   Todas las versiones de Prolog permiten el uso de números enteros. Se escriben como cualquier secuencia de números del 0 al 9, precedidos opcionalmente por un signo + o -, por ejemplo:

   623
   -47
   +5
   025

   La mayoría de las versiones de Prolog también permiten el uso de números con puntos decimales. Se escriben de la misma manera que los números enteros, pero contienen un solo punto decimal, en cualquier lugar excepto antes de un signo opcional + o -, por ejemplo:

6.43
-.245
+256.

2. **Átomos**

   Los átomos son constantes que no tienen valores numéricos. Hay tres formas en que se pueden escribir los átomos.

   a) Cualquier secuencia de una o más letras (mayúsculas o minúsculas), números y guiones bajos, que comienza con una letra minúscula, por ejemplo.

   john
   today_is_Tuesday
   fred_jones
   a32_BCD

   pero no

   Today
   today-is-Tuesday
   32abc

   b) Cualquier secuencia de caracteres entre comillas simples, incluidos espacios y letras mayúsculas, por ejemplo.

   'Today is Tuesday'
   'today-is-Tuesday'
   '32abc'

   c) Cualquier secuencia de uno o más caracteres especiales de una lista que incluye los siguientes +, -, *, /, >, <, =, &, #, @, por ejemplo:

   +++
   >=
   >
   +-

3. **Variables**

   En una consulta, una variable es un nombre que se utiliza para representar un término que se va a determinar, por ejemplo, la variable X

puede representar al átomo *dog*, el número 12.3, un término compuesto o una lista (ambos se describen a continuación).

El significado de una variable cuando se usa en una regla o hecho se describe en el Capítulo 2.

El nombre de una variable se denota por cualquier secuencia de una o más letras (mayúsculas o minúsculas), números y guiones bajos, comenzando con una letra mayúscula o un guión bajo, por ejemplo.

X
Author
Person_A
_123A

pero no

45_ABC
Person-A
author

Nota: La variable _ que consta de un solo guión bajo se conoce como variable anónima y está reservada para un propósito especial (consulte el Capítulo 2).

4. **Términos compuestos**

Los términos compuestos son de fundamental importancia al escribir programas Prolog. Un término compuesto es un tipo de datos estructurados que comienza con un átomo, conocido aquí como *funtor*. El funtor va seguido de una secuencia de uno o más argumentos, que están encerrados entre corchetes y separados por comas. La forma general es

$$functor(t_1, t_2, \cdots, t_n), n \geq 1.$$

Si está familiarizado con otros lenguajes de programación, puede que le resulte útil pensar en un término compuesto como una representación de una estructura de registro. El functor representa el nombre del registro, mientras que los argumentos representan los campos del registro.

El número de argumentos que tiene un término compuesto se llama *aridad*. Algunos ejemplos de términos compuestos son:

likes(paul,prolog)
read(X)
dog(henry)
cat(X)
>(3,2)
person('john smith',32,doctor,london)

Cada argumento de un término compuesto debe ser un término, que puede ser de cualquier tipo, incluido un término compuesto. Así, algunos ejemplos más complejos de términos compuestos son:

likes(dog(henry),Y)
pred3(alpha,beta,gamma,Q)
pred(A,B,likes(X,Y),-4,pred2(3,pred3(alpha,beta,gamma,Q)))

5. **Listas**

   A menudo se considera que una lista es un tipo especial de término compuesto, pero en este libro se tratará como un tipo separado de objeto de datos.

   Las listas se escriben como un número ilimitado de argumentos (conocidos como elementos de lista) encerrados entre corchetes y separados por comas, por ejemplo, [dog, cat, fish, man]. A diferencia de la aridad de un término compuesto, el número de elementos que tiene una lista no tiene que decidirse de antemano cuando se escribe un programa, como se explicará en el Capítulo **??**. Esto puede ser extremadamente útil.

   En esta etapa, todo lo que se necesita saber es que un elemento de una lista puede ser un término de cualquier tipo, incluido un término compuesto u otra lista, por ejemplo.

   [dog,cat,y,mypred(A,b,c),[p,q,R],z]
   [[john,28],robert,parent(victoria,albert),[a,b,[c,d,e],f],29]
   [[edinburgh,london,dover],[portsmouth,london,edinburgh],[glasgow]]

   Una lista sin elementos se conoce como *lista vacía*. Se escribe como [].

6. **Otros tipos de términos**

   Algunos dialectos de Prolog permiten otros tipos de términos, por ejemplo, cadenas de caracteres. Estos no serán descritos en este libro. Sin embargo, es posible utilizar átomos para realizar un tipo rudimentario de procesamiento de cadenas (consulte el Capítulo **??**).

   Los átomos y los términos compuestos tienen una importancia especial en las cláusulas de Prolog y se conocen colectivamente como *términos de llamada*. Volveremos sobre esto en próximos capítulos.

Resumen del capítulo

Este capítulo muestra cómo escribir programas Prolog simples, cargarlos en la base de datos de Prolog e ingresar metas que puedan evaluarse. También presenta la terminología básica y los diferentes tipos de objetos de datos (términos).

Ejercicios Prácticos 1

1. Cree un archivo de disco animals.pl que contenga el Programa de animales 1 (omitiendo los comentarios). Inicie Prolog y cargue su programa.

   Pruebe su programa con las consultas dadas en el texto y algunas otras hechas por usted.

2. Escriba un programa para poner hechos en la base de datos que indiquen que un león, un tigre y una vaca son animales y para registrar que dos de ellos (león y tigre) son carnívoros.

   Guarde su programa en un archivo de disco y cárguelo. Verifique que la base de datos sea correcta usando **listing**.

   Ingrese metas para probar si:

   a) hay un animal como un tigre en la base de datos

   b) una vaca y un tigre están ambos en la base de datos (una conjunción de dos metas)

   c) un león es un animal y también un carnívoro

   d) una vaca es un animal y también un carnívoro.

3. Trate de predecir qué generará Prolog en respuesta a cada una de las siguientes metas y luego pruébelas.

   ?-write(hello).
   ?-write(Hello).
   ?-write('Hello!').
   ?-write('Hello!'),nl.
   ?-100=100.
   ?-100=1000/10.
   ?-100 is 1000/10.
   ?-1000 is 100*10.
   ?-2 is (5+7)/6.
   ?-74 is (5+7)*6.

Capítulo 2

Cláusulas y Predicados

Objetivos del Capítulo

Después de leer este capítulo, debería ser capaz de:

- Identificar los componentes de reglas y hechos.
- Explicar el significado del término predicado
- Hacer uso correcto de variables en metas y cláusulas.

2.1. Cláusulas

Además de los comentarios y las líneas en blanco, que se ignoran, un programa Prolog consta de una sucesión de *cláusulas*. Una cláusula puede ocupar más de una línea o puede haber varias en la misma línea. Una cláusula termina con un carácter de punto, seguido de al menos un carácter de 'espacio en blanco', por ejemplo, un espacio o un retorno de carro.

Hay dos tipos de cláusula: *hechos* y *reglas*. Los hechos son de la forma

head.

head se llama la *cabeza de la cláusula*. Toma la misma forma que una meta ingresada por el usuario en el indicador, es decir, debe ser un átomo o un término compuesto. Los átomos y los términos compuestos se conocen colectivamente como *términos de llamada*. El significado de los términos de llamada se explicará en el Capítulo 3.

Algunos ejemplos de hechos son:

```
christmas.
likes(john,mary).
likes(X,prolog).
dog(fido).
```

Las reglas son de la forma:

head:-$\mathbf{t}_1$**,** $\mathbf{t}_2$**,** $\cdots$**,** $\mathbf{t}_k$**.** $(k \geq 1)$

head se llama la *cabeza de la cláusula* (o la *cabeza de la regla)* y, como con los hechos, debe ser un término de llamada, es decir, un átomo o un término compuesto.

:- se llama el *cuello de la cláusula* (o el '*operador cuello*'). Se lee como 'si'.

$t_1,\ t_2,\ \cdots,\ t_k$ se llama el *cuerpo de la cláusula* (o el *cuerpo de la regla*). Especifica las condiciones que deben cumplirse para que la conclusión, representada por la cabeza, sea satisfecha. El cuerpo consta de uno o más componentes, separados por comas. Los componentes son *metas* y las comas se leen como 'y'.

Cada meta debe ser un término de llamada, es decir, un átomo o un término compuesto. Una regla se puede leer como 'la *cabeza* es verdadera si $t_1,\ t_2,\ \cdots,\ t_k$ son todas verdaderas'.

La cabeza de una regla también puede verse como una meta con los componentes de su cuerpo vistos como submetas. Por lo tanto, otra lectura de una regla es 'para lograr la meta *head*, es necesario lograr las submetas $t_1,\ t_2,\ \cdots,\ t_k$'.

Algunos ejemplos de reglas son:

```
large_animal(X):-animal(X),large(X).
grandparent(X,Y):-father(X,Z),parent(Z,Y).
go:-write('hello world'),nl.
```

Aquí hay otra versión del programa animals, que incluye hechos y reglas.

```
/* Animals Program 2*/
dog(fido). large(fido).
cat(mary). large(mary).
dog(rover). dog(jane).
dog(tom). large(tom). cat(harry).
dog(fred). dog(henry).
cat(bill). cat(steve).
small(henry). large(fred).
large(steve). large(jim).
large(mike).
large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

fido, *mary*, *jane*, etc. son átomos, es decir, constantes, indicados por sus letras minúsculas iniciales. X y Y son variables, indicadas por sus letras mayúsculas iniciales.

Las primeras 18 cláusulas son hechos. Las dos cláusulas finales son reglas.

2.2. Predicados

El siguiente programa simple tiene cinco cláusulas. Para cada una de las tres primeras cláusulas, la cabeza es un término compuesto con *funtor* **parent** y *aridad* 2 (es decir, dos argumentos).

```
parent(victoria,albert).
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,henry).
mother(jane,henry).
```

Es posible (aunque es probable que cause confusión) que el programa también incluya cláusulas para las que el encabezado tiene un funtor **parent**, pero una aridad diferente, por ejemplo

```
parent(john).
parent(X):-son(Y,X).
/* X is a parent if X has a son Y */
```

También es posible que **parent** se use como un átomo en el mismo programa, por ejemplo en el hecho

```
animal(parent)
```

pero esto también puede causar confusión.

Todas las cláusulas (hechos y reglas) para las cuales la cabeza tiene una combinación dada de funtor y aridad comprenden una definición de predicado. Las cláusulas no tienen que aparecer como líneas consecutivas de un programa, pero hace que los programas sean más fáciles de leer si lo hacen.

Las cláusulas anteriores definen dos predicados con el nombre **parent**, uno con aridad dos y el otro con aridad uno. Estos pueden escribirse (en libros de texto, manuales de referencia, etc., no en programas) como **parent/2** y **parent/1**, para distinguirlos. Cuando no hay riesgo de ambigüedad, se acostumbra referirse a un predicado simplemente como **dog**, **large_animal**, etc.

Tenga en cuenta que una consulta como

?-listing(mypred).

da una lista de todas las cláusulas para el predicado **mypred** cualquiera que sea la aridad.

Un átomo que aparece como un hecho o como la cabeza de una regla, por ejemplo.

```
christmas.
go:-parent(john,B),write('john has a child named '),write(B),nl.
```

puede considerarse como un predicado sin argumentos, es decir, **go/0**.

Hay cinco predicados definidos en Animals Program 2: **dog/1**, **cat/1**, **large/1**, **small/1** y **large_animal/1**. Las primeras 18 cláusulas son hechos que definen los predicados **dog/1**, **cat/1**, **large/1** y **small/1**. Las dos cláusulas finales son reglas, que juntas definen el predicado **large_animal/1**.

Interpretaciones declarativas y procedimentales de las reglas

Las reglas tienen una interpretación tanto declarativa como procedimental. Por ejemplo, la interpretación declarativa de la regla

```
chases(X,Y):-dog(X),cat(Y),write(X),write(' chases '),write(Y),nl.
```

es: '**chases (X,Y)** es verdadero si **dog(X)** es verdadero y **cat(Y)** es verdadero y **write(X)** es verdadero, etc.'

La interpretación procedimental es 'Para satisfacer **chases (X,Y)**, primero satisface **dog(X)**, luego satisface **cat(Y)**, luego satisface **write(X)**, etc.'

Los hechos se interpretan generalmente de forma declarativa, es decir.

```
dog(fido).
```

se lee como 'fido es un perro'.

El orden de las cláusulas que definen un predicado y el orden de las metas en el cuerpo de cada regla son irrelevantes para la interpretación declarativa pero de vital importancia para la interpretación procedimental y, por lo tanto, para determinar si se satisface la secuencia de metas ingresada por el usuario en el indicador del sistema.

Al evaluar una meta, las cláusulas de la base de datos se examinan de arriba a abajo. Cuando sea necesario, las metas en el cuerpo de una regla se examinan de izquierda a derecha. Este tema será discutido en detalle en el Capítulo 3.

El programa de un usuario comprende hechos y reglas que definen nuevos predicados. Estos se denominan *predicados definidos por el usuario*. Además, existen predicados estándar predefinidos por el sistema Prolog. Estos se conocen como *predicados interconstruidos* (BIP) y no pueden ser redefinidos por un programa de usuario. Algunos ejemplos son: **write/1**, **nl/0**, **repeat/0**, **member/2**, **append/3**, **consult/1**, **halt/0**. Algunos BIP son comunes a todas las versiones de Prolog. Otros dependen de la versión.

Dos de los predicados interconstruidos más utilizados son **write/1** y **nl/0**.

El predicado **write/1** toma un término como argumento, por ejemplo.

write(hello)
write(X)
write('hello world')

Siempre que su argumento sea un término válido, el predicado **write** siempre tiene éxito y, como efecto secundario, escribe el valor del término en la pantalla del usuario. Para ser más precisos, se envía al flujo de salida actual, que por defecto se asumirá como la pantalla del usuario. La información sobre la salida a otros dispositivos se proporciona en el Capítulo 5. Si el argumento es un átomo entre comillas, por ejemplo, 'hola mundo', las comillas no se muestran.

El predicado **nl/0** es un átomo, es decir, un predicado que no acepta argumentos. El predicado siempre tiene éxito y, como efecto secundario, comienza una nueva línea en la pantalla del usuario.

El nombre de un predicado definido por el usuario (el funtor) puede ser cualquier átomo, con algunas excepciones, excepto que no puede redefinir ninguno de los predicados integrados del sistema Prolog. Es poco probable que desee redefinir el predicado **write/1** poniendo una cláusula como

```
write(27).
```

o

```
write(X):-dog(X).
```

en sus programas, pero si lo hace, el sistema le dará un mensaje de error como 'intento ilegal de redefinir un predicado interconstruido'.

Los predicados interconstruidos más importantes se describen en el Apéndice **??**. Es probable que cada versión de Prolog tenga otras, a veces muchas más, y si accidentalmente usa una con el mismo nombre y aridad para uno de sus propios predicados, obtendrá un mensaje de error como 'intento ilegal de redefinir un predicado interconstruido' o 'no hay permiso para modificar el procedimiento', que puede ser muy desconcertante.

En algunas versiones de Prolog, se puede permitir definir un predicado con el mismo funtor y una aridad diferente, por ejemplo, **write/3** pero definitivamente es mejor evitarlo.

Simplificando la introducción de metas

Al desarrollar o probar programas, puede ser tedioso ingresar repetidamente en el indicador del sistema una larga secuencia de metas, como

?-dog(X),large(X),write(X),write(' is a large dog'),nl.

Una técnica de programación comúnmente utilizada es definir un predicado como **go/0** o **start/0**, con la secuencia anterior de metas como el lado derecho de una regla, por ejemplo.

