Programación lógica

Un recorrido por la programación lógica y uno de sus lenguajes más representativos: Prolog, clásico de la inteligencia artificial, que se aplica de múltiples formas en el desarrollo de software comercial.

La programación lógica, junto con la funcional, forma parte de lo que se conoce como **programación declarativa**. En los lenguajes tradicionales, la programación consiste en indicar *cómo* resolver un problema mediante sentencias; en la programación lógica, se trabaja de una forma descriptiva, estableciendo relaciones entre entidades, indicando no cómo, sino qué hacer. La ecuación de Robert Kowalski (Universidad de Edimburgo) establece la idea esencial de la programación lógica: *algoritmos = lógica + control*. Es decir, un algoritmo se construye especificando conocimiento en un lenguaje formal (lógica de primer orden), y el problema se resuelve mediante un mecanismo de inferencia (control) que actúa sobre aquél.

Prolog

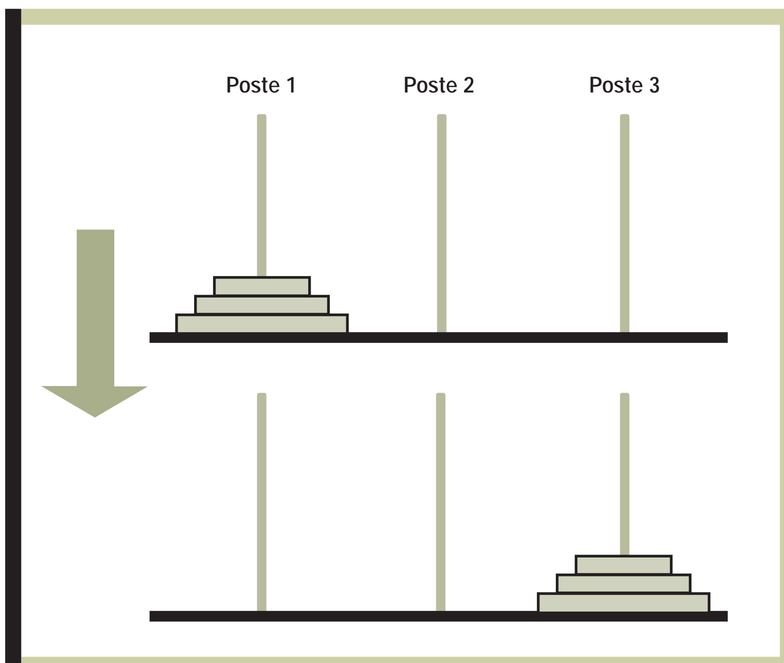
El lenguaje Prolog, principal representante del paradigma, se basa en un subconjunto de la lógica de primer orden (restricción de la forma clausal de la lógica denominada **cláusulas de Horn**). Philippe Roussel y Alain Colmerauer (Universidad de Aix-Marseille) lo crearon en 1972, y su base teórica se debe en gran parte a Kowalski.

Estructuras básicas

Prolog cuenta con dos tipos de estructuras: términos y sentencias. Los términos pueden ser constantes, variables o functores:

> Las **constantes**, representadas por una cadena de caracteres, pueden ser números o cualquier cadena que comience en minúscula.

> Las **variables** son cadenas que comienzan con una letra mayúscula.



[Figura 1] **Problema y resolución de las torres de Hanoi.**

> Los **functores** son identificadores que empiezan con minúscula, seguidos de una lista de parámetros (términos) entre paréntesis, separados por comas.

Las sentencias son reglas o cláusulas. Hay hechos, reglas con cabeza y cola, y consultas.

> Un **hecho** establece una relación entre objetos, y es la forma más sencilla de sentencia. Por ejemplo:

```
humano (socrates).
ama (juan,maría)
```

Se establece que Sócrates *es humano* y que Juan *ama* a María.

