El lenguaje de programación *Prolog*

Materia: Análisis Comparativo de Lenguajes

Departamento de Informática
Universidad Nacional de San Luis (UNSL)
San Luis. Argentina

1. Introducción

Prolog deriva su nombre de **Pro**gramming in **log**ic, es decir, la idea surgida en los 70's de usar a la lógica como lenguaje de programación. Prolog es un lenguaje de programación centrado alrededor de un conjunto pequeño de mecanismos básicos que incluyen: concordancia (*matching*) de patrones, estructuración de datos basada en *árboles* y *backtracking* automático.

Prolog es un lenguaje pensado para la computación simbólica, no numérica. Se adapta particularmente bien a problemas que involucran *objetos* y *relaciones* entre objetos. Estas características han convertido a Prolog en un lenguaje poderoso para aplicaciones de Inteligencia Artificial y programación no numérica en general.

2. Definición de relaciones mediante hechos

La forma más sencilla de especificar una relación en Prolog es mediante un conjunto de *hechos*. Cada hecho consta del nombre de la relación (que comienza con una letra minúscula), seguido por una lista de argumentos separados por coma y encerrados entre paréntesis. Cada hecho finaliza con un punto. Ejemplo: el hecho de que Tom es progenitor de Bob se puede escribir en Prolog como:

```
progenitor(tom,bob).
```

En este caso hemos elegido `progenitor` como nombre de la relación (binaria) y a `tom` y `bob` como sus argumentos. El siguiente programa Prolog define esta relación familiar en forma completa:

```
progenitor(pam,bob).
progenitor(tom,bob).
progenitor(tom,liz).
progenitor(bob,ann).
progenitor(bob,pat).
progenitor(pat,jim).
```

Una vez que este programa ha sido comunicado al sistema Prolog, se pueden realizar distintas consultas referidas a la relación `progenitor`. Por ejemplo, para preguntar: ¿Bob es progenitor de Pat?, la consulta puede ser realizada al sistema Prolog tipeando en la terminal:

```
?- progenitor(bob,pat).
```

A lo que Prolog responderá:

```
yes
```

Otra consulta puede ser:

```
?- progenitor(liz,pat).
```

A lo que Prolog responderá:

```
no
```

o bien

```
?- progenitor(tom,ben).
no
```

La pregunta ¿Quién es el progenitor de Liz? puede ser realizada tipeando:

```
?- progenitor(X,liz).
```

En este caso, Prolog no sólo responde `yes` o `no`, sino que responde cual es el valor de `X` que hace la sentencia previa verdadera. La respuesta es:

```
X=tom
```

La pregunta ¿Quiénes son los hijos o hijas de Bob? puede ser comunicada a Prolog como:

```
?- progenitor(bob,X).
```

En este caso hay más de una respuesta. La primera respuesta de Prolog es:

```
X=ann
```

Si requerimos otra solución, tipeamos un punto y coma (“;”) y Prolog responde:

```
X=pat
```

Si solicitamos otra respuesta (tipeando un ;) la respuesta será `no`. También es posible preguntar ¿Quién es progenitor y de quienes?. Otra forma de plantear la pregunta es diciendo: ”Encontrar los `X` e `Y` tal que `X` es progenitor de `Y`".

```
?- progenitor(X,Y).
X=pam
Y=bob;

X=tom
Y=bob;

X=tom
Y=liz;

...
```

La muestra de soluciones puede ser detenida tipeando un $return$ en lugar de un punto y coma. Para preguntar ¿Quién es abuelo o abuela de Jim?, debemos descomponer la consulta en dos pasos:

¿Quién es un progenitor de Jim?. Asumamos que es algún `Y`.

¿Quién es un progenitor de `Y`?. Asumamos que es algún `X`.

```
?- progenitor(Y,jim),progenitor(X,Y).
X=bob
Y=pat
```

La "," se lee como "y" (and lógico). Alterando el orden de las componentes de la consulta, el significado lógico es el mismo y producirá el mismo resultado.