```
go:-dog(X),large(X),write(X),write(' is a large dog'),nl.
```

Esto permite que las metas introducidas en el indicador sean breves, por ejemplo.

```
?-go.
```

Recursión

Una técnica importante para definir predicados, que se usará con frecuencia más adelante en este libro, es definirlos en términos de sí mismos. Esto se conoce como una definición recursiva. Hay dos formas de recursividad.

1. Recurrencia directa. El predicado **pred1** se define en términos de sí mismo.

2. Recurrencia indirecta. El predicado **pred1** se define mediante **pred2**, que se define mediante **pred3**, $\cdots$, que se define mediante **pred1**.

La primera forma es más común. Un ejemplo de ella es

```
likes(john,X):-likes(X,Y),dog(Y).
```

lo que puede interpretarse como 'a John le gusta cualquiera que le guste al menos un perro'.

Predicados y funciones

El uso del término 'predicado' en Prolog está estrechamente relacionado con su uso en matemáticas. Sin entrar en detalles técnicos (este no es un libro de matemáticas), se puede pensar en un predicado como una relación entre una serie de valores (sus argumentos) como *likes(henry,mary)* o $X = Y$, que pueden ser verdaderos o falsos.

Esto contrasta con una función, como $6 + 4$, la *raíz cuadrada de 64* o los *tres primeros caracteres* de 'hola mundo', que puede evaluarse como un número, una cadena de caracteres o algún otro valor, como verdadero y falso. Prolog no hace uso de funciones excepto en expresiones aritméticas (ver Capítulo 4).

2.3. Cargando cláusulas

El uso del predicado interconstruido **consult/1** hace que las cláusulas contenidas en un archivo de texto se carguen en la base de datos como efecto secundario. Un programa Prolog es solo una colección de cláusulas (reglas

y hechos), por lo que nos referiremos a un archivo utilizado de esta manera como un archivo de programa.

Un método común de desarrollo de programas es cargar un programa completo (conjunto de cláusulas) como un solo archivo, probarlo, luego realizar cambios, guardar los cambios en una nueva versión del archivo, consultar el archivo nuevamente para cargar las cláusulas del nueva versión del archivo, y así sucesivamente hasta lograr una versión 'perfecta' del programa.

Mostraremos cómo funciona el uso repetido de **consult/1** de esta manera utilizando un archivo que contiene solo hechos (no reglas) en aras de la simplicidad.

Digamos que el archivo *testfile.pl* contiene las líneas

```
alpha.
beta.
dog(fido).
dog(misty).
dog(harry).
cat(jane).
cat(mary).
```

entonces la consulta

?-consult('testfile.pl').

coloca las siete cláusulas anteriores en la base de datos.

Si ahora cambiamos el archivo *testfile.pl*

```
gamma.
dog(patch).
elephant(dumbo).
elephant(fred).
```

luego, después de otro

?-consult('testfile.pl').

al consultar la base de datos contiene las cláusulas

```
gamma.
dog(patch).
elephant(dumbo).
elephant(fred).
```

Todas las cláusulas colocadas en la base de datos por el primer **consult** han sido eliminadas y reemplazadas por el contenido de la segunda versión de *testfile.pl*. Las dos cláusulas **cat** se encuentran entre las eliminadas, lo que puede no haber sido la intención del usuario.

La consulta de un archivo se realiza con tanta frecuencia que existe una notación simplificada disponible para ello.

?-['testfile.pl'].

es equivalente a

?-consult('testfile.pl').

Los nombres de archivo se consideran relativos al directorio desde el que se inició Prolog. Para consultar un archivo en otro directorio, están disponibles las convenciones habituales de nomenclatura de archivos, por ejemplo.

?-consult('/mydir/testfile.pl').

o

?-['../../mydir/testfile.pl'].

Se recomienda a los usuarios de Windows que el carácter de barra invertida habitual en los nombres de archivo se reemplace por una barra diagonal.

A veces es preferible dividir un programa (conjunto de cláusulas) en varios archivos y cargarlos por separado. El siguiente ejemplo muestra el efecto de consultar archivos con diferentes nombres.

Supongamos que la primera y la segunda versión de *testfile.pl* anterior se colocan en dos archivos con nombres diferentes, de modo que el archivo *testfile1.pl* contiene las líneas

```
alpha. beta.
dog(fido).
dog(misty).
dog(harry).
cat(jane).
cat(mary).
```

y el archivo *testfile2.pl* contiene las líneas

```
gamma.
dog(patch).
elephant(dumbo).
elephant(fred).
```

entonces la consulta

?-consult('testfile1.pl'), consult('testfile2.pl').

coloca las siguientes cláusulas en la base de datos

```
gamma.
alpha.
beta.
dog(patch).
cat(jane).
cat(mary).
elephant(dumbo).
elephant(fred).
```

Podemos ver que

- El átomo **gamma** de *testfile2.pl* se ha agregado a la base de datos para unirse a los dos átomos que ya están allí desde *testfile1.pl*.
- Las dos cláusulas para **cat/1** cargadas desde *testfile1.pl* permanecen en la base de datos.
- Las dos cláusulas para **elephant/1** se han cargado en la base de datos desde *testfile2.pl*.
- En el caso del predicado **dog/1**, para el que hay cláusulas en ambos archivos, el efecto de consultarlas en el orden dado es que las tres cláusulas **dog/1** en *testfile1.pl* han sido eliminadas y reemplazadas por la única cláusula **dog/1** en *testfile2.pl*.

El último de estos bien puede no ser lo que se pretendía. También significa que las consultas

?-consult('testfile1.pl'), consult('testfile2.pl').

y

?-consult('testfile2.pl'), consult('testfile1.pl').

daría resultados diferentes en lo que se refiere al predicado **dog/1**. Parece una buena política práctica, cuando se utilizan varios archivos, mantener separados los predicados en ellos.

La notación simplificada también se puede utilizar al consultar varios archivos, por ejemplo

?-['myfilea.pl', 'myfileb.pl', 'myfilec.pl'].

tiene el mismo efecto que

consult('myfilea.pl'), consult('myfileb.pl'),consult('myfilec.pl').

2.4. Variables

Las variables se pueden usar en el encabezado o el cuerpo de una cláusula y en las metas ingresadas en el indicador del sistema. Sin embargo, su interpretación depende de dónde se utilicen.

Variables en Metas

Las variables en las metas se pueden interpretar en el sentido de 'encontrar valores de las variables que hacen que la meta se cumpla'. Por ejemplo, la meta

?-large_animal(A).

se puede leer como 'encontrar un valor de A tal que se satisfaga large_animal(A)'.

A continuación se proporciona una tercera versión del Programa Animals (sólo se muestran las cláusulas adicionales a las del Programa Animals 2 en la Sección 2.1).

```
/* Animals Program 3 */
/* Como el programa Animals 2 pero con las reglas adicionales dadas a
continuación */
chases(X,Y):- dog(X),cat(Y), write(X),write(' chases '),write(Y),nl.
/* chases es un predicado con dos argumentos*/
go:-chases(A,B).
/* go es un predicado sin argumentos */
```

Una meta como

?-chases(X,Y).

significa encontrar valores de las variables X y Y que satisfagan **chases(X, Y)**.

Para hacer esto, Prolog busca a través de todas las cláusulas que definen el predicado **chases** (solo hay una en este caso) de arriba a abajo hasta que encuentra una cláusula coincidente. Luego trabaja a través de las metas en el cuerpo de esa cláusula una por una, trabajando de izquierda a derecha, intentando satisfacer cada una por turno. Este proceso se describe con más detalle en el Capítulo 3.

El resultado producido al cargar el Programa de Animals 3 e ingresar algunas metas típicas en el indicador es el siguiente.

?-consult('animals3.pl').	*Indicador del sistema*
% animals3.pl compiled 0.02 sec.	*animals3.pl cargado*
?- chases(X,Y).	El usuario retrocede para encontrar sólo las dos primeras soluciones.
fido chases mary	
X = fido,	
Y = mary;	Note el uso de los predicados *write* y *nl*.
fido chases harry	
X = fido,	
Y = harry ?-chases(D,henry).	Nadie persigue a henry
false.	
?-go.	Tenga en cuenta que no se emiten valores de variables. (El resultado es de los predicados *write* y *nl*, seguido de la palabra 'true'.) En algunas versiones de Prolog, el usuario tiene la oportunidad de retroceder, como aquí. En otras no.
fido chases mary	
true;	
fido chases harry	
true;	
fido chases bill	

true

Cabe señalar que no hay nada que impida que se genere la misma respuesta más de una vez mediante retroceso. Por ejemplo, si el programa es

```
chases(fido,mary):-fchasesm.
chases(fido,john).
chases(fido,mary):-freallychasesm.
fchasesm.
freallychasesm.
```

La consulta **?-chases(fido,X)** producirá dos respuestas idénticas de tres, por el retroceso.

?- chases(fido,X).
X = mary;
X = john;
X = mary
?-

Vinculando Variables

Inicialmente, se dice que todas las variables utilizadas en una cláusula no están vinculadas, lo que significa que no tienen valores. Cuando el sistema Prolog evalúa una meta, algunas variables pueden recibir valores como *dog*, -6.4, etc. Esto se conoce como vincular las variables. Una variable que ha sido vinculada puede volver a desvincularse y posiblemente luego vincularse a un valor diferente mediante el proceso de retroceso, que se describirá en el Capítulo 3.

Ámbito Léxico de las Variables

En una cláusula como

```
parent(X,Y):-father(X,Y).
```

las variables X y Y no tienen ninguna relación con ninguna otra variable con el mismo nombre utilizada en otros lugares. Todas las ocurrencias de las variables X y Y en la cláusula pueden ser reemplazadas consistentemente por cualquier otra variable, por ejemplo, por *First_person* y *Second_person* dando

```
parent(First_person,Second_person):- father(First_person,Second_person).
```

Esto no cambia el significado de la cláusula (o el programa del usuario) de ninguna manera. Esto a menudo se expresa diciendo que el *alcance léxico* de una variable es la cláusula en la que aparece.

Variables Cuantificadas Universalmente

Si una variable aparece en la cabeza de una regla o hecho, indica que la regla o el hecho se aplica a todos los valores posibles de la variable. Por ejemplo, la regla

```
large_animal(X):-dog(X),large(X).
```

puede leerse como 'para todos los valores de X, X es un animal grande si X es un perro y X es grande'.

Se dice que la variable X está *universalmente cuantificada*.

Variables Cuantificadas Existencialmente

Supongamos ahora que la base de datos contiene las siguientes cláusulas:

```
person(frances,wilson,female,28,architect).
person(fred,jones,male,62,doctor).
person(paul,smith,male,45,plumber).
person(martin,williams,male,23,chemist).
person(mary,jones,female,24,programmer).
person(martin,johnson,male,47,solicitor).
man(A):-person(A,B,male,C,D).
```

Las primeras seis cláusulas (todos los hechos) comprenden la definición de predicado **person/5**, que tiene cinco argumentos con interpretaciones obvias, es decir, el nombre, apellido, sexo, edad y ocupación de la persona representada por el hecho correspondiente.

La última cláusula es una regla, definida usando el predicado **person**, que también tiene una interpretación natural, es decir, 'para todo A, A es un hombre si A es una persona cuyo sexo es masculino'. Como se explicó anteriormente, la variable A en el encabezado de la cláusula (que representa el nombre en este caso) significa 'para todo A' y se dice que está universalmente cuantificada.

¿Qué pasa con las variables B, C y D? Sería una muy mala idea que se tomaran en el sentido de 'para todos los valores de B, C y D'. Para demostrar que, digamos, *paul* es un hombre, se necesitarían cláusulas de **person** con el nombre *paul* para todos los apellidos, edades y ocupaciones posibles, lo que claramente no es un requisito razonable. Una interpretación mucho más útil sería considerar que la variable B significa 'para al menos un valor de B' y de manera similar para las variables C y D.

Esta es la convención utilizada por el sistema Prolog. Así, la cláusula final en la base de datos significa 'para todo A, A es hombre si hay una persona con nombre A, apellido B, sexo masculino, edad C y ocupación D, para al menos un valor de B, C y D'.

En virtud de la tercer cláusula para **person**, *paul* califica como hombre, con valores *smith*, *45* y *plumber* para las variables B, C y D respectivamente.

?- man(paul).
true.

La distinción clave entre la variable A y las variables B, C y D en la definición de predicado **man** es que B, C y D no aparecen en la cabeza de la cláusula.

La convención usada por Prolog es que si una variable, digamos Y, aparece en el cuerpo de una cláusula pero no en su cabeza, se entiende que significa 'hay (o existe) al menos un valor de Y'. Se dice que tales variables están *existencialmente cuantificadas*. Así la regla

```
dogowner(X):-dog(Y),owns(X,Y).
```

puede interpretarse en el sentido de 'para todos los valores de X, X es dueño de un perro si hay algún Y tal que Y es un perro y X es dueño de Y'.

La Variable Anónima

Para saber si en la base de datos existe una cláusula correspondiente a alguien llamado *paul*, solo es necesario ingresar una meta como:

?- person(paul,Surname,Sex,Age,Occupation).

en el indicador. Prolog responde de la siguiente manera:

Surname = smith,
Sex = male,
Age = 45,
Occupation = plumber

En muchos casos puede ser que conocer los valores de algunas o todas las últimas cuatro variables no tenga importancia. Si sólo es importante establecer si hay alguien con el nombre de *paul* en la base de datos, una forma más fácil es usar la meta:

?- person(paul,_,_,_,_).
true.

El carácter de subrayado _ denota una variable especial, denominada *variable anónima*. Esto se usa cuando al usuario no le importa el valor de la variable.

Si sólo interesa el apellido de alguna persona llamada *paul*, se puede encontrar anonimizando las otras tres variables en una meta, por ejemplo.

?- person(paul,Surname,_,_,_).
Surname = smith

De manera similar, si sólo interesan las edades de todas las personas llamadas *martin* en la base de datos, sería más simple ingresar la meta:

?- person(martin,_,_,Age,_).

Esto dará dos respuestas mediante retroceso.

Age = 23;
Age = 47

Las tres variables anónimas no están vinculadas, es decir, no tienen valores dados, como normalmente se esperaría.

Tenga en cuenta que no se supone que todas las variables anónimas tengan el mismo valor (en los ejemplos anteriores no lo tienen). Dando la meta alternativa

?- person(martin,X,X,Age,X).

con la variable X en lugar de la variable anónima, produciría la respuesta

false.

ya que no hay cláusulas con el primer argumento *martin* donde los argumentos segundo, tercero y quinto son idénticos.

Resumen del capítulo

Este capítulo presenta los dos tipos de cláusula Prolog, a saber, hechos y reglas y sus componentes. También introduce el concepto de predicado y describe diferentes características de las variables.

Ejercicios Prácticos 2

1. Escriba el siguiente programa en un archivo y cárguelo en Prolog.

