Tutorial básico de programación en Prolog

Indice de contenidos

Introducción

Este tutorial de programación en Prolog constituye la primera entrega de una serie de cursillos orientados a aquellas personas que desconocen la programación declarativa relacional y su lenguaje rey: **Prolog**. Hablamos de programación lógica relacional porque existe toda una gama de lenguajes que siguen este paradigma, si bien, casi todos ellos están basados en Prolog. Quizás, La familia más importante de estos lenguajes sean los denominados **CLP** - Constraint Logic Programming, que son exactamente iguales a Prolog pero con la capacidad adicional de resolver sistemas de ecuaciones.

El conjunto de cursos está organizado de forma que las características más básicas y sencillas se encuentran en este primer tutorial. El resto se adentra en cuestiones avanzadas que raramente se suelen explicar pero cuyo dominio es fundamental para trabajar profesionalmente con Prolog, y para obtener ventajas sobre otros paradigmas de programación.

¿ Es usted escéptico respecto a Prolog ?. El típico tópico muestra este lenguaje como poco eficiente, sin utilidad práctica alguna, complicadísimo de manejar, etc. Sin ánimo de ofender, si Ud. encuentra Prolog imposible de entender, es que Ud. no es un profesional de la informática, porque la verdad es que requiere una cierta formación en lógica matemática y en técnicas de programación. Pero no se desanime, porque otro objetivo de este curso es ayudarle a superar todos los desafios.

En cuanto a la escasa utilidad práctica de Prolog podemos citar:

- Generación de CGI's.
- Acceso a bases de datos desde páginas Web.
- Paralelización automática de programas.
- Programación distribuida y multiagente.
- Sistemas expertos e inteligencia artificial.
- Validación automática de programas.
- Procesamiento de lenguaje natural.
- Prototipado rápido de aplicaciones.
- Bases de datos deductivas.
- Interfacing con otros lenguajes como Java y Tcl/Tk.
- ... (la lista es interminable) ...

En cuanto a la excasa eficiencia hemos de admitir que Prolog es aproximadamente diez veces más lento que el lenguaje C. Pero también hemos de admitir que un programa en Prolog ocupa aproximadamente diez veces menos, en líneas de código y tiempo de desarrollo, que el mismo programa escrito en C. Además las técnicas de optimización de código en Prolog apenas están emergiendo en estos momentos. Algunos experimentos (optimistas) hacen pensar que la velocidad de ejecución de Prolog podría aproximarse a la de C en esta década.

Requisitos

Para hacer unos primeros pinitos en Prolog se necesita unicamente dos cosas: un editor de texto y un entorno de desarrollo Prolog. Como editor de texto resulta altamente recomendable el uso de Emacs. A continuación indicamos algunos links donde puedes descargarte entornos de desarrollo:

- CIAO Prolog.
- SWI Prolog.

Este curso también supone que el lector está familiarizado con:

- La programación imperativa tradicional.
- Tipos abstractos de datos, como listas y árboles.

- Técnicas de programación, como la recursividad.

El entorno de desarrollo Prolog

Prolog es un lenguaje de programación seminterpretado. Su funcionamiento es muy similar a Java. El codigo fuente se compila a un código de byte el cuál se interpreta en una máquina virtual denominada Warren Abstract Machine (comúnmente denominada WAM).
Por eso, un entorno de desarrollo Prolog se compone de:

- **Un compilador**. Transforma el código fuente en código de byte. A diferencia de Java, no existe un standard al respecto. Por eso, el codigo de byte generado por un entorno de desarrollo no tiene por que funcionar en el intérprete de otro entorno.
- **Un intérprete**. Ejecuta el código de byte.
- **Un shell o top-level**. Se trata de una utilidad que permite probar los programas, depurarlos, etc. Su funcionamiento es similar a los interfaces de línea de comando de los sistemas operativos.
- **Una biblioteca de utilidades**. Estas bibliotecas son, en general, muy amplias. Muchos entornos incluyen (afortunadamente) unas bibliotecas standard-ISO que permiten funcionalidades básicas como manipular cadenas, entrada/salida, etc.

Generalmente, los entornos de desarrollo ofrecen extensiones al lenguaje como pueden ser la programación con restricciones, concurrente, orientada a objetos, etc.
Sería injusto no mencionar aquí el entorno de desarrollo más popular: **SICStus Prolog**, si bién, se trata de un entorno de desarrollo comercial (no gratuito).
SICStus, CIAO Prolog, y posibiblemente otros más, ofrecen entornos integrados generalmente basados en Emacs que resultan muy fáciles de usar. CIAO Prolog además ofrece un autodocumentador similar al existente para Java además de un preprocesador de programas. Prácticamente todos ellos son multiplataforma.

Compatibilidad ISO-Prolog

Existe un standard ISO que dicta las típicas normas con respecto a la sintáxis del lenguaje y a las bibliotecas básicas que se deben ofrecer. Actualmente el standard no contempla todos los aspectos del lenguaje, y además, no todos los entornos siguen el standard al pie de la letra. Por eso, programas que funcionan en unos entornos podrían no funcionar en otros, o lo que es peor, funcionar de forma diferente.
Todos los ejemplos que aparecen en este curso siguen el standard ISO-Prolog salvo que se especifique lo contrario. En cualquier caso debe consultar la documentación de su entorno de desarrollo puesto que pueden existir pequeñas variaciones con respecto a su uso.
Los principales investigadores de la tecnología Prolog son los suecos y los españoles. Sin embargo, los españoles no tenemos voto en el comité de estandarización.

Elementos del lenguaje

En esta lección explicaremos como reconocer los diferentes elementos que componen un programa fuente en Prolog. Como observará en breve, Prolog carece de declaraciones en el sentido imperativo: secciones, declaraciones de tipo, declaraciones de variable, declaraciones de procedimientos, etc.

Después de leer está sección deber ser capaz de distinguir variables y términos lógicos entre la "maraña" de caracteres que hay en un programa fuente.

Comentarios

Los comentarios en Prolog se escriben comenzando la línea con un símbolo de porcentaje.
Ejemplo:

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

Variables lógicas

Las várables en Prolog no son variables en el sentido habitual, por eso las llamamos variables lógicas. Se escriben como una secuencia de caracteres alfabéticos comenzando siempre por mayúscula o subrayado. Ejemplos de variables:

```
Variable
_Hola
_
```

Pero no son variables:

```
variable
$Hola
p__
```

El hecho de que los nombres de variables comienzen por mayúscula (o subrayado) evita la necesidad de declarar previamente y de manera explícita las variables, tal y como ocurre en otros lenguajes.

La variable anónima

Sí, sí, existen variables sin nombre, y todas ellas se representan mediante el símbolo de subrayado `_`. Pero cuidado, aunque todas las variables anónimas se escriben igual, son todas **distintas**. Es decir, mientras que dos apariciones de la secuencia de caracteres `Hola` se refieren a la misma variable, dos apariciones de la secuencia `_` se refieren a variables distintas.

Términos

Los términos son el único elemento del lenguaje, es decir, los datos son términos, el código son términos, incluso el propio programa es un término. No obstante, es habitual, llamar término solamente a los datos que maneja un programa.
Un término se compone de un **functor** seguido de cero a N **argumentos** entre paréntesis y separados por comas. Los números enteros o decimales sin restricciones de tamaño también son términos.
Un **functor** (también denominado **átomo**) puede ser:

- Una sucesión de caracteres alfanuméricos comenzando por una letra minúscula.
- Un símbolo de puntuación o secuencia de estos. Las secuencias permitidas varian de un entorno de desarrollo a otro.
- Una sucesión cualquiera de caracteres encerrada entre comillas simples.

Veamos algunos ejemplos de functores:

```
functor
f384p12
'esto es un unico functor, eh!!'
'_functor'
$
+
```

No son functores válidos:

```
_functor
Functor
```

Los **argumentos** de un término pueden ser:

- otro término.
- una várable lógica.

La mejor forma de aprender a escribir términos es mirando algunos ejemplos:

```
termino_cero_ario
1237878837385345.187823787872344434
```

```
t(1)
'mi functor'(17,hola,'otro termino')
f(Variable)
muchos_argumentos(_,_,_,Variable,232,f,g,a)
terminos_anidados(f(g), h(i,j(7)), p(a(b)), j(1,3,2,_))
+(3,4)
$(a,b)
@(12)
```

Operadores

Algunos functores pueden estar declarados como **operadores**, bien de manera predefinida, o bien por el programador. Los operadores simplemente sirven para escribir términos unarios o binarios de una manera más cómoda. Por ejemplo, un functor definido como operador **infijo** es la suma (+). Así, la expresión `a+b` es perfectamente válida, aunque en realidad no es más que el término `+(a,b)`.