Ejercicio: Plantear las consultas que permitan saber: a) ¿Quiénes son lon nietos de Tom? y b) ¿Ann y Pat tienen algún progenitor en común?

3. Definición de relaciones mediante reglas

Comencemos agregando información sobre el sexo de las personas del ejemplo anterior:

```
mujer(pam).
hombre(tom).
hombre(bob).
mujer(liz).
mujer(pat).
mujer(ann).
hombre(jim).
```

Las relaciones `mujer` y `hombre` son relaciones unarias, mientras que la relacion `progenitor` era binaria, ya que relacionaba *pares* de objetos.

Si quiciéramos establecer la relación `descendiente`, vemos que ésta es la relacion inversa de `progenitor`. Una alternativa para la definición de esta relación consistiría en ingresar una lista de hechos:

```
descendiente(bob,pam).
descendiente(bob,tom).
....
```

que es igual a la relación `progenitor` pero con sus argumentos invertidos.

Sin embargo, si la relación `progenitor` ya está definida, y sabiendo que `descendiente` es la relación inversa, esta relación puede ser definida en base a la siguiente sentencia lógica:

Para todo X e Y

Y es descendiente de X **si** X es progenitor de Y.

Esta sentencia lógica tiene un formato similar al utilizado por Prolog. La cláusula Prolog que tiene el mismo significado es:

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

Este tipo de cláusulas se denominan *reglas* y también se pueden leer como:

Para todo X e Y

Si X es progenitor de Y **entonces** Y es descendiente de X .

Las reglas especifican cosas que son verdaderas si alguna condición es satisfecha. El símbolo "`:-`" (que se lee "if" o "si") delimita dos partes dentro de una regla, una izquierda y otra derecha. Así, en una regla podemos diferenciar:

- una parte de condición (la parte derecha de la regla).
- una parte de conclusión (la parte izquierda de la regla).

La parte de la conclusión también es llamada la *cabeza* de una cláusula y la parte de condición el *cuerpo* de la cláusula. En el ejemplo de la regla anterior `descendiente(Y,X)` constituye la cabeza de la regla y `progenitor(X,Y)` es el cuerpo de la misma. Esta regla puede ser interpretada como una sentencia que expresa que si la condición `progenitor(X,Y)` es verdadera, una consecuencia lógica de ésto es `descendiente(Y,X)`.

Para comprender la forma en que Prolog utiliza las reglas, veamos que sucede cuando realizamos la siguiente consulta:

```
?- descendiente(liz,tom).
```

Ningún hecho habla sobre la relación `descendiente` por lo que Prolog sólo podrá dar una respuesta en base a la regla definida previamente. Esta regla es general, en el sentido que puede ser aplicada a cualquier objeto `X` e `Y` y por consiguiente puede ser aplicada a objetos particulares como por ejemplo `liz` y `tom`. Para aplicar la regla a `liz` y `tom`, `Y` debe ser substituido con `liz`, y `X` con `tom`. Decimos en este caso, que las variables `X` e `Y` se han *instanciado* a:

```
X=tom , Y=liz
```

Luego de la instanciación hemos obtenido un caso especial de la regla general:

```
descendiente(liz,tom) :- progenitor(tom,liz).
```

La parte de la condición es ahora `progenitor(tom,liz)`. Prolog intenta ahora descubrir si la parte de la condición es verdadera. Así, el objetivo inicial `descendiente(liz,tom)` ha sido reemplazado con el subobjetivo `progenitor(tom,liz)`. Este (nuevo) objetivo es trivial ya que se encuentra como un hecho en nuestro programa. Esto significa que la parte de la conclusión también es verdadera y por lo tanto Prolog responderá la consulta con `yes`.

