Tutorial básico de programación en Prolog

Autor: Angel Fernández Pineda

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

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 váriable 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.

Elementos del lenguaje

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 váriable 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.

Ejecutando cosas

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

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

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 modos de 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
    - Expresiones válidas

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 m odos 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.