Los operadores binarios infijos nos permiten escribir el functor entre los dos argumentos y eliminar los paréntesis.

Los operadores tienen asociada una prioridad. Por ejemplo, la expresión `a+b*c` es en realidad el término `+(a,*(b,c))`. Esto es así porque el operador producto (*) tiene más prioridad que el operador suma (+). Si no fuese así, se trataría del término `*(+(a,b),c)`.

Los operadores también pueden ser unarios y **prefijos**, lo que nos evita escribir los paréntesis del argumento. Por ejemplo, la expresión `-5` es en realidad el término `-(5)`.

Culturilla

- Es posible escribir términos sin argumentos, en tal caso no se escriben los paréntesis (tal como muestra el ejemplo).
- Un término con N argumentos se dice que es **N-ario**, o que tiene **aridad N**.
- Un término que no contiene variables libres se dice que es **cerrado** o **ground** (en inglés).
- Para referirnos a un término con el functor f y A argumentos usamos la notación f/A. Por ejemplo: p(a,b), p(1,f(j)), p(A,_) son todos ejemplos del término **p/2**.
- Dos términos con el mismo functor pero distinta aridad son distintos, por ejemplo p(1) y p(1,2).
- Los números en Prolog no tienen límite de precisión o rango. Estan límitados únicamente por la memoria disponible.

Dando valor a las variables

El mecanismo de unificación

La unificación es el mecanismo mediante el cuál las variables lógicas toman valor en Prolog. El valor que puede tomar una variable consiste en cualquier término, por ejemplo, `j(3)`, `23.2`, `'hola que tal'`, etc. Por eso decimos que los datos que maneja Prolog son términos.

Cuando una variable no tiene valor se dice que está **libre**. Pero una vez que se le asigna valor, éste ya no cambia, por eso se dice que la variable está **ligada**.

Se dice que dos términos unifican cuando existe una posible ligadura (asignación de valor) de las variables tal que ambos términos s on idénticos sustituyendo las variables por dichos valores. Por ejemplo: `a(X,3)` y `a(4,Z)` unifican dando valores a las variables: X vale 4, Z vale 3. Obsérvese que las variables de ambos términos entran en juego.

Por otra parte, no todas las variables estan obligadas a quedar ligadas. Por ejemplo: `h(X)` y `h(Y)` unifican aunque las variables X e Y no quedan ligadas. **No obstante** , ambas variables **permanecen unificadas** entre sí. Si posteriormente ligamos X al valor j(3) (por ejemplo),

entonces automáticamente la variable Y tomará ese mismo valor. Lo que esta ocurriendo es que, al unificar los términos dados, se impone la restricción de que X e Y deben tomar el mismo valor aunque en ese preciso instante no se conozca dicho valor.

La unificación no debe confundirse con la asignación de los lenguajes imperativos puesto que representa la igualdad lógica. Muchas veces unificamos variables con términos directamente y de manera explícita (ya veremos como se hace esto), por ejemplo,`X y 355`. Esto provoca la sensación de que estamos asignando valores a las variables al estilo imperativo.

Para saber si dos términos unifican podemos aplicar las siguientes normas:

- Una variable siempre unifica con un término, quedando ésta ligada a dicho término.
- Dos variables siempre unifican entre sí, además, cuando una de ellas se liga a un término, todas las que unifican se ligan a dicho término.
- Para que dos términos unifiquen, deben tener el mismo functor y la misma aridad. Después se comprueba que los argumentos unifican uno a uno manteniendo las ligaduras que se produzcan en cada uno.
- Si dos términos no unifican, ninguna variable queda ligada.

Ejemplos paradigmáticos

- Una misma variable puede aparecer varias veces en los términos a unificar. Ejemplo: `k(Z,Z) y k(4,H)`. Por culpa del primer argumento, Z se liga al valor 4. Por culpa del segundo argumento, Z y H unifican, pero como Z se liga a un valor, entonces H se liga a ese mismo valor, que es 4.
- Recuerde que una variable no puede ligarse a dos valores distintos. Por ejemplo: `k(Z,Z) y k(4,3)` no unifican, sin embargo `k(Z,Z) y k(5,5)` sí unifican.
- ¿ Sería capaz de decir a que valores se ligan las variables de este ejemplo ? `a(b(j,K),c(X)) y a(b(W,c(X)),c(W))`. Puede estar seguro de que unifican.
- Cuidado con las variables anónimas, recuerde que son todas distintas. Por ejemplo: `k(_,_) y k(3,4)` unifican perfectamente.

Culturilla

- Existe un algoritmo de unificación más preciso que las normas dadas aqui. Además es computable, de otra manera, no estariamos hablando de Prolog.
- Una variable siempre unifica consigo misma: `Z y Z`.
- Cuando una variable empieza por el símbolo de subrayado, por ejemplo `_Var1`, Prolog sobreentiende que no estamos interesados en conocer el valor concreto que tiene esa variable. Esto sirve para evitar ciertos mensajes de advertencia (warnings) del compilador que resultan algo molestos.
- Una variable puede unificar con un término que contiene esa variable: `X y a(X)`. Esto no es muy recomendable puesto que estamos definiendo un término infinitamente anidado. No obstante, algunos entornos de desarrollo son tan potentes que pueden manejar dichos términos dependiendo, eso sí, de qué se pretenda hacer con ellos.

Ejecutando cosas

Predicados y Objetivos

Los **predicados** son los elementos ejecutables en Prolog. En muchos sentidos se asemejan a los procedimientos o funciones típicos de los lenguajes imperativos.

Una llamada concreta a un predicado, con unos argumentos concretos, se denomina **objetivo** (en inglés, goal). Todos los objetivos tiene un resultado de **éxito o fallo** tras su ejecución indicando si el predicado es cierto para los argumentos dados, o por el contrario, es falso.

Cuando un objetivo tiene éxito las variables libres que aparecen en los argumentos pueden quedar ligadas. Estos son los valores que hacen cierto el predicado. Si el predicado falla, no ocurren ligaduras en las variables libres.

Ejemplos

El caso más básico es aquél que no contiene variables: `son_hermanos('Juan','Maria')`. Este objetivo sólamente puede tener una solución (verdadero o falso).

Si utilizamos una variable libre: `son_hermanos('Juan',X)`, es posible que existan varios valores para dicha variable que hacen cierto el objetivo. Por ejemplo para `X = 'Maria'`, y para `X = 'Luis'`.

También es posible tener varias variables libres: `son_hermanos(Y,Z)`. En este caso obtenemos todas las combinaciones de ligaduras para las variables que hacen cierto el objetivo. Por ejemplo, `X = 'Juan' y Z = 'Maria'` es una solución. `X = 'Juan' y Z = 'Luis'` es otra solución.

Secuencias de objetivos

Hasta ahora hemos visto como ejecutar objetivos simples, pero esto no resulta demasiado útil.

En Prolog los objetivos se pueden combinar mediante conectivas propias de la lógica de primer orden: la conjunción, la disyunción y la negación.

La disyunción se utiliza bien poco y la negación requiere todo un capítulo para ser explicada. En cambió la conjunción es la manera habitual de ejecutar secuencias de objetivos.

El operador de conjunción es la coma, por ejemplo: `edad(luis,Y),edad(juan,Z),X>Z`. Parece sencillo, pero hay que tener en cuenta qué ocurre con las ligaduras de las variables:

- En primer lugar, hay que ser consciente de que los objetivos se ejecutan secuencialmente por orden de escritura (es decir, de izquierda a derecha).
- Si un objetivo falla, los siguientes objetivos ya no se ejecutan. Además la conjunción, en total, falla.
- Si un objetivo tiene éxito, algunas o todas sus variables quedan ligadas, y por tanto, dejan de ser variables libres para el resto de objetivos en la secuencia.
- Si todos los objetivos tienen éxito, la conjunción tiene éxito y mantiene las ligaduras de los objetivos que la componen.

Supongamos que la edad de Luis es 32 años, y la edad de Juan es 25:

- La ejecución del primer objetivo tiene éxito y liga la variable "Y", que antes estaba libre, al valor 32.
- LLega el momento de ejecutar el segundo objetivo. Su variable "Z" también estaba libre, pero el objetivo tiene éxito y liga dicha variable al valor 25.
- Se ejecuta el tercer objetivo, pero sus variables ya no estan libres porque fueron ligadas en los objetivos anteriores. Como el valor de "Y" es mayor que el de "Z" la comparación tiene éxito.

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

Varias soluciones

Hasta ahora todo parece sencillo, pero **¿ qué ocurre si uno o varios objetivos tienen varias soluciones ?**. Para entender como se ligan las variables en este caso hemos de explicar en qué consiste el **backtracking** en Prolog.