Ejercicio: Determinar cuales son las relaciones familiares que determinan las siguientes reglas:

```
ma(X,Y) :- progenitor(X,Y), mujer(X).
ab(X,Z) :- progenitor(X,Y) , progenitor(Y,Z).
```

3.1. Reglas recursivas

Supongamos que deseamos agregar a nuestro programa de familia la relación *predecesor*. Si definimos esta relación en términos de la relación *progenitor*, vemos que esta relación tiene un caso elemental que se da cuando una persona `X` es predecesora de una persona `Z` dado que `X` es progenitor de `Z`. En Prolog:

```
predecesor(X,Z) :- progenitor(X,Z).
```

Esta regla sólo sirve para el caso en que `X` sea predecesor directo de `Z`. Pero `X` puede ser un predecesor indirecto de `Z`, es decir, puede existir una cadena de progenitores que conectan `X` con `Z`. Un intento para resolver ésto, sería con el siguiente programa:

```
predecesor(X,Z) :-
      progenitor(X,Z).

predecesor(X,Z) :-
      progenitor(X,Y),
      progenitor(Y,Z).

predecesor(X,Z) :-
      progenitor(X,Y1),
      progenitor(Y1,Y2),
      progenitor(Y2,Z).

predecesor(X,Z) :-
      progenitor(X,Y1),
      progenitor(Y1,Y2),
      progenitor(Y2,Y3),
      progenitor(Y3,Z).
...
```

El programa anterior tiene dos problemas. Por un lado es muy tedioso para escribir. Por otra parte, sólo detectará los predecesores hasta una determinada profundidad. Una solución a este problema consiste en definir la regla en términos de sí mismo, es decir, en forma *recursiva*. De esta forma, la relación predecesor se puede definir:

```
predecesor(X,Z) :-
      progenitor(X,Z).

predecesor(X,Z) :-
      progenitor(X,Y),
      predecesor(Y,Z).
```

Como vemos, la primera regla servirá para manejar los casos en que la relación predecesor es directa. Usando la terminología de recursión, esta regla actuará como punto conocido del dominio de la definición recursiva. Por otra parte, la segunda regla manejará los casos en que existe una relación predecesor indirecta, y servirá para distintos niveles de profundidad entre dos personas.

Ejercicio: Especificar una consulta que determine las personas de las cuales Pam es predecesor (o sea, los sucesores de Pam), y describir cuales serán las respuestas de Prolog.

4. Variables anónimas

Hasta ahora hemos visto que en las cláusulas pueden aparecer átomos simples como `tom` y `jim` y variables como `X` e `Y`. Los nombres de las variables pueden tener en realidad un número arbitrario

de caracteres, pero deben comenzar con una letra mayúscula. Existen casos, donde el nombre de una variable puede ser irrelevante. Por ejemplo, para definir una regla que determine si una persona tiene un hijo, se podría escribir:

```
tiene_un_hijo(X) :- progenitor(X,Y).
```

Observar que la propiedad de que alguien tenga un hijo, es independiente del nombre del hijo. En estos casos, no tiene sentido inventar un nombre de variable para algo que no nos interesa. Una alternativa consiste en utilizar un tipo de variable especial denominada variable *anónima* que se denota con el símbolo "_" (*underscore*). Usando una variable anónima, la regla anterior se puede escribir:

```
tiene_un_hijo(X) :- progenitor(X,_).
```

Una regla útil para determinar si una variable debería ser reemplazada con una variable anónima, es observar si dicha variable aparece en la cláusula una única vez. Las variables anónimas también pueden aparecer en las consultas y los valores que toman no son mostrados por el sistema Prolog. Por ejemplo, para determinar todas las personas que son progenitoras de alguien, puedo escribir:

```
?- progenitor(X,_).
X=pam;
X=tom;
X=tom;
X=bob;
...
```

5. Objetos de datos

En la Figura 1 se muestra una clasificación de los objetos de datos en Prolog:

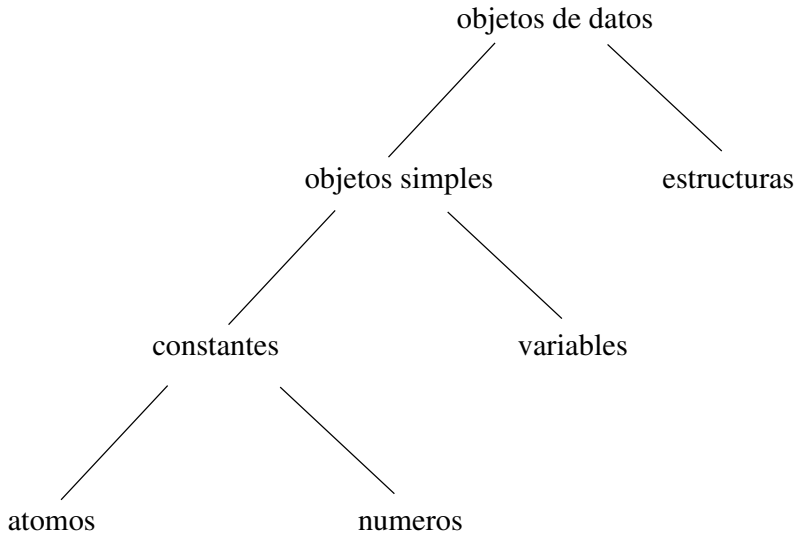

Figura 1: *Objetos de datos en Prolog*

Átomos y números

Los átomos se pueden construir de tres maneras:

1. Cadenas de letras, dígitos y el caracter underscore ("_"), comenzando con una letra minúscula. Ejemplo: `ana, x25, juan_perez`.

2. Cadenas de caracteres especiales. Ejemplo: ==> , ><:>, . : . >.

3. Cadenas de caracteres encerrada entre comillas simples. Ejemplo: `'Ana'`.

Los números utilizados en Prolog incluyen a los enteros (p.ej. 1, 1313, -97) y los reales (p.ej. 3.14, -0.035).

5.1. Variables

Las variables son cadenas de letras, dígitos y caracteres underscore. Las variables comienzan con una letra mayúscula o un caracter underscore. Ejemplo: `X, Result, Pepe2, _x23, _23`. Observar que las variables anónimas son variables especiales que sólo contienen el caracter "_".

El *alcance lexicográfico* de una variable está delimitado por la cláusula que la contiene (un hecho, una regla o una consulta). Esto significa que si el mismo nombre de variable aparece en dos reglas distintas, representa en realidad dos variables distintas. Lo mismo ocurre si el mismo nombre de variable ocurre en una consulta y en una regla o hecho.

5.2. Estructuras

Los objetos estructurados (o simplemente *estructuras*) son objetos que tienen varias componentes. Las componentes de una estructura pueden ser objetos simples (átomos, números o variables) o bien otras estructuras. Una estructura se representa con un *functor* seguido por las componentes (o argumentos) separados por comas y encerrados entre paréntesis. Ejemplo:

```
fecha(8,mayo,2003)
```

Observar la similitud entre la sintaxis de las estructuras y la utilizada para representar las relaciones. Sin embargo, es importante diferenciarlos, ya que las estructuras al igual que otro tipo de objetos como variables, átomos y números siempre las utilizaremos en los argumentos de las relaciones.

Las estructuras se pueden representar como *árboles*. La raíz del árbol es el functor y los descendientes de la raíz son sus componentes. Así, la estructura `fecha(8,mayo,2003)` puede ser representada como el árbol que se muestra en la Figura 2.

