Introducción a Prolog

Prolog es un lenguaje de programación lógica cuya primera versión fue desarrollada a principios de la década de 1970 por Colmerauer en la universidad de Marsella. Contrariamente a otros lenguajes de programación basados es estructuras de control y definición de funciones para calcular resultados, Prolog está orientado a la especificación de relaciones para responder consultas. En ese sentido Prolog es similar a un sistema de base de datos, aunque en el contexto de la inteligencia artificial se prefiere hablar de bases de conocimiento, enfatizando la complejidad estructural de los datos y de las deducciones que se pueden obtener de ellos.

Por ejemplo, para especificar la relación el padre de `X` es `Y`, se crea una base de conocimiento con *hechos* expresados mediante un predicado `padre(X,Y)` de la siguiente manera:

```
padre(juan,pedro).
padre(josé,pedro).
padre(maría,pedro).
padre(pedro,pablo).
padre(ana,alberto).
...
```

Esto es muy parecido a crear una tabla en una base de datos, sólo que cada caso se especifica mediante una cláusula independiente terminada por '.' . Si además se quiere incorporar conocimiento sobre la madre, se puede proceder de la misma manera agregando por ejemplo:

```
madre(juan,ana).
madre(josé,ana).
madre(maría,ana).
madre(pedro,juanita).
madre(ana,julia).
...
```

Esto corresponde a definiciones por extensión (caso a caso) de la relación padre y madre. Suponiendo que esta base de conocimiento está almacenada en `genealogia.pl`, para responder consultas respecto a ella se debe cargar en el ambiente de ejecución de Prolog de la siguiente manera (se subraya lo ingresado por el usuario):

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.1.13)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult(genealogia).
% genealogia compiled 0.00 sec, 2,704 bytes

Yes
```

```
?-
```

En prolog todas las cláusulas terminan con el delimitador '.' . Las cláusulas de consulta se interpretan como ecuaciones lógicas y pueden incluir variables. Por ejemplo:

```
?- padre(maría,pablo).

No
?- padre(ana,alberto).

Yes
?- padre(maría,X).

X = pedro ;

No
?- padre(X,Y).

X = juan
Y = pedro ;

X = josé
Y = pedro ;

X = maría
Y = pedro ;

X = pedro
Y = pablo ;

X = ana
Y = alberto ;

No
```

Prolog genera soluciones a estas ecuaciones lógicas dándole valores a las variables. Cuando la consulta no tiene variables, como en los dos primeros casos del ejemplo anterior, la respuesta es polar (`Yes` o `No`). Cuando hay variables, Prolog entrega la secuencia de soluciones. Para ver la lista de soluciones se debe presionar ';' (el último `No` indica que no hay más soluciones). Las variables son identificadores que empiezan siempre con una letra mayúscula. Los identificadores con letra inicial minúscula corresponden a nombres de predicados o átomos. La forma general de una consulta consiste en una secuencia de predicados que deben ser satisfechos conjuntamente en el orden especificado. Esto permite consultas complejas similares al *join* en bases de datos, por ejemplo para determinar el abuelo paterno de un individuo, aquí `maría`, se puede plantear la consulta:

```
?- padre(maría,X),padre(X,Y).

X = pedro
```

```
Y = pablo ;

No
```

Primero se obtiene el padre de `maría` en `X` y luego con ese valor el padre de `X` en `Y`, que corresponde al dato buscado. El "join" se logra compartiendo la variable `X`. Prolog resuelve consultas complejas encontrando una solución para el primer predicado y luego, con el valor obtenido para las variables, procede con el resto de la consulta. En el ejemplo anterior, el primer predicado `padre(maría,X)` produce como solución `X = pedro`. Al substituir el valor de `X` en el segundo predicado la consulta queda como `padre(pedro,Y)` lo que produce la solución `Y = pablo`. Además, Prolog intenta encontrar otras soluciones a la consulta haciendo *backtracking* (en particular cuando el usuario presiona ';'). Esto consiste en buscar otra solución para el último predicado y, si ya se agotaron, volver al predicado anterior y así succesivamente. En el ejemplo anterior, solo existe una solución para cada predicado. Sin embargo la consulta también habría podido plantearse en el orden inverso:

```
?- padre(X,Y),padre(maría,X).