Backtracking

Supongamos que disponemos de dos predicados p/1 y q/1 que tienen varias soluciones (el orden es significativo):

- `p(1)` tiene éxito.
- `p(2)` tiene éxito.
- `q(2)` tiene éxito.
- No hay más soluciones que éstas.

Y a continuación consideramos la siguiente secuencia: `p(X),q(X)`.
Ahora ejecutamos la secuencia tal y como explicamos en la lección anterior:

- Ejecutamos p(X) con éxito y la variable queda ligada al valor 1 (primera solución).
- Ejecutamos q(X), pero la variable ya no esta libre, luego estamos ejecutando realmente q(1). El predicado falla porque no es una de sus soluciones.
- La conjunción falla.

El resultado ha sido fallo, pero nosotros sabemos que para X = 2 existe una solución para la conjunción.
Aquí es donde entra en juego el **backtracking**. Esto consiste en recordar los momentos de la ejecución donde un objetivo tenía varias soluciones para posteriormente **dar marcha atrás** y seguir la ejecución utilizando otra solución como alternativa.
El backtracking funciona de la siguiente manera:

- Cuando se va ejecutar un objetivo, Prolog sabe de antemano cuantas soluciones alternativas puede tener. En un futuro capítulo veremos cómo puede llegar a saber esto. Cada una de las alternativas se denomina **punto de elección**. Dichos puntos de elección se anotan internamente y de forma ordenada. Para ser exactos, se introducen en una pila.
- Se escoge el primer punto de elección y se ejecuta el objetivo **eliminando** el punto de elección en el proceso.
- Si el objetivo tiene éxito se continúa con el siguiente objetivo aplicandole estas mismas normas.
- Si el objetivo falla, Prolog **dá marcha atrás** recorriendo los objetivos que anteriormente sí tuvieron éxito (en orden inverso) y **deshaciendo** las ligaduras de sus variables. Es decir, comienza el backtracking.
- Cuando uno de esos objetivos **tiene un punto de elección** anotado, se detiene el backtracking y se ejecuta de nuevo dicho objetivo usando la solución alternativa. Las variables se ligan a la nueva solución y la ejecución **continúa de nuevo hacia adelante**. El punto de elección se elimina en el proceso.
- El proceso se repite mientras haya objetivos y puntos de elección anotados. De hecho, se puede decir que un programa Prolog ha terminado su ejecución cuando no le quedan puntos de elección anotados ni objetivos por ejecutar en la secuencia.

Además, los puntos de elección se mantienen aunque al final la conjunción tenga éxito. Esto permite posteriormente conocer todas las soluciones posibles.

Ejemplo

La manera en que se ejecuta realmente nuestro ejemplo es la siguiente:

- Prolog tiene que ejecutar `p(X)` y sabe (en el futuro veremos por qué) que existen dos soluciones. En consecuencia, anota dos puntos de elección.
- Ejecutamos `p(X)` usando el primer punto de elección, que se elimina en el proceso. Dicho objetivo tiene éxito y la variable queda ligada al valor 1 (primera solución).
- Hay que ejecutar `q(X)` que solamente tiene un punto de elección y queda anotado.
- Ejecutamos `q(X)` eliminando su (único) punto de elección, pero la variable ya no está libre, luego estamos ejecutando realmente `q(1)`. El predicado falla porque no es una de sus soluciones.
- Comienza el backtracking, recorriendo los objetivos en orden inverso hasta encontrar un punto de elección anotado.
- Nos topamos con el objetivo `p(X)`. Se deshace la ligadura de la variable X, es decir, X vuelve a estar libre.
- Se encuentra un punto de elección. La ejecución sigue de nuevo hacia adelante.
- Ejecutamos de nuevo `p(X)`, pero esta vez se usa el punto de elección que hemos encontrado. Se liga la variable X al valor 2 que corresponde a la segunda solución. El punto de elección se elimina en el proceso.
- Hay que ejecutar `q(X)` que solamente tiene un punto de elección y queda anotado.
- Ejecutamos `q(X)` eliminando su (único) punto de elección, pero la variable ya no esta libre, luego estamos ejecutando realmente `q(2)`. El objetivo tiene éxito esta vez.
- La conjunción tiene éxito manteniendo la ligadura de la variable X al valor 2.

Predicados predefinidos (built-in)

Existen algunos predicados predefinidos en el sistema y que están disponibles en todo momento. El más importante es la igualdad: `=/2`. Este predicado tiene éxito si sus dos argumentos unifican entre sí, falla en caso contrario. Por ejemplo, el objetivo `X = 3` provoca la ligadura de X al valor 3 puesto que unifican. Otro ejemplo es `f(3) = f(X)`, que también liga X al valor 3.

Es **muy importante** no confundir la igualdad lógica con la igualdad aritmética. Por ejemplo, `X = 3 + 2` tiene éxito pero **no** resulta en X ligado a 5. De hecho, la variable X queda ligada al término `+(3,2)`. La aritmética será discutida en un posterior capítulo.

Otros predicados predefinidos muy útiles son los de comparación aritmética. Naturalmente, estos no funcionan con cualquier término como argumento. Solamente sirven para números (enteros y decimales).

Predicado	Significado
<	menor que
>	mayor que
=<	menor o igual que
>=	mayor o igual que
=:=	igualdad aritmetica
=\=	desigualdad aritmetica

El código

Cláusulas

Hasta ahora sabemos cómo ejecutar objetivos, pero no sabemos como escribir el código de los predicados. Los predicados se definen mediante un **conjunto de cláusulas**:

```
clausula1
clausula2
...
```

```
clausulaN
```

Donde el orden es significativo. Para facilitar la lectura, se suele dejar una línea en blanco entre cláusula y cláusula.
Las cláusulas son términos (como todo en Prolog) con el siguiente formato:

```
cabeza :-
  ojetivo1,
  ojetivo2,
  ...,
  ojetivoN.
```

Todo gira en torno al operador "**:-**". Lo que aparece a la izquierda se denomina **cabeza** y la secuencia de objetivos que aparece a la derecha se denomina **cuerpo**.
La cabeza es un término simple, por ejemplo, `p(X,12)` podría ser la cabeza de una cláusula del predicado p/2. Es decir, todas las cláusulas de un mismo predicado tienen en la cabeza un término con el mismo functor y aridad, aunque los argumentos pueden ser distintos.
El cuerpo no es más que el conjunto de condiciones que deben cumplirse (tener éxito) para que el predicado tenga éxito si lo invocamos con un objetivo que **unifique** con la cabeza.
Cuando invocamos un objetivo, Prolog unifica dicho objetivo con las cabezas de las cláusulas. Cada cláusula que unifique constituye un punto de elección.
A continuación se ejecuta el cuerpo de la primera cláusula. Para ello se mantienen las ligaduras que ocurrieron en el paso anterior. Si el cuerpo tiene éxito, pueden ocurrir nuevas ligaduras. Dichas ligaduras pueden afectar de nuevo a la cabeza de la cláusula. En consecuencia, el **ámbito de visibilidad de las variables** es una única cláusula.
Si el cuerpo de la cláusula falla, el mecanismo de backtracking nos lleva al siguiente punto de elección, es decir, la siguiente cláusula. El proceso se repite mientras queden cabezas que unifiquen (es decir, puntos de elección). Cuando no quedan cabezas que unifiquen, el objetivo falla.

Ejemplo simple

Veamos un predicado compuesto por una simple cláusula:

```
es_viejo(Individuo) :-
  edad(Individuo,Valor),
  Valor > 60.
```

Ahora invocamos el objetivo `es_viejo(luis)`. Para ello supongamos que la edad de Luis es 32 años, es decir, el objetivo `edad(luis,32)` tiene éxito.
Primero se unifica la cabeza de la cláusula con el objetivo. Es decir, unificamos `es_viejo(luis)` y `es_viejo(Individuo)`, produciendose la ligadura de la variable Individuo al valor luis. Como el ámbito de visibilidad de la variable es su cláusula, la ligadura también afecta al cuerpo, luego estamos ejecutando realmente:

```
es_viejo(luis) :-
  edad(luis,Valor),
  Valor > 60.
```

Ahora ejecutamos el cuerpo, que liga la variable Valor a 32. Pero el cuerpo falla porque el segundo objetivo falla (32>60 es falso). Entonces la cláusula falla y se produce backtracking. Como no hay más puntos de elección el objetivo falla. Es decir, Luis no es un viejo.