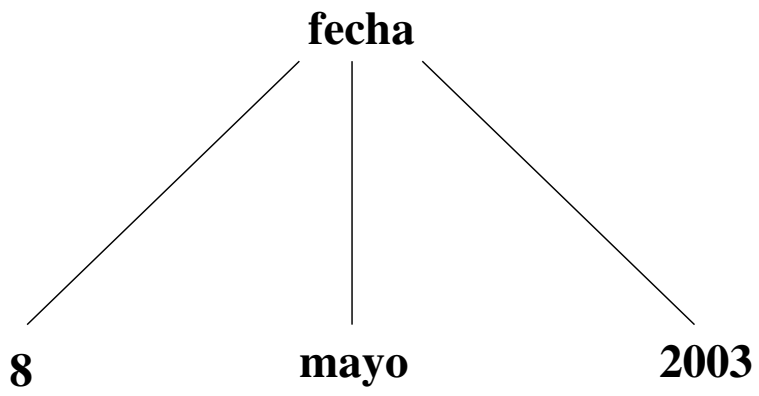

Figura 2: *La estructura `fecha(8,mayo,2003)` representada como árbol.*

Para observar como pueden estar anidadas las estructuras, veamos un ejemplo donde se utilizan estructuras para representar objetos geométricos simples. Un punto en un espacio de dos dimensiones está definido por dos coordenadas; una línea está definida por dos puntos; y un triángulo se puede definir con tres puntos. Para representar este tipo de objetos se podrían utilizar estructuras como las mostradas a continuación:

```
  punto(1,1)              punto(2,3)

     linea(punto(1,1), punto(2,3))

triangulo(punto(4,2), punto(6,4), punto(7,1))
```

Ejercicio: Dibuje los objetos representados por estas estructuras, y dé las representaciones de árbol correspondiente.

En el caso de estructuras anidadas, el functor que aparece en la raíz del árbol se denomina *functor principal*.

6. Matching (coincidencia) de términos

Todos los tipos de objetos que vimos previamente se denominan *términos*. La operación más importante que se puede realizar sobre términos se denomina *matching*. En forma informal se puede decir que esta operación determina si dos términos *coinciden* o en otras palabras *hacen juego*. Dados dos términos, decimos que éstos hacen *matching* si:

1. los términos son idénticos, o
2. las variables en ambos términos pueden ser instanciadas a objetos de tal manera que después de la substitución de las variables por estos objetos los términos se tornan idénticos.

Los términos `fecha(D,M,2003)` y `fecha(D1,mayo,Y1)` son un ejemplo de dos estructuras que hacen matching. Una instanciación que hace estos términos idénticos es:

1. D es instanciada con D1.
2. M es instanciada con mayo.
3. Y1 es instanciada con 2003.

De acuerdo al formato utilizado por Prolog para mostrar los resultados, esta instanciación sería:

```
D = D1
M = mayo
Y1 = 2003
```

El proceso de *matching* toma como entrada dos términos y controla si los mismos "hacen juego". Cuando los términos no coinciden se dice que el proceso de matching *falló*. Si en cambio los términos coinciden entonces se dice que el proceso *tuvo éxito* y se instancian las variables en ambos términos a valores que hacen los términos idénticos.

El proceso de matching se realiza siempre que se utiliza el operador "=" en una consulta:

```
?- fecha(D,M,2003) = fecha(D1,mayo,Y1).
D = D1
M = mayo
Y1 = 2003
```

La instanciación de variables que muestra Prolog no es la única posible. Otras instanciaciones válidas podrían ser:

```
D = 1
D1 = 1
M = mayo
Y1 = 2003

D = pirulo
D1 = pirulo
M = mayo
Y1 = 2003
```

Prolog siempre muestra la instanciación *más general*. Esta instanciación es aquella que compromete las variables en la menor medida posible.

Al utilizarse la instanciación más general se deja el mayor grado de libertad posible para realizar otras instanciaciones en el caso de que se requieran otras operaciones de matching.

Ejemplo:

```
?- fecha(D,M,2003) = fecha(D1,mayo,Y1),
   fecha(D,M,2003) = fecha(15,M,Y).
```

Para satisfacer el primer objetivo, Prolog internamente instancia las variables de la siguiente manera:

```
D = D1
M = mayo
Y1 = 2003
```

Sin embargo después de satisfacer el segundo objetivo la instanciación se vuelve más específica y el resultado de la consulta es:

```
D = 15
D1 = 15
M = mayo
Y1 = 2003
Y = 2003
```

Esto muestra que durante la ejecución de objetivos consecutivos, las variables usualmente se van instanciando con valores que son incrementalmente más específicos.