> Una **regla** permite definir nuevas relaciones a partir de otras ya existentes. Si queremos establecer que todo humano es mortal, en lógica estándar escribiríamos *∀(x)(humano(x)=>mortal(x))*, mientras que en Prolog escribimos:

```
mortal(X):-humano(X).
```

Esto se lee: *X (variable) es mortal si X es humano.* El símbolo :- significa "**si**" o, si lo leemos de derecha a izquierda, **entonces** o **implica**. En esta regla, *mortal(X)* es la **cabeza**, y *humano(X)* es el **cuerpo**.

> Para entender el concepto de **consulta**, veamos un ejemplo. En lógica estándar:

∀(x)(humano(x)=>mortal(x))
humano(socrates)
entonces mortal (socrates)

Partiendo de que *los humanos son mortales* y de que *Sócrates es humano*, deducimos que *Sócrates es mortal.* Para realizar esa deducción en Prolog, hay que preguntar si *es mortal Sócrates*, o *quién es mortal.* Si del programa lógico (conjunto de hechos y reglas) se deduce que Sócrates es mortal, entonces ésa será la respuesta que obtendremos. Veamos el programa:

GERARDO ROSSEL
Investigador del CAETI.
grossel@computer.org

```
mortal(X):-humano(X).
humano(Sócrates).
```

Para preguntar interactivamente, los ambientes de Prolog tienen un **listener**, un intérprete de línea de comando cuyo prompt es un signo de pregunta. Se introduce una sentencia (eventualmente con variables), y Prolog intentará demostrarla (usando un algoritmo de inferencia llamado SLD-Resolution, basado en la regla de resolución de Robinson), buscando además constantes que puedan reemplazar las variables de la pregunta. Las preguntas de nuestro ejemplo serían:

```
?- mortal(Sócrates).
Yes.
```

```
?- mortal(X)
X = socrates
```

Las segundas líneas son las respuestas de Prolog: primero, afirmando que *sí*, Sócrates es mortal; después, contestando *Sócrates* al preguntar quién es mortal. Si agregamos más conocimiento a nuestro programa lógico (por ejemplo, que Platón es mortal), podemos pedir más respuestas. Si luego de obtener *Sócrates* escribimos un punto y coma (así "pedimos más"), nos responderá *Platón*. Cuando Prolog no tenga más respuestas, nos dirá que *No*; esa negativa no significa que lo que preguntemos sea falso, sino que no lo conoce o no puede demostrarlo. Es decir que si preguntamos, por ejemplo, si Arquímedes es mortal, la respuesta será que no, pero porque nuestro programa no cuenta con el conocimiento suficiente. Esto se denomina **negación por falla**.

```
?- mortal(X)
X = socrates;
X = platon;
No
```

Este modo interactivo es muy útil para prueba y prototipación, pero dependiendo de la implementación Prolog en particular, es posible realizar invocaciones desde otros ambientes, o ejecutar un programa Prolog que tenga un predicado *main* (que será el que se intentará probar). Para facilitar el desarrollo de aplicaciones, Prolog cuenta con un conjunto de características que "ensucian" de alguna manera su pureza en lo que hace al paradigma, pero que lo vuelven utilizable. Hablamos, por ejemplo, de escritura por pantalla, pedido de datos al usuario y otros elementos específicos de control del mecanismo de inferencia (el predicado de corte es el más conocido).

Listas

Prolog manipula listas con una facilidad sintáctica que simplifica la programación: una lista es un par ordenado, donde un elemento es un término (la *cabeza*), y el otro puede ser un término, una lista o la lista vacía (la *cola*). Las listas van entre corchetes, y la cabeza se separa de la cola con el símbolo *pipe*; la lista vacía se indica con dos corchetes separados por un espacio. Haciendo un abuso de notación, también es posible enumerar todos los elementos de la lista separándolos con comas. Por ejemplo, la lista *1, 2, socrates, a, platon* se escribe en Prolog de estas dos maneras:

```
[1|[2|[socrates|[a|[platon|[]]]]]]
[1, 2, socrates, a, platon]
```

Para comprender un poco mejor el poder de la programación declarativa, comparemos un algoritmo que determina si un elemento se encuentra en una lista en un lenguaje declarativo (por ejemplo, Pascal) y en Prolog. En Pascal:

```
function member(item:Integer; L:Array
 of Integer):Boolean;
var
  i:Integer;
begin
  i:=0;
  while((i <= length(L)) and (L[i] <> item)) do
     begin
        i := i+1;
     end;
  Result := item = L[i];
end;
```

Para saber si un elemento es parte de una lista, en Prolog sólo declaramos que es su cabeza o es parte de su cola.