Ejemplo menos simple

Ahora veamos como las ligaduras que se producen en el cuerpo de la cláusula afectan también a la cabeza. Consideramos el siguiente predicado compuesto de una única cláusula:

```
mayor_que(Fulano,Mengano) :-
  edad(Mengano,EdadMengano),
  edad(Fulano,EdadFulanano),
  EdadFulano > EdadMengano.
```

Supongamos que la edad de Juan es 20 años y la de Luis es 32 años. Ejecutamos el objetivo `mayor_que(luis,Quien)`:

- Unificamos el objetivo con la cabeza: la variable Fulano se liga a luis, la variable Mengano permanece unificada con la variable Quien. Esto último es importante.
- Ejecutamos el cuerpo, que tiene éxito y liga las variables Mengano a juan, EdadMengano a 20, EdadFulano a 32.
- Como la variable Mengano ha quedado ligada, y además unificaba con Quien, la variable Quien queda ligada a ese mismo valor.
- El objetivo tiene éxito ligando la variable Quien al valor juan. Es decir, Luis es mayor que Juan.

Cláusulas sin cuerpo

Si no existen condiciones para que una cláusula sea cierta podemos omitir el cuerpo. En tal caso solamente escribimos la cabeza terminada en punto. Por ejemplo:

```
edad(juan,32).
edad(luis,20).
```

Son dos cláusulas del predicado edad/2. Las cláusulas sin cuerpo se suelen denominar **hechos**, e.g. es un hecho que la edad de Luis es 20 años.

Culturilla

- Podemos escribir las cláusulas en una sola línea, si no lo hacemos es por legibilidad: `a :- b,c,d.`
- El orden de escritura de las cláusulas determina el orden en que se suceden las soluciones.
- Recuerde que pueden aparecer puntos de elección dentro del cuerpo de una cláusula, como en toda secuencia de objetivos. Esto significa que una única cláusula puede dar lugar a varias soluciones cuando uno o más objetivos del cuerpo tienen también varias soluciones.
- Si una misma variable aparece en dos cláusulas diferentes, entonces son variables diferentes pero con el mismo nombre. Recuerde que el ámbito de visibilidad de las variables es una única cláusula.

El shell de Prolog

El shell de Prolog es una aplicación que permite **ejecutar objetivos** y ver las ligaduras de las variables de manera interactiva. Pueden existir diferencias entre unos entornos de desarrollo y otros respecto a su uso. Para ilustrar su uso, nosotros utilizaremos el shell de Ciao/Prolog.

Ejecutando el shell

El shell es una aplicación más que podemos ejecutar en nuestro sistema operativo. En nuestro caso, la aplicación se denomina ciaosh. Al ejecutarla aparece un típico mensaje de bienvenida:

```
Ciao-Prolog 1.4 #0: Sat Nov 27 19:27:11 1999
?-
```

El símbolo `?-` nos indica la zona donde podemos escribir los objetivos a ejecutar.
Para mejorar la legibilidad en los ejemplos, **destacamos** el texto que el usuario teclea para distinguirlo de la salida por pantalla del shell.

Mi primer objetivo

Cuando arrancamos el shell, los únicos objetivos que podemos ejecutar corresponden a predicados predefinidos en el sistema. Nuestro predicado predefinido favorito es la igualdad =/2. Así que vamos a probarlo:

```
Ciao-Prolog 1.4 #0: Sat Nov 27 19:27:11 1999
?- t(X,3) = t(4,Z).

X = 4,
Z = 3 ?
```

Observese que los objetivos acaban en un punto (.), si pulsamos intro antes de escribir el punto ocurre un salto de línea, pero nada más. Cuando escribimos el punto y pulsamos INTRO es cuando se ejecuta el objetivo.
A continuación, el shell nos dice si el objetivo tiene éxito o no, y cuales son las ligaduras de las variables. Después aparece un signo de interrogación (?). En este momento es cuando le podemos pedir que nos muestre **otra solución** tecleando un punto y coma (;) y pulsando INTRO:

```
Ciao-Prolog 1.4 #0: Sat Nov 27 19:27:11 1999
?- t(X,3) = t(4,Z).

X = 4,
Z = 3 ? ;

no
?-
```

Como no hay más soluciones en nuestro ejemplo, el shell dice "no" y nos permite escribir otro objetivo. Si no hubiesemos deseado más soluciones simplemente habríamos pulsado INTRO.

Compilando y cargando codigo

Puesto que en el shell solamente podemos ejecutar objetivos, la forma de compilar y cargar código es ejecutando un objetivo. Esto puede variar de un shell a otro, pero habitualmente se hace así:

```
?- consult('prog1.pl').

yes
?-
```

Obsérvese que el nombre del fichero fuente (y su ruta, si es necesario) se escribe en un término cero-ario entre comillas simples. Esta es la forma habitual de escribir nombres de fichero.

```
?- consult('c:/temp/prog1.pl').

yes
?-
```

Quiero irme de aquí

Cuando nos cansamos de jugar con el shell, podemos terminar la aplicación ejecutando el predicado halt/0, o bien pulsando CTRL-D:

```
?- halt.
Process Ciao/Prolog<1> finished
```

Mi primer programa en Prolog

Los programas se escriben en ficheros de texto, generalmente con extension .pl y pueden contener comentarios y código. Para ello puede utilizar cualquier editor de texto. Le recomendamos que intente escribir el siguiente programa desde el principio para familiarizarse con la sintáxis.

```
% Este es mi primer programa en Prolog
%
% Se trata de un arbol genealogico muy simple
%
%
% Primero defino los parentescos basicos
% de la familia.
% padre(A,B) significa que B es el padre de A...

padre(juan,alberto).
padre(luis,alberto).
padre(alberto,leoncio).
```

```
padre(geronimo,leoncio).
padre(luisa,geronimo).

% Ahora defino las condiciones para que
%  dos individuos sean hermanos
% hermano(A,B) significa que A es hermano de B...

hermano(A,B) :-
  padre(A,P),
  padre(B,P),
  A \== B.

% Ahora defino el parentesco abuelo-nieto.
% nieto(A,B) significa que A es nieto de B...

nieto(A,B) :-
  padre(A,P),
  padre(P,B).
```

Cargando el código

Para compilar y cargar el código existe el predicado consult/1. Recuerde que puede ser necesario indicar la ruta completa del fichero fuente. En este ejemplo hemos usado el top-level shell de SWI-Prolog:

```
Welcome to SWI-Prolog (Version 2.7.14)
 Copyright (c) 1993-1996 University of Amsterdam.  All rights
 reserved.

 For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- consult('arbolgenealogico.pl').
arbolgenealogico.pl compiled, 0.00 sec, 1,108 bytes.

Yes
2 ?-
```

Predicados reversibles

Una vez que hemos compilado y cargado nuestro programa vamos a estudiar sus características. Una de ellas es el backtracking, o la posibilidad de obtener varias soluciones, como ya hemos visto.

```
2 ?- hermano(A,B).

A = juan
B = luis ;

A = luis
B = juan ;

A = alberto
B = geronimo ;

A = geronimo
B = alberto ;

No
3 ?-
```

Ahora vamos a reparar en otra curiosa propiedad que no existe en otros lenguajes: la **reversibilidad**. Esto es la habilidad de los argumentos de los predicados para actuar indistintamente como argumentos de entrada y/o salida.Por ejemplo:

```
3 ?- nieto(luis,X).

X = leoncio
```

```
No
4 ?-
```

Aquí, el primer argumento es de entrada mientras que el segundo es de salida. El predicado nos dice de quién es nieto Luis. Pero ahora vamos a intercambiar los papeles:

```
4 ?- nieto(X,leoncio).

X = juan ;

X = luis ;

X = luisa ;

No
5 ?-
```

Obsérve cómo el **mismo** predicado que nos dice el abuelo de un nieto sirve para conocer los nietos de un abuelo. Estos predicados se dicen reversibles, o que sus argumentos son reversibles.

Predicados no reversibles

No todos los predicados son reversibles. Por ejemplo, los de comparación aritmética. El predicado >/2 sirve para saber si un número es mayor que otro, pero no sirve para saber todos los números mayores que uno dado (puesto que son infinitos).
Otros predicados pueden perder la reversibilidad por deseo expreso de su programador, o solamente ser reversibles para ciertos argumentos pero no otros. Así podemos hablar de las posibles formas de invocar un predicado. Esto es lo que se denomina **modos de uso**.

Modos de uso

Los modos de uso indican que combinacion de argumentos deben o no estar **instanciados** para que un objetivo tenga sentido. Se dice que un argumento está instanciado cuando no es una variable libre.
A efectos de documentación, los modos de uso se describen con un término anotado en sus argumentos con un símbolo. Los argumentos pueden ser:

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

El modo de uso que instancia todos los argumentos siempre es válido.
Por ejemplo, para el predicado hermano/2 su único modo de uso es hermano(?A,?B).
Supongamos un predicado cuyos modos de uso son:

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

Entonces son objetivos válidos:

- p(1,2,X).
- p(1,X,3).
- p(1,2,3).