```
/* Animals Database */
animal(mammal,tiger,carnivore,stripes).
animal(mammal,hyena,carnivore,ugly).
animal(mammal,lion,carnivore,mane).
animal(mammal,zebra,herbivore,stripes).
animal(bird,eagle,carnivore,large).
animal(bird,sparrow,scavenger,small).
animal(reptile,snake,carnivore,long).
animal(reptile,lizard,scavenger,small).
```

   Inventa y prueba metas para encontrar

   a) todos los mamíferos,

   b) todos los carnívoros que son mamíferos,

   c) todos los mamíferos con rayas,

   d) si hay un reptil que tiene melena.

2. Escriba el siguiente programa en un archivo

```
/* Dating Agency Database */
person(bill,male).
person(george,male).
person(alfred,male).
person(carol,female).
person(margaret,female).
person(jane,female).
```

Amplíe el programa con una regla que defina el predicado **couple** con dos argumentos, siendo el primero el nombre de un hombre y el segundo el nombre de una mujer. Cargue su programa ampliado en Prolog y pruébelo.

Capítulo 3

Satisfacción de Metas

Objetivos del Capítulo

Después de leer este capítulo, debería ser capaz de:

- Determinar si dos términos de llamada se unifican y, por lo tanto, si un objetivo puede coincidir con una cláusula en la base de datos.
- Comprender cómo Prolog utiliza la unificación y el retroceso para evaluar una secuencia de metas ingresadas por el usuario.

3.1. Introducción

Ahora podemos ver más de cerca cómo Prolog satisface las metas. Una comprensión general de esto es esencial para cualquier uso no trivial del lenguaje. Una buena comprensión a menudo puede permitir al usuario escribir programas potentes de una manera muy compacta, con frecuencia usando solo unas pocas cláusulas.

El proceso comienza cuando el usuario ingresa una secuencia de metas en el indicador del sistema, por ejemplo

?- owns(X,Y),dog(Y),write(X),nl.

El sistema Prolog intenta satisfacer cada meta en orden, trabajando de izquierda a derecha. Cuando la meta implica variables, por ejemplo **owns(X,Y)**, esto generalmente implica vincularlas a valores, en este caso, X a **John** y Y a **Fido**. Si todas las metas tienen éxito, la secuencia completa de metas tiene

éxito. El sistema generará los valores de todas las variables que se utilizan en la secuencia de metas y cualquier otra salida de texto como efecto secundario de metas como **write(X)** y **nl**.

?- owns(X,Y),dog(Y),write(X),nl.
john
X = john,
Y = fido

Si no es posible satisfacer todas las metas (simultáneamente), la secuencia de metas fallará.

?- owns(X,Y),dog(Y),write(X),nl.
false.

Aplazaremos hasta la Sección 3.4 la cuestión de qué hace exactamente Prolog si, por ejemplo, el primer objetivo tiene éxito y el segundo falla.

Términos de Llamada

Cada meta debe ser un término de Prolog, como se define en el Capítulo 1, pero no cualquier tipo de término. Puede ser sólo un átomo o un término compuesto, no un número, variable, lista o cualquier otro tipo de término proporcionado por alguna implementación particular de Prolog. Este tipo restringido de término se denomina *término de llamada*. Las cabezas de las cláusulas y las metas en los cuerpos de las reglas también deben ser términos de llamada. La necesidad de que los tres tomen la misma forma (restringida) es esencial para lo que sigue.

Cada meta como **write('Hello World')**, **nl**, **dog(X)** y **go** tiene un predicado correspondiente, en este caso **write/1**, **nl/0**, **dog/1** y **go/0** respectivamente. El nombre del predicado (**write**, **nl**, etc.) se llama *funtor*. El número de argumentos que tiene se llama *aridad*.

Las metas relacionadas con los predicados interconstruidos se evalúan de una manera predefinida por el sistema Prolog, como se analizó para **write/1** y **nl/0** en el Capítulo 2. Los objetivos relacionados con los predicados definidos por el usuario se evalúan examinando la base de datos de reglas y hechos cargados por el usuario.

Prolog intenta satisfacer una meta comparándolo con las cabezas de las cláusulas en la base de datos, trabajando de arriba hacia abajo.

Por ejemplo, la meta

?-dog(X).

podría coincidir con el hecho

```
dog(fido).
```

para dar la salida

X=fido

Un principio fundamental de la evaluación de metas definidas por el usuario en Prolog es que cualquier meta que no se pueda cumplir utilizando los hechos y las reglas de la base de datos *falla*. No hay una posición intermedia, como 'desconocido' o 'no probado'. Esto es equivalente a hacer una suposición muy fuerte acerca de la base de datos llamada *suposición de mundo cerrado*: cualquier conclusión que no se pueda probar a partir de los hechos y las reglas de la base de datos es falsa. No hay otra información.

3.2. Unificación

Dada una meta para evaluar, Prolog trabaja a través de las cláusulas en la base de datos tratando de hacer coincidir la meta con cada cláusula, trabajando de arriba a abajo hasta que se encuentra una coincidencia. Si no se encuentra ninguna coincidencia, la meta falla. La acción tomada si se encuentra una coincidencia se describe en la Sección 3.3.

Prolog utiliza una forma muy general de coincidencia conocida como *unificación*, que generalmente implica que se asignan valores a una o más variables para hacer que dos términos de llamada sean idénticos. Esto se conoce como *vincular* las variables a los valores. Por ejemplo, los términos **dog(X)** y **dog(fido)** se pueden unificar vinculando la variable X al átomo **fido**, es decir, dando a X el valor **fido**. Los términos **owns(john,fido)** y **owns(P,Q)** pueden unificarse vinculando las variables P y Q a los átomos **john** y **fido**, respectivamente.

Inicialmente, todas las variables no están vinculadas, es decir, no tienen ningún valor. A diferencia de la mayoría de los otros lenguajes de programación, una vez que se ha vinculado una variable, se puede desvincular nuevamente y luego quizás vincularse a un nuevo valor mediante *retroceso*, lo cual se explicará en la Sección 3.4.

Primero se explica el proceso de unificar una meta con la cabeza de una cláusula. Después esa unificación se utilizará para explicar cómo Prolog satisface las metas.

Advertencia: una nota sobre la terminología

Las palabras unificado, unificar, etc. se usan de dos maneras diferentes, lo que a veces puede causar confusión. Cuando decimos que 'dos términos de llamada están unificados', queremos decir estrictamente que se intenta hacer que los términos de llamada sean idénticos (lo que generalmente implica vincular variables a valores). Este intento puede tener éxito o fallar.

Por ejemplo, los términos de llamada **likes(X,mary)** y **likes(john,Y)** pueden hacerse idénticos vinculando la variable X al átomo **john** y la variable Y al átomo **mary**. En este caso decimos que *la unificación tiene éxito*. Sin embargo, no hay forma de vincular variables a valores que hagan que los términos de llamada **likes(X,mary)** y **dog(Z)** sean idénticos. En este caso decimos que *falla la unificación* o que *los términos de llamada no logran unificarse*.

Expresiones como 'la unificación de los dos términos de llamada tiene éxito' a menudo se abrevian simplemente como 'los dos términos de llamada están unificados' o 'los dos términos de llamada se unifican'. El significado pretendido (el intento o el intento exitoso) suele ser obvio por el contexto, ¡pero es una trampa potencial para los inexpertos!

3.2.1. Unificando Términos de Llamada

El proceso se resume en el siguiente diagrama de flujo (Figura 3.1).

Hay tres casos a considerar. La más simple es cuando un átomo se unifica con otro átomo. Esto tiene éxito si y solo si los dos átomos son iguales, entonces

- la unificación de átomos **fido** y **fido** tiene éxito
- la unificación de los átomos **fido** y **'fido'** también tiene éxito, ya que las comillas circundantes no se consideran parte del átomo en sí
- falla la unificación de los átomos **fido** y **rover**.

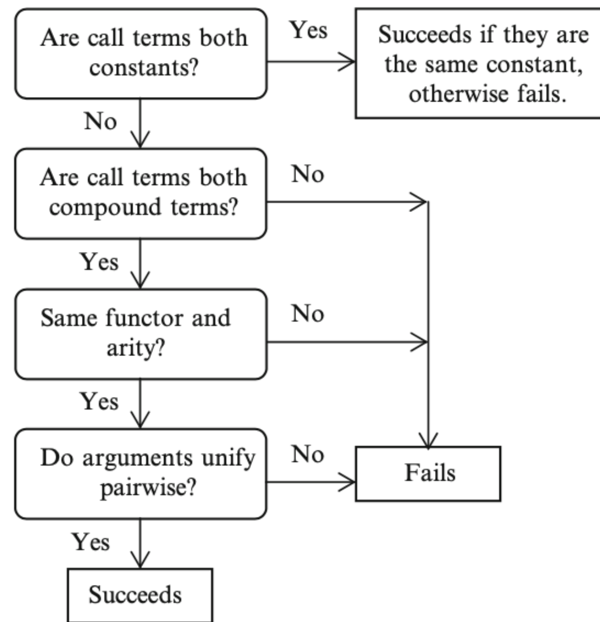


Figura 3.1: Unificación de dos términos de llamada.

Una segunda posibilidad es que un átomo esté unificado con un término compuesto, por ejemplo, **fido** con **likes(john, mary)**. Esto siempre falla.

El tercer caso, y con mucho el más común, es que se unifican dos términos compuestos, por ejemplo, **likes(X,Y)** con **likes(john,mary)** o **dog(X)** con **likes(john,Y)**. La unificación falla a menos que los dos términos compuestos tengan el mismo funtor y la misma aridad, es decir, el predicado es el mismo, por lo que la unificación de **dog(X)** y **likes(john,Y)** falla inevitablemente.

Unificar dos términos compuestos con el mismo funtor y aridad, por ejemplo, la meta **person(X,Y,Z)** con la cabeza **person(john,smith,27)**, requiere que los argumentos de la cabeza y la cláusula se unifiquen 'por pares', trabajando de izquierda a derecha, es decir, los primeros argumentos de los dos los términos compuestos se unifican, luego sus segundos argumentos se unifican, y así sucesivamente.

Entonces, X se unifica con **john**, luego Y con **smith**, luego Z con **27**. Si todos los pares de argumentos pueden unificarse (como pueden en este caso), la unificación de los dos términos compuestos tiene éxito. Si no, falla.

Los argumentos de un término compuesto pueden ser términos de cual-

quier tipo, es decir, números, variables y listas, así como átomos y términos compuestos. Unificar dos términos de este tipo sin restricciones implica considerar más posibilidades que unificar dos términos de llamada (Figura 3.2).

La unificación es probablemente más fácil de entender si se ilustra visualmente, para mostrar los pares de argumentos relacionados. Algunas unificaciones típicas se muestran a continuación.

person(X,Y,Z) person(john,smith,27)
Tiene éxito con las variables X, Y y Z vinculadas a **john**, **smith** y **27**, respectivamente.

person(john,Y,23) person(X,smith,27)
Si bien X se puede unificar con **john** y Y con **smith**, falla porque **23** no se puede unificar con **27**.

pred1(X,Y,[a,b,c]) pred1(A,prolog,B)
Tiene éxito con las variables X y A vinculadas entre sí, Y vinculada al átomo **prolog** y B vinculada a la lista **[a,b,c]**.

Variables Repetidas

Un caso un poco más complicado surge cuando una variable aparece más de una vez en un término compuesto.

pred2(X,X,man) pred2(london,dog,A)
?

- Dos átomos se unifican si y sólo si son iguales.
- Dos términos compuestos se unifican si y solo si tienen el mismo funtor y la misma aridad (es decir, el predicado es el mismo) y sus argumentos se pueden unificar por pares, trabajando de izquierda a derecha.
- Dos números unifican si y solo si son iguales, por lo que **7** unifica con **7**, pero no con **6.9**.
- Dos variables independientes, digamos X y Y, siempre se unifican, con las dos variables vinculadas entre sí.
- Una variable no vinculada y un término que no es una variable siempre se unifican, con la variable vinculada al término.
  - X y **fido** se unifican, con la variable X vinculada al átomo **fido**
  - X y [**a,b,c**] se unifican, con X vinculada a la lista [**a,b,c**]
  - X y **mypred(a,b,P,Q,R)** se unifican, con X vinculada a **mypred(a,b,P,Q,R)**
- Una variable vinculada se trata como el valor al que está vinculada.
- Dos listas se unifican si y solo si tienen el mismo número de elementos y sus elementos se pueden unificar por pares, trabajando de izquierda a derecha.
  - [**a,b,c**] se puede unificar con [**X,Y,c**], con X vinculada a **a** y Y vinculada a **b**
  - [**a,b,c**] no se puede unificar con [**a,b,d**]
  - [**a,mypred(X,Y),K**] se puede unificar con [**P,Z,third**], con las variables P, Z y K vinculadas al átomo **a**, el término compuesto **mypred(X,Y)** y el átomo **third**, respectivamente.
- Todas las demás combinaciones de términos no se unifican.

Figura 3.2: Unificación de dos términos.

Aquí los primeros argumentos de los dos términos compuestos se unifican con éxito, con X vinculada al átomo **london**. Todos los demás valores de X en el primer término compuesto también están vinculados al átomo **london** y, por lo tanto, se reemplazan efectivamente por ese valor antes de que tenga lugar cualquier unificación posterior. Cuando Prolog llega a examinar los dos segundos argumentos, ya no son X y **dog**, sino **london** y **dog**. Estos son átomos diferentes y, por lo tanto, no logran unificarse.

pred2(X,X,man)
pred2(london,dog,A)
Falla porque X no puede unificarse simultáneamente con los átomos **london** y **dog**.

En general, después de unificar cualquier par de argumentos, todas las variables vinculadas se reemplazan por sus valores.

El siguiente ejemplo muestra una unificación exitosa que involucra variables repetidas.

pred3(X,X,man)
pred3(london,london,A)
Tiene éxito con las variables X y A vinculdas a los átomos **london** y **man**, respectivamente

Este ejemplo muestra una variable repetida en uno de los argumentos de un término compuesto.

pred(alpha,beta,mypred(X,X,Y))
pred(P,Q,mypred(no,yes,maybe))
Falla

P se unifica con éxito con **alpha**. A continuación Q se unifica con **beta**. Luego, Prolog intenta unificar los dos terceros argumentos, es decir, **mypred(X,X,Y)** y **mypred(no,yes,maybe)**. El primer paso es unificar la

variable X con el átomo **no**. Esto tiene éxito con X vinculada a **no**. A continuación se comparan los dos segundos argumentos. Como X está vinculado a no, en lugar de X y **yes**, los segundos argumentos ahora son **no** y **yes**, por lo que la unificación falla.

En este ejemplo, el segundo argumento de **mypred** ahora es **no** en lugar de **yes**, por lo que la unificación tiene éxito.