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

El predicado *member* (parte del estándar de Prolog) dice que un elemento (representado por la variable *X*) es miembro de una lista si es la cabeza de ella (la cabeza de la lista [*X*/*Xs*] es *X*), y también que un elemento es miembro de una lista si es miembro de la cola (la cola de la lista [*Y*/*Ys*] es *Ys*, que es,

Operadores

MATEMATICOS

+	Suma
-	Resta
*	Multiplicación
/	División (retorna siempre en punto flotante)
//	División entera (trunca)
mod	Resto de división
**	Potenciación

RELACIONALES

>	Mayor que
<	Menor que
>=	Mayor o igual que
=<	Menor o igual que
=:=	Aritméticamente igual
=\=	Aritméticamente diferente

a su vez, una lista, un término o la lista vacía, según definimos anteriormente). En un caso indicamos *cómo* determinar que un elemento es parte de la lista, mientras que en el otro declaramos *qué significa* que un elemento sea parte de la lista. En esta última situación, el motor de inferencia se encargará de responder cuando preguntemos.

Operadores aritméticos y relacionales

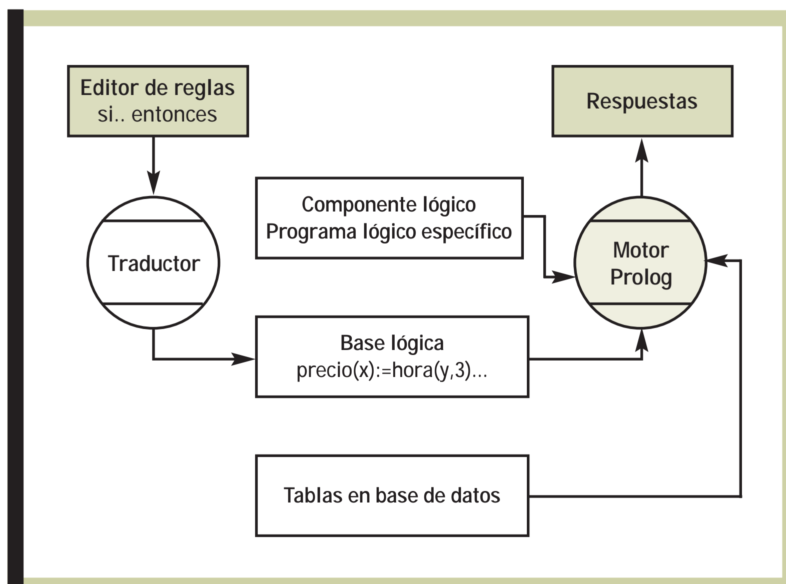
El predicado *igual* (=) compara dos términos, y es verdadero solamente cuando ambos son idénticos o cuando alguna sustitución de variables los hace idénticos (es más correcto decir "cuando unifican"). Las siguientes consultas nos dan un ejemplo:

```
?- 1 = 1.
yes
?- 1 = 2.
no
?- 2 = 1+1.
no
?- a=A.
A = a ;
No
```

Puede parecer extraño que la consulta 2 = 1+1 nos arroje un *no* como respuesta; en realidad, la constante 2 no unifica con el predicado + con dos argumentos 1 y 1. Para estas comparaciones, y para conseguir asignaciones, se usa el predicado *is*:

```
?- 2 is 1+1.
yes
?- X is 1+2.
X = 3 ;
no
?- 1 is X+2.
Error
```

En el primer caso, 1 + 1 es 2, el *is* responde *yes*. En el segundo, responde que X=3 es la sustitución adecuada a X = 1+2; al pedir otra respuesta (con el punto y coma), no la encuentra. El último ejemplo es un error: al usar *is*, la parte derecha debe estar instanciada. El estándar establece que en otro caso es un error, aunque algunas distribuciones de Prolog responden *no*. En la tabla [Operadores] vemos otras posibilidades.