Pero no son válidos:

- p(X,Y,3).
- p(X,2,3).
- p(X,Y,Z).

Los modos de uso se suelen indicar a modo documentativo, pero actualmente los entornos de desarrollo más avanzados los pueden utilizar para detectar errores en tiempo de compilación.

Culturilla

- Un compilador que no utiliza los modos de uso no detecta objetivos inválidos. En ese caso, el programa se ejecuta sin más. Cuando le llega el turno al objetivo mal formado pueden ocurrir dos cosas: que el objetivo falle sin más, o que se lance una excepción. Cualquiera de ellas suele ir acompañada de un horrendo mensaje por pantalla.
- La forma de describir los modosde uso "formalmente" varia de un entorno de desarrollo a otro. Si no es posible especificarlos "formalmente", entonces conviene escribir un comentario explicativo.
- Si el compilador no detecta modos de uso, es responsabilidad del programador no invocar objetivos mal formados, tarea que no resulta nada trivial puesto que hay que saber que variables van a estar ligadas cuando el programa se ejecute.
- Otra buena práctica es escribir predicados que siempre sean reversibles.

Evaluación de expresiones aritméticas

LLega el momento de utilizar Prolog para nuestros complejísimos cálculos aritméticos. ¿ Acaso existe algún programa donde no se sumen dos y dos ?.

En Prolog es fácil construir expresiones aritméticas. Algún avispado se habrá percatado de que las expresiones matemáticas en general son términos, puesto que corresponden a teorías lógicas de primer orden.

El problema es reducir esas expresiones según las leyes matemáticas para obtener lindos numeritos. Eso se hace en Prolog mediante el predicado **is/2**, cuyo modo de uso es is(-Var,+Expr). Además, el argumento Expr debe ser un término cerrado (es decir, que no contenga variables libres). Por ejemplo, vamos a sumar dos y dos:

```
Welcome to SWI-Prolog (Version 2.7.14)
Copyright (c) 1993-1996 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-  X is 2 + 2.

X = 4

yes
2 ?-
```

El predicado is/2 no es reversible, por eso debemos estar seguros de que las variables del segundo argumento siempre están instanciadas. El modo de uso que instancia todas las variables solo tiene éxito si el primer argumento es un número y coincide con la evaluación del segundo argumento. Por ejemplo:

```
2 ?-  5 is 2 + 2.

no
3 ?-
```

Expresiones válidas

Las expresiones que podemos utilizar en el segundo argumento pueden variar de un entorno de desarrollo a otro, pero vamos a citar las más comunes:

Término	Significado	Ejemplo
+/2	Suma	X is A + B.

*/2	Producto	X is 2 * 7.
-/2	Resta	X is 5 - 2.
'/'/2	División	X is 7 / 5.
-/1	Cambio de signo	X is -Z.
'//'/2	División entera	X is 7 // 2.
mod/2	Resto de la división entera	X is 7 mod 2.
'^'/2	Potencia	X is 2^3.
abs/1	Valor absoluto	X is abs(-3).
pi/0	La constante PI	X is 2*pi.
sin/1	seno en radianes	X is sin(0).
cos/1	coseno en radianes	X is cos(pi).
tan/1	tangente en radianes	X is tan(pi/2).
asin/1	arcoseno en radianes	X is asin(7.2).
acos/1	arcocoseno en radianes	X is acos(Z).
atan/1	arcotangente en radianes	X is atan(0).
floor/1	redondeo por defecto	X is floor(3.2).
ceil/1	redondeo por exceso	X is ceil(3.2).

Resumen y ejercicios

Hasta este momento, el lector debería haber aprendido:

1. Qué es un entorno de desarrollo Prolog.
2. Qué es una variable lógica.
3. Qué es un término.
4. Cómo funciona la unificación.
5. Cómo se ejecutan objetivos desde el top-level shell.
6. Cómo se ejecutan secuencias de objetivos.
7. Cómo el backtracking permite explorar varias soluciones.
8. Cómo se escribe un fichero fuente en Prolog.
9. Cuál es el motivo de que aparezcan puntos de elección.
10. Qué es la reversibilidad.
11. Qué son los modos de uso y para qué sirven.
12. Cómo se realizan cómputos aritméticos.

Ejercicios sobre términos y variables

A continuación aparecen una serie de expresiones. Trate de identificar si se trata de variables, términos o si están mal construidos.

- p(j(G),h(12),j(3),a+b)
- p(j(G),H(12),j(3),a+b)
- __abc
- aBc

- AbC
- 3 $ 2
- ' '(_,_)
- _'A'(12)
- 32.1
- pepe > 32.2

Ejercicios sobre unificación

Indique si los siguientes pares de términos unifican entre sí. En caso de que unifiquen, indique a que valores se ligan las variables.

- p(a) y p(A)
- p(j(j(j(j(j))))) y p(j(j(j(j))))
- p(j(j(j(j(j))))) y p(j(j(j(X))))
- q(_,A,_) y q(32,37,12)
- z(A,p(X),z(A,X),k(Y)) y z(q(X),p(Y),z(q(z(H))),k(z(3)))
- z(A,p(X),z(A,X),k(Y)) y z(q(X),p(Y),z(q(z(H)),z(H)),k(z(3)))

Compruebe los resultados del ejercicio utilizando el top-level shell y el predicado igualdad =/2.
A continuación, ejecute las siguientes secuencias de objetivos en el top-level shell y observe las ligaduras de las variables:

- f(X) = f(Y).
- X = 12, f(X) = f(Y).
- f(X) = f(Y), X = 12.
- f(X) = f(Y), Y = 12.
- X = Y, Y = Z, X = H, Z = J, X = 1.
- X = 1.
- 1 = X.

Ejercicios sobre predicados

A continuación indicamos las soluciones de tres predicados (el orden es significativo):

- p(5,2) tiene éxito.
- p(7,1) tiene éxito.
- q(1,3) tiene éxito.
- z(3,1) tiene éxito.
- z(3,7) tiene éxito.
- No hay más soluciones que las anteriores.

Indique los pasos de ejecución para la secuencia `p(A,B),q(B,C),z(C,A)`.
Defina el predicado sumar_dos/2 que toma un número en el primer argumento y retorna en el segundo argumento el primero sumado a dos. ¿ Cuáles son los modos de uso permitidos para dicho predicado ?.
Editando el programa de ejemplo (arbolgenealogico.pl), defina el predicado `tio/2` donde `tio(A,B)` significa que A es el tío de B. Utilize dicho predicado desde el top-level shell para averiguar quienes son los sobrinos de `geronimo`. Recuerde que cada vez que modifique el fichero fuente debe volver a compilarlo mediante el predicado consult/1.

Curso intermedio de programación en Prolog

Indice de contenidos

Introducción y licencia de uso

Este curso intermedio de programación en Prolog constituye la segunda entrega de una serie de cursillos orientados a aquellas personas que desconocen la programación declarativa relacional y su lenguaje rey: **Prolog**.

Requisitos

Para seguir este curso es imprescindible leer el tutorial de Prolog que le precede.

Créditos y licencia

Tipos de datos

Todos sabemos que los datos que maneja Prolog son los términos. Sin embargo, podemos construir otros tipos de datos a partir de estos. De hecho, algunos estan predefinidos para mayor gloria del programador, son el caso de las listas y las cadenas de caracteres.

En cualquier caso, el lector debe asumir que Prolog **no es un lenguaje tipado**, puesto que no existen declaraciones explícitas de tipo tal y como ocurre en los lenguajes imperativos. El hecho de que no existan dichas declaraciones se debe sencillamente a que **no hacen falta**.