```
pred(alpha,beta,mypred(X,X,Y))

pred(P,Q,mypred(no,no,maybe))
```

Tiene éxito con las variables P, Q, X y Y vinculadas a los átomos **alpha**, **beta**, **no** y **jmaybe**, respectivamente.

3.3. Evaluación de Metas

Dada una meta como **go** o **dog(X)**, Prolog busca en la base de datos de arriba a abajo examinando aquellas cláusulas que tienen cabezas con el mismo funtor y aridad hasta que encuentra la primera para la que la cabeza unifica con la meta. Si no hay ninguna, la meta falla. Si logra una unificación exitosa, el resultado depende de si la cláusula es una regla o un hecho.

Si la cláusula es un hecho, la meta tiene éxito inmediatamente. Si es una regla, Prolog evalúa las metas en el cuerpo de la regla una por una, de izquierda a derecha. Si todas tienen éxito, la meta original tiene éxito. (El caso en que no todas tengan éxito se tratará en la Sección 3.4.)

Usaremos la frase 'una meta *coincide* con una cláusula' para indicar que unifica con la cabeza de la cláusula.

Ejemplo

En este ejemplo, la meta es

?-pred(london,A).

Se supone que la primera cláusula en la base de datos con predicado **pred/2** y una cabeza que unifica con esta meta es la siguiente regla, a la que llamaremos Regla 1 para facilitar la referencia.

```
pred(X,'european capital'):- capital(X,Y),european(Y),write(X),nl.
```

La unificación vincula a X con el átomo **london** y a A con el átomo **'european capital'**. La vinculación de X con **london** afecta a todas las apariciones de X en la regla. Podemos mostrar esto esquemáticamente como:

```
?-pred(london,A).

pred(london,'european capital'):-capital(london,Y),european(Y),
write(london),nl.
```

X está vinculada con **london**, A está vinculada con **'european capital'**.

A continuación Prolog examina las metas en el cuerpo de la Regla 1 una por una, trabajando de izquierda a derecha. Todas ellas tienen que ser satisfechas para que la meta original tenga éxito.

La evaluación de cada uno de estas metas se lleva a cabo exactamente de la misma manera que la evaluación de la meta original del usuario. Si una meta se unifica con la cabeza de una regla, esto implicará la evaluación de las metas en el cuerpo de esa regla, y así sucesivamente.

Supondremos que la primera cláusula que coincidió con la meta **capital(london, Y)** es el hecho **capital(london,england)**. La primera meta en el cuerpo de la Regla 1 se cumple, con Y vinculada al átomo **england**. Este vínculo afecta a todas las apariciones de Y en el cuerpo de la Regla 1, no sólo al primero, por lo que ahora tenemos

```
?-pred(london,A).

pred(london,'european capital'):- capital(london,england),
european(england),write(london),nl.

capital(london,england).
```

X está vinculada con **london**, A está vinculada con **'european capital'**, Y está vinculada con **england**.

Ahora es necesario tratar de satisfacer la segunda meta en el cuerpo de la Regla 1, que en forma reescrita es **european(england)**.

Esta vez supondremos que la primera cláusula en la base de datos que tiene una cabeza que unifica con la meta es la regla

```
european(england):-write('God Save the Queen!'),nl.
```

la llamaremos Regla 2.

Prolog ahora intenta satisfacer las metas del cuerpo de la regla 2: **write('God Save the Queen!')** y **nl**. Lo hace con éxito, escribiendo la línea de texto

God Save the Queen!

como efecto secundario.

Las primeras dos metas en el cuerpo de la Regla 1 ahora se han cumplido. Hay dos metas más, que en forma reescrita son **write(london)** y **nl**. Ambos tienen éxito, escribiendo la línea de texto

london

como efecto secundario.

Todas las metas en el cuerpo de la Regla 1 ahora han tenido éxito, por lo que la meta que forma su cabeza tiene éxito, es decir, **pred(london,'european capital')**.

Esto, a su vez, significa que la meta original ingresada por el usuario

?-pred(london,A).

tiene éxito, con *A* vinculada a **'european capital'**.

La salida producida por el sistema Prolog sería:

?- pred(london,A).
God Save the Queen!
london
A = 'european capital'

Ahora podemos ver por qué la salida de las metas **write/1** y **nl/0** se denomina con el término ligeramente desdeñoso 'efecto secundario'. El enfoque principal del sistema Prolog es la evaluación de metas (ya sea ingresadas por el usuario o en los cuerpos de reglas), mediante unificación con las cabezas de las cláusulas. Todo lo demás es incidental. Por supuesto, con frecuencia son los efectos secundarios los que más interesan al usuario.

Este proceso de satisfacer la meta del usuario crea vínculos entre la meta, las cabezas de las cláusulas y las metas en los cuerpos de las reglas. Aunque el proceso es largo de describir, por lo general es bastante fácil visualizar los vínculos.

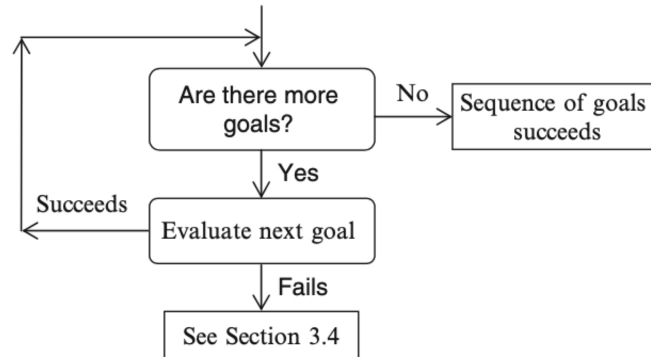


Figura 3.3: Evaluación de una secuencia de metas.

```
?-pred(london,A).

pred(london,'european capital'):- capital(london,england),
european(england),write(london),nl.

capital(london,england).

european(england):-write('God Save the Queen!'),nl.
```

X está vinculada con **london**, *A* está vinculada con **'european capital'**, *Y* está vinculada con **england**.

El proceso de evaluación de una meta se resume (en forma muy simplificada) en las Figuras 3.3 y 3.4. Tenga en cuenta que el diagrama de flujo para evaluar una secuencia de metas se refiere al de evaluar una (única) meta, y viceversa (Figuras 3.3 y 3.4).

El tema principal que se ha dejado sin considerar en este relato es qué sucede si falla la evaluación de cualquiera de las metas. Si lo hace, el sistema Prolog intenta encontrar otra forma de satisfacer la meta previa satisfecha más recientemente. Esto se conoce como *retroceso* y es el tema de la siguiente sección. La unificación y el retroceso juntos comprenden el mecanismo que utiliza Prolog para evaluar todas las metas, ya sea que las ingrese el usuario en el indicador o que estén en el cuerpo de una regla.

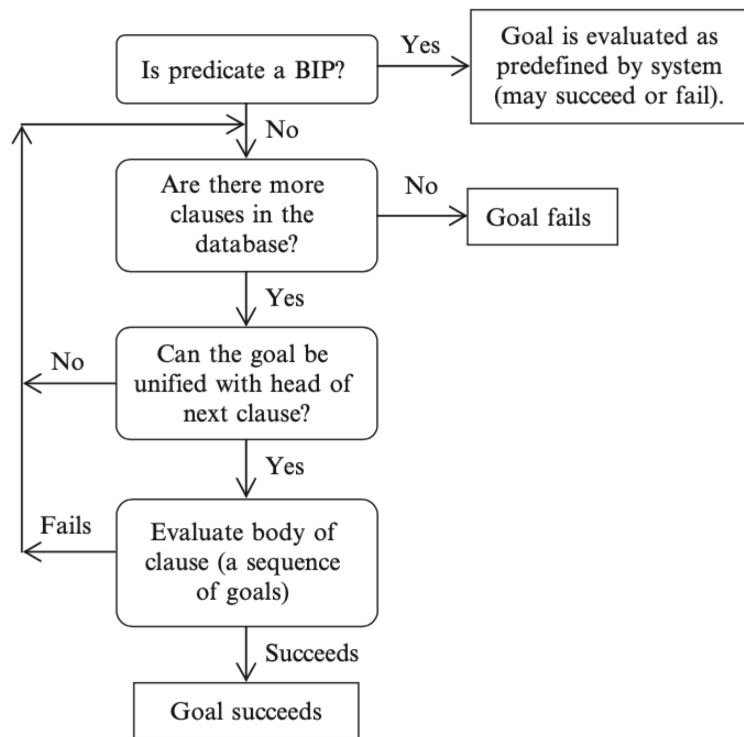


Figura 3.4: Evaluación de una meta.

3.4. Retroceso

Retroceso es el proceso de volver a un objetivo anterior y tratar de volver a satisfacerlo, es decir, encontrar otra forma de satisfacerlo.

Esta sección ofrece dos explicaciones muy detalladas de la forma en que Prolog intenta satisfacer una secuencia de metas mediante la unificación y el retroceso. Con la práctica, es bastante fácil calcular la secuencia de operaciones mediante una inspección visual de la base de datos. Sin embargo, puede ser útil tener una cuenta detallada disponible como referencia.

El Ejemplo de las Relaciones Familiares

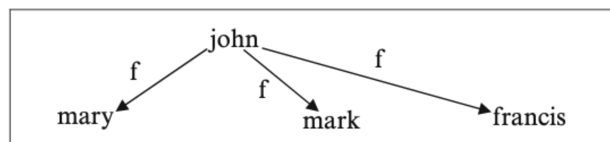
Este ejemplo se refiere a las relaciones familiares entre un grupo de personas. Las cláusulas que se muestran a continuación comprenden 10 hechos que definen el predicado **mother/2**, 9 hechos que definen el predicado **father/2** y 6 cláusulas que definen el predicado **parent/2**.

Hechos como

```
mother(jane,mark).
father(john,mark).
```

puede interpretarse en el sentido de 'jane es la madre de mark' y 'john es el padre de mark', respectivamente.

Tenga en cuenta que las etiquetas como [M1] se han agregado aquí solo con fines de referencia. No forman parte de las cláusulas y no deben incluirse en ningún archivo de programa. Los hechos relevantes para los siguientes ejemplos se pueden mostrar esquemáticamente de la siguiente manera (con 'f' en lugar de 'padre').



```
[M1] mother(ann,henry).
[M2] mother(ann,mary).
[M3] mother(jane,mark).
]lñ´++‘+
[M4] mother(jane,francis).
[M5] mother(annette,jonathan).
[M6] mother(mary,bill).
[M7] mother(janice,louise).
[M8] mother(lucy,janet).
[M9] mother(louise,caroline).
[M10] mother(louise,martin).
[F1] father(henry,jonathan).
[F2] father(john,mary).
[F3] father(francis,william).
[F4] father(francis,louise).
[F5] father(john,mark).
[F6] father(gavin,lucy).
[F7] father(john,francis).
[F8] father(martin,david).
[F9] father(martin,janet).
[P1] parent(victoria,george).
[P2] parent(victoria,edward).
[P3] parent(X,Y):-write('mother?'),nl,mother(X,Y), write('mother!'),nl.
[P4] parent(A,B):-write('father?'),nl,father(A,B), write('father!'),nl.
[P5] parent(elizabeth,charles).
[P6] parent(elizabeth,andrew).
```

Ejemplo 1

Dada la consulta

?-parent(john,Child),write('The child is '),write(Child),nl.

Prolog intenta satisfacer todos las metas de la secuencia (simultáneamente) y, al hacerlo, encontrará uno o más valores posibles para la variable *Child*. Comienza con la primer meta **parent(john,Child)** e intenta unificarlo con la cabeza de cada una de las cláusulas que definen el predicado **parent/2** a su vez, trabajando de arriba hacia abajo.

Primero llega a las cláusulas [P1] y [P2] pero no logra hacer coincidir la meta con (es decir, unificar la meta con la cabeza de) ninguno de ellos. Luego llega a la cláusula [P3] y esta vez la meta se unifica con éxito con la cabeza de la cláusula, con X vinculada a **john** y las variables Y y *Child* vinculadas entre sí.

```
?-parent(john,Child),write('The child is'),write(Child),nl.

[P3] parent(john,Y):-write('mother?'),nl,mother(john,Y),
write('mother!'),nl.
```

X se vincula con **john** y las variables Y y *Child* se vinculadan entre sí.

El sistema ahora funciona a través de las metas en el cuerpo de la regla [P3] tratando de hacer que cada uno tenga éxito a su vez. Evalúa con éxito las metas **write('mother?')** y **nl**, generando la línea de texto

mother?

como efecto secundario.

Luego llega a la tercera de las metas, es decir, **mother(john, Y)**. Esto no se unifica con la cabeza de ninguna de las cláusulas [M1] a [M10] que definen el predicado **mother/2**, por lo que la meta *falla*.

El sistema ahora *retrocede*. Vuelve a la última meta satisfecha en el cuerpo de [P3], moviéndose de derecha a izquierda, que es nl, y trata de *volver a satisfacerla*, es decir, de encontrar otra forma de satisfacerla.

Como muchos predicados interconstruidos (pero no todos), **nl/0** *no se puede volver a satisfacer*, lo que significa que siempre falla cuando se evalúa durante el retroceso.

Prolog ahora se mueve una posición más a la izquierda en el cuerpo de [P3], a la meta **write('mother?')**. El predicado **write/1** tampoco se puede volver a satisfacer, por lo que esta meta también falla.

No hay más metas en el cuerpo de la regla [P3], trabajando de derecha a izquierda, por lo que el sistema rechaza la regla [P3]. Ahora tenemos simplemente

```
?-parent(john,Child),write('The child is '),write(Child),nl.
```

La variable *Child* no está vinculada.

con la variable *Child* sin vincularse.

Prolog ahora vuelve a la meta anterior evaluada más recientemente, que en este caso es **parent(john, Child)**, e intenta volver a satisfacerla. Continúa buscando en la base de datos cláusulas que definan el predicado **parent/2** desde el punto al que había llegado previamente, es decir, la cláusula [P3]. Primero examina la cláusula [P4] y unifica con éxito la meta con su cabeza, con la variable *A* vinculada a **john** y las variables *B* y *Child* vinculadas entre sí.

```
?-parent(john,Child),write('The child is '),write(Child),nl.