[Figura 2] **Reglas de negocios y Prolog.**

Las torres de Hanoi

Este problema puede mostrar la potencia del paradigma lógico. Se comienza ordenando los discos de mayor a menor (de abajo hacia arriba) sobre un eje, y se trata de llevarlos a otro (del primero al último) para que queden de la misma forma. Es posible desplazar los discos de a uno (siempre tomando el de arriba), y nunca puede ubicarse un disco sobre otro de menor diámetro. La [Figura 1] muestra los estados inicial y final, luego de realizar los movimientos. Existen tres postes, y uno de ellos se usa como auxiliar (con sólo dos sería imposible). Se trata, entonces, de hacer un programa que pueda mover *N* discos desde el poste 1 hasta el poste 3, usando el poste 2 como auxiliar. Para lograrlo, establecemos una regla cuya cabeza sea el predicado Hanoi con un parámetro indicando la cantidad de discos por mover; en el cuerpo de la regla pondremos el predicado *mover* con los parámetros: cantidad de discos (N), desde dónde (poste 1), hasta dónde (poste 3) y qué usaremos como auxiliar (poste 2).

```
hanoi(N) :- mover(N, poste1,
  poste3, poste2).
```

Luego debemos especificar *mover*, para lo cual usaremos dos reglas. Trabajar en Prolog nos exige pensamiento recursivo: podemos decir que, para mover N discos desde el poste 1 al 3 usando el 2 como auxiliar:

- deben moverse N-1 discos del poste 1 al poste 2, usando el 3 como auxiliar;
- debe moverse el disco restante al poste 3, quedando en el poste correcto;
- luego hay que mover N-1 discos del poste 2 al poste 3 usando el 1 como auxiliar.

Asumimos que es fácil mover N-1 discos, pero podemos aplicar este razonamiento y quedarnos con N-2, y así sucesiva o recursivamente. Esto se implementa en una regla:

```
mover(N,A,B,C) :-
   M is N-1, mover(M,A,C,B),
    escribir_mov(A,B), mover(M,C,B,A).
```

No olvidar el caso base: cuando no hay discos para mover, no hay que hacer nada. Esa es justamente la regla que aquí necesitamos:

```
mover(0,_,_,_).
```

El carácter _ representa variables anónimas (en este caso no nos importa con qué postes

Enlaces relacionados

AMBIENTES PROLOG:

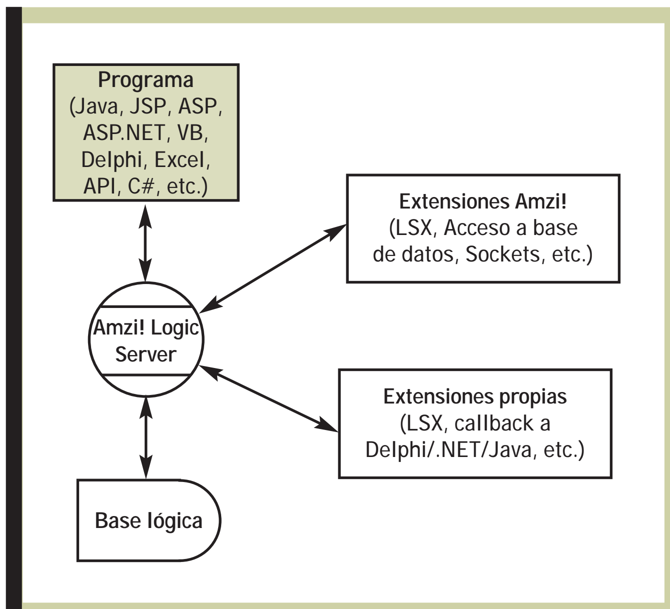
- > Amzi! Prolog: www.amzi.com
- > SWI Prolog: www.swi-prolog.org
- > VisualProlog: www.visual-prolog.com
- > Logic Programming Associates: www.lpa.co.uk
- > GNU Prolog: gprolog.inria.fr