Registros

Los registros son agrupaciones ordenadas de datos que en Prolog podemos escribir como términos que almacenan cada dato en un argumento. Por ejemplo, supongamos que queremos un registro para representar los datos personales de la gente:

```
persona('Eva','Fina','Y Segura',15)
persona('Fulanito','De Tal','Y Tal',32)
```

Mediante el término **persona/4** representamos a un individuo. El primer argumento es el nombre, el segundo y tercero son los apellidos y el cuarto es la edad.
Puesto que los términos son anidables podemos crear registros complejos:

```
persona('Menganito',edad(32),direccion('Leganitos',13,'Madrid'))
```

Árboles

Puesto que los términos pueden ser recursivos es fácil crear estructuras de datos recurrentes. Como ejemplo, veamos como definir árboles binarios. Para ello representamos el árbol vacío mediante una constante, por ejemplo, **empty/0**, y un nodo cualquiera puede ser representado mediante el término **tree/3**. El primer argumento representa un dato cualquiera asociado al nodo. El segundo argumento representa la rama izquierda, y el tercer argumento la correspondiente rama derecha. Son ejemplos de árboles:

```
empty
tree(dato1,empty,empty)
```

```
tree(dato1,tree(dato2,empty,empty),tree(dato3,empty,empty))
tree(dato4,empty,tree(dato5,tree(dato6,empty,empty),empty))
```

Listas

Las listas en Prolog podrian definirse del mismo modo que los árboles puesto que los términos se pueden anidar todas las veces que sea necesario. Por ejemplo, la lista de números del uno al cinco se puede representar así:

```
lista(1,lista(2,lista(3,lista(4,lista(5,vacio)))))
```

Afortunadamente, las listas están predefinidas en el lenguaje para una mayor comodidad. De modo que la lista anterior la podemos escribir así:

```
[1, 2, 3, 4, 5]
```

Esta es la forma de escribir las listas definiendo todos los elementos, pero podemos manipular las listas distinguiendo cabeza y resto: `[C|R]`. Donde la variable C representa la cabeza, y R el resto. Por ejemplo:

```
L = [1, 2, 3, 4, 5],
M = [0|L].
```

La lista M sería equivalente a `[0,1,2,3,4,5]` . Es importante no confundir los términos `[C|R]` y `[C,R]`. La diferencia es muy sutil:

```
L = [1, 2, 3, 4, 5],
M = [0,L].
```

El resultado sería `M = [0,[1,2,3,4,5]]`, que es una lista de dos elementos.
Naturalmente, existe la lista vacía, que se representa como `[]`. Además resulta conveniente tener en cuenta que:

- Existen bibliotecas para hacer cosas más complicadas con las listas, como concatenar, aplanar, etc.
- Los elementos de las listas son términos y pueden ser heterogéneos. Por ejemplo: `[1,p(a),[a,b],f(g(h))]`.
- Las listas también son términos, solamente las escribimos de una manera más cómoda. Así, la lista `[1,a,2]` es en realidad el término `'.'(1,'.'(a,'.'(2,[])))`.

Cadenas de caracteres

Las cadenas de caracteres son en Prolog **listas de códigos ASCII**. Afortunadamente se pueden escribir de una manera cómoda poniendo los caracteres entre comillas dobles. Por ejemplo, la expresión `"ABC"` es en realidad la lista `[65,66,67]`. Así, podemos tratar las cadenas de caracteres como cadenas o como listas según nos interese. Naturalmente, todo el código que nos sirve para listas nos sirve para cadenas. Por ejemplo, el predicado que concatena dos listas támbien sirve para concatenar dos cadenas de texto (¿ no es genial ?).

Constantes

Como ya es sabido, las constantes en Prolog son términos cero-arios (átomos). A pesar de su simpleza, pueden ser muy útiles para representar información ya que pueden contener cualquier caracter. Se utilizan, por ejemplo, para representar nombres de ficheros. Recuerde que las constantes numéricas también son términos cero-arios (pero no son átomos).

Conversión entre números, átomos y cadenas de caracteres

Existe cierta correspondencia entre estos elementos. Tanto los números como los átomos se pueden convertir a cadena de caracteres mediante los predicados **number_codes/2** y **atom_codes/2** respectivamente. Utilizando las cadenas de caracteres como elemento intermedio, es posible convertir de átomos a números y viceversa: `atom_codes(Atomo,Aux),number_codes(Numero,Aux)`. Observe que dichos predicados son reversibles.

Tests de tipo

Si en Prolog no existen declaraciones de tipo, ¿ como demonios estamos seguros de que un argumento es de un tipo determinado ?. La respuesta está en los **tests de tipo**. Éstos són predicados que (habitualmente) reciben un dato como argumento y fallan si el argumento no es del tipo esperado.

Como ejemplo vamos a escribir el test de tipo para comprobar si un argumento es una lista:

```
es_una_lista( [] ).
es_una_lista( [ _ | Resto ] ) :-
   es_una_lista(Resto).
```

La lista vacía es una lista, y si no, [A|B] será una lista si y sólo si B es una lista. En cuanto a A, nos trae al fresco lo que valga, por eso usamos una variable anónima en el código.
Ahora podemos comprobar el tipo de un argumento llamando al test de tipo:

```
mi_predicado(Lista1,Lista2) :-
   es_una_lista(Lista1),
   es_una_lista(Lista2),
   ...,
```

Tests de tipo predefinidos

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

Predicado	Test
integer/1	Comprueba si su argumento es un número entero
float/1	Comprueba si el argumento es un número decimal
number/	Comprueba si el argumento es un número (entero o decimal)
atom/1	Comprueba si el argumento es un término cero-ario excluyendo las constantes numéricas
var/1	Comprueba si el argumento es una variable libre
nonvar/1	Comprueba si el argumento está instanciado
ground/1	Comprueba si el argumento es un término que no contiene variables libres (está cerrado)

Ejecución de los tests

La desventaja de los tests de tipo es que resulta necesario ejecutarlos. Esto añade un tiempo extra de ejecución a nuestra aplicación que no sirve para nada útil. Sin embargo, esto es solamente una verdad a medias:

- La mayoría de los predicados no requieren test de tipo como es el propio predicado que implementa el test de tipo. Todo gracias a la unificación.
- Los compiladores más avanzados son capaces de suprimir los test de tipo cuando pueden asegurar en tiempo de compilación que el predicado no se llamará con tipos inadecuados.
- Analogamente, algunos compiladores son capaces de insertar automáticamente el test de tipo cuando al programador "se le olvida" (es decir, siempre).
- Cuando la aplicación esta terminada y probada, el propio programador puede suprimir los test de tipo si está seguro de que no pueden producirse errores de tipo.
- El polimorfismo que aporta la ausencia de declaraciones de tipo es deseable en muchas ocasiones. Por eso, los test de tipo no siempre son necesarios.

En cualquier caso, el programador tiene libertad para decidir si es necesario ejecutar tests de tipo en su programa.

Tests de tipo versus sublenguaje de tipos

La mayoría de los lenguajes tradicionales (imperativos) incorporan un sublenguaje para la declaración de los tipos. Se dice que es un sublenguaje porque no sigue la misma sintáxis, ni dispone de los mismos recursos, que el propio lenguaje de programación. Estos sublenguajes solamente contemplan un tipado de tipo estructural, pero no de tipo **semántico**. Por ejemplo, es posible declarar el tipo de datos Lista de números enteros, pero dificilmente podemos declarar el tipo de datos Lista de números enteros ordenados de menor a mayor que además son todos menores que cien. A decir verdad, lo podemos declarar, pero no espere que el compilador compruebe si las lista de su programa estan ordenadas o no.
Sin embargo, en Prolog esto **sí** es posible incluso **en tiempo de compilación**. Hoy por hoy las técnicas para conseguir esto están aún en investigación, motivo por el que no va ha encontrar muchos entornos de desarrollo que lo incorporen. Lo importante es ser consciente de que la ausencia de sublenguaje de tipos no es una desventaja sino todo lo contrario.

Tipos paramétricos

También es posible escribir test de tipos paramétricos, es decir, aquellos que dependen de otro tipo. Por ejemplo, para evitar tener que definir un test de tipo para listas de números y otro para listas de átomos, podríamos definir el tipo "lista de X". La declaración de estos test de tipo requiere el uso de predicados de orden superior (o metapredicados) que estudiaremos posteriormente.

El corte

El **corte** es un predicado predefinido que no recibe argumentos. Se representa mediante un signo de admiración (`!`). Sin duda, es el predicado más difícil de entender. El corte tiene la espantosa propiedad de **eliminar los puntos de elección** del predicado que lo contiene.

Es decir, cuando se ejecuta el corte, el resultado del objetivo (no sólo la cláusula en cuestión) queda comprometido al éxito o fallo de los objetivos que aparecen a continuación. Es como si a Prolog "se le olvidase" que dicho objetivo puede tener varias soluciones.

Otra forma de ver el efecto del corte es pensar que solamente tiene la propiedad de detener el backtracking cuando éste se produce. Es decir, en la ejecución normal el corte no hace nada. Pero cuando el programa entra en backtracking y los objetivos se recorren marcha atrás, al llegar al corte el backtracking se detiene repentinamente forzando el fallo del objetivo.