X = pedro
Y = pablo ;

No
```

Se obtiene el mismo resultado pero en este caso el proceso es más complejo. La primera solución para el predicado `padre(X,Y)` es `{X = juan, Y = pedro}`, con esto se intenta resolver `padre(maría,juan)` lo que falla, *backtracking*, la segunda solución es `{X = josé, Y = pedro}` con lo que se intenta resolver `padre(maría,josé)`, falla, *backtracking*, la tercera solución es `{X = maría, Y = pedro}` con lo que se intenta resolver `padre(maría,maría)`, también falla, *backtracking*, la cuarta solución es `{X = pedro, Y = pablo}` con lo que se intenta resolver `padre(maría,pedro)`, funciona, se muestra el resultado. Si el usuario presiona ';', *backtracking*, la quinta solución para `padre(X,Y)` es `{X = ana, Y = alberto}` con lo que se intenta resolver `padre(maría,ana)`, falla y ya no hay más soluciones. Este proceso de búsqueda de la solución se puede explicitar intercalando un predicado para desplegar resultados intermedios como sigue:

```
?- padre(X,Y),print([X,Y,padre(maría,X)]),padre(maría,X).
[juan, pedro, padre(maría, juan)][josé, pedro, padre(maría, josé)]
[maría, pedro, padre(maría, maría)][pedro, pablo, padre(maría, pedro)]

X = pedro
Y = pablo ;
[ana, alberto, padre(maría, ana)]

No
```

Prolog permite abstraer conceptos como la relación abuelo paterno definiendo *reglas* que precisan las condiciones en que se cumple la relación. Estas reglas constituyen una definición por comprensión que se pueden agregar a la base de conocimiento:

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

Una regla puede verse como una consulta empaquetada. El primer término corresponde a la relación que se está definiendo. La parte a la derecha de ':-' indica bajo qué condiciones se cumple la relación definida, es decir, la consulta que se debe hacer. La definición termina con '.' . Un hecho corresponde a un caso particular de regla en que no hay condiciones en la parte derecha, lo que puede escribirse como:

```
padre(juan,pedro) :- true.
```

Las consultas relativas a relaciones definidas por comprensión mediante reglas se realizan de la misma manera que en el caso de definiciones por extensión con hechos. Por ejemplo:

```
?- abuelo_paterno(maría,X).

X = pablo ;

No
```

La única diferencia es que al tratar de satisfacer el predicado se busca una cabeza de regla que calce y se procede a resolver la parte derecha de la regla substituyendo las variables que fue necesario asignar para el calce. Así al consultar `abuelo_paterno(maría,X)` se resuelve la parte derecha de la regla substituyendo `X1 = maria` y `Y1 = X`, es decir `padre(maría,Z1),padre(Z1,X)`. **Nota:** para cada invocación de una regla se crean variables locales a esa invocación precisadas aquí mediante un índice.

Cuando existen varios casos en que una relación se cumple, es decir una disyunción, se debe crear una regla para cada caso. Al intentar responder una consulta sobre la relación definida, Prolog intentará succesivamente con cada caso en el orden de definición. Por ejemplo, si se desea definir la relación hijo, se debe considerar las relaciones padre y madre:

```
hijo(X,Y) :- padre(Y,X).
hijo(X,Y) :- madre(Y,X).
```

Uno de los aspectos más interesantes de Prolog es que las reglas se prestan para definiciones recursivas. Por ejemplo para definir la relación descendiente:

```
descendiente(X,Y) :- hijo(X,Y).
descendiente(X,Y) :- hijo(X,Z),descendiente(Z,Y).
```

Se está precisando que un descendiente es el hijo o un descendiente del hijo. La relación ancestro se puede definir como:

```
ancestro(X,Y) :- padre(X,Y).
ancestro(X,Y) :- madre(X,Y).
ancestro(X,Y) :- padre(X,Z),ancestro(Z,Y).
ancestro(X,Y) :- madre(X,Z),ancestro(Z,Y).
```

Aquí se precisa que un ancestro es el padre, o la madre, o un ancestro del padre, o un ancestro de la madre. También se habría podido definir como:

```
ancestro(X,Y) :- descendiente(Y,X).
```

Para que una relación definida recursivamente esté correcta se necesita por lo menos un caso no recursivo.

Otro aspecto notable de Prolog es que las relaciones pueden establecerse no sólo entre átomos sino que también entre términos estructurados. Un *término* es ya sea un átomo (identificador, número, string, variable) o un nombre de término (o *functor*) asociado a una lista de argumentos $t(t_1,t_2,...,t_n)$. Así, cuando se define una relación entre términos estructurados mediante reglas, lo que en realidad se está haciendo es definir como se construyen los valores de respuesta a las variables de la consulta.

Los términos estructurados se utilizan en particular para representar listas. Internamente, una lista se construye utilizando el functor binario `.(_,_)` y el
átomo `[]` que representa la lista vacía. Sin embargo en general las listas se expresan mediante una notación sintáctica más legible. Así una lista de tres elementos `a`, `b` y `c` se escribe como `[a,b,c]` y se interpreta internamente como el término `.(a,.(b,.(c,[])))`. La sintaxis de notación de listas también autoriza a describir en forma separada los primeros elementos de la lista del resto de la lista separando con '|'. Por ejemplo `[a | [b,c]]` se lee como la lista que empieza con el elemento `a` y sigue con la lista `[b,c]`, lo que es otra manera de representar la misma lista `[a,b,c]`.

```
?- L = .(1,.(2,.(3,[]))).

L = [1, 2, 3]

Yes
?- L = [1|[2,3]].

L = [1, 2, 3]

Yes
?- L = [1,2|[3]].

L = [1, 2, 3]

Yes
```

Teniendo este esquema de representación, se pueden proponer definiciones de relaciones para operar sobre listas. Por ejemplo para concatenar listas está predifinida en Prolog la relación `append(A,B,C)` que indica que concatenado las listas `A` y `B` se obtiene la lista `C`:

```
append([],L,L).
append([E|R1],L,[E|R2]) :- append(R1,L,R2).
```

La primera regla nos dice que concatenado una lista vacía con cualquier lista se obtiene como resultado esta última lista. La segunda regla expresa que al concatenar una lista que empieza con el elemento `E` y sigue con la lista `R1` con una lista `L` se obtiene una lista que empieza con el mismo elemento `E` y sigue con la lista `R2`. La restricción para que se cumpla esta la relación está descrita en la parte derecha de la regla y dice que la lista `R2` debe ser el resultado de la concatenación de `R1` y `L`. En la primera regla no fue necesario considerar restricciones adicionales, por lo que su parte derecha está vacía. La relación definida de aquí se puede utilizar de varias maneras. Para consultar sobre la veracidad de la relación:

```
?- append([1,2],[3,4],[1,2,3,4]).

Yes
```

Para encontrar el resultado de la concatenación:

```
?- append([alfa,[1,2]],[beta,gama],X).

X = [alfa, [1, 2], beta, gama] ;

No
```

Para encontrar prefijos y sufijos:

```
?- append(X,_,[1,2,3]).

X = [] ;

X = [1] ;

X = [1, 2] ;

X = [1, 2, 3] ;

No
?- append(_,X,[1,2,3]).

X = [1, 2, 3] ;

X = [2, 3] ;

X = [3] ;

X = [] ;

No
```

Utilizando `append` se pueden definir otras relaciones, por ejemplo:

```
sublista(S,L) :- append(_,X,L),append(S,_,X).
```

Una sublista `S` está definida como un prefijo de un sufijo `X` de `L`. **Nota:** el orden de las condiciones de la regla es importante ya que la consulta `append(S,_,X)` generaría una infinidad de soluciones si estuviera en primera posición debido a que la variable `X` no estaría *instanciada* (asociada a un valor). Generalmente en Prolog se debe tener cuidado de colocar primero la condición más restrictiva, es decir aquella que genera menos soluciones. Aquí primero se encuentra un sufijo, después un prefijo de ese sufijo.

La definición anterior de `sublista` entrega también como resultado la lista vacía, varias veces. Estas respuestas se pueden evitar redefiniendo la relación:

```
sublista([E|R],L) :- append(_,X,L),append([E|R],_,X).
```