RECURSOS

- > Libro en inglés: cblpc0142.leeds.ac.uk/~paul/prologbook
- > Association for Logic Programming (ALP): www.cs.kuleuven.ac.be/%7Edtai/projects/ALP
- > Blog en español: programacionlogica.blogspot.com

trabajamos, ya que no hay discos). Usamos, además, un predicado auxiliar que escriba por pantalla el movimiento actual (se usa *write*, que es una facilidad extra lógica, y *nl*, que significa *new line*). El resultado final:

```
% Torres de Hanoi %
hanoi(N) :- mover(N, poste1, poste3, poste2).
mover(0,_,_,_).
mover(N,A,B,C) :-
   M is N-1, mover(M,A,C,B), escribir_mov(A,B),
    mover(M,C,B,A).
escribir_mov(Desde,Hasta) :-
   write('mover desde: '),
   write( Desde ),
   write(' hasta: '),
   write(Hasta),
   nl.
```



[Figura 3] Interacción a través de Amzi! Logic Server.

Reglas de negocios y Prolog

Uno de los aspectos críticos en el desarrollo de aplicaciones empresariales es el de la programación de reglas de negocio. Existen numerosos trabajos acerca de la mejor forma de modelarlas: el problema se relaciona con el tipo de conocimiento que requieren. Tenemos tres tipos:

- > **Factual.** Puede ser modelado e implementado como datos (por ejemplo, filas en una tabla relacional). Establece hechos.
- > **Procedural.** Serie de pasos para resolver problemas.
- > **Lógico.** Representa un conjunto de reglas para modelar relaciones entre objetos (por ejemplo, en una aplicación de tarifas telefónicas, serían las relaciones entre horarios, distancia, descuentos, plan del usuario, tiempo de duración, costo de la llamada, etc.).

Los dos primeros tipos de conocimiento pueden modelarse e implementarse simplemente con las herramientas tradicionales de bases de datos y lenguajes imperativos (C#, Java, Smalltalk, Eiffel, etc.), pero ni el paradigma de orientación a objetos provee una semántica adecuada para modelar conocimiento lógico. Este tipo no es fácil de representar en un esquema tradicional: no es suficiente una base de datos, dado que las relaciones son demasiado complejas para una tabla. Tampoco es adecuado el modelo procedural, ya que fuerza a transformar relaciones lógicas en secuencias de instrucciones (en este último aspecto, hay que diferenciar un conjunto de reglas *si-entonces* de un conjunto de instrucciones de un lenguaje procedural *if-then*; en la primera no hay sentido de secuencia explícito; en la segunda, sí). Prolog, así como otras extensiones específicas, es adecuado para modelar este tipo de conocimiento lógico. Una implementación eficiente de dicho conocimiento lógico puede requerir un paso intermedio: modelar primero reglas de negocio en un formato claro para los encargados de establecerlas (alguna sintaxis tipo *si-entonces*) y luego trasladarlas automáticamente a un programa lógico que use las facilidades del motor de inferencias. Veámoslo gráficamente en la [Figura 2].