Ejemplo

Para entender de manera simple el uso del corte vamos a comparar dos predicados que solamente se diferencian en un corte:

```
% Sin corte.
p(X,Y) :-
  X > 15,
  Y > 50.

p(X,Y) :-
  X > Y,

% Con corte.
q(X,Y) :-
  X > 15,
  !,
  Y > 50.

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

Veamos que ocurre si ejecutamos el objetivo `p(25,12)`:

- Obsérve que ambas cláusulas unifican con la cabeza, luego existen **dos** puntos de elección que se anotan.
- Prolog entra por el primer punto de elección (primera cláusula) eliminandolo.
- Prolog ejecuta el primer objetivo del cuerpo (`X>15`), que tiene éxito.
- Prolog ejecuta el segundo objetivo del cuerpo (`X>50`), que falla.
- Empieza el backtracking.
- Se recorren ambos objetivos hacia atrás pero no hay variables que se hayan ligado en ellos.
- Encontramos el segundo punto de elección (segunda cláusula) que detiene el backtracking eliminandolo en el proceso. La ejecución continúa hacia delante.
- Prolog ejecuta el cuerpo de la segunda cláusula que consiste en `X>Y`. Este objetivo tiene éxito.
- El objetivo `p(25,12)` tiene éxito.

Ahora comprobamos lo que ocurre cuando éxiste el corte, ejecutamos `q(25,12)`:

- Ambas cláusulas unifican con la cabeza, luego existen **dos** puntos de elección que se anotan.
- Prolog entra por el primer punto de elección (primera cláusula) eliminandolo.
- Prolog ejecuta el primer objetivo del cuerpo (`X>15`), que tiene éxito.
- Se ejecuta el segundo objetivo del cuerpo que es el corte. Por tanto, se eliminan todos los puntos de elección anotados que son debidos al objetivo `q(25,12)`. Solamente teníamos uno, que se elimina.
- Prolog ejecuta el tercer objetivo del cuerpo (`X>50`), que falla.
- Empieza el backtracking.
- Se recorren ambos objetivos hacia atrás pero no hay variables que se hayan ligado en ellos.
- No encontramos ningún punto de elección porque fueron eliminados por el corte.
- El objetivo `p(25,12)` falla.

Como puede comprobar, los resultados son sustacialmente diferentes. La segunda cláusula del predicado `q/2` ni siquiera ha llegado a ejecutarse porque el corte ha comprometido el resultado del objetivo al resultado de `Y>15` en la primera cláusula.

Usos del corte

El corte se utiliza muy frecuentemente, cuanto más diestro es el programador más lo suele usar. Los motivos por los que se usa el corte son, por orden de importancia, los siguientes:

1. Para optimizar la ejecución. El corte sirve para evitar que por culpa del backtracking se exploren puntos de elección que, con toda seguridad, no llevan a otra solución (fallan). Para los entendidos, esto es **podar el árbol de búsqueda** de posibles soluciones.
2. Para facilitar la legibilidad y comprensión del algoritmo que está siendo programado. A veces se situan cortes en puntos donde, con toda seguridad, no van a existir puntos de elección para eliminar, pero ayuda a entender que la ejecución sólo depende de la cláusula en cuestión.
3. Para implementar algoritmos diferentes según la combinación de argumentos de entrada. Algo similar al comportamiento de las sentencias case en los lenguajes imperativos.
4. Para conseguir que un predicado solamente tenga una solución. Esto nos puede interesar en algún momento. Una vez que el programa encuentra una solución ejecutamos un corte. Así evitamos que Prolog busque otras soluciones aunque sabemos que éstas existen.

Corte y fallo

Es muy habitual encontrar la secuencia de objetivos corte-fallo: `!,fail`. El predicado **fail/0** es un predicado predefinido que siempre falla. Se utiliza para detectar prematuramente

combinaciones de los argumentos que no llevan a solución, evitando la ejecución de un montón de código que al final va a fallar de todas formas.

Algoritmos y técnicas de programación

Los algoritmos utilizados en Prolog estan intimamente ligados a los términos y su estructura anidada/recursiva. Por eso, la técnica de programación por excelencia es la **recursividad**. Sin embargo existen técnicas propias del lenguaje como son los **bucles de fallo**. En este capítulo repasamos todas ellas.

Recursividad

La recursividad es la técnica por antonomasia para programar en Prolog. El lector ya habrá notado que en Prolog no existen bucles for, while, do-while, ni sentencias case, ni otras construcciones absurdas. En Prolog no hacen falta.

Todos los términos en Prolog pueden ser recursivos, y gracias a la unificación, podemos recorrer sus argumentos a voluntad. La estructura de datos más significativa con respecto a la recursividad son las listas, por eso centraremos nuestros ejemplos en ellas.

La estructura de las cláusulas de un predicado recursivo es muy simple. Como ejemplo veamos un predicado que calcula la longitud de una lista:

```
% La longitud de la lista vacia es cero

longitud([],0).

% La longitud de una lista es la longitud
% del resto mas uno. Como el contenido
% de la cabeza no nos interesa,
% utilizamos la variable anonima

longitud( [_|Resto], Longitud) :-
  longitud(Resto,LongitudResto),
  Longitud is LongitudResto+1.
```

Observe como el primer objetivo de la segunda cláusula es una llamada al propio predicado que estamos definiendo. Para evitar que un predicado se llame a sí mismo infinitamente debemos estar seguros de que existe almenos un caso en el que termina. Este caso se contempla en la primera cláusula y se denomina **caso básico**.
Otro ejemplo interesante es el predicado que concatena dos listas, que es reversible:

```
% Concatenar vacio con L es L...

concatena([],L,L).

% Para concatenar dos listas, sacamos
% la cabeza de la primera lista,
% luego concatenamos el resto con la segunda lista
% y al resultado le ponemos la cabeza
% de la primera lista como
% cabeza del resultado...

concatena([Cabeza|Resto],Lista,[Cabeza|RestoConcatenado]):-
  concatena(Resto,Lista,RestoConcatenado).
```

Parámetros de acumulación

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

```
longitud2_aux([],Parcial,Parcial).

longitud2_aux([_|Resto],Parcial,Result) :-
  NParcial is Parcial+1,
  longitud2_aux(Resto,NParcial,Result).

longitud2(Lista,Longitud) :-
  longitud2_aux(Lista,0,Longitud).
```

En este ejemplo, el valor inicial del parámetro de acumulación es cero. Este valor inicial es importante para que la solución sea correcta. Por eso hemos creado el predicado `longitud2/2`, que se asegura el correcto uso del parámetro de acumulación. El predicado `longitud2_aux/3` no debería ser ejecutado directamente.
La ventaja del parámetro de acumulación es que genera **recursividad de cola**, esto es, la llamada recursiva es siempre la última en ejecutarse. Esto permite a los compiladores optimizar considerablemente el uso de recursos ocasionado por la recursividad. La desventaja es que los predicados podrían resultar no reversibles.

Sentencias "case"

A modo meramente anecdótico indicamos como podría simularse una típica estructura "case" (de selección) propia de los lenguajes imperativos. Así el siguiente algoritmo:

```
case Dato of
  1 : corromper_archivos;
  2 : cancelar;
  3 : formatear_disco;
end;
```

Se expresaría en Prolog de la siguiente manera:

```
case(1) :-
   !,
   corromper_archivos.
case(2) :-
   !,
   cancelar.
case(3) :-
   !,
   formatear_disco.
```

Bucles de fallo

Los bucles de fallo constituyen una técnica de programación que permite recorrer una serie de elementos y aplicarles una operación. De la misma manera que un bucle for o while.
Los bucles de fallo están basados en la capacidad para obtener varias soluciones y el backtracking para recorrerlas. La estructura general de un bucle de fallo es la siguiente:

```
bucle :-
  generador(UnaSolucion),
  filtro(UnaSolucion),
  tratamiento(UnaSolucion),
  fail.

bucle.
```

El predicado generador/1 es el encargado de enumerar los datos a tratar en cada paso del bucle. Es decir, cada una de sus soluciones será un elemento a tratar en el bucle.
El predicado filtro/1 es opcional y permite seleccionar qué elementos se van a tratar y cuales no.
El predicado tratamiento/1 es el encargado de hacer algo con el dato. Es algo así como el cuerpo de un bucle for.
Finalmente, el predicado fail/0, que esta predefinido, se encarga de que ocurra el bucle forzando el backtracking. Además incluimos una cláusula para que el bucle en sí no falle después de haberse ejecutado.

Ejemplo

El siguiente ejemplo recorre los números del uno al diez y los escribe por pantalla.