[P4] parent(john,B):-write('father?'),nl,father(john,B),write('father!'),nl.
```

A se vincula con **john**. Las variables *B* y *Child* se vinculan entre sí.

El sistema ahora funciona a través de las metas en el cuerpo de la regla [P4] tratando de hacer que cada una tenga éxito. Las dos primeras metas tienen éxito, escribiendo la línea de texto

father?

como efecto secundario.

El sistema ahora intenta satisfacer la tercer meta, es decir, **father(john, B)**. Busca a través de las cláusulas que definen el predicado **father/2**, de arriba a abajo.

La primera cláusula con la que coincide es [F2], lo cual es un hecho. Esto hace que la variable *B* esté vinculada al átomo **mary**. Esto, a su vez, hace que la variable *Child* (que está vinculada a la variable *B*) esté vinculada al átomo **mary**.

```
?-parent(john,Child),write('The child is '),write(Child),nl.

[P4] parent(john,mary):-write('father?'),nl,father(john,mary),
write('father!'),nl.

[F2] father(john,mary).
```

A se vincula con **john**. Las variables *B* y *Child* se vinculan entre sí y al átomo **mary**.

Hay dos metas adicionales en el cuerpo de la regla [P4], es decir, **write('father!')** y nl. Ambas tienen éxito con la línea de texto.

father!

como efecto secundario de salida. Todas las metas en el cuerpo de [P4] ahora han tenido éxito, por lo que la cabeza de la cláusula, que en forma reescrita es **parent(john,mary)**, tiene éxito. Por lo tanto, la meta **parent(john,Child)** en la consulta del usuario tiene éxito.

La primera de las metas en la secuencia ingresada por el usuario ahora se ha cumplido. Hay tres metas más en la secuencia: **write('The child is ')**, **write(Child)** y **nl**. Todas tienen éxito, como efecto secundario se genera la línea de texto.

The child is mary

Todos las metas de la consulta del usuario ahora se han cumplido. El sistema Prolog genera el valor de todas las variables utilizadas en la consulta. En este caso, el único es *Child*.

parent(john,Child),write('The child is '),write(Child),nl.
mother?
father?
father!
The child is mary
Child = mary

Obligando al Sistema a Retroceder para Encontrar más Soluciones

El usuario ahora puede obligar al sistema a retroceder para encontrar una solución o soluciones adicionales ingresando un carácter de punto y coma. Esto funciona al forzar el fallo de la meta satisfecha más recientemente, es decir, **nl** (la última meta en la consulta del usuario).

El sistema ahora retrocede a la meta anterior en la secuencia, es decir, **write(Child)**. Esto también falla al retroceder, al igual que la meta anterior, es decir, **write('The child is ')**. El sistema retrocede un paso más, hasta la primera meta de la consulta, que es **parent(john,Child)**.

El sistema intenta encontrar otra forma de satisfacerla, comenzando por tratar de encontrar otra forma de satisfacer la última meta en el cuerpo de [P4]. Esto es **nl**, que falla al retroceder. Así también falla la meta anterior **write('father!')**.

?-parent(john,Child),write('The child is '),write(Child),nl.

[P4] parent(john,mary):-write('father?'),nl,father(john,mary),
write('father!'),nl.

[F2] father(john,mary).

A se vincula con **john**. Las variables *B* y *Child* se vinculan entre sí y al átomo **mary**.

Ahora intenta volver a satisfacer la meta anterior en el cuerpo de [P4], trabajando de derecha a izquierda, que es **father(john,B)**. Este proceso comienza por rechazar la unificación con la cabeza de [F2]. Prolog ahora continúa buscando a través de las cláusulas que definen el predicado **father/2** para más unificaciones.

La próxima unificación exitosa es con la cabeza de la cláusula [F5]. Los términos **father(john, B)** y **father(john, mark)** se unifican con la variable *B* vinculada a **mark**. Esto hace que la variable *Child* también se vincule a **mark**.

?-parent(john,Child),write('The child is '),write(Child),nl.

[P4] parent(john,mark):-write('father?'),nl,father(john,mark),
write('father!'),nl.

[F5] father(john,mark).

A se vincula con **john**. Las variables *B* y *Child* se vinculan entre sí y al átomo **mark**.

Esto da una segunda solución a la meta del usuario, es decir, una segunda forma de satisfacerlo. Retrocediendo más se encontrará una tercera solución, usando la cláusula [F7].

?-parent(john,Child),write('The child is '),write(Child),nl.

[P4] parent(john,francis):-write('father?'),nl,father(john,francis),
write('father!'),nl.

[F7] father(john,francis).

A se vincula con **john**. Las variables *B* y *Child* se vinculan entre sí y al átomo **francis**.

?- parent(john,Child),write('The child is '),write(Child),nl.
mother?
father?
father!
The child is mary
Child = mary ;
father!
The child is mark
Child = mark ;
father!
The child is francis
Child = francis

Si el usuario vuelve a introducir un punto y coma para obligar al sistema a retroceder, el sistema volverá a pasar por la secuencia de retroceso descrita anteriormente, hasta llegar a la etapa de intento de volver a satisfacer a **father(john, B)**, rechazando la unificación con la cabeza de la cláusula [F7] encontrada previamente y continuando la búsqueda a través de las cláusulas que definen el predicado **father/2** para más coincidencias. Como no se encuentran más unificaciones, la meta **father(john, B)** en el cuerpo de la regla [P4] ahora fallará.

El sistema ahora intenta volver a satisfacer la meta a su izquierda en el cuerpo de la regla [P4]. Esto es **nl**, que siempre falla al retroceder. La siguiente meta, de nuevo moviéndose hacia la izquierda, es **write('father?')**, que también falla. No hay más metas en el cuerpo de [P4], moviéndose de derecha a izquierda, por lo que el sistema rechaza la regla [P4]. Esto lo lleva de regreso a la meta original **parent(john,Child)**, que intenta volver a satisfacer.

Continúa buscando en las cláusulas que definen el predicado **parent/2** desde el punto al que llegó previamente ([P4]), pero no encuentra más coincidencias, por lo que la meta falla. Como este es la primera de la secuencia de metas ingresadas por el usuario, no es posible retroceder más y la consulta del usuario finalmente falla.

```
?- parent(john,Child),write('The child is '),write(Child),nl.
mother?
father?
father!
The child is mary
Child = mary ;
father!
The child is mark
Child = mark ;
father!
The child is francis
Child = francis ;
?-
```

El indicador del sistema se muestra para indicar que no hay más soluciones disponibles al retroceder.

Ejemplo 2

En el siguiente ejemplo, las cláusulas en la base de datos son como antes, con la adición de las cláusulas

```
[R1] rich(jane).
[R2] rich(john).
[R3] rich(gavin).
[RF1] rich_father(X,Y):-rich(X),father(X,Y).
```

Se han agregado nuevamente etiquetas como [R1] para facilitar la referencia. No forman parte de las cláusulas.

Dada la meta

?-rich_father(A,B).

Prolog comienza tratando de unificar la meta con las cabezas de todas las cláusulas que definen el predicado **rich_father/2**. Sólo hay uno, es decir, la cláusula [RF1]. La unificación tiene éxito y las variables A y X están vinculadas entre sí. Las variables B y Y también están vinculadas entre sí.

```
?-rich_father(A,B).

[RF1] rich_father(X,Y):-rich(X),father(X,Y).
```

Las variables A y X están vinculadas entre sí. Las variables B y Y están vinculadas entre sí.

A continuación, Prolog intenta encontrar un valor de A que satisfaga la primer meta en el cuerpo de la regla [RF1]. Lo hace buscando en las cláusulas que definen el predicado rich/1. La primera unificación que encuentra es con la cabeza de [R1], es decir, **rich(jane)**. X está vinculada a Jane.

```
?-rich_father(A,B).

[RF1] rich_father(jane,Y):-rich(jane),father(jane,Y).

[R1] rich(jane).
```

Las variables A y X están vinculadas entre sí y al átomo **jane**. Las variables B y Y están vinculadas entre sí.

El sistema ahora intenta satisfacer la meta **father(jane,Y)** examinando las cláusulas que definen el predicado **father/2**, es decir, [F1] a [F9]. Ninguna de ellas se unifica con la meta, por lo que el sistema retrocede e intenta volver a satisfacer (es decir, encontrar otra solución) la meta satisfecha más recientemente, que es **rich(X)**.

Continúa buscando a través de las cláusulas que definen el predicado **rich/1**, la siguiente unificación encontrada es con **rich(john)** (cláusula [R2]). Ahora X está vinculada a **john**, lo que a su vez hace que A esté vinculada a **john**.

```
?-rich_father(A,B).

[RF1] rich_father(john,Y):-rich(john),father(john,Y).

[R2] rich(john).
```

Las variables A y X están vinculadas entre sí y al átomo **john**. Las variables B y Y están vinculadas entre sí.

El sistema ahora intenta satisfacer la meta **father(john,Y)** examinando las cláusulas que definen el predicado **father/2**, es decir, [F1] a [F9]. La primera unificación encontrada es con [F2], es decir, **father(john,mary)**. Y está vinculada a **mary**.

?-rich_father(A,B). [RF1] rich_father(john,mary):-rich(john),father(john,mary). [R2] rich(john). [F2] father(john,mary).
Las variables A y X están vinculadas entre sí y al átomo **john**. Las variables B y Y están vinculadas entre sí y al átomo **mary**.

No hay más metas en el cuerpo de [RF1], por lo que la regla tiene éxito. Esto, a su vez, hace que la meta **rich_father/2** tenga éxito, con A y B vinculadas a **john** y **mary**, respectivamente.

?- rich_father(A,B).
A = john ,
B = mary

El usuario ahora puede obligar al sistema a retroceder para encontrar más soluciones ingresando un carácter de punto y coma. Si es así, intenta volver a satisfacer el objetivo que coincidió más recientemente, es decir, **father(john,Y)** al rechazar la coincidencia con [F2] encontrada anteriormente. Esto hace que B y Y ya no estén vinculadas a **mary** (pero todavía están vinculados entre sí).

El sistema continúa buscando las cláusulas que definen el predicado **father/2** para encontrar más coincidencias. La siguiente unificación encontrada es con la cabeza de la cláusula [F5]. La variable Y está vinculada a **mark**.

?-rich_father(A,B).

[RF1] rich_father(john,mark):-rich(john),father(john,mark).

[R2] rich(john).

[F5] father(john,mark).

Las variables A y X están vinculadas entre sí y al átomo **john**. Las variables B y Y están vinculadas entre sí y al átomo **mark**.

Esto da una segunda solución a la meta del usuario. Si el usuario obliga al sistema a retroceder nuevamente, encontrará una tercera solución utilizando la cláusula [F7] **father(john, francis)**.

?-rich_father(A,B).

[RF1] rich_father(john,francis):-rich(john),father(john,francis).

[R2] rich(john).

[F7] father(john,francis).

Las variables A y X están vinculadas entre sí y al átomo **john**. Las variables B y Y están vinculadas entre sí y al átomo **francis**.

Si el usuario obliga al sistema a retroceder nuevamente, comenzará por considerar que la meta cumplida más recientemente, es decir, **father(john,Y)** ha fallado. Esto hace que B y Y ya no estén vinculadas a **francis** (pero todavía están vinculados entre sí).

?-rich_father(A,B).

[RF1] rich_father(john,Y):-rich(john),father(john,Y).

[R2] rich(john).

Las variables A y X están vinculadas entre sí y al átomo **john**. Las variables B y Y están vinculadas entre sí.

El sistema no podrá encontrar más coincidencias para la meta **father(john, Y)**. A continuación, intentará encontrar más soluciones a la meta anterior satisfecha más recientemente en [RF1], trabajando de derecha a izquierda. Esta es **rich(X)**. Esta tendrá éxito con X ahora vinculada a **gavin** (cláusula [R3]).

```
?-rich_father(A,B).

[RF1] rich_father(gavin,Y):-rich(gavin),father(gavin,Y).

[R3] rich(gavin).
```

Las variables A y X están vinculadas entre sí y al átomo **gavin**. Las variables B y Y están vinculadas entre sí.

Trabajando nuevamente de izquierda a derecha, el sistema ahora intentará satisfacer la meta **father(gavin, Y)**. Ésta se unificará con la cabeza de sólo una de las cláusulas **father/2**, es decir, con la cláusula [F6] **father(gavin,lucy)**, con la variable Y vinculada a **lucy**.

```
?-rich_father(A,B).

[RF1] rich_father(gavin,lucy):-rich(gavin),father(gavin,lucy).

[R3] rich(gavin).

[F6] father(gavin,lucy).
```

Las variables A y X están vinculadas entre sí y al átomo **gavin**. Las variables B y Y están vinculadas entre sí y al átomo **lucy**.

Todos las metas en el cuerpo de [RF1] ahora han tenido éxito, por lo que la cabeza **rich_father(gavin,lucy)** tiene éxito y, a su vez, **rich_father(A,B)** tiene éxito con A y B vinculadas a **gavin** y **lucy**, respectivamente.

Obligar al sistema a retroceder nuevamente conducirá a la misma secuencia de operaciones que la anterior, hasta el intento de encontrar más coincidencias para la meta **rich(X)** en el cuerpo de [PF1]. Esto fallará, lo que a su vez hará que [RF1] falle. Esto hará que el sistema Prolog retroceda un paso más para intentar encontrar otra coincidencia para meta original

rich_father(A,B) con cláusulas que definan el predicado **rich_father/2**. Dado que solo existe una cláusula de este tipo, no se encontrarán más coincidencias y la meta del usuario finalmente fallará.