Integración total: lógica para .NET y Java

Para que el uso de programas lógicos pueda aplicarse al desarrollo de software moderno, es necesario que sean verdaderos **componentes lógicos** que puedan utilizarse desde entornos como .NET, JSP, Delphi, Java, etc. Muchas implementaciones actuales de Prolog proveen mecanismos para invocar programas Prolog desde otros ambientes. El entorno **Amzi! Prolog** + **Logic Server** brinda una integración total multiplataforma (ver [Figura 3]) con los ambientes más conocidos. Además, podemos invocar desde Prolog rutinas hechas en otros lenguajes, mediante predicados extendidos (así Prolog puede actualizar, por ejemplo, el estado de un botón en pantalla habilitándolo o no dependiendo de un complejo conjunto de relaciones lógicas). Naturalmente, existen diversos IDEs muy completos para el desarrollo de programas Prolog: el Amzi! utiliza Eclipse, muy conocido por la comunidad Java.

Conclusiones

Hemos presentado las características principales de la programación lógica encarnadas en su principal representante: Prolog. Luego del ímpetu inicial con que contó, hoy se la puede ver como un paradigma adecuado para determinados dominios, como sistemas expertos y reglas de negocios. Integrándolo con ambientes de desarrollo, podemos dotar a nuestras aplicaciones de inteligencia y flexibilidad. Existen ampliaciones con soporte para otras lógicas, como la difusa, para la programación con restricciones, soporte de paralelismo y concurrencia, etc. ●

Gerardo Rossel es Licenciado en Ciencias de la Computación y doctorando de la Facultad de Ciencias Exactas y Naturales de la UBA. Es docente e investigador (a cargo del Grupo de Investigación y Desarrollo de Agentes y Sistemas Inteligentes del CAETI) en la Facultad de Tecnología Informática de la UAI. En el campo profesional, donde actúa desde hace más de quince años, es Chief Scientist y responsable de ingeniería en Uppersoft Ingeniería de Software, representante de Amzi! Prolog en Sudamérica, y se especializa en inteligencia artificial, desarrollo de software (.NET y J2EE) y el diseño por contratos. Además, es consultor de proyectos especiales para Omnisis S.A.

Ejemplo: Java y Prolog

Veamos un caso real de utilización de Prolog en una aplicación Java: se trata de una compañía que brinda servicios para manejar el financiamiento de propiedades. El centro de sus servicios es una aplicación web que permite a sus clientes buscar la mejor solución para un préstamo hipotecario. El módulo de cotizaciones fue desarrollado utilizando 5000 líneas de código Java y varias tablas de una base de datos; es el más crítico y el que soporta el mayor peso de las reglas de negocio. Era necesario cambiarlo todo el tiempo para adaptarse a nuevas reglas y factores de tasación. A esto se agregaba un largo ciclo de afirmación de calidad, dado que la modificación de código procedural para realizar las adaptaciones era proclive a producir errores nuevos. Se precisaba una solución con menos errores y que permitiera una rápida adaptación a nuevas reglas: se reemplazó el módulo de tasaciones, construyendo un módulo lógico con Amzi!, y en dos meses las 5000 líneas Java y las 18 tablas de la base habían dado lugar a sólo 500 líneas Prolog. La base lógica resultante estaba casi libre de errores, y el ciclo de modificación/prueba se redujo en gran forma. El resto de la aplicación sigue en Java, aunque se planea migrar módulos particularmente complejos.

Esta Pocket PC puede ser TUYA!

Y ganá doble:

Porque visitando www.msdntrial.com y con sólo registrarte participás por el sorteo de una **Pocket PC HP IPAQ 1930.**

Además vas a ingresar a un sitio que te da acceso a los trials de los mejores programas de **software empresarial Microsoft** y la posibilidad de conocer en profundidad los beneficios de la Suscripción MSDN. Un producto que te da acceso al mayor repositorio de software de Microsoft del mundo y a soporte técnico para desarrolladores.

Te esperamos en www.msdntrial.com, participá y ganá!

Microsoft
Tu potencial. Nuestra pasión.™

msdn subscriptions

Para más información de la Suscripción MSDN comunicate al 011-4317-2606.
Promoción válida en Argentina, desde el 15/3/2004 hasta el 15/12/2004. Hay en juego 1 (una) Pocket Pc Hewlett Packard IPAQ 1930. Consulte bases y condiciones en http://www.msdntrial.com/promo.html