```
generador(Desde,_,Desde).

generador(Desde,Hasta,Valor) :-
  Desde < Hasta,
  NDesde is Desde+1,
  generador(NDesde,Hasta,Valor).

tratamiento(Numero) :-
  display(Numero),
  nl.

bucle :-
  generador(1,10,Numero),
  tratamiento(Numero),
  fail.
bucle.
```

En este caso no hemos utilizado filtro. El predicado generador/3 se encarga de generar los números del uno al diez. El predicado display/1 está predefinido y simplemente escribe un término por la salida standard. El predicado nl/0 también está predefinido y se encarga de escribir un salto de línea por la salida standard.

La negación por fallo

La negación en Prolog consiste en un predicado predefinido llamado `'\+'/1`. La negación recibe como argumento un objetivo. Si dicho objetivo tiene éxito la negación falla y viceversa. Por ejemplo: `\+ (X > 5)` es equivalente a `X =< 5`.

Parece simple, pero la negación encierra una pequeña trampa. Dicha negación no es la negación lógica sino la **negación por fallo**.

Esto significa que Prolog **asume** que aquellos objetivos que no tienen solución (fallan) son falsos. Esto se denomina asunción de mundo cerrado porque damos por supuesto que todo aquello que no se puede deducir (porque no ha sido escrito en el programa) es falso.

La negación por fallo solamente coincide con la negación lógica cuando el objetivo negado es un término **cerrado** (no contiene variables libres). El programador es el responsable de asegurarse esta condición.

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

Ejemplo

Consideremos el siguiente programa:

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

informatico(luis).

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

Y ahora ejecutamos el siguiente objetivo: `\+ hobby(X,musica)`, es decir, queremos saber a quién no le gusta la música como hobby.

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

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

Entrada/Salida

Streams

Como en todos los lenguajes, en Prolog existe la posibilidad de manejar entrada/salida, esto es, ficheros, pantalla, impresoras, etc. Todo ello se hace a través de **streams** (concepto idéntico al de otros lenguajes).

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

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

Existen dos grupos de predicados para manejar streams:

- Aquellos donde el stream se indica explícitamente. Se suelen utilizar para manipular ficheros y sockets.
- Aquellos donde el stream es ímplicito. Manipulan la entrada y salida standard, lo que habitualmente supone el teclado y la pantalla.

Entrada/Salida standard

Cada vez que ejecutamos un programa, éste tiene predefinidos tres streams correspondientes a la entrada, salida y error standard. Los siguientes predicados sirven para conocer cuales son dichos streams:

Predicado	Modo de uso	Descripción
current_output/1	current_output(-Stream)	Retorna el stream asociado a la salida standard
current_input/1	current_input(-Stream)	Retorna el stream asociado a la entrada standard

Un programa siempre puede cambiar dichos streams para que se dirijan a otros dispositivos. El nuevo stream debe obtenerse mediante alguno de los predicados que veremos más tarde.

Predicado	Modo de uso	Descripción
set_output/1	set_output(+Stream)	Cambia el stream asociado a la salida standard
set_input/1	set_input(+Stream)	Cambia el stream asociado a la entrada standard

Entrada standard

El predicado estrella para leer de la entrada standard es `read/1` cuyo modo de uso es `read(-Term)`. Dicho predicado es capaz únicamente de leer términos que estarán separados unos de otros por un punto (`.`) y un salto de línea, igual que cuando tecleamos en el top-level shell. Cuando ya no hay más términos para leer en la entrada standard, el predicado retorna el término `end_of_file/0`.

Si leer términos no es lo nuestro, simpre podemos leer caracteres uno a uno mediante el predicado `get/1`. Su modo de uso es `get(-Char)` y retorna el código ASCII del caracter leido. Cuando ya no hay más caracteres retorna `-1`.

Salida standard

Para escribir en la salida standard disponemos del predicado `write/1` que recibe un término como argumento. Análogamente a la entrada standard, el predicado `put/1` escribe caracteres simples, recibiendo como argumento el codigo ASCII correspondiente.

Adicionalmente existen los predicados `display/1` cuyo efecto es el mismo que `write/1`, y el predicado `nl/0` que escribe un salto de línea en la salida standard.

Ejemplo

Este es un programa que lee números de la entrada standard (separados por punto) hasta que se encuentra algún término que no sea un número. Entonces suma todos los números y escribe el resultado por la salida standard.

```
sumar_lista([],Parcial,Parcial).

sumar_lista([Num|Resto],Parcial,Result) :-
  NParcial is Num+Parcial,
  sumar_lista(Resto,NParcial,Result).

leer_numero(Num) :-
  read(Num),
  number(Num).

suma :-
  suma_aux([]).

suma_aux(Lista) :-
  leer_numero(Num),
  suma_aux([Num|Lista]).

suma_aux(Lista) :-
  sumar_lista(Lista,0,Result),
  nl,
  display('LA SUMA ES : '),
  display(Result),
  nl.
```

Entrada/Salida explícita

A diferencia de la entrada/salida standard, estos predicados podrían no estar predefinidos, en tal caso deben ser importados previamente. La biblioteca a importar puede variar de un entorno de desarrollo a otro, por lo que debe consultar la documentación correspondiente. El modo de importar bibliotecas también varía de un sistema a otro.

El predicado read_term/3

Modo de uso: `read_term(+Stream,-Termino,+Opciones).`

Este predicado sirve para leer términos del stream indicado. Su peculiaridad radica en que podemos indicar algunas opciones sobre como leer el término. El argumento Opciones consiste en una lista (posiblemente vacía) de términos que pueden ser cualquiera de los siguientes:

- `variables(-ListaVariables)`. Unifica ListaVariables con la lista de variables libres que aparecen en el término leido.
- `variable_names(-ListaNombres)`. Unifica ListaNombres con una lista de terminos cero-arios que representan los nombres de las variables libres que aparecen en el término leido.
- Otras opciones dependen del entorno de desarrollo usado. Consulte la documentación asociada.

Recuerde que, cuando ocurre el evento de fin de fichero, este predicado retorna el término `end_of_file/0`.

Ejemplo

```
?-
current_input(I),read_term(I,Term,[variables(L1),variable_names(L2)]).
  |:  t(A,b(C,A)).

I = user_input,
L1 = [_A,_B],
L2 = ['A'=_A,'C'=_B],
Term = t(_A,b(_B,_A)) ?
yes
?-
```

El predicado write_term / 3

Modo de uso: `write_term(+Stream,+Termino,+Opciones).`
Este predicado es similar al anterior puesto que nos permite escribir un término en un stream controlando una serie de opciones. El argumento Opciones debe ser instanciado a una lísta (posiblemente vacía) de términos que se encontrarán entre alguno de los siguientes:

- `quoted(true)`. Aquellos functores que contengan "caracteres raros" (no alfanuméricos) serán escritos entre comillas simples. Esta opción es muy importante si los términos van a ser leidos posteriormente con `read_term/3` o similar.
- `ignore_ops(true)`. Normalmente, aquellos functores que esten declarados como operadores se escriben de manera infija o postfija según corresponda. Con esta opción obligamos a que siempre se escriban de manera prefija. Por ejemplo, `a+b` se escribiría forzosamente como `+(a,b)`.
- Otras opciones dependen del entorno de desarrollo usado. Consulte la documentación asociada.

Ejemplo

```
?-
current_output(O),write_term(O,'1 y 2'+Z,[quoted(true),ignore_ops(true)]).

+('1 y 2',_347)
O = user_output ?
yes
?-
```

Ficheros

Para manipular ficheros solamente necesitamos abrirlos y/o cerrarlos. Cuando abrimos un fichero obtenemos un stream que podemos manejar mediante los predicados anteriormente descritos.

Apertura de ficheros

Se hace mediante el predicado `open/3`, con el modo de uso `open(+Fichero,+Modo,-Stream)`.
Fichero es un término cero-ario con la ruta del fichero (normalmente entre comillas simples).
Modo puede ser uno de estos términos:

- `read/0`. Abrir en modo de solamente lectura.
- `write/0`. Abrir en modo de solamente escritura. Si el fichero contiene datos, se truncan.
- `append/0`. Abrir en modo de solamente escritura. El fichero no se trunca, los nuevos datos se añaden al final.

Cierre de ficheros

Para ello existe el predicado `close/1`, cuyo modo de uso es `close(+Stream)`. El stream queda invalidado una vez que se cierra el fichero. Este predicado vuelca a disco los contenidos del buffer antes de cerrarlo. Si deseamos hacer esto sin cerrar el fichero, disponemos del predicado `flush_output/1`.

Culturilla

- Cuando `read/1` intenta leer un término pero se encuentra con un error sintáctico (el término está mal construido), emite un mensaje de error y el predicado falla. Afortunadamente, hay medios para cambiar este comportamiento, que veremos más adelante.
- Nada impide que `read/1` lea un término que contenga **variables libres**. Tenga en cuenta que dichas variables serán siempre **distintas** a las variables del programa en ejecución y de las variables leídas en llamadas anteriores a `read/1`, independientemente del nombre que se les ponga a dichas variables. Naturalmente, si una variable aparece dos veces en el mismo término leido, en efecto se tratará de la misma variable.

- Cuando la entrada standard es el teclado e intentamos leer de él, Prolog muestra un símbolo de "prompt" para indicarnos que está preparado para que tecleemos. Dicho símbolo tiene este curioso aspecto: ¦