```
?- rich_father(A,B).
A = john ,
B = mary ;
A = john ,
B = mark ;
A = john ,
B = francis ;
A = gavin ,
B = lucy ;
?-
```

3.5. Satisfacción de Reglas: Un Resumen

El método descrito en las secciones anteriores se muestra en forma de diagrama en las Figuras 3.5 y 3.6. Observe cómo los dos diagramas de flujo se refieren entre sí.

Evaluación de una secuencia de metas: Resumen

Evalúe las metas por turnos, trabajando de izquierda a derecha. Si todas tienen éxito, toda la secuencia de metas tiene éxito. Si una falla, retroceda sobre las metas anteriores en la secuencia una por una de derecha a izquierda tratando de volver a satisfacerlas. Si todas fallan, toda la secuencia falla. Tan pronto como alguna se satisfaga, comience a trabajar de nuevo en las metas de izquierda a derecha.

Evaluación/Reevaluación de una meta: Resumen

Busque en la base de datos las cláusulas, trabajando de arriba a abajo[1] hasta encontrar una cuya cabeza coincida con la meta. Si la cláusula con la que coincide es un hecho, la meta tiene éxito. Si es una regla, evalúe la secuencia de metas en su cuerpo. Si la secuencia tiene éxito, la meta tiene éxito. Si no es así, continúe buscando en la base de datos para encontrar más coincidencias. Si se alcanza el final de la base de datos, la meta falla.

[1] Comience en la parte superior (para evaluación) o después de la cláusula con la que coincidió cuando se satisfizo la meta por última vez (para reevaluación).

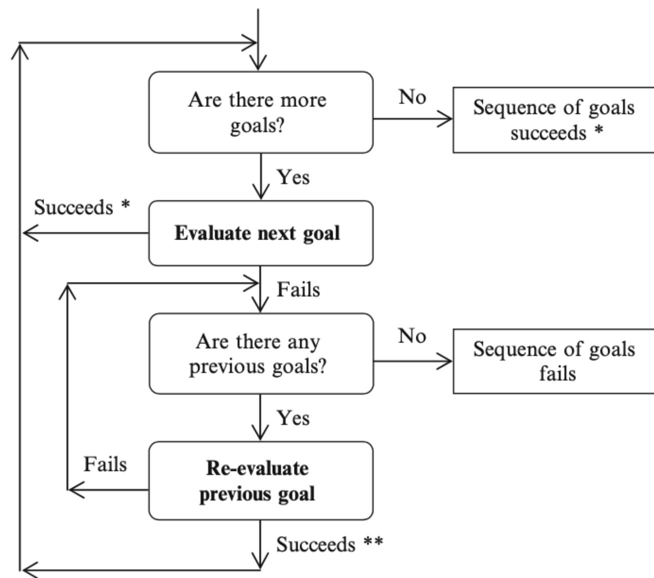


Figura 3.5: Evaluación de una secuencia de metas.

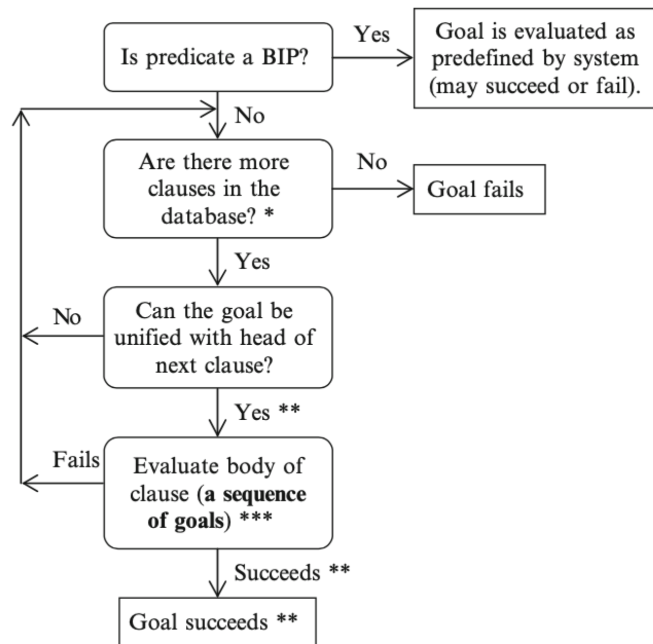


Figura 3.6: Evaluación/Reevaluación de una meta.

3.6. Eliminación de variables comunes

Al unificar una meta con la cabeza de una cláusula, existe una complicación importante, que todos los ejemplos de este capítulo hasta ahora han sido cuidadosamente diseñados para evitar: ¿qué sucede si la meta y la cabeza tienen una o más variables en común?

Supongamos que la meta es **mypred(tuesday,likes(Z,Y),X)** y que la cabeza es **mypred(X,Y,Z)**. Las variables *X*, *Y* y *Z* de la meta parecen ser las mismas que las variables *X*, *Y* y *Z* de la cabeza, pero en realidad no hay conexión entre ellas. La cláusula de la que **mypred(X,Y,Z)** es la cabeza puede ser, por ejemplo,

```
mypred(X,Y,Z):-pred2(X,Q),pred3(Q,Y,Z).
```

Las variables en esta regla son solo 'marcadores de posición'. Se pueden reemplazar consistentemente por cualquier otra variable sin ningún cambio en el significado de la cláusula, como se explica en el Capítulo 2, por lo que no sería sensato considerar que *X*, *Y* y *Z* en la cláusula son las mismas variables que en la meta. **mypred(tuesday,likes(Z,Y),X)**.

Antes de intentar unificar la meta y la cabeza de la cláusula, primero es necesario reescribir la cláusula para garantizar que no tenga variables en común con la meta. Para ser precisos, la cláusula no debe tener ninguna variable en común con ninguno de las metas en la secuencia de la cual forma parte la meta actualmente en consideración.

Prolog reemplaza automáticamente las variables *X*, *Y* y *Z* en la cláusula sistemáticamente por otras variables que no aparecen en la secuencia de metas (o en cualquier otra parte de la cláusula). Por ejemplo, pueden ser reemplazados por *X1*, *Y1* y *Z1* .

```
mypred(X1,Y1,Z1):-pred2(X1,Q),pred3(Q,Y1,Z1).
```

Luego de esta reescritura solo queda unificar la cabeza **mypred(X1,Y1,Z1)** y la meta **mypred(tuesday,likes(Z,Y),X)**, que no tienen ninguna variable en común. La unificación tiene éxito.

```
?- mypred(tuesday,likes(Z,Y),X).

mypred(X1,Y1,Z1):-pred2(X1,Q),pred3(Q,Y1,Z1).
```

Tiene éxito con la variable *X1* vinculada al átomo tuesday, la variable *Y1* vinculada al término compuesto **likes(Z,Y)** y las variables *Z1* y *X* vinculadas entre sí. Las variables *Y* y *Z* no están vinculadas.

3.7. Una Nota sobre la Programación Declarativa

De este capítulo queda claro que el orden en que aparecen las cláusulas que definen un predicado en la base de datos y el orden de las metas en el cuerpo de una regla son de vital importancia al evaluar la consulta de un usuario.

Es parte de la filosofía de la programación lógica que los programas deben escribirse para minimizar el efecto de estos dos factores en la medida de lo posible. Los programas que lo hacen se denominan *total o parcialmente declarativos*.

Un ejemplo de un programa totalmente declarativo es el siguiente, basado en el Programa Animals 2 del Capítulo 2.

```
dog(fido). dog(rover). dog(jane). dog(tom). dog(fred). dog(henry).

cat(bill). cat(steve). cat(mary). cat(harry).

large(rover). large(william). large(martin).
large(tom). large(steve).
large(jim). large(mike).

large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

La consulta

?- large_animal(X).

mediante retroceso producirá tres posibles valores de X

X = rover ;
X = tom ;
X = steve ;
false.

Reorganizar las cláusulas en el programa en cualquier orden producirá las mismas tres respuestas pero posiblemente en un orden diferente (¡inténtelo!).

Reorganizar el orden de las metas en los cuerpos de las dos reglas que definen **large_animal/1**, por ejemplo

```
large_animal(X):- large(X),dog(X).
large_animal(Z):- large(Z),cat(Z).
```

también dará las mismas tres respuestas.

A menudo es muy difícil o imposible definir un predicado de tal manera que el orden de las metas en el cuerpo de cada regla sea irrelevante, especialmente cuando se trata de predicados interconstruidos como **write/1**.

Sin embargo, con un poco de esfuerzo a menudo es posible escribir las cláusulas que definen un predicado de tal manera que si se cambiara el orden, las respuestas a cualquier consulta (incluidas las producidas por retroceso) serían las mismas. Por ejemplo, si deseamos probar si un número es positivo, negativo o cero, podríamos definir un predicado **test/1** como este

```
test(X):-X>0,write(positive),nl.
test(0):-write(zero),nl.
test(X):-write(negative),nl.
```

Esto se basa en que la tercera cláusula sólo se alcanza cuando el valor de X es negativo. Una forma más declarativa (y mejor) de definir **test/1** sería

```
test(X):-X>0,write(positive),nl.
test(0):-write(zero),nl.
test(X):-X<0,write(negative),nl.
```

donde se hace explícita la prueba de que X es negativo en la tercera cláusula.

No sólo se considera un buen estilo de programación en Prolog hacer que los programas sean lo más declarativos posible, sino que puede reducir en gran medida la probabilidad de cometer errores que son difíciles de detectar, especialmente cuando se utiliza el retroceso. El capítulo **??** da algunos ejemplos de esto.

3.8. Nota Importante sobre el Retroceso Controlado por el Usuario

El uso del retroceso ‘controlado por el usuario’ para encontrar más soluciones a una consulta se ilustró en capítulos anteriores y se explicó en detalle anteriormente en este capítulo.

El uso del retroceso ‘detrás de escena’ es fundamental para cualquier sistema Prolog y el retroceso controlado por el usuario ciertamente puede ser valioso en algunas situaciones. Sin embargo, hay otras situaciones en las que no se debe utilizar esta herramienta.

Para muchos de los ejemplos dados en este libro, ejecutar una consulta produce una solución después de la cual el sistema se detiene para permitir que el usuario intente retroceder. En algunos casos, no se pueden encontrar más soluciones y si el usuario ingresa un carácter de punto y coma, el sistema simplemente responderá

false.

En otros casos, el sistema intentará encontrar una solución alternativa aunque no sea posible otra solución significativa, como encontrar el mayor de dos números o combinar el contenido de dos archivos de texto en un solo archivo. A menudo, el efecto de hacerlo será producir una salida que es ‘misteriosa’ u obviamente incorrecta. En algunos casos, el sistema entrará en un bucle infinito y/o se bloqueará a medida que se llene la memoria disponible.

La solución a estos problemas es simplemente que el usuario presione la tecla ’retorno’ para suprimir el retroceso en todas las ocasiones, excepto cuando haya una buena razón para creer que puede haber soluciones adicionales disponibles. Esa es la política seguida en este libro, generalmente sin llamar la atención sobre ella.

Los problemas relacionados con el retroceso y cómo tratarlos se discutirán más adelante en el Capítulo **??**.

Resumen del capítulo

Este capítulo demuestra cómo Prolog usa la unificación para hacer coincidir las metas con las cabezas de las cláusulas y cómo usa la combinación de unificación y retroceso para evaluar las metas ingresadas por el usuario y encontrar múltiples soluciones si es necesario. El capítulo termina con una advertencia sobre el uso del retroceso ’controlado por el usuario’.

Ejercicios Prácticos 3

El programa a continuación es una variante del programa de relaciones familiares utilizado en la Sección 3.4. Como antes, [M1] etc. son etiquetas añadidas para facilitar la referencia a las cláusulas.

```
[M1] mother(ann,henry).
[M2] mother(ann,mary).
[M3] mother(jane,mark).
[M4] mother(jane,francis).
[M5] mother(annette,jonathan).
[M6] mother(mary,bill).
[M7] mother(janice,louise).
[M8] mother(lucy,janet).
[M9] mother(louise,caroline).
[M10] mother(caroline,david).
[M11] mother(caroline,janet).
[F1] father(henry,jonathan).
[F2] father(john,mary).
[F3] father(francis,william).
[F4] father(francis,louise).
[F5] father(john,mark).
[F6] father(gavin,lucy).
[F7] father(john,francis).
[P1] parent(victoria,george).
[P2] parent(victoria,edward).
[P3] parent(X,Y):-mother(X,Y).
[P4] parent(X,Y):-father(X,Y).
[P5] parent(elizabeth,charles).
[P6] parent(elizabeth,andrew).
[A1] ancestor(X,Y):-parent(X,Y).
[A2] ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
```

El cambio más importante es la adición de dos cláusulas que definen el predicado **ancestro/2**. La cláusula [A1] simplemente establece que X es un antepasado de Y si X es un padre de Y. La cláusula [A2] es una definición recursiva de relaciones de antepasados más distantes que se puede leer como 'X es el antepasado de Y si hay algún persona Z tal que X es el padre de Z y Z es el antepasado de Y'.

1. Amplíe el programa anterior ideando reglas para definir cada uno de los siguientes. Cargue su programa extendido y pruébelo.

   a) child_of(A,B)

   b) grandfather_of(A,B)

   c) grandmother_of(A,B)

   d) great_grandfather_of(A,B)

2. Construya una secuencia de diagramas similares a los de la Sección 3.4 para mostrar la secuencia de eventos cuando el sistema Prolog intenta satisfacer la meta.

3. Dada la siguiente consulta

   ?-ancestor(louise,Desc).

   Encuentre (retrocediendo) a las dos primeras personas que el sistema Prolog identificará como descendientes de louise.

   Prediga la salida que se producirá si el usuario obliga repetidamente al sistema a retroceder. Verifique su predicción cargando el programa y probándolo.

Capítulo 4

Operadores y Aritmética

Objetivos del Capítulo

Después de leer este capítulo, debería ser capaz de:

- Convertir predicados binarios y unitarios en operadores
- Evaluar expresiones aritméticas y comparar sus valores
- Probar por igualdad términos y expresiones aritméticas
- Utilizar los operadores 'no' y de disyunción.

4.1. Operadores

Hasta ahora, la notación utilizada para los predicados en este libro es la estándar de un funtor seguido de una serie de argumentos entre paréntesis, por ejemplo **likes(john, mary)**.

Como alternativa, cualquier predicado definido por el usuario con dos argumentos (un predicado binario) se puede convertir en un *operador infijo*. Esto permite escribir el funtor (nombre del predicado) entre los dos argumentos sin paréntesis, por ejemplo.

john likes mary

Algunos usuarios de Prolog pueden encontrar esto más fácil de leer. Otros pueden preferir la notación estándar.

Cualquier predicado definido por el usuario con un argumento (un predicado unario) se puede convertir en un *operador prefijo*. Esto permite escribir el funtor antes del argumento sin paréntesis, por ejemplo.

isa_dog fred

en lugar de

isa_dog(fred)

Alternativamente, un predicado unario se puede convertir en un *operador posfijo*. Esto permite que el funtor se escriba después del argumento, por ejemplo.

fred isa_dog

La notación operador también se puede usar con reglas para facilitar la lectura. Algunas personas pueden encontrar una regla como

```
likes(john,X):-is_female(X),owns(X,Y),isa_cat(Y).
```

más fácil de entender si se escribe como

```
john likes X:- X is_female, X owns Y, Y isa_cat.
```

La notación estándar de "funtor y argumentos" entre paréntesis, por ejemplo, **likes(john,X)** todavía se puede usar con operadores si se prefiere. También se permite la notación 'mixta', por ejemplo, si **likes/2**, **is_female/1**, **owns/2** and **isa_cat/1** son todos operadores

```
likes(john,X):-is_female(X),X owns Y,isa_cat(Y).
```

es una forma válida de la regla anterior.

Cualquier predicado definido por el usuario con uno o dos argumentos se puede convertir en un operador ingresando una meta usando el predicado **op/3** en el indicador del sistema. Este predicado toma tres argumentos, por ejemplo

```
?-op(150,xfy,likes).
```

El primer argumento es la 'precedencia del operador', que es un número entero de 0 en adelante. El rango de números utilizados depende de la implementación particular. Cuanto menor sea el número, mayor será la precedencia. Los valores de precedencia de operadores se utilizan para determinar el orden en que se aplicarán los operadores cuando se utilice más de uno en un término.