6.1. Reglas de Matching

Las reglas generales para determinar si dos términos S y T hacen matching son las siguientes:

1. Si S y T son constantes, S y T hacen matching sólo si son el mismo objeto.

2. Si S es una variable y T es cualquier cosa, los términos hacen matching y S es instanciada a T. A su vez, si T es una variable entonces T es instanciada a S.

3. Si S y T son estructuras, éstas hacen matching si:

   a) S y T tienen el mismo functor principal, **y**

   b) Las componentes correspondientes hacen matching.

   La instanciación resultante es determinada por el matching de cada una de las componentes.

Ejemplo:

```
?- triangulo(punto(1,1),A,punto(2,3)) =
           triangulo(X,punto(4,Y),punto(2,Z)).
X = punto(1,1)
A = punto(4,Y)
Z = 3
```

Hasta el momento sólo hemos observado la forma en que se realiza el proceso de matching cuando se utiliza el operador "=". Sin embargo, este proceso se realiza repetidamente durante el funcionamiento del intérprete Prolog. En particular, el matching de términos se realiza cada vez que un objetivo debe ser probado y se busca un hecho o la cabeza de una regla que coinicidan con este objetivo.

Ejemplo: Supongamos que se tiene el siguiente programa Prolog, que permite determinar si una línea es horizontal o vertical:

```
vertical(linea(punto(X,Y),punto(X,Y1))).
horizontal(linea(punto(X,Y),punto(X1,Y))).
```

"Vertical" es una propiedad de las líneas y es representada en este caso como una relación unaria. Vemos en este caso que una línea es vertical si las coordenadas x de sus puntos extremos son iguales. Consideraciones similares se pueden realizar con el predicado "horizontal". Ahora es posible realizar la siguiente conversación con este programa:

```
?- vertical(linea(punto(1,1),punto(1,2))).
yes

?- vertical(linea(punto(1,1),punto(2,Y))).
no

?- horizontal(linea(punto(1,1),punto(2,Y))).
Y = 1
```

Una pregunta que podemos realizar es: ¿hay alguna línea vertical que comienza en el punto (2,3)?:

```
?- vertical(linea(punto(2,3),P)).
P = punto(2,Y1)
```

La respuesta en este caso significa que *si*, y dice además que será cualquier línea que finaliza en algún punto (2,Y1) (la línea vertical $x = 2$). Otra consulta interesante sería preguntar si existe alguna línea que es simultáneamente vertical y horizontal:

```
?- vertical(L) , horizontal(L).
L = linea(punto(X,Y),punto(X,Y))
```

7. Listas

Las listas constituyen las estructuras más simples y útiles que se pueden utilizar en Prolog. Una lista es una secuencia de cualquier número arbitrario de ítems. Una lista con los ítems `ana,tenis,tom,sky` se puede escribir en Prolog como:

```
[ana,tenis,tom,sky]
```

Para representar listas se deben considerar dos casos: la lista es vacía o no vacía. Para el primer caso, se utiliza un átomo especial de Prolog: `[]`. Cuando la lista no es vacía, se puede decir que ésta consiste de dos cosas:

1. el primer ítem, llamado la *cabeza* de la lista;
2. el resto de la lista, llamada la *cola*.

En la lista de ejemplo, `ana` es la cabeza y `[tenis,tom,sky]` es la cola. En general, la cabeza puede ser cualquier objeto Prolog pero la cola siempre será una lista.

Si bien la notación utilizada con la lista de ejemplo no respeta el formato de las estructuras, es importante notar que las listas son en realidad estructuras. Una lista puede ser representada combinando la cabeza y la cola con un functor especial (“.”):

```
.(Cabeza,Cola)
```

Así, la lista de ejemplo podría haber sido representada con esta notación de la siguiente manera:

```
.(ana,.(tenis,.(tom,.(sky,[]))))
```

La notación `[Item1, Item2,....]` es la usada normalmente en los programas Prolog. No obstante esto, es importante observar que sólo se trata de una mejora cosmética en la sintaxis. Internamente, las listas son estructuras de dos componentes que se representan mediante árboles binarios.