El uso práctico más importante de esto es para los operadores que se usan para la aritmética, como se explicará más adelante. En la mayoría de los demás casos será suficiente usar un valor arbitrario como 150.

El segundo argumento normalmente debería ser uno de los siguientes átomos:

- **xfx**, **xfy** o **yfx**, lo que significa que el predicado es binario y debe convertirse en un operador infijo

- **fx** o **fy**, lo que significa que el predicado es unario y debe convertirse en un operador prefijo

- **xf** o **yf**, lo que significa que el predicado es unario y debe convertirse en un operador sufijo.

(La diferencia entre las alternativas disponibles para cada tipo se explicará en la Sección **??**.)

El tercer argumento especifica el nombre del predicado que se va a convertir en un operador.

Un predicado también se puede convertir en un operador colocando una línea como

?-op(150,xfy,likes).

en un archivo de programa Prolog que se cargará mediante consulta. *Tenga en cuenta que se debe incluir el indicador (los dos caracteres ?-).* Cuando se usa una meta de esta manera, la línea completa se conoce como directiva. En este caso, la directiva debe colocarse en el archivo antes de la primera cláusula que usa el operador likes.

Si un archivo de programa que contiene

```
?-op(150,xfy,likes).
?-op(150,xf,is_female).
?-op(150,xf,isa_cat).
?-op(150,xfy,owns).

john likes X:- X is_female, X owns Y, Y isa_cat.
is_female(mary).
owns(mary,fido).
isa_cat(fido).
```

se carga mediante **consult** algunas consultas posibles son:

?- john likes mary.
true.

?- john likes X.
X = mary.

?- X likes mary.
X = john.

?- X likes Y.
X = john,
Y = mary.

?- is_female(X).
X = mary.

Varios predicados interconstruidos se han predefinido como operadores. Estos incluyen operadores relacionales para comparar valores numéricos, incluidos < que denota 'menor que' y > que denota 'mayor que'.

Así, los siguientes son términos válidos, que pueden ser incluidos en el cuerpo de una regla:

X>4
Y<Z
A=B

La notación entre paréntesis también se puede usar con predicados interconstruidos que se definen como operadores, por ejemplo >**(X,4)** en lugar de **X**>**4**.

4.2. Aritmética

Aunque los ejemplos utilizados en los capítulos anteriores de este libro no son numéricos (animales que son mamíferos, etc.), Prolog también brinda facilidades para hacer aritmética utilizando una notación similar a la que ya será familiar para muchos usuarios del álgebra básica.

Esto se logra utilizando el predicado incorconstruido **is/2**, que está predefinido como un operador infijo y, por lo tanto, se escribe entre sus dos argumentos.

La forma más común de usar **is/2** es donde el primer argumento es una variable independiente. Evaluar la meta **X is -6.5** hará que X esté vinculada al número -6.5 y la meta tenga éxito.

El segundo argumento puede ser un número o una expresión aritmética, por ejemplo.

X is 6*Y+Z-3.2+P-Q/4 (* denota multiplicación)

Cualquier variable que aparezca en una expresión aritmética ya debe estar vinculada (como resultado de evaluar una meta anterior) y sus valores deben ser numéricos. Siempre que lo sean, la meta siempre tendrá éxito y la variable que forma el primer argumento estará vinculada al valor de la expresión aritmética. De lo contrario, aparecerá un mensaje de error.

```
?- X is 10.5+4.7*2.
X = 19.9

?- Y is 10,Z is Y+1.
Y = 10,
Z = 11
```

Los símbolos como $+ - * /$ en expresiones aritméticas son un tipo especial de operador infijo conocido como *operadores aritméticos*. A diferencia de los operadores que se usan en otras partes de Prolog, no son predicados sino *funciones* que devuelven un valor numérico.

Además de números, variables y operadores, las expresiones aritméticas pueden incluir *funciones aritméticas*, escritas con sus argumentos entre paréntesis (es decir, no como operadores). Al igual que los operadores aritméticos, estos devuelven valores numéricos, por ejemplo, para encontrar la raíz cuadrada de 36:

?- X is sqrt(36).
X=6

El operador aritmético - puede usarse no sólo como un operador infijo binario para indicar la diferencia de dos valores numéricos, por ejemplo **X-6**, también se puede usar como un operador unario prefijo para denotar el negativo de un valor numérico, por ejemplo.

?- X is 10,Y is -X-2.
X = 10,
Y = -12

La siguiente tabla muestra algunos de los operadores aritméticos y funciones aritméticas disponibles en Prolog.

X+Y	La suma de X y Y
X-Y	La diferencia de X y Y
X*Y	El producto de de X y Y
X/Y	La división de X entre Y
X//Y	El cociente de X entre Y
X mod Y	El residuo de X entre Y
X ˆ Y	El valor de X a la potencia Y
-X	El negativo de X
abs(X)	El valor absoluto de X
sin(X)	El seno de X (X medido en radianes)
cos(X)	El coseno de X (X medido en radianes)
max(X,Y)	El máximo entre X y Y
round(X)	El valor de X redondeado al entero más cercano
sqrt(X)	La raíz cuadrada de X

Ejemplo

?- X is 30,Y is 5,Z is X+Y+X*Y.
X = 30,
Y = 5,
Z = 185.

Aunque el predicado **is** normalmente se usa de la forma descrita aquí, el primer argumento también puede ser un número o una variable vinculada con un valor numérico. En este caso, se calculan los valores numéricos de los dos argumentos. La meta tiene éxito si estos son iguales. Si no, falla.

?- X is 7,X is 6+1.
X = 7

?- 10 is 7+13-11+9.
false.

?- 18 is 7+13-11+9.
true.

Unificación

La descripción anterior se puede simplificar diciendo que se evalúa el segundo argumento del operador **is/2** y luego se unifica este valor con el primer argumento. Esto ilustra la flexibilidad del concepto de unificación.

1. Si el primer argumento es una variable no vinculada, se vincula al valor del segundo argumento (como efecto secundario) y la meta tiene éxito.

2. Si el primer argumento es un número, o una variable vinculada con un valor numérico, se compara con el valor del segundo argumento. Si son iguales, la meta es exitosa; de lo contrario, falla.

Si el primer argumento es un átomo, un término compuesto, una lista o una variable vinculada a uno de estos (nada de lo cual debería suceder), el resultado depende de la implementación. Es probable que ocurra un error.

Tenga en cuenta que una meta como **X is X+1** siempre fallará, ya sea que X esté vinculada o no.

?- X is 10,X is X+1.
false.

Aumentar un valor en uno requiere un enfoque diferente.

```
/* Versión incorrecta */
increase(N):-N is N+1.
```

?- increase(4).
false.

```
/*Versión correcta*/
increase(N,M):-M is N+1.
```

?- increase(4,X).
X = 5

Precedencia de Operadores en Expresiones Aritméticas

Cuando hay más de un operador en una expresión aritmética, por ejemplo **A+B*C-D**, Prolog necesita un medio para decidir el orden en el que se aplicarán los operadores.

Para los operadores básicos como + - * y / es muy deseable que este sea el orden 'matemático' habitual, es decir, la expresión **A+B*C-D** debe interpretarse como 'calcular el producto de B y C, sumarlo a A y luego restar D', no como 'sumar A y B, luego multiplicar por C y restar D'. Prolog logra esto dando a cada operador un *valor de precedencia* numérica.

Los operadores con una precedencia relativamente alta, como * y /, se aplican antes que los de menor precedencia, como + y -. Los operadores con la misma precedencia (por ejemplo, + y -, * y /) se aplican de izquierda a derecha. El efecto es dar a una expresión como **A+B*C-D** el significado que un usuario familiarizado con el álgebra esperaría que tuviera, es decir, **A+(B*C)-D**.

Si se requiere un orden diferente de evaluación, esto se puede lograr mediante el uso de paréntesis, por ejemplo, **X is (A+B)*(C-D)**. Las expresiones entre paréntesis siempre se evalúan primero.

Operadores Relacionales

Los operadores infijos =:=, =\=, >, >=, < y =< son un tipo especial conocido como *operadores relacionales*. Se utilizan para comparar el valor de dos expresiones aritméticas.

La meta tiene éxito si el valor de la primera expresión es igual, distinto, mayor, mayor o igual, menor, menor o igual que el valor de la segunda expresión, respectivamente. Ambos argumentos deben ser números, variables

vinculadas o expresiones aritméticas (en las que las variables están vinculadas a valores numéricos).

?- 88+15-3=:=110-5*2.
true.

?- 100=\=99.
true.

4.3. Operadores de Igualdad

Hay tres tipos de operadores relacionales disponibles en Prolog para probar la igualdad y la desigualdad. El primer tipo se utiliza para comparar los valores de expresiones aritméticas. Los otros dos tipos se utilizan para comparar términos.

Igualdad de Expresiones Aritméticas =:=

E1=:=**E2** tiene éxito si las expresiones aritméticas *E1* y *E2* dan como resultado el mismo valor.

?- 6+4=:=6*3-8.
true.

?- sqrt(36)+4=:=5*11-45.
true.

Para verificar si un número entero es impar o par, podemos usar el predicado **checkeven/1** definido a continuación.

```
checkeven(N):-M is N//2,N=:=2*M.
```

?- checkeven(12).
true.

?- checkeven(23).
false.

?- checkeven(-11).

false.

?- checkeven(-30).
true.

El operador cociente de enteros // divide su primer argumento por su segundo y trunca el resultado al entero más cercano entre él y cero. Entonces **12//2** es 6, **23//2** es 11, **-11//2** es -5 y **-30//2** es -15. Dividir un número entero por 2 usando // y multiplicarlo por 2 nuevamente dará el número entero original si es par, pero no en caso contrario.

Desigualdad de Expresiones Aritméticas =\=

E1=\=**E2** tiene éxito si las expresiones aritméticas *E1* y *E2* no se evalúan al mismo valor

?- 10=\=8+3.
true.

Términos Idénticos ==

Ambos argumentos del operador infijo == deben ser términos. La meta **Term1==Term2** tiene éxito si y sólo si *Term1* es idéntico a *Term2*. Cualquier variable utilizada en los términos puede o no estar vinculada, pero ninguna variable queda vinculada como resultado de la evaluación de la meta.

?- likes(X,prolog)==likes(X,prolog).
true.

?- likes(X,prolog)==likes(Y,prolog).
false.
(*X* y *Y* son variables diferentes)

?- X is 10,pred1(X)==pred1(10).
X = 10

?- X==0.
false.

?- 6+4==3+7.
false.

El valor de una expresión aritmética sólo se evalúa cuando se usa con el operador **is/2**. Aquí **6+4** es simplemente un término con el funtor + y los argumentos **6** y **4**. Esto es completamente diferente del término **3+7**.

Términos No Idénticos \ ==

Term1\ ==Term2 comprueba si *Term1* no es idéntico a *Term2*. La meta tiene éxito si **Term1==Term2** falla. De lo contrario falla.

?- pred1(X)\ ==pred1(Y).
true.

La salida significa que tanto *X* como *Y* no están vinculadas y que son variables diferentes.

Términos Idénticos con Unificación =

El operador de igualdad de términos = es similar a == con una diferencia vital (y a menudo muy útil). La meta **Term1=Term2** tiene éxito si los términos *Term1* y *Term2* se unifican, es decir, hay alguna forma de vincular variables a valores que harían que los términos fueran idénticos. Si la meta tiene éxito, dicha vinculación realmente se lleva a cabo. La unificación se analiza en detalle en el Capítulo 3.

?- pred1(X)=pred1(10).
X = 10

La variable *X* se vincula con 10, lo que hace que los dos términos sean idénticos.

?- likes(X,prolog)=likes(john,Y).
X = john ,
Y = prolog

Vinculando *X* al átomo **john** y *Y* al átomo **prolog** hace que los dos términos sean idénticos.

?- X=0,X=:=0.
X=0

X=0 hace que *X* se vincule a 0. La meta **X=:=0** tiene éxito, lo que confirma que *X* ahora tiene el valor cero.

?- 6+4=3+7.
false.

Por la razón explicada en ==.

?- 6+X=6+3.
X=3

Vinculando X a 3 hace que los dos términos sean idénticos. Ambos son **6+3**, no el número 9.

?- likes(X,prolog)=likes(Y,prolog).
X = Y.

Vinculando X y Y hace que los términos sean idénticos.

?- likes(X,prolog)=likes(Y,ada).
false.

Ninguna unificación puede hacer que los átomos **prolog** y **ada** sean idénticos.

No Unificación entre Dos Términos \ =

La meta **Term1**\ =**Term2** tiene éxito si **Term1=Term2** falla, es decir, los dos términos no se pueden unificar. De lo contrario falla.

?- 6+4 \ = **3+7.**
true.

?- likes(X,prolog) \ = **likes(john,Y).**
false.

Porque vinculando X a **john** y Y a **prolog** hará que los términos sean idénticos.

?- likes(X,prolog) \ = **likes(X,ada).**
true.

4.4. Operadores Lógicos

Esta sección ofrece una breve descripción de dos operadores que toman argumentos que son términos de llamada, es decir, términos que pueden considerarse metas.

El operador *not*

Nota: en algunas versiones de Prolog **not/1** se define como un operador. En otros, se define simplemente como un predicado con un argumento. Si su sistema Prolog es uno de estos últimos, puede convertirlo en un operador mediante una directiva como

?-op(1000,fy,not).

Para los propósitos de este libro, trataremos **not/1** como un operador prefijo.

El operador prefijo **not/1** puede colocarse antes de cualquier meta para dar su negación. La meta negada tiene éxito si la meta original falla y falla si la meta original tiene éxito.

Los siguientes ejemplos ilustran el uso de **not/1**. Se supone que la base de datos contiene la cláusula única

```
dog(fido).
```

?- not dog(fido).
false.

?- dog(fred).
false.

?- not dog(fred).
true.

?- X=0,X is 0.
X=0

?- X=0,not X is 0.
false.

El Operador Disyunción

El operador disyunción **;/2** (escrito como un carácter de punto y coma) se usa para representar ‘o’. Es un operador infijo que toma dos argumentos, ambas metas. **Goal1;Goal2** tiene éxito si **Goal1** o **Goal2** tienen éxito.

?- 6<3;7 is 5+2.
true.

?- 6*6=:=36;10=8+3.
true.

4.5. Más sobre Precedencia de Operadores

El predicado **op/3** se introdujo en la Sección 4.1. Para los propósitos de este libro, se supone que los operadores presentados en este capítulo son declarados automáticamente por el sistema Prolog con los siguientes valores:

Precedencia	Tipo	Operador(es)
1100	xfy	;
1000	fy	not
700	xfx	is, <, >, =<, >=, =:=, =\=, =, \ =, ==, \ ==
500	yfx	+, -
400	yfx	*, /, //
200	xfy	^
200	fy	+, -

Estos valores de precedencia varían de un sistema Prolog a otro al igual que el valor de precedencia más alto permitido. El valor de precedencia más bajo es siempre cero. De manera bastante confusa, cuanto menor sea el valor de precedencia, mayor será la precedencia.