Prolog provee otra extensión notacional que permite diferenciar la cabeza y la cola de una lista, utilizando la barra vertical (“|”):

```
[Cabeza|Cola]
```

Con esta notación es posible referenciar la cola completa de una lista como un objeto simple. Ejemplo:

```
?- [1,2,3] = [C|T].
C = 1
T = [2,3]

?- [a,b,c] = [a|[b|[c]]].
yes

?- [ana,pepe,tom,lucas] = [A|[pepe|B]].
A = ana
B = [tom,lucas]
```

La notación con barra vertical puede ser generalizada para listar cualquier número de elementos seguidos por una "l" y la lista de elementos restantes.

```
?- [a,b,c,d] = [a,b|[c,d]].
yes

?- [a,b,c,d] = [a,b|c,d].
no

?- [1,2,3,4,5] = [A,B,C|T].
A = 1
B = 2
C = 3
T = [4,5]

?- [a|[b,c]] = [a,b|[c]].
yes
```

7.1. Operaciones sobre listas

Una operación importante al trabajar con listas es aquella que determina si un objeto es un elemento de la lista. Supongamos que para implementar la operación de membresía utilizamos la relación:

```
member(X,L)
```

donde `X` es un objeto y `L` es una lista. El objetivo `member(X,L)` es verdadero si `X` ocurre en `L`. Ejemplo:

```
?- member(b,[a,b,c]).
yes

?- member(b,[a,[b,c]]).
no

?- member([b,c],[a,[b,c]]).
yes
```

El programa Prolog que implementa la relación `member` es:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Ejercicio: Ejecute manualmente la consulta `?- member(R,[1,2]).`

Otra operación útil sobre listas es aquella que, dadas dos listas, obtiene una tercera lista que representa la concatenación de las dos primeras. Ejemplo:

```
?- concat([a,b,c],[1,2,3],L).
L = [a,b,c,1,2,3]

?- concat([a,b,c],[1,2,3],[a,b,c,1,2,3]).
yes
```

El programa Prolog que implementa la operación `concat` es:

```
concat([],L2,L2).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

Un aspecto interesante de esta relación es que se puede hacer una consulta:

```
?- concat(L,[a,b],[1,2,a,b]).
L = [1,2]
```

Esto muestra un aspecto que difícilmente es logrado por otros lenguajes de programación. Las relaciones permiten obtener resultados, dados ciertos datos de entrada. Lo sorprendente es que en algunos casos, dado un resultado, se puede determinar cuales serían los datos de entrada que permiten obtenerlo.

8. Aritmética en Prolog

Supongamos la siguiente consulta en Prolog:

```
?- X = 1 + 2.
X = 1 + 2
```

El resultado no es `X = 3` como uno esperaría. El problema es que el operador "=" trata de igualar dos términos pero no fuerza la evaluación de operaciones. Para lograr dicho efecto, se utiliza el operador "`is`":

```
?- X is 1 + 2.
X = 3
```

El operador "`is`" también permite evaluar otras operaciones como la resta ("-"), multiplicación("*"), división("/"), potencia("**") y módulo("mod"). Ejemplo:

```
?- X is 5/2, Y is 2 ** 3, Z is 5 mod 2.
X = 2.5
Y = 8
Z = 1
```

Existen operadores que permiten comparar valores numéricos y que automáticamente fuerzan la evaluación de los términos. Algunas operaciones relacionales válidas son la de mayor (">"), menor ("<"), mayor o igual (">=") y menor o igual ("=<"). Ejemplo:

```
?- 3 * 4 > 10.
yes
```

El operador "\=" sirve para controlar si dos términos son distintos.