En la Sección 4.2 se indicó que “los operadores con una precedencia relativamente alta, como * y /, se aplican antes que aquellos con una precedencia más baja, como + y -. Los operadores con la misma precedencia (por ejemplo, + y -, * y /) se aplican de izquierda a derecha”.

En la tabla podemos ver que los valores de precedencia de los operadores infijos * y / son ambos 400 y el valor de precedencia de + y - (cuando se usan como operadores infijos) es 500. Cuando el sistema Prolog evalúa la

expresión **A+B*C-D**, el operador de precedencia más alta (valor más bajo) es *, por lo que el producto de B y C se calcula primero.

A continuación, existe la opción de aplicar el operador + o el operador -. Como ambos operadores tienen una precedencia de 500, se aplican de izquierda a derecha, por lo que el valor de A se suma al valor de **B*C** y luego se resta el valor de D.

Otros principios relacionados con la precedencia son:

- Un término encerrado entre paréntesis tiene precedencia cero.
- La precedencia de un término es cero, a menos que su funtor principal sea un operador.
- La precedencia de un término cuyo funtor principal es un operador es la precedencia del operador.

Esto explica lo que sucede cuando el sistema Prolog evalúa la expresión **(A+B)*C-D**. La expresión entre paréntesis **(A+B)** tiene precedencia cero, la precedencia más alta posible, por lo que se evalúa primero. Luego, el valor resultante se multiplica por C, ya que * tiene el siguiente valor de precedencia más alto (400) y luego se le resta D al resultado, ya que el operador restante tiene la precedencia más baja (500).

Los operadores + y - aparecen dos veces en la tabla. Las versiones con infijo, como en **A+B** o **A-B**, tienen precedencia 400 y las versiones con prefijo, como en **-A*B** o **+A*B**, tienen precedencia 200. Esto es útil ya que garantiza que la expresión **-A+B** se interprete como **(-A)+B** en lugar de **-(A+B)**, lo que parece deseable.

```
?- X is 10, Y is 25, Z is -X+Y.
X = 10,
Y = 25,
Z = 15.
```

Como se señaló anteriormente, hay tres tipos de operadores infijos **xfx**, **xfy** y **yfx**. También hay dos tipos de operadores de prefijo **fx** y **fy** y dos tipos de operadores de sufijo **xf** y **yf**.

En todos los casos, la **f** indica la posición del operador cuando se usa en una expresión (**f** significa funtor). Para operadores infijos, está entre los dos argumentos, etc. Las letras **x** y **y** indican la posición de los argumentos a uno o ambos lados del operador.

La diferencia entre **x** y **y** es la precedencia que deben tener los argumentos del operador:

- **x** denota un argumento que tiene una precedencia estrictamente menor que la del operador
- **y** denota un argumento que tiene una precedencia menor o igual que la del operador.

La diferencia entre estos a menudo no es importante, pero en algunos casos es importante. Si en la Sección 4.4 el operador **not** hubiera sido declarado erróneamente como de tipo **fx**, y la base de datos contiene la cláusula única

```
dog(fido).
```

entonces el uso de not todavía sería posible

?- dog(fido).
true.

?- not dog(fido).
false.

Sin embargo, una doble negación como

?- not not dog(fido).

habría generado un error de sintaxis.

Con **not** declarado correctamente como del tipo **fy**, se permite una secuencia de dos o más operadores **not**:

?- dog(fido).
true.

?- not dog(fido).
false.

?- not not dog(fido).
true.

?- not not not dog(fido).

false.

?- not not not not dog(fido).
true.

Resumen del capítulo

Este capítulo introduce la notación de operadores para predicados y describe los operadores proporcionados para evaluar y comparar los valores de expresiones aritméticas, para probar la igualdad de expresiones aritméticas o términos y para probar la negación de una meta o la disyunción de dos metas.

Ejercicios Prácticos 4

1. Este programa está basado en el Programa Animals 3, dado en el Capítulo 2.

```
dog(fido). large(fido).
cat(mary). large(mary).
dog(rover). small(rover).
cat(jane). small(jane).
dog(tom). small(tom).
cat(harry).
dog(fred). large(fred).
cat(henry). large(henry).
cat(bill).
cat(steve). large(steve).
large(jim).
large(mike).
large_dog(X):- dog(X),large(X).
small_animal(A):- dog(A),small(A).
small_animal(B):- cat(B),small(B).
chases(X,Y):-
        large_dog(X),small_animal(Y),
        write(X),write(' chases '),write(Y),nl.
```

Convierta los siete predicados utilizados en forma de operador y pruebe su programa revisado. La salida debe ser la misma que la salida del programa anterior. Incluya directivas para definir los operadores en su programa.

2. Defina y pruebe un predicado que tome dos argumentos, ambos números, y calcule y genere los siguientes valores:

   a) su promedio

   b) la raíz cuadrada de su producto

   c) el mayor de *a*) y *b*).

Capítulo 5

Entrada y Salida

Objetivos del Capítulo

Después de leer este capítulo, debería ser capaz de:

- Usar los predicados interconstruidos que leen y escriben en la terminal del usuario (teclado y pantalla) o en un archivo, tanto término por término como carácter por carácter en sus propios programas.
- Usar valores ASCII para manipular cadenas de caracteres.

5.1. Introducción

Prolog tiene facilidades para permitir la entrada y salida de términos o caracteres. Usar términos es más simple y se describirá primero. Inicialmente, se supondrá que todas las salidas son a la pantalla del usuario y todas las entradas son desde el teclado del usuario.

La entrada y salida usando archivos externos, por ejemplo, en un disco duro o CD-ROM, se describirá en la Sección **??** más adelante. Tenga en cuenta que, al igual que muchos otros predicados interconstruidos, los de entrada y salida descritos en este capítulo no se pueden volver a satisfacer, es decir, siempre fallan si se hace retroceso.

5.2. Escritura de Términos

El principal predicado interconstruido proporcionado para la salida de términos es **write/1**, que ya se ha utilizado muchas veces en este libro.

El predicado **write/1** toma un único argumento, que debe ser un término de Prolog válido. La evaluación del predicado hace que el término se escriba en el *flujo de salida actual*, que de forma predeterminada es la pantalla del usuario. (El significado del *flujo de salida actual* se explicará en las Secciones **??** y **??**. En este momento, puede interpretarse simplemente como la pantalla del usuario).

El predicado interconstruido **nl/0** también se ha utilizado muchas veces anteriormente en este libro. No necesita argumentos. La evaluación de una meta **nl** hace que se envíe una nueva línea al flujo de salida actual.

Ejemplos

```
?- write(26),nl. 26
true.

?- write('a string of characters'),nl.
a string of characters
true.

?- write([a,b,c,d,[x,y,z]]),nl.
[a,b,c,d,[x,y,z]]
true.

?- write(mypred(a,b,c)),nl.
mypred(a,b,c)
true.

?- write('Example of use of nl'),nl,nl,write('end of example'),nl.
Example of use of nl

end of example
true.
```

Tenga en cuenta que los átomos que deben entrecomillarse cuando se ingresan (por ejemplo, 'Paul', 'hola mundo') no se entrecomillan cuando se escriben usando **write**. Si es importante generar las comillas, se puede usar el predicado **writeq/1**. Es idéntico a **write/1** excepto que los átomos que necesitan comillas para la entrada se escriben entre comillas (otros átomos no).

?- writeq('a string of characters'),nl.
'a string of characters'
true.

?-writeq(dog),nl.
dog
true.

?- writeq('dog'),nl.
dog
true.

5.3. Lectura de Términos

El predicado interconstruido **read/1** se proporciona para leer términos. Toma un solo argumento, que debe ser una variable.

Al evaluarlo, se lee el siguiente término del *flujo de entrada actual*, que de forma predeterminada es el teclado del usuario. (El significado de *flujo de entrada actual* se explicará en las Secciones **??** y **??**. En este momento, puede interpretarse simplemente como el teclado del usuario).

En el flujo de entrada, el término debe ir seguido de un punto ('.') y al menos un *carácter de espacio en blanco*, como un espacio o una nueva línea. Los caracteres de punto y espacio en blanco se leen pero no se consideran parte del término.

Tenga en cuenta que para la entrada desde el teclado, generalmente se mostrará un 'mensaje' como | : (una barra vertical seguida de dos puntos) para indicar que se requiere la entrada del usuario. El término de entrada debe ir seguido de un punto. Probablemente también será necesario presionar la tecla 'retorno' antes de que Prolog acepte la entrada.

Cuando se evalúa una meta **read**, el término de entrada se *unifica* con la variable que se dio como argumento. Si la variable no está vinculada (que suele ser el caso), se vincula con el valor de entrada.

?- read(X).
| :jim.
X = jim

```
?- read(X).
| :26.
X = 26

?- read(X).
| :mypred(a,b,c).
X = mypred(a,b,c)

?- read(Z).
| : [a,b,mypred(p,q,r),[z,y,x]].
Z = [a,b,mypred(p,q,r),[z,y,x]]

?- read(Y).
| : 'a string of characters'.
Y = 'a string of characters'
```

Si la variable que se dio como argumento ya está vinculada (lo que para la mayoría de los usuarios es mucho más probable que ocurra por error que por diseño), la meta tiene éxito si y sólo si el término de entrada es idéntico al valor previamente vinculado.

```
?- X=fred,read(X).
| :jim.
false.

?- X=fred,read(X).
| :fred.
X = fred
```

5.4. Entrada y Salida Usando Caracteres

Aunque la entrada y salida de términos es sencilla, el uso de comillas y puntos puede ser engorroso y no siempre es adecuado. Por ejemplo, sería tedioso definir un predicado (usando **read**) que leyera una serie de caracteres del teclado y contara el número de vocales. Un enfoque mucho mejor para problemas de este tipo es ingresar un carácter a la vez. Para hacer esto, primero es necesario conocer el *valor ASCII* de un carácter.

9	tab	40	(	59	;	94	∧
10	end of record	41	)	60	<	95	_
32	space	42	*	61	=	96	`
33	!	43	+	62	>	97–122	a to z
34	"	44	,	63	?		
35	#	45	-	64	@	123	{
36	$	46	.	65-90	A to Z	124	|
37	%	47	/	91	[	125	}
38	&	48-57	0 to 9	92	\	126	~
39	'	58	:	93	]		

Figura 5.1: Tabla de caracteres ASCII.

Todos los caracteres impresos y muchos caracteres no impresos (como el espacio y el tabulador) tienen un valor ASCII (Código estándar estadounidense para el intercambio de información) correspondiente, que es un número entero de 0 a 255.

La tabla en la Figura 5.1 proporciona los valores ASCII numéricos correspondientes a los principales caracteres imprimibles y algunos otros.

Los caracteres cuyo valor ASCII es menor o igual a 32 se conocen como *caracteres espacio en blanco.*

5.5. Escritura de Caracteres

Los caracteres se escriben utilizando el predicado interconstruido **put/1**. El predicado toma un solo argumento, que debe ser un número de 0 a 255 o una expresión que se evalúe como un número entero en ese rango.

La evaluación de una meta **put** hace que se envíe un solo carácter al flujo de salida actual. Este es el carácter correspondiente al valor numérico (valor ASCII) de su argumento, por ejemplo

```
?- put(97),nl.
a
true.

?- put(122),nl.
z
true.
```

?- put(64),nl.
@
true.

5.6. Lectura de Caracteres

Dos predicados interconstruidos se proporcionan para leer un solo carácter: **get0/1** y **get/1**. El predicado **get0** toma un único argumento, que debe ser una variable. La evaluación de una meta **get0** hace que se lea un carácter del flujo de entrada actual. A continuación, la variable se *unifica* con el valor ASCII de este carácter.

> Tenga en cuenta que para la entrada desde el teclado, generalmente se mostrará un 'indicador' como | : (una barra vertical seguida de dos puntos) para indicar que se requiere la entrada del usuario. Probablemente también será necesario presionar la tecla 'retorno' antes de que Prolog acepte la entrada. Esto también se aplica al predicado **get** que se describe a continuación.

Suponiendo que la variable que se dio como argumento no está vinculada (que suele ser el caso), ésta se vincula con el valor ASCII del carácter de entrada.

?- get0(N).
| : **a**
N = 97

?- get0(N).
| : **Z**
N = 90

?- get0(M)
| :)
M = 41

Si la variable que se dio como argumento ya está vinculada, la meta tiene éxito si y sólo si tiene un valor numérico que es igual al valor ASCII del carácter de entrada.

?- M is 41,get0(M).
| :)
M = 41

?- M is 50,get0(M).
| :)
false.

El predicado **get** toma un único argumento, que debe ser una variable. La evaluación de una meta **get** hace que el siguiente carácter que *no sea un espacio en blanco* (es decir, un carácter con un valor ASCII menor o igual a 32) se lea del flujo de entrada actual. A continuación, la variable se unifica con el valor ASCII de este carácter de la misma forma que para **get0**.

?- get(X).
| : **Z**
X = 90

?- get(M).
| :   **Z**
M = 90

5.7. Uso de Caracteres: Ejemplos

El primer ejemplo muestra cómo leer una serie de caracteres del teclado que terminan con * y generar sus valores ASCII correspondientes uno por línea (para todos los caracteres excepto *).

El predicado readin se define recursivamente. Hace que se lea un solo carácter y que la variable **X** se vincule a su valor ASCII (numérico). La acción tomada (la meta **process(X)**) depende de si **X** tiene o no el valor 42 que significa un carácter *. Si lo ha hecho, se detiene la evaluación de la meta.

De lo contrario, se escribe el valor de **X**, seguido de una nueva línea, seguida de una nueva llamada a readin. Este proceso continúa indefinidamente hasta que se lee un carácter *. (En el siguiente ejemplo, los valores ASCII de los caracteres P, r, o, etc. se muestran correctamente como 80, 114, 111, etc.)

```
readin:-get0(X),process(X).
process(42).
process(X):-X=\=42,write(X),nl,readin.
```

?- readin.
| : **Prolog Example***
80
114
111
108
111
103
32
69
120
97
109
112
108
101
true.

El siguiente ejemplo es una versión extendida del anterior. Esta vez no se escriben los valores ASCII de los caracteres de entrada, sino el *número* de caracteres (excluyendo *). El predicado *count* se define con dos argumentos que se pueden leer como 'el número de caracteres contados hasta ahora' y 'el número total de caracteres antes del *'.

```
go(Total):-count(0,Total).
count(Oldcount,Result):-
        get0(X),process(X,Oldcount,Result).
process(42,Oldcount,Oldcount).
process(X,Oldcount,Result):-
        X=\=42,New is OldcountC1,count(New,Result).
```

?- go(T).
| :**The time has come the walrus said***
T = 33

```
?- go(T).
| :*
T = 0
```